ManyDSL
One Host for All Language Needs

Piotr Danilewski

March 2017

A dissertation submitted towards the degree
(Dr.-Ing.) of the Faculty of Mathematics
and Computer Science of Saarland University.
Saarbrücken

UNIVERSITÄT
DES
SAARLANDES

Dean

Prof. Dr. Frank-Olaf Schreyer

Date of Colloquium

June 6, 2017

Examination Board:

Chairman

Prof. Dr. Sebastian Hack

Reviewers

Prof. Dr.-Ing. Philipp Slusallek

Prof. Dr. Wilhelm Reinhard

Scientific Asistant

Dr. Tim Dahmen

Piotr Danilewski, danilewski@cs.uni-saarland.de
Saarbrücken, June 6, 2017

Statement

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any degree.

Saarbrücken, June 6, 2017

(Piotr Danilewski)

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, June 6, 2017

(Piotr Danilewski)

Zusammenfassung

Die Sprachen prägen die Denkweise. Das ist die Tatsache für die gesprochenen Sprachen aber auch für die Programmiersprachen. Da die Computer immer wichtiger in jedem Aspekt des menschlichen Lebens sind, steigt der Bedarf um entsprechend neue Konzepte in den Programmiersprachen auszudrücken.

Jedoch, damit unsere Denkweise sich weiterentwicklen könnte, müssen sich auch die Programmiersprachen weiterentwickeln. Aber welche Hilfsmittel gibt es um die Programmiersprachen zu schaffen und aufzurüsten? Wie kann man Entwickler ermutigen damit sie eigene Sprachen definieren, die dem Bereich in dem sie arbeiten am besten passen?

Heutzutage gibt es zwei Methoden. Die erste Methode: es gibt spezifische Werkzeuge und Parser-Generatoren, die zum Schaffen der unabhängigen Programmiersprache von Anfang an dienen. Die zweite Methode: man kann die ausreichend flexiblen exisitierenden Hostsprachen ausnutzen, um in sie kleine DSL einzubetten.

Die beiden Methoden haben eigene Beschränkungen. Einerseits braucht man viel Aufwand um die unabhängige Programiersprache zu schaffen. Diese Sprache ist es schwer mit den anderen Sprachen zu verbinden.

Andererseits sind die eingebetteten DSLs durch Syntax der Hostsprache eingeschränkt. Außerdem wenn die eingebetteten DSLs einmal definiert sein werden, sind sie ständig gegenwärtig.

Es gibt keine Abgrenzung zwischen den eingebetteten DSLs und der Hostsprache. Wenn man viele eingebettete DSLs verwendet, führt es zur Sprachenmischung, die Syntax durcheinander hat. Diese Sprachenmischung hat auch unerwartete Interaktionen zwischen den Sprachen.

In der vorliegenden Arbeit wird die alternative Lösung dargestellt: ManyDSL. Das ist ein einzigartiger Interpreter und Compiler, die aus diesen Lösungen Kraft schöpft und meidet die Schwächen dieser Lösungen.

ManyDSL hat den eigegen LL1 Parser-Generator, der die Beschränkungen meidet, die von der Hostsprache aufgedrängt sind. Beschreibung der Grammatik ist definiert in derselben Programmiersprache wie die anderen Teile des Programms. Die Fragmente der Grammatiken können parametrisiert werden und aus diesen Fragmenten können Funktionen geschaffen werden. Diese Funktionen können zum Schaffen der nächsten Sprachen benutzt werden. Die Sprachen werden während des Interpretationsprozesses geschaffen und sie können benutzt werden um nächste Fragmente des Quellecodes zu parsen.

Ähnlich den eingebetteten DSLs übersetzt ManyDSL alle Sprachen in die Hostsprache. Die Hostpsprache verwendet Continuation-Passing Style (CPS) mit der neuartigen, dynamischen Methode für Staging. Staging erlaubt Partial Evaluation und Ausführung von Quellecode in vielen Phasen. Das kann zum Definieren der Optimierung und der 'zusätzlichen Berechnung' benutzt werden — alles das in der Funktionalen Methode, ohne Abstrakten Syntaxbaum (ASTs) zu benutzen.

Mit der Hilfe von ManyDSL kann der Benutzer neue Sprachen mit der erkennbaren Syntax bauen. Außerdem kann er viele Sprachen innerhalb eines Projektes verwenden. Diese Sprachen haben genaue Grenzen und der Benutzer kann zwischen diesen Sprachen umschalten. Dank diesen Grenzen treten diese Sprachen miteinander in die Interaktion auf kontrollierte Art und Weise.

ManyDSL ist der erste Schritt zum Sprachwechsel in den Programmiersprachen. Mit der Hilfe von ManyDSL möchte ich die Entwickler zum Schaffen der Sprachen, die denen am besten passen, ermutigen. Ich hoffe, dass jeder Entwickler mit der Zeit mit der Hilfe von grammatischen Bibliotheken neue Sprachen schaffen kann.

Abstract

Languages shape thoughts. This is true for human spoken languages as much as for programming languages. As computers continue to expand their dominance in almost every aspect of our lives, the need to more adequately express new concepts and domains in computer languages arise.

However, to evolve our thoughts we need to evolve the languages we speek in. But what tools are there to create and upgrade the computer languages? How can we encourage developers to define their own languages quickly to best match the domains they work in?

Nowadays two main approaches exists. Dedicated language tools and parser generators allows to define new standalone languages from scratch. Alternatively, one can "abuse" sufficiently flexible host languages to embed small domain-specific languages within them.

Both approaches have their own respective limitations. Creating standalone languages is a major endeavor. Such languages cannot be combined easily with other languages. Embedding, on the other hand, is limited by the syntax of the host language. Embedded languages, once defined, are always present without clear distinction between them and the host language. When used extensively, it leads to one humungous conglomerate of languages, with confusing syntax and unexpected interactions.

In this work we present an alternative: ManyDSL. It is a unique interpreter and compiler taking strength from both approaches, while avoiding the above weaknesses.

ManyDSL features its own LL1 parser generator, breaking the limits of the syntax of the host language. The grammar description is given in the same host language as the rest of the program. Portions of the grammar can be parametrized and abstracted into functions, in order to be used in other language definitions. Languages are created on the fly during the interpretation process and may be used to parse selected fragments of the subsequent source files.

Similarly to embedded languages, ManyDSL translates all custom languages to the same host language before execution. The host language uses a continuation-passing style approach with a novel, dynamic approach to staging. The staging allows for arbitrary partial evaluation, and executing code at different phases of the compilation process. This can be used to define domain-specific optimizations and auxiliary computation (e.g. for verification) — all within an entirely functional approach, without any explicit use of abstract syntax trees and code transformations.

With the help of ManyDSL a user is able to create new languages with distinct, easily recognizable syntax. Moreover, he is able to define and use *many* of such languages within a single project. Languages can be switched with a well-defined boundary, enabling their interaction in a clear and controlled way.

ManyDSL is meant to be the first step towards a broader language pluralism. With it we want to encourage developers to design and use languages that best suit their needs. We believe that over time, with the help of grammar libraries, creating new languages will become accessible to every programmer.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Solutions Today . . . 4
1.3 Our Work . . . 6
1.4 The Structure of this Work . . . 10

2 Background **13**
2.1 Language Clasification . . . 13
2.1.1 Language Generations . . . 13
2.1.2 Language Paradigms . . . 15
2.2 Compilation and Interpretation . . . 17
2.2.1 Compilation Phases . . . 17
2.2.2 Type Checking . . . 18
2.3 Staging . . . 22
2.3.1 Fixed Staging . . . 23
2.3.2 Textual Staging . . . 25
2.3.3 Structured Staging . . . 26
2.3.4 Dynamic Code Generation . . . 28
2.3.5 Code as a First Class Citizen . . . 29
2.3.6 Automated Staging . . . 32
2.3.7 Staging in ManyDSL . . . 33
2.4 Language Construction . . . 34
2.4.1 Parser Generators . . . 34
2.4.2 Parser Combinators . . . 35
2.4.3 Code Generation . . . 36
2.5 Language Embedding . . . 37
2.5.1 Plain Shallow Embedding . . . 38
2.5.2 Deep Embedding . . . 39
2.5.3 Shallow Embedding with Staging . . . 42
2.6 Metamorphic Languages . . . 43
2.6.1 The Importance of Syntax . . . 43
2.6.2 Macro Languages . . . 48
2.6.3 Racket . . . 50
2.6.4 Grammar Extension . . . 51
2.6.5 Grammar replacement . . . 53
2.6.6 Metamorphism in ManyDSL . . . 54

3 ManyDSL Overview **57**
3.1 The Main Goal . . . 57
3.2 Properties . . . 59
3.3 Design Decisions . . . 64
3.4 Separation of Concerns . . . 67
3.5 Structure . . . 68

4 ManyDSL Components **71**
4.1 Dynamic Staging . . . 71
4.1.1 Continuation Passing Style . . . 71
4.1.2 Stageless DeepCPS . . . 75
4.1.3 Definition of Dynamic Staging . . . 79
4.1.4 Staging in DeepCPS . . . 83
4.1.5 Using Dynamic Staging . . . 84
4.1.6 Staging as a Graph . . . 87
4.1.7 Programing Patterns . . . 90
4.2 Language Creation . . . 96
4.2.1 Interleaved Parsing . . . 96
4.2.2 Syntax-Directed Execution . . . 97
4.2.3 SDE: Formal Definition . . . 100
4.2.4 LangDSL . . . 103
4.2.5 Grammar Composition . . . 106
4.2.6 Abstraction over Parameters . . . 108
4.3 Parser Actions . . . 110
4.3.1 Building Code . . . 110
4.3.2 Removing the Overhead . . . 115
4.3.3 Branching . . . 119
4.3.4 Converging . . . 123
4.3.5 Loops and Recursion . . . 125
4.4 Conclusion . . . 126

5 Implementation **129**
5.1 ManyDSL Core . . . 129
5.1.1 Target Representation . . . 130
5.1.2 Interpretation . . . 134
5.1.3 Performance Concerns . . . 138
5.1.4 Binding with C/C++ . . . 139
5.1.5 Pitfalls . . . 141
5.1.6 Hardware Reflection . . . 144
5.2 Lang Module . . . 144
5.2.1 Components . . . 145
5.3 Executor . . . 148
5.4 Parsers . . . 148
5.4.1 The DeepCPS Parser . . . 149
5.4.2 The LangDSL Parser . . . 150
5.5 Compiler . . . 150
5.6 Application . . . 151
5.7 Interleaved Parsing . . . 151
5.7.1 Halting in DeepCPS . . . 151
5.7.2 Action Halting . . . 154

5.7.3 Halt and Interpretation 155
5.8 Roads not Taken 156

6 Usage Examples 159
6.1 Stageless DeepCPS 159
6.1.1 Defining Control Flow Structures 159
6.1.2 Extending DeepCPS 163
6.1.3 Memory Operations 166
6.2 Staging 168
6.3 Compilation and Performance 177
6.4 Language Creation Challenges 179
6.4.1 Multiple Passes 180
6.4.2 Building Recursive Functions 182
6.4.3 Environment 186
6.4.4 Type System 196
6.4.5 Language Switching 199
6.5 Language Example: The Array Processing DSL 200

7 Conclusion 209
7.1 Goals Evaluation 211
7.1.1 The Main Goal 215
7.2 Future Work 216

Appendices 229

A Example Execution of Staged Builders 229
A.1 Gluing 229
A.2 Build-time Staging Chain 232

B Manual Building of Fix Nodes 237

C Y-combinator in CPS 239

Chapter 1

Introduction

1.1 Motivation

Language and Thought

In the first half of the XX century two linguists, Edward Sapir and Benjamin Lee Whorf, stated that the language used by humans affects how they view the world [150, 120]. In short — languages shape thoughts.

This Sapir-Whorf Hypothesis refers to human-speaking languages and the perception of the real world. However, the hypothesis can be easily extended to programming languages and the human view on algorithms and computation. The concepts provided by the language we use determine on how we try to solve problems. Moreover, the programming languages we learn early can have a profound influence on how we think about algorithms in general, beyond just the actual implementation. A language style, such as procedural, object-oriented, or functional can inadvertently put us in one mindset, sometimes preventing us from seeing a different, simpler solution [34].

For that reason, we believe it is important to know a wide range of languages and to know which one to choose for a particular problem. Using SQL for physics simulations and FORTRAN for database operations is certainly possible, but inefficient and impractical.

Evolution of Languages

The converse of this hypothesis is also true: Thoughts shape languages. Human spoken languages grow to reflect new concepts we encounter. This way our language and our understanding of the world naturally evolves in small steps to better fit the actual, ever-changing world. New terms, constructs, or even grammar rules are invented to represent new concepts. This evolution of a language can occur on a global scale when the new construct becomes commonly accepted by all its users. However, the change may also occur at a smaller scale,

e.g. in a scientific community. Additions to a language can even be made by a single author for the purpose of a single work or paper.

Programming languages cannot evolve as easily as spoken languages. Any change must be well-defined and then incorporated into the corresponding compiler or interpreter. Users of a language cannot easily change it to fit their needs. Instead, they must refer to compiler experts to apply such a change. The compiler experts on the other hand often lack sufficient domain-specific knowledge to fully understand what is actually needed.

The problem is further compounded by the fact that changes to well adopted languages must fit the needs of all programmers simultaneously. Only then is it likely for the change to be supported by every compiler of that language, and the code written in such language to remain portable among them.

On the other side, a change needed only by a small group of developers appears as unwanted noise to others. Such change would likely not receive much attention by most compiler vendors. Ultimately, such isolated changes lead to incompatible dialects, making source code hard to port between them.

Consider briefly the history of C++: The first C++ reference book was released in 1985 [130]. In the next few years the language received many important additions, such as member pointers, multiple inheritance, or templates [129]. However, as the language became more popular and was widely adopted, the evolution stagnated. The language received only minor improvements and polish, leading to the first international standard only in 1998 [113]. The focus shifted towards supporting libraries and improved compilers, while keeping the language constant. Attention was made to actively prevent it from fragmenting into dialects [128]. Only a decade later a major addition of new features was added in the form of C++11 [60].

Despite the efforts, C++ dialects appeared in the past, some even not having their own name. We had different kinds of C++ supported by GCC, clang, Visual Studio, and others. Writing libraries, such as Boost, with an intention to be used in any of those platforms, soon become a major challenge.

Another problem is backward compatibility. Programmers naturally seek languages that are stable: Code written today should remain valid for many years to come. As a result, it is hard to *remove* features from languages even if they seem irrelevant by today's standards. For example, the C/C++ syntax for function pointer types is often frowned upon, yet it remained the same throughout all revisions.

To summarize – the language needed two decades to make a major step, and even now as the C++ standard is actively being worked on, changes come only every few years. This is a long time, compared to how fast new concepts may appear in everyday development, when their usability scope is limited to a narrow domain and few developers. We would like to be able to adjust the language to the needs of a developer in a matter of days, and not wait years until the language catches up to their demands!

The Need for Languages

When we ask for adjusting a language in a matter of days, surely we do not mean that C++ should change every week. Instead, we advocate that application or domain experts should have the tools to adjust the language of their choosing to meet their needs.

The simplest form of adjusting the programming environment is done simply by writing or adopting a proper set of libraries. These do not change the actual language, but introduce functions, classes, or objects to convey the domain concepts. However, not every concept can be captured in such way. For example, in software development, many design patterns have been identified [43] that remain only as general coding rules that are written verbatim in the source each time they are used. This can easily lead to excess of boilerplate code. The underlying logic of the program becomes hard to comprehend, as the reader must identify these patterns while reading the source code.

Consider for example an observer pattern [43, Chap. 5] — so called *subject* objects maintain a list of dependencies – *observers* – that are notified when the subject is modified. A considerable amount of code needs to be written, spread across several classes to realize this simple concept. Both subject and observer classes must agree on how they interact. Moreover, attention must be taken in the implementation as naive approach can lead to hard to track errors. For example, when observers or subjects are destroyed additional notification must be sent to avoid dangling references or memory leaks known as the *lapsed listener problem*.

The pattern as a whole refers to multiple classes and cannot be encapsulated by a single function. If a language would support the pattern through a dedicated construct, the code would be shorter, easier to comprehend, and the potential pitfalls could be permanently avoided.

The search for further patterns is part of ongoing research, having their reflection in conferences such as "Pattern Languages of Programs". The programming environment must often be adjusted on the language level in order to capture these patterns in a concise way.

Language adjustments are not only about adding new constructs. The converse is almost equally important: restricting a language, preventing unwanted patterns from appearing at all. While general-purpose languages must support multiple coding styles and use scenarios, a language adjusted to a specific project or domain can and should forbid uses that violate their rules. This would allow project architects, following examples from [12]:

- Define and enforce coding style: Does one enforce functional approach, or permit unpure operations? Is using global variables permitted, and if so are there any restrictions accessing them?
- Control resource utilization for performance or security reasons. For example, should program have access to the file system or network?
- Prevent environment-specific API usage in favor of generic libraries to

ensure portability and prevent vendor lock-in.

These rules imposed by the architect would no longer be unwritten rules that the developers of the project must willingly abide to, but an actual language design that enforces proper use and locates mistakes if the rules are violated.

The languages and the set of rules set by project architects may be different between domains, between different applications or even between different parts of the same project. For example, the same project if built in layers, can have different set of rules at each level. Moreover, we expect the project leaders to be able to further adjust and change the languages over time if such need arises.

1.2 Solutions Today

What languages and tools are available to developers today that allow to cover all the use cases described above?

First, if the feature list is the primary concern, one could try using languages that encompass as many possible concepts and paradigms as possible. This however leads to languages that are difficult to learn. Such monstrosities, like PL/1, are overloaded with constructs and keywords [35]. It becomes difficult to create and maintain compilers for such languages and the code is hard to follow by a human as well. Moreover, as future requirements are impossible to predict, such languages still need to evolve over time, increasing their complexity even further.

Building Languages

We accept that there is no single ultimate language to handle all possible user requirements. Instead, we shift our attention towards tools that allow developers to create their own Domain-Specific Language (DSL) as needed. Such tools would simplify the evolution of languages and permit more fine-grained adjustments to match the need of smaller groups of programmers.

First major aid for language creation comes in the form of parser generators, such as YACC [65] or ANTLR [106]. These create independent parsers which, upon reading the input, generate code pieces based on user-supplied actions. These pieces are then connected together, ready to be compiled or interpreted by other tools.

Parser generators mainly focus on the creation of independent compilers from scratch. Even with their help, it still requires a considerable effort to make a functioning compiler: Tasks such as managing names and scopes, type checking, Abstract Syntax Tree (AST) transformation, or code generation need to be explicitly defined by the language creator. Moreover, combining or extending existing languages is not within the scope of these tools. When a user program should cover multiple domains, a different solution is needed that would permit using multiple languages interchangeably.

Embedding Languages

Combining different domains is possible through *language embedding*. In this approach a single host language is used, which is flexible enough for the user to define their own small Embedded Domain-Specific Language (EDSL). This approach is often much easier for the DSL designer, but it struggles in areas where independent languages do not:

- Host language imposes limitation on the syntax of the custom language.
- Embedded languages introduce an additional overhead because of the additional layer of abstraction tied to the syntax of the DSL.

Since we address these problems in our work, let us explain them in a bit more detail here, and show how they are typically addressed:

The host language almost always has a fixed syntax and additional semantics can be introduced only in specific constructs. This is typically achieved through operator and function overloading coupled with a carefully crafted types for a specific domain. Although some languages support higher order functions that allow for a more flexible syntax, there are always boundaries that cannot be broken. These boundaries lead to a syntactic "noise", i.e. code fragments and symbols that must always appear that provide no additional information with respect to the EDSL. A language obfuscated in such a way is harder to use, compared to a stand-alone DSL with its own, separate grammar. In Section 2.6.1 we give detailed examples of that happening.

Only a few *metamorphic languages* permit extensive alteration to the syntax, such as Racket [141] or SugarJ [37]. These systems, however, are complex, requiring the user to manually inspect or modify an AST generated for the host language. Secondly, metamorphs usually extend the host language, instead of creating a new separate one. Without clear separation between host and many custom DSLs the user may once again get trapped with a monstrous syntax that is hard to learn and follow. Moreover, the plenitude of extensions may lead to unexpected interactions and conflicts, something we face for example in the TeX language [78].

The second problem is the performance of the generated code that is tied with how the semantics of the DSL is defined. Traditionally, one distinguish two categories of embedding with respect to semantic definition [16, 44]:

1. In *shallow embedding* the DSL semantic is specified directly in the definition of the language, e.g. in the form of functions.
2. In *deep embedding* a code object, e.g. in a form of AST is created, and the semantic meaning is added later.

In a typical *shallow embedding* the semantics is defined directly via well crafted functions in the host language. The EDSL code is the program, and it is up to the host's compiler to remove any overhead caused by it. Sometimes, the compiler

is not able to elide the extra complexity completely. Moreover, domain-specific optimizations may be impossible to express, as the compiler of the host language lacks the higher-level knowledge to recognize them.

In the alternative solution, the *deep embedding* approach, the EDSL code in the host language generates a new program. Executing the EDSL code, the host creates a program structure, for example in a form of AST. Then, the EDSL provides additional tools in the form of support library to optimize and finally compile such structure into an actual runnable program.

Deep embedding solves the problem of overhead of embedded languages, but at a cost of manual AST manipulation known from standalone language building we described earlier. Moreover, the communication between multiple EDSLs and the host language is restricted as this execution pipeline needs two steps, with a clear separation between host code and each separate EDSL.

Staging in Language Embedding

Staging is a form of partial evaluation, guided explicitly by the programmer. A staged code typically consists of *immediate* and *deferred* parts of code. The immediate part performs symbolic computation, specializing the deferred section. The specialized version of the deferred code is executed only at a later phase, e.g. at run-time.

Deep embedding is a form of embedding that uses staging: The deferred part is represented as the AST, and immediate code handles its creation and transformation. The DSL acts as a generic program, which is specialized for a particular input. The deferred part is the program that the DSL produces.

However, such a structural approach that involves AST creation is not the only way to define staged programs. For example, MetaML [136] and Impala [85] provide means for partial evaluation through staging annotations of a regular source code. With these techniques it is possible to use shallow embedding, but with partial evaluation to reduce the overhead. This is in contrast to the more typical *plain shallow embedding* that does not use any form of staging at all.

In our work we explore a novel, more flexible form of staging: *Dynamic staging*. Especially, we show how it can be used in the context of language embedding. Through staging, our shallow embedding approach is actually brought very close to deep embedding, blurring the line between these two categories.

1.3 Our Work

In this work we present a new solution for multi-domain programing: ManyDSL. It is a comprehensive approach, dealing with both problems stated above at the same time: Syntactic flexibility and program efficiency.

Goals

The main goal of our project is to provide a tool for language creation and manipulation that can be used by programmers, especially in a multi-domain project. Through multiple customizable languages, we want to let programmers achieve a higher degree of expressiveness and abstraction. Custom constructs can capture both domain-specific constructs and algorithms as well as hardware-dependent implementation details.

We want to design a tool with the following properties:

- G1 Languages can be defined easily by non-experts, for the purpose of a very narrow domain or even a single project.
- G2 Separate languages may have their own syntax. The tool should empose as few syntactic constrain as possible. On the other hand, the custom DSL should be able to define its own restrictions if so desired.
- G3 The tool should impose as few syntactic or semantic constraints as possible on the user-defined languages.
- G4 Language definitions should be modular and composable.
- G5 ManyDSL should support and aid the use of many small DSLs.
- G6 Languages should be easily shared between developers, ensuring portability of the code.
- G7 It should be possible to specify domain-specific optimization strategies specified within the DSL definitions.
- G8 The tool should produce efficient machine code despite the additional language layers.

We discuss these goals in detail in Section 3.1.

Design

With these goals in mind we made the following design decisions:

- D1 We use language embedding as a mean to specify languages (G1, G6).
- D2 We use a single flexible host language for language embedding. The language is in Continuation Passing Style (CPS) [69, 6], allowing its user to specify arbitrary control flow (G3). With the host language acting as a base, communication between different languages can be achieved (G5).
- D3 We use shallow embedding with a functional approach to language definitions. Shallow embedding, compared to deep embedding, is easier to comprehend by an average programmer and produces less programming artefacts. (G1)

D4 We let the programmer define its own grammars to break the limitations of the host language syntax (G3)

D5 Grammar components of the language are to be represented as objects in the host language, allowing for parametrization and composability (G4, G6)

D6 We provide a way to clearly separate different DSLs from each other and from the host language. Each DSL may have its own rule set without interference from other languages (G2)

D7 We use partial evaluation as a way to specify optimization and reduce the overhead coming from the embedding (G7, G8)

At present, no tool satisfying our goals and design decisions exist. We are limited by both the theory as well as existing software:

- The staging mechanism as of now is still inflexible (Section 2.3). In Section 4.1 we show how it can be improved by intuitively and formally defining a *dynamic staging* approach, where staging may be a result of computation rather than a fixed annotation in the code.
- As we show in Section 2.6 only a few languages that permit language embedding allow for a custom grammar to be used. These solutions either operate on the AST of the host language or use plain shallow embedding. We provide the first solution for shallow embedding with grammar specification and staging.

Potential Problems

P1 It may be tempting for each developer to define their own language, containing their own syntactic sugars that they deem convenient. In a team project involving multiple developers it may lead to a mixture of languages, each doing the same thing but with a slightly different syntax. Software designers and architects will have to control such misuse of our tool.

It is theoretically possible, for the purpose of such a project, to explicitly define which languages are to be used and combined. The tool itself however does not impose any such constraints — it is up to the architect to define them.

P2 While out tool provide means for inter-language communication, it does not automatically translate the messages. Different languages need to understand each other. To achieve that some standardization of the data representation may be needed, but our tool does not enforce any such format. We hope that over time a proper protocol for such communication will emerge.

P3 Some optimization strategies that require careful analysis of the code, beyond its execution, may be hard or impossible to specify in a system

that relies solely on partial evaluation (D7). It is a design compromise between deep and shallow embedding approaches favoring the latter (D3).

We believe that higher-level, domain-specific optimizations *can* be defined by partial evaluation and code analysis needs to be performed only for low-level optimization — but this is not in our focus.

P4 Interpreting programs represented directly in CPS, especially when coupled with a flexible staging mechanism, may be inefficient. Our focus is on usability and efficiency of the generated machine code, but not on the speed of interpretation or compilation into that code. We hope that the actual implementation of our solution will improve over time.

ManyDSL Structure

ManyDSL is an interpreter and a compiler created with the goals and design decisions described above.

At the core of ManyDSL lies a functional language *DeepCPS* with novel *dynamic staging*, that we introduce in Section 4.1. In Section 4.3.2 and Section 6.4 we show that with staging provided as a first-class citizen of the language, the partial evaluation mechanism can be triggered multiple times for different layers of the program: Staging can be used to peel off the overhead of language layers, it can control the domain-specific optimizations, and finally the DSL itself can expose staging to its user.

In Section 4.2.2 we introduce the Syntax Directed Execution (SDE) scheme for language definition. While similar to tranditional translation grammars [88, 104], we focus on functional execution of the language grammar and its actions. We avoid AST creation, tree grammars, and do not perform any code transformation that are typical for standalone generators or deep embedding. Instead, the action code is executed immediately without any explicit code objects, and uses dynamic staging as means for code generation.

The action code with dynamic staging represents all phases of compilation as executable code. This may include for example syntax dispatching, name resolution, and type checking. With the help of dynamic staging all these phases can, but do not have to be, resolved early to produce efficient and practical code. Any language feature may be *static*, i.e. resolved in an early stage, or *dynamic* such that it is executed at late stage or becomes part of the final compiled code. Consequently, our flexible approach to staging blurs the distinction between the terms “static” and “dynamic”.

A ManyDSL interpreter is coupled with its parser, allowing bi-directional exchange of data. The parser builds a program that in turn can alter the parser again. These changes allow the user to control the grammar that is being used for parsing.

ManyDSL includes a mechanism for interleaved parsing and execution of partially built programs, happening across a single source. Thanks to this, custom DSLs can be included as source libraries, together with the actual program source.

All custom languages defined in ManyDSL are translated into DeepCPS, but the input of each language is a separate entity with its own syntax and semantics. This allows the developer to draw clear lines between domains that are being addressed in the program. The user is encouraged to use several small DSLs rather than a single big language that tries to address all domains simultaneously.

Contributions

Our main contributions of this thesis are:

- A new formal definition of staging at the lambda-calculus level (Section 4.1.3). The staging may be dynamic and value-dependent. The lambda abstraction may encapsulate not only computation but also any staging strategy.
- Introduction of a convenient syntax for writing programs in Continuation-Passing Style (Section 4.1.2). With its help we are able to write and reason about long CPS programs with ease.
- New functional view on parsing using a *Syntax-Directed Execution* scheme, both formally and practically (Section 4.2.2). The parser itself, together with its input becomes an executable program, rather than a source code translation. In this view, the act of parsing is merely a specialization of the parser with respect to the input.
- Design and implementation of the DeepCPS parser and interpreter (Section 5.1). This includes the built-in partial evaluator needed to realize the dynamic staging.
- Realization of interleaved parsing and interpretation with bi-directional communication between the program and the parser (Section 5.7).
- Design of code building blocks that allow creation of programs piece by piece (Section 4.3). The blocks are entirely functional and use staging to remove the overhead in the final program.

1.4 The Structure of this Work

The following chapters are structured as follows:

In Section 2 we lay down the background of our work. In Section 2.1 and Section 2.2 we start by giving a broad classification of the languages, the compilation process and typical work associated with it. In Section 2.3 we then explain the concept of staging and how it is realized in different languages today.

In the second part of the background chapter we focus on the problem of language creation. In Section 2.5 we detail the language embedding techniques and their relation to staging. Finally, in Section 2.6 we discuss the languages that can manipulate their own grammar and syntax. Within it, as a motivation

in Section 2.6.1 we provide examples how syntax limitations can negatively affect the usability of the embedded languages.

In Section 3 we give the oveview of ManyDSL: listing the design goals and decissions and the general structure. We then follow in Section 4 with an in-depth description of the main ManyDSL components. First, in Section 4.1, we describe the theory and application of dynamic staging and define our language DeepCPS. In Section 4.2 we explain how language grammars can be defined in ManyDSL. Finally, in Section 4.3 we show how dynamic staging in the context of language grammar actions allows us to specify language semantics, specify optimization and remove any overhead.

In the Section 5 we focus more on the technical aspect of ManyDSL. We explain the underlying software and algorithms that make ManyDSL possible. In addition, in Section 5.8 we also report on our attempts to implement ManyDSL in a different way that ultimately we had to abandon.

With the theory and implementation of ManyDSL explained, in Section 6 we follow with extensive examples on how to actually use ManyDSL. We give examples to all aspects of the language: direct DeepCPS programming (Section 6.1, Section 6.2), as well as language creation (Section 6.4). In Section 6.3 we explain how DeepCPS can be compiled and what performance of the produced code to expect. Finally, in Section 6.5 we combine the knowledge of all previous examples to define a new language in ManyDSL from scratch.

At the end of the thesis, in Section 7 we evaluate the goals laid out at the beginning and hint on how the project may evolve in the future.

Chapter 2

Background

2.1 Language Clasification

The main subject of this work are programming languages. In order to put our ManyDSL system into perspective, let us first show how languages came into being and how they are classified.

2.1.1 Language Generations

Since the dawn of computing machinery humans searched for a way to describe their algorithms to the computer more concisely and readably. The first programming languages were closely coupled with the computers upon which the algorithms were implemented. Over time languages were capable to encapsulate and hide more and more hardware and implementation details, allowing their users to describe the problems at higher levels of abstraction. Depending on the level of encapsulation, we clasify languages into five *generations* [103].

The first electronic, general-purpose, Turing-complete computer – ENIAC[47] – was originally programmed by manual manipulation of its switches and connectors. Mapping an abstract algorithm onto the machine was not an easy assignment. It was followed by equally difficult task of maintaining the program or finding and fixing any errors. To make the matter worse, the machine itself was unreliable and tended to break down every day. As a result, the whole process of writing a single small program onto ENIAC typically took several weeks [102].

The situation improved with the introduction of *instruction sets*. A program consisted of a sequence of elementary instructions, where each was given a unique number. The sequence, known as *machine code* could be stored in binary form in the machine memory, and reused at a later time when needed [64].

Programming was no longer a matter of closing the right circuits. Instead, it was

a translation from the abstract algorithm into the binary machine code. This concept remains in use in almost all computers today. The binary encoding of programs is referred as a first-generation programming language.

The next step was the introduction of textual *assembly languages*, marking the second generation of languages. These languages aim to encapsulate the arbitrary numeric values assigned to instructions. In this form, instructions and their arguments are represented as short textual mnemonics. This eases the programming effort as the code is more humanly readable, but the code remains machine dependent. The execution process is usually two-step: translation of assembly into the machine code, and then its actual execution on the hardware.

As the number of different computer architectures and program complexity grew, the need for portability arose. This was delivered, at least partially, with the introduction of third generation languages. The first hardware-independent language was Plankalkül [152] in 1945 but it did not draw much attention at the time. Only later, languages such as FORTRAN [10] or ALGOL [108, 9] and later C [71] made third generation languages popular. These general purpose languages allow the programmers to express the intended sequence of instructions independently of the machine it is run on. Each architecture, in turn, provides a *compiler* or and *interpreter* that translates this *source code* into the machine dependent assembly code and eventually the machine code that is ready for execution.

Forth-generation languages step even further towards human readability and productivity: The programmer specifies the task that needs to be performed, without actually providing an algorithm. It is up to the language interpreter to decide how the task has to be done. By its very nature, these languages are designed to solve specific problems rather than providing general purpose functionality. One of the common examples is the Structure Query Language (SQL), taking its origins from SEQUEL [22], which facilitates data manipulation in relational databases. The user specifies a query, and the interpreter decides — based on the database scheme, its indexes, caches, profiling etc. — which strategy or algorithm is the best to fetch or modify the data.

Languages that focus on specific domains – the Domain Specific Languages (DSLs) – try to abstract implementation details away from its user, letting them focus solely on the domain specific problems. As such, most DSLs are considered forth-generation languages. Their user specify the task, and the DSL implementation chooses the best algorithm to realize it. DSLs however, as they mature, tend to gain some of the capabilities of general purpose language.

Finally, fifth-generation languages allow the programmer to specify problem constraints, letting the computer find the solution on its own. The most notable example is Prolog [119, 18, 25], which is used for logic programming, theorem proving, and natural language processing by applying basic logical rules. While powerful, fifth-generation languages are used today to solve only a small set of tasks because of their limitations and slow performance. Today, most focus is on 3rd and 4th generation languages.

2.1.2 Language Paradigms

The third generation of languages allow programs to be specified independently of the hardware. This gives unprecedented freedom in how actually programs are formulated, giving birth to multiple ways of expressing oneself – multiple *programming paradigms* [38]. Many of such paradigms can be identified, each of varying scope and impact. We can identify broad paradigms such as structured programming, dynamic programming, rule-based paradigm, state-machine paradigm, but also identify simple ones – the simultaneous assignment paradigm for example.

For each of these concepts we can ask how a programming language can support them: What paradigms are encouraged in the language by having a simple syntax to express them? In other words, we ask what kind of thinking is stimulated by a given language? Let us give the most common top-level paradigms of third generation of languages.

Imperative Programming

The *imperative* language treats a program as a sequence of statements evaluated in the order of their appearance. Each statement acts as a command that changes the overall state of the machine.

It is the traditional view on programming, stemming from the lower generation languages, but having a generic, hardware-independed meaning. The instruction may ask, for example, to allocate memory, store data, and modify it afterwards, but abstract away how the allocation and modification is performed by any given machine.

The earliest imperative high-level languages, such as FORTRAN [8, 10] or COBOL 60 [62] are unstructured, often consisting of long monolithic sequences of statements operating on global variables. The execution flow is controlled through simple conditionals and jump instructions. These programs are difficult to maintain because of their size, execution flow is hard to follow and any change in the code may have unexpected influence on the behavior of other parts.

Over time, these problems were gradually resolved by introducing subroutines and procedures, giving birth to *procedural programming*. Later on introducing more advanced control-flow structures and introducing local variables gave birth to *structured programming* paradigm. In this approach programs are still seen as a sequence of operations, but most procedures can be treated as separate modules, limiting unwanted interaction between them.

Object-Oriented Programming

An *object-oriented* language groups the data and code that affects this data in the form of *objects* and *classes*. An object can represent a physical or conceptual entity that has a set of attributes and can be acted upon. A *class* acts as a template for objects, defining their behavior, initial value, and acting as a type classification.

A typical class consists of:

- A set of attributes defining objects of this class. Typically, these are object's local and private data values that are inaccessible from the rest of the program.
- A set of *methods* – program code acting on the object of given class. These methods act as the interface of the object and modify its internal state in a controlled way.

Object Oriented Programming (OOP) can be seen as a special case of imperative and structured programming. Each class method can be seen as a self-contained imperative program. However, OOP defines strict rules on how data can be accessed that were lacking in imperative programming. With these rules it is easier for both human and compiler to understand when values are changed and what invariants are maintained.

OOP also introduces *class inheritance* as a form of subtyping. A derivative class *inherits* all the attributes and methods of its parent class, but may extend it with additional set of parameters and functions. OOP also introduce dynamic dispatch through *virtual procedures*, currently more commonly known as virtual methods.

The first language to introduce many of the object-oriented concepts is Simula 67 [29, 100], which later had a strong influence on other languages such as C++, Java, and C#.

Functional Programming

A completely different approach to programing is seen in *functional languages*: A program is not a sequence of statements executed in the specified order, but as a function with its solely goal to compute a final result. The necessary computation to obtain the result is formulated as a mathematical expression over immutable values.

Functional languages treat functions as first-class citizens of the language. Functions can be created and passed as parameters the same way as any data values. This *higher-order* functional programming gives a level of expressiveness that cannot be easily achieved in a pure imperative or object-oriented programs.

In pure functional programming there is no global state and functions do not have any side effects. The compiler can freely choose the order of evaluation. Some subexpressions can be *lazily* evaluated – delaying their computation in hope that their result is ultimately not needed. Through lazy evaluation it is possible to define structures that are infinite, knowing that only the part that is actually needed is ever computed.

Multi-Paradigm Programming

The most generic programming languages try to combine several top-level paradigms described above. This is because more complex problems can be split

into subproblems where each benefit from different kind of thinking. For example, the C and later C++ languages initially embrace imperative and object-oriented programming. However, with the introduction of C++11 standard some aspects of functional programming has been enabled through built-in support for lambda functions and garbage collection.

The ManyDSL core language is also a 3rd generation language with higher-order functional programming in mind. However, we forgo lazy evaluation in favor of granting the user explicit information on the order of evaluation. We permit memory operations and functions having side effects in the style of imperative programming. Finally, the ManyDSL core language supports the staging paradigm, which we discuss separately in Section 2.3.

2.2 Compilation and Interpretation

2.2.1 Compilation Phases

Programs written in every language have to be translated or interpreted before any execution can be actually performed. In a way, it is true even in the case of a machine code — the instruction code has to be interpreted by the hardware and appropriate circuit selected to perform a desired operation. In some cases, complex machine instructions may be actually slower than a set of simpler instructions performing the same action, even on the same processor [107]. The only programs not needing interpretation are those which have been translated to a physical circuit. Such circuit programming was common in the early computers (Section 2.1.1), but remains relevant today when designing new hardware, especially where speed or security are important.

Starting from the second-generation languages, the source code needs to be first translated into machine code. This translation is almost always performed in software, although some machines are specifically designed to support a specific higher-level language – for example the Symbolics 3600 had hardware support for executing Lisp code [99].

The translation to machine code can be done on the fly, by an *interpreter*, for every instruction that is about to get executed. Alternatively, a two-step approach is used: First, the source code is translated by a *compiler*, producing a machine code that later can be executed fast and multiple times. Compilation often involves additional optimization in order to maximize the efficiency of the produced code.

Some compilers, such as Java, do not produce code for any physical machine. Instead, they target an idealized, Virtual Machine (VM) [127]. The produced code remains portable, yet a VM interpreter has a much easier work translating it to actual machine instructions during run-time.

Virtual machine representation is also used for online just-in-time (JIT) compilation.

In modern compilers, the compilation step can be split into even more steps, such as preprocessing, parsing, linking, optimizing, and translating between different representations. Some languages adopt even more intermediary steps, each using different code representation suited for different kind of operations. For example, GCC 5.0 parses the C source into GENERIC trees, which are then translated to GIMPLE representation. Then, after various optimization passes, control flow graphs (CFG) are produced representing the code in Static Single Assignment (SSA) form [124]. After that the code is yet again transformed into Register Transfer Language (RTL) where another set of optimizations is applied. Ultimately, with that many steps, it is hard to directly relate pieces of initial source code and the final machine code that they produce.

2.2.2 Type Checking

Let us define the terms *type checking* and *type system* by first describing the problem it tries to solve:

The plain lambda calculus [24] is a formal system and a language for expressing computation through function abstraction and application. Every value in the lambda calculus is either a symbolic parameter, or a concrete function. Any such value can be bound to any parameter through function application. Numerical values such as boolean and integers are also represented as functions, e.g. through a Church encoding [24].

Such encoding however is impractical, as they are hard to interpret by a human, and incur an additional processing cost for a machine. For that reason, lambda calculus is often extended by explicit primitive terms with an intrinsic meaning. Such terms, such as literal integers, cannot be decomposed further and often have a restricted use. For example, it makes no sense to invoke a literal integer value, as opposed to a lambda of a Church-encoded number. Such an invalid construct cannot be further reduced. When the program cannot be reduced in any way, we say that a program is *stuck*.

In order to prevent this kind of mistakes, *type checking* has been invented to statically analyze lambda expressions before they are evaluated. In a *typed lambda calculus* [23] each term is annotated with a special value called a *type*. The type holds information about what kind of value the term may hold, without actually evaluating the value. For example, if we can deduce that at one point of the evaluation an integer is bound to a parameter x, we annotate x as **`int`**. If we can then find that the x of type **`int`** is invoked at some point, we can flag it as an error — all without actually evaluating what x is.

A *type system* is a set of type values and inference rules which allow to reason about the types. The rules specify how types can be deduced and which constructs are valid. A *type checking* is a process of evaluating the program with respect to a type system.

The above theoretical problem is resembled in real-life programming languages as well. Values, variables and whole expressions are assigned a type – either directly by the programmer, or indirectly through the type deduction. Any

expression that misuses its components, that could potentially lead to a program getting stuck, is recognized as a type error. This way, in the type checking phase, compilers try to detect statically, without actually executing a program, if it can get stuck.

In practice, the type system can be seen as arbitrary annotation explaining the meaning of given data. The type specifies if some given binary data should be interpreted as a number, a string, or some more complex object. Types can be more detailed, for example specifying the physical units that the given quantity is measured with. Lack of such annotation can lead to misinterpretation of data, which in the most extreme cases can lead to disastrous effects such as the crash of the Mars Climate Orbiter in 1999 — where a value given in a pound-force units was treated as if it was given in Newtons instead [91]. If the value had been properly annotated with a type describing the units, the problem would have been found early by the compiler.

Types can be either explicitly specified by the programmer, or the y can be inferred by the compiler statically from other terms. If at some point, the type of an expression cannot be deduced, or types are in conflict, this is considered an error. The type system helps the compiler to statically reject programs that could get stuck. However, at the same time, depending on the constraints of the system — other, possibly meaningful programs can be rejected as well. A wide branch of computer science research is focused on type theory, looking for solutions which successfully find most errors, but are permissive enough for one's needs.

Examples

Simply typed lambda calculus is one of the simplest and most basic example of type systems [23, 53]. Each value is either an atomic type T or a function type of the form $\alpha \to \beta$, where α and β are also types (atomic or function types).

At the same time, the simply typed lambda calculus imposes many restrictions severely limiting the language. Most notably, many generic combinators – functions without free variables – even as simple as generic identity are forbidden. Instead, we need an array of identity functions, specific for a each type.

A particularly important combinators are the fix-point combinators [109]. These are higher-order functions f that for any function argument x, yield $x(fx)$:

$$fx = x(fx)$$

The expression fx is a *fix point* of the function x, i.e. it does not change after applying to x. These kind of combinators pay a central role in defining recursion in lambda calculus. However, in a simple-typed lambda calculus these cannot be typed in their generic way. For that reason, languages using just the simple typing must support recursion through different means, usually by explicitly introducing a special construct.

The simply typed lambda calculus can be expanded in various ways. The most common extensions are:

- **Parametric Polymorphism** — various type systems that allows functions to be parametrized by a type. For example, the identity function can be defined as:

$$\Lambda\alpha.\lambda x^{\alpha}.x$$

  where:

  - Λ indicates a distinct category of functions operating on types
  - α is the type parameter
  - λ indicates a regular lambda function
  - x^{α} is a regular parameter with a type annotation α

  One example is the Hindley-Milner type system [54] that is used in Standard ML with great success [97]. This particular type checker is capable to deduct all the type annotations without any input from the user [31, 96].

  Another example of parametric polymorphism is the type system *System F* [45, 115] that formulates parametric polymorphism through introduction of universal quantification.

- **Type Construction** — allows new types to be constructed from other types. The so called *type constructors* are n-array type operator taking n types as an argument. The constructors are integral part of each type system. Some examples of such usage are:

  - Tuple type for defining a type of arbitrary structures having members.
  - Pointer and reference types

- **Recursive Types** — allows type definitions to be recursive [20]. With this mechanism, every lambda calculus expression can be typed. In particular, it is useful for defining recursive combinators. Taking a simple lambda function as an example:

$$\omega = \lambda x.xx$$

  can be typed as

$$\mu\alpha.\alpha \to T$$

  with any type T. The notation $\mu\alpha$ means that we assign the name α to the whole type that follows the dot, permitting recursion in the definition. In our case it names the whole type that we are just defining. Thus, the above is equivalent to:
  $(\mu\alpha.\alpha \to T) \to T$
  $((\mu\alpha.\alpha \to T) \to T) \to T$
  and so on.

  In practice recursive types are used to define recursive data types, such as lists, trees, or other graphs.

- **Subtyping** — introduces a partial order among types. We say that T is a subtype of U, i.e. $T <: U$ when T is considered a special case of U. In practice it is often seen in object-oriented languages, for example through inheritance, where a derived class is a special case of the parent class.

- **Dependent Types** — one of the most flexible systems, allowing the type value be dependent on arbitrary expression. Dependent typing is hard to capture in a programming language however, thus many mainstream languages do not support it. A few examples that do give such possibility are: Coq [13], Epigram [5], or Matita [7]. Languages with such complex type systems often act as theorem provers or allow the programmer to define program specifications for additional compiler checks.

Each of the type systems described above extend the basic system in a different, ortogonal direction. The systems can be combined together, for example we identify:

- System F_ω – parametric polymorphism combined with type construction
- System $F_<$ – parametric polymorphism with subtyping

Most programming languages adopt one of the systems above, a combination of them, or introduce their own type system, treating the above examples as a starting point.

Dynamic Typing

So far we discussed types in the context of compile-time verification. However, run-time type information may be useful to further strengthen the verification and increase the expresiveness of the language.

Dynamic languages that avoid extensive static type checking often annotate run-time values. Every computation step is preceded by checks of the supplied arguments. Dynamic typing allows for using simpler type systems while permitting highly complex constructs, at a cost of delayed checks. For example, type-checking polymorphic types at run-time is as simple as checking a regular simply-typed language. This is because, when a polymorphic type is used in a check, it is already given as a concrete type constant. The type-checker must verify this single use instance, rather than consider all possible types that may fit the given polymorphic definition.

Annotating data with additional information at runtime allows to realize some advanced concepts, such as dynamic dispatching. Even languages known for their static typing and run-time efficiency do that occasionally. For example, most C++ compilers store a pointer to a table of virtual functions within every instance of a given class in order to manage the virtual functions of that. This pointer is a form of additional information, acting as a type identification.

Type Checking as Computation

Type checking in general can be seen as an additional computation performed by the compiler or interpreter [52]. Such auxiliary computation is a second stream of instructions focused primarily on data properties, aiming at providing early proofs on the correctness of the program. The computation is performed at

compile time (static typing), but may also be a part of the code that is generated (dynamic typing).

Traditionally, the type system is given as an integral part of a language. However, in most languages the computation on types is significantly different than the regular code. This is a highly specialized DSL in disguise, defining the auxiliary computation necessary for checking the program correctness.

2.3 Staging

Traditionally, a *staged execution* is an iterative execution process of a program under the programmer's direct control. In each iteration, called *stage*, only a part of the program is executed. As a result of such stage, a new partially-evaluated version of the program is produced. The new program is specialized with respect to some of the parameters, and is often able to perform the remaining work faster than the original, generic version.

Such execution strategy is a realization of the S-mn theorem [76]. The theorem says that there exists an algorithm that can specialize a function f with $m+n$ parameters, into f_m with known m values and taking only the remaining n arguments. Formally the theorem is defined and proven using the Gödel numbering [46]; in practice, it is applicable to any programming language in use.

The term *staging* refers to any mechanism in the language which allows the user to explicitly control the aforementioned staged execution process.

In Sections 2.3.7 and 4.1.3 we provide a more generic definition of staging, that does not stem from the S-mn theorem. Our approach however does encapsulate the current, more traditional view on staging and the practical techniques remain relevant.

Staged execution should not be confused with automatic partial evaluation, where parts of some code are executed early due to optimization, rather than explicit language specification. For example, one of the common optimizations done by compiler is constant propagation, also known as constant folding [74, 148]. While the technique itself is much more powerful, one of its effects is that expressions that are recognized as constant, are actually evaluated and reduced to a single value. In such situation, the compiler acts as an interpreter, executing parts of the code at *compile-time*, instead of translating it to code that would run at *run-time*. This may seem as an example of staged execution, but the user does not have a full control over it: the compiler may choose to avoid constant folding or perform it only partially, e.g. to reduce the register pressure of the produced code. As such, when the mechanism is not part of the language semantics, but just an optional compiler optimization – it cannot be regarded as *staging*.

With ongoing research, automatic partial evaluation becomes increasingly powerful. Nevertheless, automatic optimization cannot be exhaustive. Due to the halting problem, the compiler cannot always predict if a given optimization is feasible: For example, attempting to evaluate a given expression at compile-time

would actually terminate [70]. On the practical side, too aggressive optimization may also be slow or produce large amounts of code.

The programmer often has a better, high-level understanding about the underlying algorithm that their code represents and how it is intended to be used. They may know that a particular function is safe to run at compile-time, while the compiler itself may not be able to deduce that. For that reason, user-guided staging remains relevant regardless of how extensive the automatic optimization may be.

As we will show in this chapter, deferring an execution of a piece of code is central to defining staging. This is however not equivalent to function abstraction. In a call-by-value or call-by-name λ-calculus, a function has a static body that is never reduced before the said function is invoked [111, Chap 5.1]. Staged code on the other hand can be algorithmically constructed, where parts within the generated code are reduced in an early stage. Such mechanism that allows the user precisely control when such reductions happen requires fundamentally different mechanism than abstraction.

2.3.1 Fixed Staging

As we have shown in Section 2.2.1, the process of compilation and execution consists of several phases. Many modern languages provide ways to explicitly instruct the compiler to execute a piece of code at some of these phases. For example, C++ templates metaprogramming allows arbitrary computation to be staged in the type-checking phase [144, 145]. Similarly, C++11 `constexpr` keyword ensures compile-time evaluation. We refer to these approaches as *fixed staging* as they allow the user to assign additional work to an already existing, fixed stage provided by the compilation pipeline. The user is unable to create a completely new, separate stage this way.

Thanks to this fixed kind of staging, generic libraries emerged. One of the early adoption of it is the Standard Template Library for C++ [126]. The library provides a generic set of algorithms and containers for any data type `T`. When `T` is specified, the library's functions are specialized at compile-time. The produced code often remains within a few percent of the efficiency of the corresponding hand-written routines.

While early adoption of C++ templates focused on specialization with respect to types, it can be used for any computation [144]. Such template C++ programming is refered as *template metaprogramming*. To give a concrete, simple example consider the integer power function written in C in Listing 2.1. Each time the function is called, the recursion is resolved at run-time. As long as the arguments remain unknown a good compiler may, at best, try to replace the recursion with a loop or reduce the amount of recursive calls e.g. by inlining once `power` within itself. In the end however, the depth of the recursion is unknown and must be controlled by a conditional expression over the run-time exponent value.

However, if the programmer knows beforehand that the power function is going

```
int power(float base, int exp) {
  if (n==0) return 1;
  if (odd(n)) /* n%2 */
    return power(base,exp-1)*base;
  else {
    float part = power(base,exp/2)
    return part*part;
  }
}
```

Listing 2.1: *An unstaged power function. Every time the function is called, the recursion is executed in full..*

```
template <int N, bool isOdd> struct rpow {};

template <int N> struct rpow<N, true> {
    static inline float value(float base) {
        return base*rpow<N-1, false>::value(base);
}   };
template <int N> struct rpow<N, false> {
    static inline float value(float base) {
        float part = rpow<N/2, (N/2) & 1>::value(base);
        return part*part;
}   };
template <> struct rpow<0, false> {
    static inline float value(float base) {
        return 1;
}   };

template <int exp> float pow(float base) {
    return rpow<exp, exp & 1>::value(base);
}
...
float pow10(float base) {
    return pow<10>(base);    //transforms at compile-time to:
                             //float partA = base*1;
                             //float partB = partA*partA;
                             //float partC = base*partB*partB;
                             //return partC*partC;
}
...
pow10(2); //returns 1024
```

Listing 2.2: *A staged power function in C++. The template parameter exp is known at compile time. C++ will unroll the recursion at compile time, creating a sequence of plain multiplications to be executed at run-time.*

to be invoked often for a certain constant exponent, he may be interested in a specialized version of this function. When the exponent is known, the recursion, looping, or branching is not needed and the whole expression can be represented as a set of multiplication operators. In order to avoid manual implementation of each specialized copy of the function, staging can be used to generate the necessary code from a generic version. As in example Listing 2.2, with the help of template metaprogramming, we guide the compiler to resolve the recursion at compile time. The multiplication instruction remains in the code to be executed at run-time.

Templates of C++ were originally intended merely for generic programming with respect to unknown types, and became a tool for staging only by accident [144].

```
sub powgenrec {
  if ($_[0] == 0) { return "my \$part = 1;\n"; }
  if ($_[0] % 2) {
    return powrec($_[0]-1) . "\$part = \$part * \$_[0];\n";
  } else {
    return powrec($_[0]/2) . "\$part = \$part * \$part;\n";
  }
}
sub powgen {
  return "sub pow$_[0] {\n" . powrec($_[0]) . "return \$part;\n}\n";
}
eval powgen(10);
...
pow10(2); #returns 1024
print powgen(10) #prints the code:
#                 sub pow10 {
#                   my $part = 1;
#                   $part = $part * $_[0];
#                   $part = $part * $part;
#                   $part = $part * $part;
#                   $part = $part * $_[0];
#                   $part = $part * $part;
#                   return $part;
#                 }
```

Listing 2.3: *Textual staging with PERL, assembling code for power computation for a fixed exponent. After invoking* `eval` *a specialized version of the function is available*

Consequently, such method of staging is cumbersome in practice. For example, as in Listing 2.2, we need to create several partial specializations of a `struct` over an additional helper boolean parameter just to realize a template-based conditional – something that in normal code is realized through a simple `if` statement.

Let us explore other methods of staging, with mechanisms specifically designed for that purpose.

2.3.2 Textual Staging

The simplest way to add an additional stage, is to make a program produce source code of another program, which would be then compiled or executed later.

Many scripting languages allow new code to be pieced together at run-time from character string fragments. Such code encapsulated in a string is then passed into an `eval` function, where it is interpreted, such as in Listing 2.3. Thus, conceptually, the code within this string can be seen as a deferred fragment, that gets executed at a later stage — namely when `eval` is used.

The above, "eval" approach usually uses the same language for both the quoted fragment, and the surrounding code. This is not the only way of doing textual staging though. Sometimes completely different languages are combined. For example, the C compiler actually contains two languages within itself: the actual C, and a preprocessor language [114, Chap. 6.10]. While the C preprocessor is part of the standard, it can actually be supplemented or even replaced by

```
Expression( Apply( Variable(f), Add(Variable(x), Variable(y))))
```

Listing 2.4: *Explicit structural staging of an expression* `"f(x+y)"`. *Each term of the staged language is represented as a structure node with arguments.*

another preprocessing languages, such as M4 [72]. The preprocessing language acts as a first stage, typically performing string manipulations and producing C source as an input to the actual C parser.

Apart from general-purpose preprocessors, there are also domain-specific ones. For instance, tools like `lex` or `yacc` read special user input files and generate C/C++ source code [87].

This approach is inelegant for several reasons:

- The language provides no guarantee that the character string produced in the first stage actually represents a type-checking or even syntactically correct code for the second stage. Any errors are detected late — when attempting to actually parse or compile it. It may also happen that such a composition is exploited through code injection, directing second-stage program to perform different actions than intended by the original author [112].
- Since second-stage variables are created and referred by their names, it is easy to create faulty references or fall victim of name conflicts.
- The syntax and semantics of the first-stage language is often different from the subsequent ones. Even if both stages are technically in the same language, some symbols in the second stage must be escaped because of the string context. For example, in Listing 2.3, we had to introduce `\` in front of every `$` symbol that we produced. This prevents any form of code reusability between different stages. Same functionality has to be reimplemented twice for each language or stage.
- Even if the staged code is semantically correct, it may still fail in type-checking phase. Values produced in the early stage may not match the types used in the following stages.
- There is little communication possible between the different stages. If `x` represents a complex object, it is not enough to put the string `"x"` in the generated code, because at that time the variable is no longer available. Instead, `x` has to be serialized as text and put verbatim in the code of the subsequent stage.

2.3.3 Structured Staging

One step towards a more secure staging is to represent the second-stage code in a more structured way, for example as an Abstract Syntax Tree (AST). This way the staged code no longer needs to be parsed, and the first language controls what code is actually being produced. For example, a fragment code `"f(x+y)"` could be represented as in Listing 2.4. This way the code of the first stage can

```
trait Pow { this: Arith =>
  def pow(base:Rep[Int], exp:Int) : Rep[Int] = {
    if (exp==0) 1
  else if ((exp % 2) == 0) {
    val part = pow(base, exp/2);
    part * part
  } else
    base * pow(base, exp-1)
  }
}
val o = new Pow with ArithExp
val pow72 = o.pow(fresh[Int],72)
...
val res = pow72(3)
```

Listing 2.5: *Staging in LMS. The argument* `base` *as well as the result value are of type* `Rep[Int]`*. Consequently, any operations on these variables produces code rather instead of performing the actual computation. The* `ArithExp` *component that is used with the trait class* `Pow` *holds the definition of the arithmetic performed on the* `Rep[Int]` *type. The* `fresh[Int]` *creates a new* `Int` *node with a fresh, unique symbol that can be later used for identification and referencing, e.g. when binding a value to a function argument. The* `pow72` *becomes a specialized version of* `pow`*, containing only the code operating on what was before a* `Rep[Int]`*.*

ensure that the produced second stage is syntactically correct. However, when code is assembled in such an explicit way, such solution is not reusable between different stages and is hard to read. This becomes even more problematic when stages are nested, i.e. the produced code constructs yet another code piece.

Building of the structure can also be hidden behind overloaded functions and operators. For example, in C++ an expression tree can be built with templates, where each object represents a separate node of the produced AST [143]. That approach however cannot easily handle control flow structures and the supporting template library is complex to use.

A more practical approach was shown recently in Scala, with the introduction of Lightweight Modular Staging (LMS) [116]. A higher-kinded type `Rep[T]` is introduced that denotes a computation that evaluates to `T` in the next stage. With this approach, a code fragment `"f(x+y)"` often can be represented as `"f(x+y)"`, but `x` and `y` need to be declared of type `Rep[_]`. Moreover, the binary `+` and the function `f` must be overloaded for `Rep` types. These overloaded operators hold the generation of the code, and when invoked they produce the result similar to Listing 2.4. Moreover, if `f` is intended to be specialized with respect to its non-Rep types, it first needs to be put in a special trait class, as in example Listing 2.5.

The LMS provides an extensive library already containing overloaded operators for common types. That is why in typical use-cases of LMS, only the actual implementation of the intended algorithm is needed.

Unfortunately, the nature of deep embedding is not entirely hidden. First and foremost, LMS requires a change in function signature, using `Rep` types instead of normal ones. More complex code generators need to use advanced typing techniques such as type tagging [68].

Moreover, since staging is guided by the type system, any change to the default staging behavior requires the programmer to define new types. For example, prohibiting computation even when operands are known one can achieve by introducing dummy types such as `NoRep`, or deriving a new type that would handle both `Rep[T]` and `T` cases [101]. It is neither simple nor intuitive as the programmer must alter staging indirectly, through type manipulation.

2.3.4 Dynamic Code Generation

It is much easier to write staged code when all stages are written using the same syntax and semantics. The compiler is then able to track the relationship between variables used in different stages, in addition to normal semantic verification.

One example of such addition to a widely adopted language is ‘C (Tick-C) — an extension to ANSI C supporting dynamic generation of code [36]. In this approach, the staged program is actually compiled to machine code, and at run-time it dynamically produces new pieces of code. For example, in Listing 2.6 the function `pow` for a given exponent produces a new function at run-time. The newly produced function is given as a pointer and can be invoked in the same way as any regular function.

A backquote unary operator ‘ creates *code* for its argument. The argument may be a value, a function call, but also an arbitrary code block put within curly braces. Two fragments of code can be pieced together with the `@` operator.

```
typedef int (*powtype)();
powtype pow(int exp) {
  int vspec base = param(int, 0); /* Parameter: the base */
  int vspec result = local(int);  /* Local temporary value */
  void cspec code = `{ result=1; };
  int bit = 1;
  while (bit <= exp) {
    code = `{ @code; base *= base; };
    if (bit & exp) code = `{ @code; result *= base; };
    bit <<= 1;
  }
  return compile(`{ @code; return result; }, int);
}
...
powtype pow10 = pow(10);
     /* generates and compiles a code for:
        int pow10(int base) {          //param(...)
          int result;                  //local(...)
          result = 1;                  //code = ...
          base *= base;                //bit=1
          base *= base; result *= base;//bit=2
          base *= base;                //bit=4
          base *= base; result *= base;//bit=8
          return result;               //compile(...)
        } */
```

Listing 2.6: *Building code piece-by-piece in ‘C. The generated function signature is assembled in declerative way using a built-in API. In each step of the loop, another piece of code is appended to the* `code`*. Finally, the complete code is compiled and returned as a function pointer.*

Constant values of the surrounding early stage can be used within the code through the `$` operator.

All the code of Tick-C, both the main code and the dynamically created pieces, is type-checked at compile-time. Code fragments have a type `T cspec`, where `T` is the type returned by the fragment. Dynamically created l-values are of type `T vspec` and can be used to control the variables between multiple code sections. In the Listing 2.6, for example, the **`int`** `vspec` values `base` and `result` are used to reference the parameter and local variable of the function being generated. These values are seamlessly accessed within all of the quoted code fragments.

It is possible to receive a symbolic value from one code section and pass it as a parameter into another code section. Unfortunately, it is not a simple operation. Tick-C provides a set of C library functions to build more complex relationships between code sections and pass the necessary arguments in between them.

Code sections can build a new, complete function F. Through library functions calls, a list of parameters for F can be built incrementally. This creates local variables of F that can be referenced within the code sections. Ultimately, one produces a function pointer that can be immediately used.

However, Tick-C does not support nesting of ` operators. It is not possible to generate dynamic code that would, in turn, generate another dynamic code. Effectively, ` provides a single additional level of staging.

2.3.5 Code as a First Class Citizen

Dynamic code generation adds a single additional stage. Ideally, code fragments should not differ from ordinary values, making them the first class citizen of the language. Same as any other compound value, one should allow code composition, nesting, and allow them to be passed freely between functions.

Lisp

In fact, such flexible solution existed for much longer than ‘C in the Lisp family of languages. The original Lisp language [93] — List Processing — serves both as a programming language and as a formalism for recursive function theory [92]. It is a unique functional language representing the program structure as well as data in the form of nested lists. Any program can be represented as data by a simple quote operator `'( ... )` and evaluated through an eval function. The eval function itself, is relatively easy to define in Lisp itself.

The above mechanism is further supplemented through a *quasiqotation* [11]. The mechanism was originally introduced as part of one of Lisp's dialect, the Conniver language [94], but was later adopted by other dialects as well, including the mainstream Standard Lisp and Common Lisp.

The quasiquotation `` `( ... ) ``, similary to normal quotation, treats the list not as a program, but as a data value. However, within the context of quasiquotation, an *unquote* operator may appear: `,x` which causes `x` to be evaluated again and the result is placed into the quasiquoted list. Furthermore, a convenience syntax

```
(define odd
  (lambda (x) (= (modulo x 2) 1))
)
(define powgen (lambda (exp)
  `(lambda (base)
    ,(if (= exp 0) 1
      (if (odd exp)
        `(let ((part (,(powgen (- exp 1)) base)))
          (* part base))
        `(let ((part (,(powgen (/ exp 2)) base)))
          (* part part))
      )
    )
  )
))
(define pow72 (eval (powgen 72)))
```

Listing 2.7: *Staging through backquoting and unquoting in Scheme — one of the well adopted dialects of Lisp.*

`,@x` evaluates `x` to a list which is then spliced into the containing quasiquoted list.

With these operators staging can be easily achieved, as shown in example Listing 2.7. Quasiquotations can be arbitrary nested and can be freely passed between functions as values, although deeply nested quotations can be hard to read by humans. It is also easy to introduce errors due to staging, e.g. trying to use a symbol with a yet unknown value. As an example, if in the Listing 2.7 we would remove ` before the **let** statements, invoking `powgen` would cause an attempt to multiply with `part` and `base` values, which are not specified at that point in time.

MetaML

Despite these difficulties, quasiquotation – in different forms – became the dominant method for staging. One of the most successful adoption of Lisp's quasiquotations is MetaML [136]. It adds 4 basic operators to handle staging to the standard ML language:

- The angular brackets `< ... >` indicate a *piece of code* that is deferred, defining the next stage.
- The unary escape operator ~ allows splicing multiple pieces of code into one. The escape operator can appear only with the angular brackets. If the argument is a variable representing a piece of code, it is inserted as-is. However, if it is a more complex expression, it is evaluated first before splicing.
- The `lift` function converts any value to a piece of code representing that value.
- The `run` function executes a piece of code.

Similar as with quasiquotation, the pieces of code can be arbitrarily nested. MetaML introduces a notion of *level* of staging. The level at the beginning of

```
fun sqr(x) = <let val y = ~x in y*y end>

fun pow(base,exp) =
  if exp=0 then <1.0>
  else if even exp then
    sqr(pow(base, exp div 2))
  else
    <~base * ~(pow(base,exp-1)>

val pow72 = <fn base => ~(pow(<base>,72))>
```

Listing 2.8: *Staging in MetaML. Pieces of code quoted in* `<...>` *are deferred in execution, but fragments followed by the* ~ *are executed and the value is spliced into the generated code.*

the program is 1, each opened angular bracket increases the level of staging, while an escape operator reduces it. Level 1 code is executed immediately, while higher levels remain intact as pieces of code. Only `run` can dynamically reduce the level of staging by striping off the outher `< ... >` of its argument.

Variables set in one stage can be directly used in another stage without any special mechanism. Only the typical scoping rules define where a variable is visible and where it is not. The ability to refer to variables across stages is referred as *cross-stage persistence* [134, Chap 2.3.1] and is one of the highlights of MetaML.

MetaML is strongly typed. A considerable effort has been made to ensure that any program that type-checks — would execute. Apart from standard type checks, as discussed in Section 2.2.2, it has to ensure that staging is resolved in the correct order. The MetaML type system has to ensure that:

1. No variable is used before it is bound.
2. `run` and escape operator operate on pieces of code.
3. The invocation of `run` does not cause the violation of rule 1.

In a functional language, such as the original ML, the first requirement is trivial since variables are used only within the scope of their definition – a value assignment precedes its usage. With staging this is no longer the case: Names can be used before their assignment. Some operations can still be performed through symbolic evaluation, but other require concrete values. For example `fn a => <fn b => ~(a+b)>` is invalid, as the addition `a+b` is spliced at the stage level 1, before the anonymous function `fn b` is invoked.

The third requirement has shown to be challenging to enforce by the static type system. The `run` operator essentially reduces the level of staging of the piece of code passed as an argument. As a result, a free variable may reach a staging level that would be lower than the stage at which it is being bound, thus violating the requirement 1.

The original MetaML paper [136] allowed `run` to be executed only on pieces of code not having free variables at all. This was shown to be too strong a restriction. For instance, the term `fn x => run x` was rejected. It was observed that `run`

```
fn pow(base,exp) {
  if exp == 0 { 1 }
  else if exp % 2 == 0 {
    let part = pow(base, exp/2);
    part * part
  } else {
    base * pow(base, exp-1)
  }
}
fn pow72(base) {
  @pow(base,72);
}
```

Listing 2.9: *Staging in Impala. The @pow triggers the partial evaluation of pow for the known exponent. The interpreter automatically selects which operations may be performed given incomplete information. As a result, only the multiplication instructions that take the symbolic values base and part remain and are spliced into the context of pow72.*

can be used freely in a top level bindings, at the first staging level [135]. Later, the type checker was improved further, by conservatively estimating possible run-time staging levels and allowing free variables to appear at sufficiently high staging levels [137].

In order to make `run` more flexible while maintaining type safety, MetaML was extended to AIM (An Idealized MetaML) [98]. AIM expands on the idea of pieces of code and makes a distinction between *closed code* that has no free variables, and *open code* that may contain free variables. An `execute` command (counterpart of MetaML's `run`) operates only on the closed code, but its use is no longer limited as it was in MetaML. The downside of AIM is that its use is more verbose than of MetaML.

2.3.6 Automated Staging

MetaML is very verbose: the user must explicitly annotate both the staged section of the code, as well as any expressions within it that should be evaluated and spliced. Manually staging the spliced values is error-prone. Moreover, as we discussed above, the splicing is problematic for the type checker as well. In most cases however, splicing should simply occur whenever it is possible to perform.

Such automated staging is available for example in Impala [85]. Partial evaluation is triggered by putting a `@` in front of a function call. The function is then executed, but sections that depend on symbolic values (e.g. a branch with yet-unknown condition) are automatically skipped. Symbols annotated explicitly by the user through `$` are also skipped.

Consider the example Listing 2.9. The `pow` function looks exactly as a non-staged version and can be used as such. Staging is triggered only at the call site, by putting `@` within the context of `pow72`. Consequently, a symbolic value `base` and a known constant `72` is passed into the generic `pow` and the partial evaluator computes as much as possible, leaving only those operations that need a concrete value of `base`.

Since Impala partial evaluator explicitly skips over operations that would fail during staging, the type system does not need to perform any additional checks. More importnantly however, Impala partial evaluator guarantees that the staged section of the program halts as long as the original, unstaged version of it halts as well. Infinite execution can be induced through staging only when a recursive call is annotated with `@` — something that is detected statically by Impala and raised as a warning.

Automatic staging easily enables *staging polymorphism* – a single function can be partially evaluated in different ways, depending on which parameters are known and which are not.

The ease of use of Impala comes at a cost. Impala imposes only 2 levels of staging, that means the result produced by the partial evaluator cannot contain any staging annotations anymore. Moreover, while Impala staging is automated, it is static: the decision if given instruction is executed early or late cannot be made as a result of a computation.

2.3.7 Staging in ManyDSL

All the solutions we presented so far treat staging as layers of code. The interpreter performs a complete pass over a function, producing a new, specialized one. We find this mechanism not robust enough, however. In a more complex staging scenarios, such as in Listing 2.10, treating staging as complete passes makes programming and reasoning about the code needlessly difficult. When several stages are involved, the programmer must track which level of stage is at any given point and include the corrent number of splice and lift operators, to move between the levels, one by one. If, for example, a code block needs to be put in another staging level, it often does not suffice to put `< ... >` around it. The programmer must inspect all its contents to ensure if the splicing remains correct. Ultimately, as in this example, the amount of stage-related code may actually exceed the amount of regular code, making it hard to understand what is the underlying arithmetic logic of the given function.

For that reason throughout our work we define staging differently. It is no longer a transformation of a function f into its specialized form f_m. *Staging* is any explicit method that allows symbolic computation "under a lambda", that is: performing β-reduction within bodies of functions that have not been invoked yet.

```
fun back2 f = <fn x => <fn y => ~~(f <x> <<y>>)>>
fun dotF2 n v w =
  if n '>' 0
    then <<(~(lift (nth ~v ~(lift n)))
           * (nth ~~w ~(lift (~lift n))))
           + ~~(dotF2 (n-1) v w)>>
fun dot n = back(dotF2 n)
```

Listing 2.10: *The staged dot product function in MetaML [135]. The function can be specialized over the size of the vectors (*`n`*), as well as over the actual value of the second vector (*`w`*). Several escape and lift operators are needed to correctly switch between different levels of staging.*

Staging defined in such a way can still be used for function specialization. However, the way it is achieved is not so rigid: It is not required to specify which function is being specialized or which staging level is currently being evaluated.

In Section 4.1.3 we define *dynamic staging.* It is an extension of the plain lambda calculus at the lowest formal level. Staging is defined through a relation between lambda headers and their bodies, without introducing the concepts of staging levels. With it the user is able to explicitly control the execution order.

2.4 Language Construction

With the increasing amount and complexity of programming languages, an important question is how those languages are created. Naturally, the early compilers had to be written directly in machine code, but this is no longer the case with modern languages.

With the advent of general-purpose languages, compilers could use those languages to build on top of them. In fact, many of the compilers became *self-hosted* — that is, the compiler was written in the language that it was defining. This trend began with a dialect of ALGOL — NELIAC [58] and later with Lisp 1.5 compiler [50]. Today, many general-purpose languages are self-hosted. This is beneficial for practical reasons:

- Upgrades to the compiler back-end may improve not only the normal programs produced by it, but also the compiler itself.
- It serves as a major test of the compiler correctness and language usability.
- Using high-level language to describe the compiler makes it easier to maintain it.

Let us fist inspect the tools available to the programmer to define custom languages.

2.4.1 Parser Generators

The process of compilation or interpretation can be split into a sequence of tasks, such as scanning (lexical parsing), parsing, semantic analysis, optimization, code generation, etc. [3]. Each task, treated separately, can benefit from a description using a higher level of abstraction than the level provided by a general purpose language.

In particular, the description of the front-end of the language – it's syntax and semantics – is often represented in a Backus-Naur form [9]. Instead of implementing a new parser from scratch, the grammar of the language represented in BNF can be used directly as an input for a parser generator. The generator transforms its input, detects any potential ambiguity in the grammar, and produces code — realizing a pushdown automaton either as an executable or as a source code of another language.

One of the first examples of such an approach is the META II language [121]. It accepts a language grammar given as a set of rewrite rules. While it is not explicitly stated, it is using a top-down parser generator, similar to LL(1) [88, 118]. Ambiguities are resolved by a simple backtracking strategy, referred as "back-up".

In addition, the rules contain *output commands* that are produced whenever the given rule is used during the parsing process. The output command can generate an arbitrary character string.

A more robust parser generator that gained popularity is YACC [65]. Based on the grammar description in a format similar to BNF, it produces an LALR(1) parser [2] – a practical approach to parsing of a subset of LR(1) grammars [77].

YACC can parse text input directly, but it is also possible to use it on an already tokenized stream. The tokenization can be done manually by the user, or with the help of another tool, such as lex [86].

Each grammar rule can contain an *action* – a piece of code that is executed each time the rule is taken during the parsing process. Actions are given either in Ratfor or C programming languages. The latter became the dominant use of the tool. YACC often acts as an early stage preprocessor, producing C files that are then used within bigger projects.

After the initial YACC success, further tools were developed, supporting more powerful classes of grammars. We have bottom-up parsers using Generalized LR (GLR) [142] that in worst case perform their task in cubic time but often are able to achieve near-linear parse times. We also have top-down parsers, such as *packrat parses* [40] using Parser Expression Grammars [41], and LL(*) parsers – used for example in ANTLR [106].

While the classes of grammars change, the basic principle remains: The languages are specified as a set of rules, together with their semantic action.

2.4.2 Parser Combinators

Instead of using a dedicated tool, a parser can be defined in a functional language using *parser combinators* [147, 39, 59]. In this approach, a language is constructed by first defining simple, basic parsers represented as ordinary functions. Then, so called combinators are used to take one or more parser functions and combine them to create more complex parsers. In this way, the parsers are gradually enriched until the whole language is denoted by a single parser function.

Because such parsers are created in a general-purpose functional language, the compiler has no insight into the purpose of these functions and does not process them in any special way. It is easy for the user to create a parser that is ambiguous, highly redundant, and possibly backtracking many times. Moreover, if there is an actual error in the input, the parser is slow to recognize it — it first tries to exhaust all possible parsing alternatives in search for a successful solution.

It is impossible to analyze the functions the same way as it is done with a grammar description using a dedicated tool. However, other solutions are available to the language designer, as he can embed arbitrary support code within the parsing functions. For example it is common for the functions to memorize the tokens as they are being consumed. This way, the first/follow sets for LL(1) grammars can be computed dynamically, without analyzing the grammar structure [133].

Other solutions try to explicitly forbid or limit the ambiguities. For example, in the Parsec combinator parser [82], the alternative operator `x <|> y` will try parsing with `y` only if parsing with `x` fails without consuming any tokens. On the other hand if `x` consumes anything, it is assumed that this is the correct path and any further errors do not cause backtracking. If both `x` and `y` can potentially start with the same prefix, it must be left-factored or an explicit `try` keyword has to be used explicitly allowing for backtracking in this particular spot.

Finally, parser combinators can be specialized. Often, when the arguments to the combinators are known, a simpler parsing function can be created through staging and partial evaluation [67]. This results in a parser with less function calls and fewer situations where backtracking may occur.

Unfortunately, all these approaches need to be explicitly implemented by the language designer. There is no support from the language to automatically realize the optimization strategies described above.

2.4.3 Code Generation

Regardless of the approach to parsing, whether through generation or combination, the semantic meaning has to be provided as well. We do not want to just read the input, but also perform some action based on it. This is typically defined by mixing pieces of executable code into the grammar. This pieces, typically referred to as *actions*, can be executed immediately, during parsing, but more often than not they generate code that is to be executed at some later time. In fact, performed action usually spans into multiple stages. We can distinguish for example:

- A *generation stage*, when the generation creates the parser and its actions.
- An *early stage*, such as compile-time, when tasks such as variable lookup or type checking are performed by the actions.
- A *late stage*, e.g. run-time, when the algorithm described by the parsed code is executed.

For that reason staging, as defined in Section 2.3 pays an important role when defining actions.

Many parser generators, such as yacc and most of its derivatives [65], realize the generatoin stage through textual staging (Section 2.3.2). Only few special constructs within actions are treated specially and are substituted by the generator,

the rest is used as-is to generate code for the following stages. Early and late stages are compiled by a different language, e.g. C, and only then action code is actually parsed in a syntactically correct way.

Parser combinators, as well as more advanced parser generators, such as ANTLR v4, typically create a parse tree. Such abstract syntax tree is then later combined with a user-defined visitors or listeners [105]. By doing so, the description of semantics is in the visitor, separate from the grammar description, making the whole language definition more robust. Moreover, the user-defined visitors and listeners are fully parsed in the language of choice and do not suffer from the drawbacks of the textual staging.

The code for the late stage is harder to define. It often depends on the backend, and usually involves calling appropriate API functions, building the target code piece by piece. This resembles the structured staging approach described in Section 2.3.3.

Today, one of the most popular backends is LLVM [80]. It defines a common, low-level code representation for an abstract machine. The user can then specify further transformations within the framework. Many useful optimization strategies and further backends for concrete machines are already available in LLVM itself. This makes the framework a particularly convenient backend for parsers of higher-level languages.

2.5 Language Embedding

We have shown representative examples of tools allowing a programmer to define a new language completely from scratch. However, even with their help it can be a costly process. All aspects of the language must be defined, even if the overall shape closely resembles that of general purpose language maybe with the exception of a few key features for a specific domain. As a result, language building requires time and people who are experts both in compiler construction and the specific domain.

To circumvent this problem, a different approach can also be used. A sufficiently flexible *host language* is used to define all the terms of the new language within itself – usually as a mix of functions, overloaded operators, or special types and generics. When all the definitions are given, the new constructs can be used immediately, as a part of the single program. This kind of language is typically referred as Embedded Domain Specific Language (EDSL) or, in different order: Domain Specific Embedded Language[56].

Examples of languages defined this way date back before the concept itself had been defined. For example Haskell has been used to define geometric operations for naval military applications [57], and later for other domains such as COM component scripting [66], hardware design [15], server-side web scripting [95, 140], etc.

As already mentioned in Section 1.2, language embedding has numerous benefits compared to a standard parser generators, most importantly the cost of pro-

gramming an EDSL is much lower [56]. The listed benefits become even more relevant when multiple embedded languages are used together.

- The complete host language is available at all times. No special handling is needed to switch between host and embedded languages, and back.
- Variables defined in one language can be directly used in the another.
- The host compiler is able to verify the correctness of all constructs, even across multiple languages.
- The embedded languages can benefit from all optimizations available for the host language.

Depending on the implementation, when the EDSL construct is used, two things can happen:

- In *shallow embedding*, the associated semantics is represented entirely within the action. We further subdivide this category into *plain shallow embedding* and *staged shallow embedding*. The plain subcategory does not incorporate staging and the action code is executed immediately during parsing. In staged embedding, the execution of the action or its part may be delayed through staging.
- In *deep embedding*, the action specifies only how a program structure is created. Independently of the grammar, separate programs define how to traverse the structure, how to optimize and transform, and finally how to interpret it.

Let us evaluate these approaches in detail.

2.5.1 Plain Shallow Embedding

Typical shallow embedding does not use staging and is often easy to implement: it does not differ much from defining a library. A set of functions is designed to serve a specific domain. The main difference from a straightforward API is that the functions use the features of the language to increase readability.

Common techniques involve function and method chaining, or consistent overloading of operators for domain-specific types. Consider for example the C++ standard I/O stream library. In native C++ the "<<" and ">>" operators signify shift-left and shift-right bit operations on integers. However, for the purpose of the library these are used as streaming operators, allowing the user to read or write a sequence of data. The library is not just a set of functions to achieve a certain goal – it incorporates a new syntax, through operator overloading, to make the library easier to use.

Shallow embedding is an *extension* to the host language. New constructs often can be easily intermixed with the native code of the host language.

The downside of plain shallow embedding is that the additional language layer may lead to suboptimal performance. The execution order matches the order of the invocations, which for an EDSL may be suboptimal.

Consider for example a problem of multiplying a chain of matrices of statically known dimensions. Finding the optimal order of multiplications is a classic problem that can be solved through dynamic programming [26, Chap. 15.2]. The goal of such optimization is the reduce the dimensionality of the intermediate results. In a plain shallow EDSL this can be solved in two ways:

- Load all matrices into memory, find the optimal sequence and then perform the multiplication.
- Ignore the optimization and multiply the matrices in the order of their appearance.

Both approaches are suboptimal. The first one requires all matrices to be held in memory at the same time and the algorithm itself is performed at run-time. The second approach requires only two matrices to be held in memory at a time, but their dimensions may be higher than necessary due to suboptimal order of multiplication.

Ideally, the dimensions and the optimal order should be computed in an early stage. This would produce instructions in the right sequence for loading and multiplying the matrices at a later stage. This however is a solution that uses staging and is not within reach of plain shallow embedding.

2.5.2 Deep Embedding

In order to circumvent the problems of plain shallow embedding, a different approach is typically proposed [151, 116, 143]. The embedded language does not execute the intended semantics immediately, instead, a structure of computation is created in a form of an Abstract Syntax Tree (AST) or some other intermediate representation (IR). It is a kind of structural staging and domain-specific optimizations can be expressed in two ways:

- As a computation that is performed when the IR is being created.
- As IR transformations.

The dynamic creation of the IR, resulting of an arbitrary computation, is a form of structured staging that we described in Section 2.3.3, with all its benefits and limitations.

On the other hand, IR transformations provide a whole new way of doing optimization. As the language can include additional algorithms that can explore the IR in any way, it has a better chance to deduct the programmer's intention. The optimization can be expressed as an offline algorithm, possibly taking even more time to execute, but producing a highly-efficient final code.

Let us consider the same example as before: The chained matrix multiplication problem. The DSL parser produces an AST representing a sequence of matrix multiplications in the order of their appearance. Then, the language runs an algorithm that inspects the matrices in the AST and transforms the tree to represent the optimal order of the multiplication. Finally, the optimized AST is transformed into a runnable code, that can later be executed or be compiled to machine code.

Typical embeddings with explicit AST creation and transformation are done in functional languages due to their flexibility. Prime examples are Haskell [81] and Scala [55]. It is not uncommon however to encounter embedded DSLs creating an AST in other languages as well, such as C++ [143].

However, one should carefully approach the topic of IR transformations. Specifying them can be compared to imperative programming (Section 2.1.2): It is a sequence of operations that changes the overall state of a machine — where "state" in this context means the current IR of the program. The order of the transformation steps, similarly to imperative programming, can have a huge impact on the end result. When introducing a new optimization one must always investigate how it interacts with other, already existing transformations. Consequently, as the number of transformations grows, introducing new ones for a custom DSL becomes increasingly difficult.

Delite

Consider the usage of LMS for embedded DSL creation [116]. Recall from Section 2.3.3 that LMS is a form of structural staging, hidden from the user through overloaded operators for `Rep` types. These operators construct a program representing the intended computation.

The definition of these operators can be separated from the interface. By swapping different definitions, the same DSL program can produce different IRs and be used with different optimization engines.

One such engine, Delite [117], provides optimization, compilation, runtime support for parallelism, and execution on heterogeneous targets, such as multi-core CPUs and GPUs. Delite is performing several standard optimizations such as common subexpression elimination, dead code elimination, constant folding, code motion, as well as more specialized ones such a loop fusion. Moreover, Delite framework permits usage of higher level IR nodes that encode DSL-specific information. For example, OptiML [21] treats some of the IR nodes as linear algebra operations and may perform simplifications based on the algebraic rules.

The Delite framework assume that all actions are pure by default, that means they do not affect the state of the machine. Imperative operation that modify the state, either globally or locally, must be manually defined by the DSL creator. In particular, to reach maximum performance, the DSL creator must track the possible aliasing of pointers. In OptiML, more advanced optimization strategies are immediately skipped when an unpure function is encountered [132].

Polly

As another example of a typical IR transformation pipeline, consider the Polly tool for polyhedral optimizations [49]. Polly, similarly to Delite, is not a complete language and provides no front-end DSL. It is an an infrastructure for the the IR transformation, operating on a LLVM-IR.

Polly can perform polyhedral optimization only on so called *static control parts* (SCoP) of a function. Several requirements must be met to constitute a block as SCoP, some of which are:

- The only looping control flow is realized through `for` loops, using single integer induction variable incremented by a constant stride.
- An `if` conditional is permitted, comparing two affine expressions with respect to induction variables and block parameters (values immutable within the SCoP).
- All operations may only perform computation using induction variables, block parameters, and values read from arrays.
- All array subscripts must be affine expressions with respect to induction variables and block parameters.
- Arrays represented in memory must not alias

Several transformations on LLVM-IR are performed to produce a block that is equivalent to the requirements of SCoP. However, if the transformations fail to create a code part with the desired properties, it is excluded from further optimization. Unfortunately, there is no mechanism for the programmer to identify the problems and possibly guide the transformations to successfully obtain a SCoP.

Once a SCoP is identified, it is translated from LLVM-IR into its own *Polyhedral Representation*. Only in this form, the actual polyhedral optimizations are performed. When completed,the old code is replaced by new LLVM-IR code.

Evaluation

Let us identify common properties of the tools described above. Both perform optimizations by recognizing special structures in the IR and transforming them. Unfortunately, many complex transformations assume certain properties of the representation, which if cannot be met, prevent the optimization from occurring. Even an attentive developer aware of these limitations may accidentally cause it to fail. When that happens, tracking the cause may be difficult due to the amount of transformations performed in the pipeline.

In order to prevent such mistakes from happening, a DSL must be designed such that is prevents these situations from happening, or provide a way to track the issues to minimize their impact on the underlying optimization.

Moreover, any domain-specific optimization requires usage of domain-specific IR, either through an extension of the basic representation (Delite) or as a complete replacement and translation (Polly).

Effectively, for a successful DSL, its creator must:

- Extend the default IR, or create a new one from scratch
- Define the intended optimization pass over the IR
- Correctly order the transformation passes over the IR to satisfy all the requirements of each

Recall from the beginning of Section 2.5 that the motivation for language embedding is to make DSL creation accessible for non-experts. However, the tasks needed for IR-based optimization are not easy and the overarching goal of embedded language is missed.

2.5.3 Shallow Embedding with Staging

So far we discussed two of the canonical approaches to language embedding: The plain shallow embedding (typically referred to just as “shallow embedding”) where the semantic action is directly represented in the language and executes as soon as the given language construct is encountered. On the other end of the spectrum we have *deep embedding* where a language constructs a program representation, that is to be transformed and executed at a later stage.

In between those two extreme approaches lies a less explored solution – expressing the semantic action directly in the host language, but using staging to optimize the produced program through partial evaluation. No explicit program structure is created and as a result no IR transformations are possible. However, domain-specific optimizations can still be expressed as a staged specialization. This approach for EDSL construction has been used both with MetaML [28] and Impala [85].

Consider the loop fusion optimization: A set of loops are recognized to iteratve over the same range, and the operations within the loop are either independent or have very specific dependency. This kind of loops are very typical, for example, in component-wise array expressions. In those conditions, if an IR-based compiler recognizes this pattern, it may choose to replate the loops by a single one.

In a specialization-based EDSL one tries to avoid creating separate loops in the first place. In Impala, for example, language constructs are defined as a higher-order functions. This way the DSL is able to define its own control-flow constructs, such as a loop with several bodies. By passing body arguments to such a loop, the DSL creates a single fused-loop that can be simplified through specialization.

This way of defining an EDSL requires attention by the language developer: As in the example above, actions may need to be defined differently, compared to how they are typically specified. The domain-level information must be encoded

in a form of abstractions, rather than a set of augmented IR nodes. However, once this is done, it is more natural to reason about the behavior of the DSL. The DSL definition is given as a functional library, rather than as an imperative set of IR transformations.

In our work, we employ a flexible staging mechanism, as mentioned in Section 2.3.7. This direclty improves the staged shallow embedding approach we use in Section 4.3. Together with grammar-based language definitions, explained in Section 4.2, we achieve both syntactic and semantic flexibility.

2.6 Metamorphic Languages

Most programming languages have an immutable grammar and its is definition hidden from the user. Doing otherwise – allowing the user to inspect and modify the grammar, especially on the fly from the source code, is both challenging from the compiler's design perspective and often not deemed necessary.

A *metamorphic language* is a language that may have its grammar modified dynamically. The source code can, during its interpretation, modify or even replace the grammar. Usually, as a result of such change, the later parts of the source can be parsed with the grammar containing user-defined constructs. It is also possible for a metamorphic language to define *island grammars* [33] which describe only some general properties of source in areas where no definite language is yet known. For example, an island grammar may specify that within a specified block of code parenthesis must match and otherwise ignore the contents of that block. Only later, when a user-defined grammar is established, these sections of source code are reparsed.

2.6.1 The Importance of Syntax

Metamorphic languages naturally become good candidates for host languages for DSL embedding: The ability to change the grammar allow embedded languages to have their own, unique syntax.

However, before exploring what metamorphic languages are available, let us first argue if they are needed. Most articles on the embedded DSLs focus on the semantics and correctness of the program [56]. The actual syntax of these DSLs is often secondary or given just as an afterthought. Some articles take a one step further, arguing that a lack of a unique syntax is a positive trait of a language: The most common argumentation is that reusing the known syntax lowers the learning curve of the EDSL.

Is that really the case? We agree that when a DSL can be concisely embedded with the concepts provided by the host language, no new distinct syntax is needed. As we will show in a moment, however, for many DSLs this is not true. Often, DSLs introduce new paradigms that cannot wasily be captured by the host language syntax. Moreover, additional syntax is needed for the host language to distinguish its native code from the domain-specific code. This *syntactic noise* has to be repeatedly injected into the DSL, but it serves no

Embedding	Host	Example
	SQL	`SELECT Name, Surname FROM Members` `WHERE Age = 18`

Haskell/DB	Haskell	`do r <- table Members;`
		`   restrict $ r!Age .==. constant 18`
		`   project $ Name << r!Name`
		`           # Surname << r!Surname`

LINQ	C#	`Members`
		`  .Where(row => row.Age == 18)`
		`  .Select(row => new {row.Name, row.Surname});`

jOOQ	Java	`create`
		`  .select(MEMBERS.NAME,MEMBERS.SURNAME)`
		`  .from(MEMBERS)`
		`  .where(MEMBERS.AGE.eq(18))`
		`  .fetch();`

Slick	Scala	`for {m <- Members if m.Age === 18}` `  yield (m.Name, m.Surname)`

Table 2.1: *Comparison of different SQL embeddings into general-purpose languages.*

additional semantic meaning. Such code becomes less readable, harder to learn, and more prone to mistakes, e.g. when these constructs are omitted.

SQL Embedding

Consider for example a common problem of embedding SQL into a general-purpose language. This scenario is very common in applications that use relational databases in any way.

Many attempts have been made to do so, some of which we show in Table 2.1. Let is explain each entry of that table:

In Haskell/DB [81] the main problem is that additional keywords are needed to distinguish between native Haskell code and embedded code. For example constants in the context of database statement have to be preceded by a keyword `constant`. Moreover, any operator within the database query uses additional dots at the beginning and the end, e.g. "`.==.`" instead of "==".

The C# language integrates SQL queries through LINQ [14]. Such queries can be written in two forms:

- Through *query expressions* which are an actual extension of the C# grammar, specific for SQL.
- Through standard C# syntax.

When using LINQ with normal C# syntax, SQL statements are represented as chained method calls. Each method represents a single construct such as `SELECT` or `WHICH`. The contents of these are encoded as lambda function parameters. The lambda syntax `row => ...` provides no additional meaning with respect of the domain, but must be written anyway for native C# to process it correctly.

In Java, one can create queries through EDSLs such as jOOQ [138], using similar approach of chaining methods. This time no lambda functions are used, but the

content is no longer arbitrary host code. Instead, special predefined values based on the database structure must be used, combined with special operators where needed.

An interesting approach of integrating SQL into Scala is provided by Slick [30]. Instead of trying to mimic SQL closely, it tries to use Scala's own syntax to represent queries. This however effectively means that those familiar with SQL would have to completely relearn how to access a database using this tool. Despite this approach, pitfalls still exist due to limitations of the host language. For example, a regular equality operator == cannot be used in the context of the query, and a new === operator is used instead. Similarly, a not-equal operator is written as =!= instead of !=.

In all these cases we obtain a DSL showing some similarity to SQL, but with additional syntactic noise coming from the fact that the language is embedded. These additional constructs add nothing meaningful to the language but need to be learned by the user. In order to overcome these problems it is not uncommon for a language to actually extend its own grammar in order to explicitly support an SQL syntax.

For example, the *query expressions* introduced with LINQ, alter the C# grammar to support query syntax which is very similar to native SQL [14]. It is possible to write:

```
from row in Members
  where row.Age == 18
  select new {row.Name, row.Surname};
```

Another approach is to have SQL statements preprocessed, before they are included in a general-purpose language. For example PostgreSQL features a preprocessor ECPG for the C language[1]. The user writes C code, together with SQL statements within it. Such .pcg file is then preprocessed into a plain C file, where the SQL statements are converted to C function calls working with PostgreSQL API.

C# query expressions and ECPG preprocessor are examples of how much effort compiler experts can put to introduce a custom syntax into a language that is not metamorphic. Unfortunately, this is a one-time solution specific for SQL. If one wants to introduce another DSL with its own syntax, the same hard work of altering the existing compiler has to be repeated.

Regular Expression Embedding

So far we discussed only the problem of embedding SQL in other languages. This is probably the most common problem, but it is not the only one. For example, a regular expression can be seen as a small DSL for describing a regular automaton over characters. Many scripting languages support regular expressions natively. Other languages introduce regular expressions through some form of library.

Most implementations take the regular expression hidden in a text string. This bypasses the problem of syntactic embedding, but defers any form of correctness

[1] https://www.postgresql.org/docs/current/static/ecpg-concept.html, retrieved on 01.06.2016

checking and optimization until runtime. The expression provided in a string may be incorrect, but the compiler has no way of knowing it.

Moreover, embedding regular expressions in a string can actually interfere with the host language syntax of the string itself. Consider for example a regular expression for a C string:

```
"(\\.|[^\\"\n])*"
```

The formula finds any double-quoted sequence of entries, where each entry is either:

- A single character, with an exception of backslash, double-quotation mark or end-line character
- An escape sequence starting with backslash, followed by any single character, including quotation mark or end-line character

However, if the above formula is put into a literal string as part of some general-purpose language, e.g. C/C++, it becomes much more complicated. The formula must handle the escaping mechanism of the string it is contained in:

```
"\"(\\\\.|[^\\\\\"\\n])*\""
```

Some libraries try to provide their own syntax embedded in the host language. For example, the `boost::xpressive`[2] library introduces its own set of regular expression operators embedded in C++. In such setting, the above formula can be written as:

```
'"' >> *(( '\\' >> _ ) | ~(set='\\','"','\n')) >> '"'
```

As it can be seen in this example, when a language does not support regular expressions on the grammar level, it is hard to reintroduce it through other means. We obtain a DSL with a complex and long syntax to something that is supposed to be short.

SIMD Computing

Another domain we would like to draw attention to is SIMD programming. While it is a general-purpose paradigm, it is still not well adopted in existing languages. The simplest approach is to simply introduce a series of hardware-specific, intrinsic functions for SIMD programming. This kind of programming however is very difficult to write as it effectively lowers the language generation down to low-level assembly programming. In addition to inconvenient syntax, the user must manually handle divergent branches in the form of masking.

Some of the intrinsic SIMD functions can be hidden behind overloaded operators, e.g. for vector computation. However, more advanced problems such as efficient memory organization and diverging branches require changes both on the syntactic and semantic level.

[2] http://www.boost.org/doc/libs/1_61_0/doc/html/xpressive/user_s_guide.html, retrieved on 02.06.2016

```
kernel<<<grid_dim, block_dim>>>(args)
```
(a)

```
var worker = Worker.Default;
var lp = new LaunchParam(grid_dim, block_dim);
worker.Launch(kernel, lp, args);
```
(b)

Listing 2.11: *Comparison between kernel calls in native CUDA **(a)**, and CUDA embedded in C# **(b)**. The native CUDA call uses a clear specialized syntax which can be reused in dynamic parallelism. The embedded CUDA syntax is more verbose but adds no information. Moreover it cannot be used for GPU-side subsequent kernel calls.*

These changes cannot be done through a simple embedding. However, standalone languages such as ISPC [110] or Sierra [83] show that with a single new concept suffices to provide higher-level support for SIMD programming on top of C++. These languages introduce the `varying` and `uniform` type specifier. Variables which are `varying` may hold a different value for each SIMD thread (typically referred as *lane*). On the other hand, `uniform` variables are always the same between across all lanes.

With such information, the compiler has enough information to identify when to generate SIMD code. It can identify which conditionals may branch within the SIMD and how to apply necessary masks.

At the same time, this type extension allows the compiler to reorganize the data in structure-of-arrays (SoA) layout, or a hybrid arrays of short SoA. In plain C it is also possible to define such structures, but are cumbersome to define and use.

While neither ISPC nor Sierra are embedded DSLs, these examples show how C++ syntax with just a few adjustments can convey a new programming paradigm in a concise way.

SIMD programming is not limited just to CPUs. While hardware differences exist, GPU programs can be regarded as SIMD as well. Most GPU-oriented programs are standalone languages, however. This includes highly specialized shading languages such as the RenderMan Shading Language (RSL) [139, part II] or the OpenGL Shading Language (GLSL) [73], which both includes many types and operators specific for graphics processing. A general-purpose programming languages also exist, such as CUDA [27] and OpenCL [17].

However, even the general-purpose languages introduce special syntax to represent parallel computation. For example, in CUDA, when launching a function on the GPU, called a *kernel*, it is invoked with a special syntax as shown in Listing 2.11a.

Such a call launches a kernel on the GPU, and assigns a series of threads organized in an array to execute it. The new syntax <<< ... >>> typically appears as part of the CPU code and invokes the necessary driver functions to launch the kernel. However, with the introduction of dynamic parallelism in CUDA 5.0[3], the same

[3] http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf, retrieved on 07.06.2016

syntax may appear as part of the GPU code, spawning additional threads in effect. The syntax hides all the differences between CPU and GPU invocation of that kind.

There are a few examples where the CUDA programming model has been adopted as an embedded DSL into other higher-level language. For example, the Alea GPU language embeds CUDA programs into the .NET family of languages, such as C#[4]. Such embedding cannot use the kernel call syntax and has to rely on manual function calls that achieve the goal as in Listing 2.11b. Once again, we obtain more verbose code that does not actually convey any additional domain-specific information over the shorter version provided by native CUDA. Furthermore, such an approach with a `Worker` object cannot be used in the GPU code for the CUDA dynamic parallelism.

Conclusion

We have given examples where domain-specific programming benefits from using its own unique syntax. Every time a language brings a new concept or tries to establish a new paradigm, support from the syntax helps to convey the meaning. The code becomes shorter, easier to maintain, and understand.

Furthermore, recent studies show that syntax matters among novices learning a new programming language [125]. For any potentially new DSL – with or without custom syntax – every programmer has to learn the language and can be considered novice. For that reason, the easier, more intuitive syntax of that DSL, the lower its learning curve and a higher chance for the language to be actually used in any given project.

For these reason we claim that using the same homogeneous syntax in all DSLs is counterproductive. Instead, in a multi-domain project, it is actually beneficial to use many languages with different syntax. At each point in the code it is clear which domain one is dealing with and can program with ease in an environment that is suited specifically for that domain.

Now that we have stated the reasons why we claim that syntax is important, let us look at some languages that permit syntax manipulation. This property of a host language is necessary to embed DSLs with arbitrary syntax.

2.6.2 Macro Languages

One of early approaches for syntax manipulation is in the form of macros – small source-to-source transformation patterns. Simplest macro languages, such as the C-preprocessor [114, Chap. 6.10] and m4 [72] operate lexically. That means, the definition of a macro is treated as a sequence of characters or tokens. Macros can have parameters, that are used as a placeholders within the definition. A recognized macro invocation is replaced by its definition "as-is" and arguments replace the placeholders without checking if the produced code is syntactically correct. Often, the macro languages are not Turing complete and recursion is

[4] http://www.aleacubase.com/temp/manual/compilation-overview.html, retrieved on 24.02.2017

limited or not possible at all. TeX [78] is a notable exception of a macro language that is fully programmable.

More advanced macro systems however can operate with additional syntax knowledge, for example by transforming Abstract Syntax Trees. For example Lisp and Scheme introduced the concept of *hygenic macros* [1, 79], which are referentially transparent and prevent accidental name capturing. These advanced macro language can also be combined with already well adopted languages, such as C [149].

One of the more powerful macro languages is the macro system of the `<bigwig>` compiler [19]. New syntactic constructs are defined by:

```
syntax <context> id params ::= { body }
```

- *context* — The name of the host language nonterminal for which the macro can be invoked. The macro evaluates to a node of a parse tree matching this nonterminal.
- *id* — The macro identifier, acting as the first terminal allowing the parser to recognize the invocation
- *params* — A sequence of parameters which may be a mix of terminals and host language's nonterminals.
- *body* — The body of the macro, which is a code snippet, with references to the macro parameters put in angular brackets.

All nonterminals of the host grammar can be used, and can be extended with the macro system. In addition, macros can be grouped into packages, which can be used selectively or be individually extended. Each package can be seen as a different DSL extension of the host language.

Moreover, the macro system allows for the creation of its own nonterminals and grammar rules, through so called *metamorphs*. Metamorphs direct the process of macro expanding and parsing. They can be used only within the context of other metamorphs and macros. They are never used directly as an extension of the host language.

However, the `<bigwig>` macro system is not programmable and recursion is explicitly rejected. While macros are hygenic, metamorphs that define their own nonterminals are not — when using multiple metamorphs, names of one can be captured by another.

In general, macros are not considered first-class citizens of the language and are non-compositional. Their arguments and body are treated differently than the normal code. For example in Scheme when evaluating a macro with nested macro arguments, the arguments will expand only if placed at certain positions while at others they remain as literal invocations. As a result functional composition and higher-level combinators do not easily apply. A careful study and a proposed solution was given, but it requires writing macros in a continuation passing style [75].

Macros can be even more powerful when using semantic knowledge. Semantically-sensitive macroprocessing has been explored in the experimental language XL [90]. XL resembles Scheme but is statically-typed, allowing for the language to provide better assistance, finding possible errors or providing optimizations. With semantic macros, the user can define for example:

- Custom classes of types with their own rules for instantiation, destruction, and type equivalence.
- Atypical control structures (e.g. resembling a finite-state machine)
- Custom resource types, with their own accessibility rules.

The XL language can be enriched semantically, but its syntax is fixed and cannot be altered. While a custom DSL can be embedded in XL, it faces similar problems to other language embedding strategies as described in Section 2.5.

2.6.3 Racket

There are only a few metamorphic languages that allow such extensive modification as Racket [141]. The Racket, a descendant of Scheme, allows for the manipulation of all aspects of the programming language: Its syntax and semantic, type system, linking, and optimizations.

The parsing process of Racket is two-step.

- The *reader* transforms the input characters into syntax objects, which act as Racket's internal representation.
- The *expander* repeatedly process the syntax objects, using user-defined and built-in macros, until the produced code consists only of Racket's core expressions.

Both steps can be customized by the programmer.

A reader can be redefined from scratch by defining all the lexing rules through a combination of regular expressions and other built-in functions. Readers can also be extended by *readtables*, akin to macros, but working on the lexical level. Effectively, this defines the syntax of the user's language.

The expander can be extended through pattern-based macros or general macro transformers. In both cases, macros operate on syntax objects, transforming it to other syntax objects. By supplying a set of custom macros, the user can define or modify the semantics of the language.

Different languages can be put into *modules*, which can be used directly or be installed as a package. This allows them to be used the same way as a regular library. A special syntax at the beginning of a file `#lang myDSL` allows the remainder of the file to be parsed using the `myDSL` syntax.

Racket employs numeric *phases*, similar to staging [141]. The run-time is Phase 0, but compile-time computation — that is, a computation at a higher phase level — is needed when defining custom macros. Since macros can generate further macros, phases of even higher level may be needed. This resembles the staging of MetaML, as described in Section 2.3.5, but it employs different rules of accessing identifiers. The only means of communication between phases is through macro expansion. The output of a macro in one phase, becomes the code executed in the next phase.

While Racket is powerful, it is big and cumbersome. The user needs a deep understanding on the behavior of the Racket parser and its internal representation in order to avoid mistakes. Building custom languages also requires familiarity with the big set of functions provided by the core Racket. For example, there are 41 core procedures specific for macro syntax transformers[5].

Furthermore, composing languages and building new ones on top of previous ones can create additional challenges. For example, pattern matching – the primary mechanism of Racket – can be affected by the amount of macros and the order of their expansion. Thus, the amount and order of language layers can influence how they work. The order of expansion can be manually guided through core functions, such as `local-expand`. Their proper usage however assumes deep understanding of all the expansion steps, and the implementation of all language layers. It can be used as a "quick-fix" when expansion order becomes a problem, but it is not necessarily a methodological solution to the problem in general.

2.6.4 Grammar Extension

A less explored route is to let the source of the program alter the behavior of the parser in a more direct way, by manipulating the grammar via which the language is being parsed. The user defined rules become indistinguishable from the native constructs.

Fortress

The prime and often cited example of grammar-extending language is Fortress [4]. It allows the user to use template-like constructs to define nearly arbitrary syntactic constructs, which become indistinguishable from the core language. The grammar rules are given using the formalism of Parsing Expression Grammars [41].

The parsing is performed in two steps. In the first step, all grammar rules are parsed but their actions are loaded as raw character strings. The new rules modify the original grammar, using the Rats! tool [48] to produce packrat parsers [40]. In the second step, the new extended grammar is used to parse all the actions. This way, actions can use all the new grammar constructions, even if it leads to recursion.

The grammar rules are grouped into *grammar* objects, which can be indepen-

[5] http://docs.racket-lang.org/reference/stxtrans.html, retrieved on 23.03.2015

dently extended, inherited into new grammars, or even combined together. Each grammar can represent a different aspect of the same domain specific language, or of a completely different language. In the end, however, the program has to choose a single grammar that is used to parse it. There is no mechanism to switch from one DSL to another within the program code.

The downside of this approach is that macros and rules (similarly to `<bigwig>` from Section 2.6.2) are not first-class citizens of the language and may appear at global scope only. Also, macros cannot be formed dynamically as a result of some computation.

The grammar transformations are performed on the AST level. The authors argue that transformations could be performed by a multi-staged system, using arbitrary computation, but that would prevent the compiler from confirming the correctness of the produced AST nodes. Since the authors do not want to abandon the AST, they choose to use a template mechanism, where user-defined portions of the grammar are represented by placeholder items that are replaced with concrete nodes during parsing.

Finally, Fortress faces syntactic limitations. Some macros must be split into two to be correctly parsed. For example, in some scenarios the way ellipsis arguments are expanded prevent pattern matching used in Fortress to correctly recognize the macro usage [4]. This can erroneously leading to a syntax error.

SugarJ

A more recent solution is SugarJ [37] – a language build on top of Java, SDF [51], and Stratego [146]. It is capable of extending the Java syntax and semantics with a user-defined grammar through *sugar libraries*. As soon as a sugar library is imported, the new extensions are merged with the original parser of the SugarJ language. After that, the extensions can immediately be used. Sugar libraries can be composed and the definition of new ones can already use the extended syntax provided by preceeding ones.

New grammar is defined in three parts:

- The specification of syntax is described in SDF, typically in a `context-free syntax` section. An existing nonterminal is extended with a new syntax production, generating a new AST node.
- The specification of semantics is described in a Stratego section, referred as *desugaring rules*. These rules, specified as templates, define how the new AST nodes should be handled. During the process of desugaring the node is replaced by the template construct.
- In the final section, the desugaring entry-points are specified.

With the templated approach, the user does not have to deal with the AST explicitly. On the other hand, similarly as in Fortress, templates prevent arbitrary computation at compile-time and manipulation of the generated code. SugarJ also does not provide any advanced staging mechanism. This prevents any custom

languages from expressing complex compile-time domain-specific reasoning or optimization.

2.6.5 Grammar replacement

Grammar extensions provide no clear distinction between the host language and the custom DSL. Instead of creating a collection of useful small languages, one obtains a single language overloaded with potentially many features. Let us investigate systems that permit a complete grammar replacement.

OMeta

OMeta is an embedded language in Cola[6] and Squeak Smalltalk[61] designed for pattern-matching. It adopts an object-oriented methodology to define Parsing Expression Grammars [41]. Grammars are put into *classes* and use a similar extension mechanism to classes known from other object-oriented languages.

OMeta use a single inheritance mechanism. When combining multiple grammars, inheritance cannot be used. Instead, a *foreign production invocation* (FPI) is used. When using a FPI during parsing, the forein grammar of the target production completely replaces the current grammar. From that point onward, the parser operates in a new language. Foreign functions allow the grammars to remain separate, preventing any unwanted interactions and ambiguities.

OMeta rules and grammars can be used as parameters to further rules, creating higher-order rules. OMeta also supports semantic predicates in the grammar productions.

Unfortunately, the grammar actions must be implemented in the host language (COLA or Squeak Smalltalk). Any staging mechanism is limited by those languages. While shallow embedding is directly possible, deep embedding is not supported by those languages – if desired, it must be done manually by the user.

Katahdin

Katahdin is another example of an object-oriented programming language that let programmers define their own grammars [122]. As in other similar solutions, grammar rules and productions are represented as classes. Fixed, pragmatic keywords are introduced to handle typical problems, such as left recursion and rule precedence. Different languages can be used explicitly through the language identification, e.g. a simple keyword `python` can trigger parsing in python-like language:

```
python {
    ... python code ...
}
```

Grammars can also be augmented. By doing so, special constructs can be made enabled in the context of objects of that class.

For example, consider a possible implementation of an SQLite class. Objects of

[6] Cola homepage as of 24.03.2015: http://www.cola-lang.org

```
class SqliteExpression : Expression {
  pattern {
    option recursive = false;
    database:Expression "?" statement:Sqlite.Statement
  }
  method Get() {
    ... the semantic of the "?" operator ...
  }
}
precedence SqliteExpression > CallExpression;
```

Listing 2.12: *Katahdin definition of a new ? operator designed to be used with a database object. The Sqlite.Statement defines the grammar for an SQL statement.*

```
database = new Mono.Data.SqliteClient.SqliteConnection(...);
database.Open();
database? insert into films values(
  "Indiana Jones and the Last Crusade",
  1989, "Steven Spielberg");
print database? select director from films
  where title = "Indiana Jones and the Last Crusade";
```

Listing 2.13: *Example usage of a different DSL associated with an object in Katahdin. Example given in [122], orange SQL fragments highlighted for clarity.*

such a class represent an SQL connection to a database. However, in addition to typical methods of such class, in Katahdin one can define a new pattern that switches to the SQL language in order to represent a query, as shown in Listing 2.12. This allows to write a query directly, as in example Listing 2.13. Compared to traditional approaches where a query is represented as a string, the Katahdin approach is much safer: The query is parsed before it is submitted and any variable arguments are passed as objects and do not rely on string concatenation.

As a result of parsing, Katahdin creates an AST, with each node being an object of some class representing the grammar rule. No code is being executed during parsing. The AST can be traversed afterward, performing the intended actions. The actions must be defined in the host language, which provides no built-in staging mechanism.

2.6.6 Metamorphism in ManyDSL

In our work we want to extend the capabilities of the metamorphic languages described above.

The problem that is prevalent in the previous approaches is that the grammar rules or macros are not first-class citizens of the language. Instead they are treated in a special way by a language. In particular, they cannot be put in a separate scope or function or be created dynamically. In ManyDSL we want new language constructs – both on syntax and semantics level – to be defined as regular functions, removing these limitations.

With grammar rules expressed as functions, it becomes simple to combine them or dynamically create them. Moreover, the grammars and languages themselves can be parametrized – a property we have not seen in any other approach.

The DSL-defining functions should be capable of specifying nearly all aspects of the language, including lookup rules, type checking, or unique control flow routines. We also want to permit a custom DSL to specify its own syntax for staging. We approach this, by defining staging itself as a fist-class citizen of the language, rather than a fixed construct in Section 4.1. Then, we let the DSL designer create the parameters controlling staging. This is one of the most unique goals of ManyDSL not present in any other metamorphic language.

Chapter 3

ManyDSL Overview

3.1 The Main Goal

Consider a group of developers that is working on a highly-specialized programming project. As general-purpose languages lack expressivity of their domain, they consider creating a Domain Specific Language (DSL) to match their criteria. However, if they choose to create one from scratch, they need a person who is both an expert in their domain, knows techniques of compiler construction, and preferably understands the target architecture that the program is going to be run on. Finding such a person with all these traits simultaneously may be difficult.

These requirements are reduced when using tools for language construction. Ideally, with their help, the language designer no longer needs to be a compiler expert. In Section 2.4 we reviewed the most common tools and approaches. Unfortunately neither is satisfactory:

- Plain shallow embedding of a DSL is limited in terms of syntax and the produced program is often inefficient (Section 2.5.1).
- Deep embedding still requires good understanding of IR transformations and compiler technology in general. At the same time, the syntax is still limited by the host language (Section 2.5.2).
- Shallow embedding with staging simplifies DSL creation and the produced code is efficient. Unfortunately, not many tools exist and they still remain limited by the host language's syntax (Section 2.5.3).
- Metamorphic languages give freedom in terms of syntax, but require the DSL developer to dwell even deeper in AST representations and their transformations (Section 2.6).

ManyDSL tries to address this issue. We want to simultaneously:

- Make the tool accessible for non-experts.
- Provide freedom of syntax.
- Produce an efficient code in the end.

The ManyDSL system aims at bridging the gap between hardware, programming languages, scientific domains, and application creation. ManyDSL itself is not trying to specify how any of these aspects should look like, as they are continuously growing and changing. Instead, it is a platform allowing different kinds of developers to specify those aspects *independently*, and then use them or abstra ‘ct them away into other parts of software development.

Ideally, ManyDSL should allow different people to focus independnetly on a different aspects of a project built in this platform. The complete set of developers that lead to creation of an application would consist of three kinds of people:

- The core compiler programmer: The maintainer of ManyDSL and any underlying compilers with deep knowledge of this process and machines that the produced programs are intended to be run on. All hardware-specific or standard optimizations would be handled by the ManyDSL itself, without any user interaction. The work of the compiler programmer would only be done once, after which the ready tool is released to public.
- DSL programmer: Uses ManyDSL to create new languages suitable for a specific domain. ManyDSL itself should allow for abstracting away the difficult aspects of creating languages from scratch, letting the DSL programmer focus on the domain aspect. The DSL creator should be able to define any domain-specific optimizations that are not handled automatically by the ManyDSL itself.

  The DSL programmer may be specific to a single project, who fine-tunes the constraints and capabilities of languages in use to best suit their concrete project.
- Application programmer: A programmer writing concrete applications using the DSLs provided. Typical projects span over multiple domains. ManyDSL should provide mechanisms to connect between the languages of the multiple domains without requiring deep understanding of their inner workings.

Finally, at the end we have the *end user* of the application. At this point any overhead of using ManyDSL should be resolved and the application should behave comparatively to one written in another way. No ManyDSL knowledge should be required to run the application, unless the nature of the project demands it.

Today, many language tools do not allow for splitting between the roles of “DSL programmers” and “Core compiler programmers”. Even when a tool abstracts away from the hardware, e.g. by targeting Java Virtual Machine, its efficient usage still requires deep understanding of compiler construction, such as:

- Abstract Syntax Tree construction, manipulation and traversing.

- Default and alternative macro expansion orders.
- Name substitution and capturing.
- Typing rules and inference.

3.2 Properties

Recall, in Section 1.3 we briefly listed the properties we want ManyDSL to have in order to achieve the aforementioned main goal. Let us bring up the list again and explain each point in detail.

G1 Languages can be defined easily by non-experts, for the purpose of even a very narrow domain or even a single project.

G2 Languages should have their own syntax. The tool should empose as few syntactic constrain as possible. On the other hand, the custom DSL should be able to define its own syntactic restrictions if so desired.

G3 The tool should impose as few semantic constraints as possible on the custom-defined languages.

G4 Language definitions should be modular and composable.

G5 ManyDSL should support and aid the use of many small DSLs.

G6 Languages should be easily shared between developers, ensuring portability of the code.

G7 It should be possible to specify domain-specific optimization strategies specified within DSL definitions.

G8 The tool should produce efficient machine code despite the additional language layers.

G1: Simplicity of DSL Definitions

We want to provide a tool for language construction that does not require deep compiler knowledge in order to use it. Language definition should use such constructs that are easy to comprehend, follow, and later maintain for an average programmer.

By making the DSL definition accessible, we hope to make language design more common. Every developer or team should be able to create a new or adjust an existing language to best match their needs. With such level of control, the adjusted language should help the programmers enforce the project design choices, even if they are unique to their team.

This does not mean that we expect that every language will be easy to implement. Similarly to ordinary programming and algorithm design, there may be challenges that require careful thinking and finesse. However, once the programmer finds a solution, using ManyDSL should no longer pose additional obstacles when implementing it.

G2: Freedom of Syntax

As we discuss in Section 2.6.1, having an expressive but compact syntax in a DSL is important for the application programmer. Any, even most unique concept — if needed — should be representable in a DSL without any additional syntactical noise.

To meet this goal, we expect that the languages defined in ManyDSL can have its own arbitrary syntax. It should be possible for every new language construct to be recognized as a new, unique thing simply by looking at the text, without the need of looking things up, e.g. object types or scopes. By using a unique syntax, it helps the reader recognize which actual DSL is being used at a given place and what is the meaning of the code.

On the other hand, the opposite should also be possible: Sometimes it is prudent to represent a new language construct with a familiar syntax. This is particularly true when the logic remains similar and the difference can be derived easily from the context.

Choosing which syntax should be overloaded and which should be made explicit is a delicate matter that is best understood by the domain expert. For that reason, it should be left to the designer's capable hands, rather than being forced upon by ManyDSL.

Freedom of syntax does not only mean adding new constructs to the language. It also means, that we should be able to remove unwanted elements. By restricting the language, the designer may direct the programmers to a specific way of thinking or coding style. They may enforce coding patterns and idioms such as the Resource Acquisition is Initialization (RAII) [131, Chap. 13.3], and discourage practices that deem dangerous for the stability of the project. Moreover, language restrictions ensure that concepts that the DSL is supposed to encapsulate, are indeed hidden from the application programmer.

G3: Core Flexibility

Any language built with ManyDSL will share the constraints of what the ManyDSL core language can do. For that reason the core itself should impose as few restrictions as possible.

In particular, attention has to be paid to control flow structures. While general-purpose language typically use if-statements and loops, the domain-specific languages may need more specific and unique kinds of loop. The execution flow may be guided by work queues, communication streams, grammar rules, graph nodes, function sampling, database contents etc. It should be possible to define such control flow in a straightforward way.

Consider a more concrete example of an array programming language that define expressions over whole arrays. An array expression, as the one in Listing 3.1, denotes a component-wise operation over all indices of the arrays. A naive implementation would create a loop iterating over all array elements for each operation. For more complex expressions this creates more loops than necessary.

```
A=B*C+D

for i in 1..n temp[i] = B[i]*C[i]   | loop = for i in 1..n;
for i in 1..n A[i] = temp[i]+D[i]   | loop.append { temp = B[i]*C[i]; }
                                    | loop.append { A[i] = temp+D[i]; }
```

Listing 3.1: *Two approaches for the implementation of an array-processing DSL. Given an expression over arrays `A`, `B`, `C`, `D`, the language may convert each subexpression into a separate loop. The produced code needs to be later processed to perform loop fusion and remove the temporary `temp` array. Alternatively, a different type of control flow object may be created, which allows pieces of code to be appended as its body. This way the generated loop is already fused and no code analysis is needed.*

These additional loops causes more time to be spent managing them, but also has a higher intermediate memory footprint as well as worse cache utilization.

A better DSL for array programming should try to perform all computation for a single index and discard all temporaries, before the next cell is processed. One solution to achieve this is to create separate loops as in the naive approach and then merge them through a careful analysis of the code – an operation known as a "loop fusion". Alternatively each expression can generate code that is not a traditional loop in the first place, but allows code concatenation forming a single loop body from the start.

The algorithm can be more complicated when a value of one cell depends on the contents of other, neighboring cells. Not all temporaries can be immediately discarded if their values are still needed by multiple cells. Furthermore, the neighbor access pattern may differ between array indices, most prominently near the border of the array. Thus, most efficient implementation would process the data in a pipelined fashion, possibly split into a few top-level loops, each handling a different access pattern case.

We would like the DSL programmers to be able to specify all these cases where needed, or use libraries provided by others if that kind of optimization is not their primary concern.

G4: Language Modularity and Composability

Even with the most powerful tools, writing a complete programming language from scratch can be a long process. On the other hand, one can identify language constructs that are shared between many languages. For example, basic infix mathematical operators are common between many general-purpose as well as domain-specific languages. For example, within a certain context, the notation *`v1+v2`* is valid in a wide variety of languages, such as C, SQL, CSS (calc notation[1]) or LaTeX (calc package[2]).

ManyDSL should be able to abstract out these common constructs in the form of *language fragments*, that would later be used as building blocks to form a complete language. With their help, the basis of powerful languages would be constructed easily from already existing fragments, requiring the DSL developer

[1] http://www.w3.org/TR/css3-values/#calc-notation, retrieved on 30.03.2015
[2] https://www.ctan.org/pkg/calc, retrieved on 30.03.2015

to manually add only the necessary unique constructs.

Language fragments must be able to take parameters to be properly configured. For example, consider that `infixmath` represents a language for infix mathematical operations. For such abstraciton to be useful in many DSLs, ManyDSL cannot limit itself to only one type of data, such as integers. The `infixmath` should be able to create a part of language for different kind of objects specific to the DSL, such as unit or unitless numeric values, strings, signals, or images. If `T` names the type of data, then `infixmath(T)` should create part of a language operating specifically on `T`. The author of `infixmath` may ask for even more parameters, specifying which actual operators are available and what is their semantics.

Languages should also be able to be built one on top of another. This way very specific DSLs can be defined incrementally, possible by many independent developers. Some DSL developers may focus on low-level optimization strategies, e.g. with the help of hardware reflection. Other DSL creators may include these strategies as a basis to write higher-level languages. Finally, application programers may fine-tune these DSL definitions to precisely meet the requirements of their specific project.

G5: DSL Interoperability

Since we expect projects written in ManyDSL to comprise multiple languages, care has to be taken to handle the transition between these languages as fluently as possible.

The application developer should be able to change the language whenever the domain changes. This may happen not only between files, or at the top level of a file. Domain change may be needed within a construct of another language, such as within a subexpression or a fragment of a procedure body, like in example Listing 3.2. Allowing a language change *anywhere* is an unrealistic requirement though. Instead, we expect DSLs to define points in their syntax where a change is possible.

Secondly, ManyDSL should provide a mechanism to reference entities defined in one language within the context of another. This may refer to simple variables, regular objects or types, but also objects characteristic to a particular DSL should be available from the context of another language. Names should respect

```
SQL sql;
sql.open(...);
SQL::Result result = sql.SELECT * FROM Films WHERE Year=1989;
foreach (auto row : result) {
  printhtml <tr><td>row.title</td><td>row.director</td></tr>;
}
```

Listing 3.2: *An example of how DSL mixing could be used in ManyDSL. Within the C++11 code, using a method `SELECT` on an object of type `SQL` triggers DSL switch into actual SQL syntax. Later on, in order to show the results in a tabular way, a function `printhtml` is used, which switches the DSL into actual HTML syntax. Within the HTML code, existing variables can be used. Moreover, each row of the `result` is an object with fields corresponding to the fields of the table `Films`*

scoping rules, even when they differ between DSLs.

G6: Language Sharing

Languages must be be easily shared. This will permit DSL programmer to work independently from application programmers. When a DSL is ready, it can be used by many developers in different projects.

Multi-language programming also creates a potential danger when distributing source-code libraries. Suppose a library is defined using a custom language, or a combination of several such languages. If another person tries to use such a library, they must first obtain all the necessary languages.

Ideally, we would like languages to be shared as easily as libraries. This way languages could be simply part of libraries without any special distinction.

G7: Domain-Specific Optimization

Domain Specific Languages not only help the programmer write more concise code. The code can also convey more information to the compiler about the programmer's intend, letting the compiler perform much more aggressive optimizations. It is up to the DSL designer to specify those optimizations and ManyDSL must provide means to specify them. This must be achieved without exposing the inner compiler workings in order to maintain the separation between the compiler and domain experts.

The domain specific optimization may depend not only on the domain itself, but also on the hardware the code is ultimately compiled for. We do not want the hardware characteristics to be entirely hidden from the language or the application programmer. At the same time, we do not want the programemrs to tediously rewrite code for any new machine he targets to achieve the maximum performance.

Instead, ManyDSL should provide *hardware reflection* – a description of the hardware provided as a set of data structures. The language and application programmers may use this information to fine-tune and specialize the code without source duplication. Such reflection should be simple enough for the DSL developer with limited hardware knowledge to successfully harness the power of the specific hardware.

The code produced by the DSL would be *hardware generic*. Given a specific hardware description, it would be specialized for it, possibly in some early stage of the compilation process.

G8: Generation of Efficient Code

Finally, we want the code produced by multiple DSL layers to be efficient. In particular, we need to ensure that the fact of mixing languages and dynamic treatment of syntax and semantics does not cause overhead in the code we produce. At runtime there should be no visible distinction between different origins of the code.

Moreover, ManyDSL should feature a compiler, capable of producing machine code that would be competitive with hand-written code.

3.3 Design Decisions

DSL Interoperability through Embedding

How should ManyDSL support custom DSLs? Through what means should different DSLs communicate, e.g. share variable values, permit cross-language function calls, etc?

We have chosen to embed custom DSLs in a single, flexible host language. The language acts as a bridge, achieving the degree of interoperability that we require (G5).

Moreover, the language descriptions themselves are expressed in the host language. With it, parts of language descriptions can be abstracted out or can interact with each other in the same way as two regular pieces of code (G3). It is also possible to define custom DSLs specifically designed for the domain of designing new languages, hiding inconvenient intrinsics of the host language. As we show in Section 4.2.4 we use this mechanism to define one such language-defining language – LangDSL. Our solution does not prevent other programmers from designing another, possibly better, DSL with similar purpose in the future.

Having language definitions embedded in the host language also satisfies our language sharing goal (G7). The language description does not differ from other normal code and can be shared as any other regular library.

Metamorphism of ManyDSL

Embedding languages may potentially limit the allowed syntax of custom DSLs. We believe that the answer lies in metamorphic languages (Section 2.6) as they can provide full syntactical and semantic flexibility (G6). Using them requires certain knowledge of formal language theory, but other aspects of compiler construction can be hidden (G1).

However, unlike some of existing solutions, we do not want to gradually extend a single host language, but to create grammatically separate languages. While we permit language switching at predefined locations, we want to avoid creation of huge languages trying to encompass everything. By working with different languages, each can be as flexible or as restrictive as its creator wants, without any unexpected interaction from other languages (G6).

Core Language

One of the fundamental decisions is the choice of the core language. As we explained in the previous chapter, the core should be flexible to support a wide array of constructs. It should be sufficiently low-level so that it does not limit the DSL built on top of it (G4).

```
r = (2+3)*(4-5)
...
```

(a)

```
+ 2 3 (λx.
  - 4 5 (λy.
    * x y (λr. ... )))
```

(b)

Listing 3.3: *Representing nested expression(3.3a) in CPS(3.3b). Each mathematical binary expression takes 3 arguments, the last one being the continuation. Subexpressions must be explicitly computed before their partial results are used.*

We have decided that the core to be a functional language in a Continuation Passing Style (CPS)[69, 42, 6]. CPS imposes addition constraints to functional programming:

- Functions never return. Instead, they typically take an additional parameter – a *continuation* – representing the rest of the program. When the function finishes its main computation task, it invokes the continuation so that the rest of the program can execute.
- Functions cannot return values. Instead, the result is passed as an argument to the continuation, so that it can be used in the rest of the program.
- Subexpressions are disallowed. Having them would be pointless, since such subexpression – being another function call – cannot return. Instead, subexpressions have to be explicitly computed earlier and the partial result, given as a concrete value, used later.
- The order of execution is well-defined in CPS – it follows the order of nesting. There is no ambiguity, for example, coming from the order of function arguments, since the arguments must be concrete values and cannot be subexpressions.

CPS can be formally defined with very few rules [111, Chap. 5]. All built-in operations and control flow structures can be represented as functions. A branch instruction can be defined as a core intrinsic function and it is the only one needed to define more advanced constructs. High-level language flows, such as exceptions, can be represented as a continuation function.

Programming directly in CPS directly can be hard. In order to reduce the entry level difficulty and increase readability, we have chosen to use our own syntax to represent CPS programs. A full description of the syntax of our core language DeepCPS is given in Section 4.1. The key aspect is that while a complex function body must be put in curly braces, it can be skipped for the last argument, which – in case of CPS programming – is typically the continuation. This allows to reduce the apparent nesting. Consequently, DeepCPS code looks like a functional assembly language.

Code Generation through Staging

Another question requiring an answer is: How to generate code? We believe that efficiently using Abstract Syntax Trees requires deep compiler knowledge. Defining optimization as tree transformations is imperative in nature, as we

discussed in Section 2.5.2. For that reason, such approach can be difficult and unreliable when the tree is constructed from multiple languages. It is generally impossible to predict all DSL combinations that the application developer may use. ASTs can also be a limiting factor: New language constructs require either new AST node types, or subtree patterns, or both.

For that reason, we decided to forgo the AST approach completely. Instead, language definition contains action code that is executed immediately by the interpreter. We rely on staging to have the actions create pieces of code. We originally looked at MetaML-like approach (Section 2.3.5), but then explored a more flexible solution, giving birth to *dynamic staging* (Section 4.1). Dynamic staging, defined as a part of our core language DeepCPS, treats all staging information as a first-class citizens of the language. This property allows for passing staging information as parameters between grammar actions, as well as letting the custom language define its own explicit or implicit staging constructs. It is a powerful and intuitive approach, albeit verbose.

As we show in Section 4.3.1, dynamic staging can be used to define so called *builder* functions. These builders can be used to construct code piece by piece. Staging can then resolve any glue code overhead, and produce a program as if it was written in native DeepCPS.

The builders themselves bare similarity to structured staging again, but their semantic meaning is defined entirely in a functional way within themselves. No external visitor patterns or manipulation of the tree is needed. Domain-specific optimizations are defined by the contents of the nodes, using dynamic staging (G2).

Type System for the Core Language

The type system of a language may greatly improve the productivity of a programmer, but may also be a great limiting factor. Therefore, a flexible core language needs a type system that is most robust, so that no DSL is limited by it. We would need to support the dependent types (explained in Section 2.2.2 together with other typing solutions, or have no static typing at all.

We choose the latter. We treat type checking as an auxiliary computation, provided as a library or in the code of a custom DSL. This way the language designer may define its own typing rules without any negative interference from the core language's type system. In the examples in Section 6.4.4 we show how simple typing can be defined in ManyDSL. In general, however, any of the type system models described in Section 2.2.2 can be adopted by custom DSLs and languages focused on type checking can be created.

Note that by using staging, this custom-defined type checking can be performed in any stage of the compilation process. The code may act as static or dynamic checking or any mixture of both. In particular, such additional computation can be staged for early execution and have no negative impact on the runtime performance of the final code.

Compilation

The whole ManyDSL approach will not be useful if it is not equipped with a compiler capable of producing a highly efficient code. Staging can help produce highly specialized functions and hardware reflection allows the code to adapt to a given hardware. That is not enough however: The host language, even when specialized, remains in continuation-passing style. It is a representation relying heavily on higher-order functions with no explicit distinction between different control flow structures. While it is a highlight of the host language, it may be potentially difficult to translate to efficient binary code.

Fortunately, there exist translators that can achieve just that. Rather than reinventing the wheel, we decided to use existing solutions, such as the Thorin compiler [84]. This allows us to focus primarily on the front-end of the host language as discussed in this section.

3.4 Separation of Concerns

Having our design decisions explained, let us ask briefly if they fulfill our ultimate goal, allowing a new kind of programmers – the DSL creators – to emerge, separately from compiler experts. We give a full answer to this question in the conclusions (Section 7), but let us address most pressing concerns that one may have at the moment.

Some may view CPS and Dynamic Staging programming too complex for an average programmer. It is true, that understanding these is necessary for developing custom DSLs at the lowest level, directly in ManyDSL's host language. This does not mean that all DSLs have to be written this way:

- Languages can build on top of each other. A higher-level, more restrictive but simpler language can be used to define new DSLs, leaving the host language merely as a fallback solution.
- Since Dynamic Staging is a first-class citizen in the ManyDSL core language, the way of doing staging can also be abstracted or restricted by custom DSLs. A programmer using the custom DSL instead of DeepCPS would be exposed to a less verbose staging mechanism.
- Common language constructs, e.g. arithmetic expressions, can be given as functions in a library to be included into any custom-built language. This way, the major part of DSL development can be spent on these parts of language which are truly unique.
- Similarly, common type systems, expressed as an auxiliary computation, can be given as libraries for the DSL creator to choose from. The creator may focus just on extending these type systems, or using them as they are, without changes.

With sufficient support in DSL-building libraries and languages we believe that the verbosity of the host language is not an issue.

3.5 Structure

The ManyDSL structure and its workflow are sketched in Figure 3.1. The first step is parsing the source code into ManyDSL Target Representation (TR). This process can be interrupted however. The parser halts, and the interpreter executes the part that is already in TR format. When the execution process hits the interruption spot, parsing resumes. This interleaved parsing and execution puts ManyDSL at unique position where code and the parser can communicate with each other.

The interpreter and the parser can communicate and affect each other's behavior. TR code can define new grammar constructs and whole languages which can entirely replace the original parser language. The parser also has access to all the values and variables of the partially executed program, allowing the process to be further parametrized by the code. This interaction as well as the language creation process is further described in Section 5.7 and Section 4.2.

The source code – at least the beginning of it – must be written in DeepCPS, which is a low-level functional continuation-passing style language. DeepCPS is almost a direct textual representation of TR. Once new languages are loaded into the parser, the rest of the source code can be written in a completely different language. New languages do not have to hold any resemblance to DeepCPS. In the future one will be able to start with another parser preloaded.

TR code can be interpreted or compiled. As a result of interpretation, a new optimized TR code can be produced. This is particularly true when dynamic staging is used [32], which is supported by TR. We explain in detail how DeepCPS is defined and how dynamic staging works in Section 4.1.

Alternatively, a fragment of TR can be compiled. We translate TR to Thorin [84] which is another language that uses continuation-passing style, making the

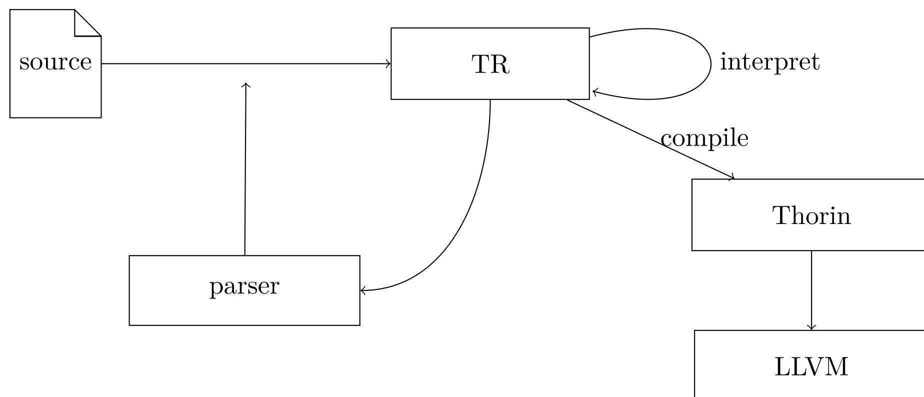


Figure 3.1: *The structure of ManyDSL. Source code is parsed into ManyDSL Target Representation (TR). TR can be interpreted and partially evaluated, producing a more efficient TR code. It can also modify the parser of ManyDSL, so that different DSLs can be read. Finally, TR code can be translated to Thorin[84] and then compiled to LLVM[80].*

translation process easy. Thorin transforms the code to eliminate the overhead caused by higher-order functions, and then compiles into LLVM bytecode [80]. From there, one can use one of many LLVM backends to produce machine code for different architectures supported by LLVM.

In the following chapters we discuss each part of ManyDSL. First, in Section 4.1 we provide a formal and practical view DeepCPS and dynamic staging. In Section 4.2 we show how new grammar can be defined in ManyDSL. Finally, in Section 4.3 we combine dynamic staging with grammar actions, allowing user-defined DSLs to generate code.

The actual implementation details of ManyDSL, including its Target Representation and caveats of dynamic staging are given in Section 5.

Finally, in Section 6 we provide more practical examples of how each of the ManyDSL components can be used. We start with plain DeepCPS and staging examples, but follow it with more advanced challenges typically encountered when creating languages. In the end, we include an example how languages can actually be switched, marking one of the main goals of our work.

Chapter 4

ManyDSL Components

In this chapter we describe in detail the main contribution of our work, which constitute the core of ManyDSL. First we describe the core language – DeepCPS, a functional language with dynamic staging. This allows the programmer to define partial evaluation strategies and produce specialized versions of functions.

We then describe the parsing and execution process of ManyDSL. We show how grammars can be redefined on the fly to produce different DSLs.

Finally, we show how DeepCPS can be used to specify actions which are executed when the language constructs are recognized. We define *builders*, functions which incrementally build code. We show how dynamic staging removes any overhead coming from the builders.

4.1 Dynamic Staging

ManyDSL relies heavily on functions defining language semantics and building code (Section 4.3). Every syntactic construct, control flow, domain-specific optimization strategy, or simple variable lookup routine, is a function that potentially can be defined or redefined by different DSLs. This makes ManyDSL flexible, but also potentially slow.

For that reason, partial evaluation and staging plays a central role, not only in custom DSL optimization, but in almost every language definition within ManyDSL. Let us then focus on this core host language – DeepCPS – that everything in ManyDSL is translated into, and which enables partial evaluation through the *dynamic staging*.

4.1.1 Continuation Passing Style

Dynamic Staging is defined on top of the Continuation Passing Style (CPS) functional language [6]. We have briefly explained what is CPS and why we have

```
int power(float base, int exp) {
  ...
  float part = power(base,exp/2)
  return part*part;
  ...
}
...
```

(a) *C code*

```
letrec power = λbase exp .
  ...
  let part = power base
    (div exp 2) in
  mul part part
  ...
in ...
```

(b) *Standard functional code*

```
letrec power = λbase exp cont .
  ...
  div exp 2 (λexph .
    power base exph (λpart .
      mul part part cont
    )
  )
  ...
in ...
```

(c) *Continuation-passing style*

```
let power(base, exp, cont) {
  ...
  div . exp 2 (exph)
  power . base exph (part)
  mul . part part cont
  ...
} ...
```

(d) *DeepCPS code*

Listing 4.1: *Comparison of the same code ("else" branch of power function, given in Listing 2.1) implementation between C, standard lambda calculus, continuation-passing style and our language DeepCPS.*

chosen this style (Section 3.3). Let us reiterate them once again, in more detail and in the context of dynamic staging.

What is CPS?

The CPS-based lambda calculus is similar to the ordinary lambda calculus [24] with a single, yet important, restriction: In CPS functions never return. As a result the following typical lambda constructs are not possible:

- Functions cannot return any value. Formally, an application term is not a value.
- Currying is not possible. Partial application can be achieved only in a more verbose and explicit way.
- The body of a lambda consists of a single application term.

What is then possible with such restriction? How can any meaningful computation be represented in CPS?

Note, that it is not necessary to return from a function when the function body or at least one of its arguments represents the complete remainder of the program. We name a *continuation* any function that contains the remaining computation. In CPS programming all functions are either continuations or take a continuation as an argument. Note, that there is no formal distinction between continuations and regular functions, it is merely an informal term indicating what a given function represents.

Typically, CPS functions take one more argument compared to normal functions. After the primary operation of the function is performed, it invokes its continua-

tion. In the context of CPS program, we informally say “return a value” when a value is passed to the continuation.

Consider, for example, the division operator in the example Listing 4.1. In normal lambda calculus, `div` is a binary function and its invocation is used as a subexpression. In CPS it actually takes 3 arguments: `exp`, 2 and `λexph`. When the division is computed, the continuation is invoked and the result is bound to its `exph` parameter.

Why CPS?

From the theoretical standpoint, the order of execution bares no impact on the program in purely functional programming, as long as the execution terminates. However, when discussing efficiency the execution order is crucial. Functional language semantics provides freedom for the interpreters and compilers to use heuristics to find an optimal execution plan. For example, the order of evaluation of function arguments is often not defined, or it is defined only in a few selected cases (e.g. Boolean operators).

In DeepCPS we let the programmer to specify the most suitable order. For that to accomplish, however, DeepCPS must have general unambiguous rules describing the execution order; not based up on special cases and ad-hoc solutions. Continuation Passing Style satisfies this requirement. The canonical order of execution in a CPS program is well defined. There is always exactly a single application at the top level, containing no subexpressions. In a single execution step (formally: single β-reduction of the top application), we obtain a new program with another, single application at the top level. At no times is there an ambiguity on which subexpression to execute first.

Secondly, in standard CPS every lambda function contains a single application building up its complete body. Vice-versa, every instruction (application) is contained within its own lambda function. The existence of this one-to-one relation between applications and lambdas is important for the definition of dynamic staging. In canonical CPS, when a lambda function is invoked, its body is executed in the next step. Dynamic staging allows for manipulating this relation.

Finally, in a programming language represented in CPS, any control flow can be expressed as a function. This is in contrast to most other programming languages where control flow structures are given as special constructs, having their own syntax and semantics. In CPS no new syntax is needed.

Formal CPS

The syntax and semantics of an untyped lambda calculus is defined as in Figure 4.1 [111, Chap. 5]. For practical reasons, we include an additional category of values: Constants, which may be numbers, but also Boolean values, intrinsic functions, etc. For the sake of simplicity of the rules, we restrict lambdas to include a single parameter. Multi-parameter functions can be represented through higher-order functions with currying.

$$
\begin{aligned}
t ::= & & \text{(term)} \\
& x & \text{(parameter)} \\
& t\ t & \text{(application)} \\
& v & \text{(value)} \\
v ::= & & \text{(value)} \\
& c \in \{0, 1, 2, \ldots, \} & \text{(constant)} \\
& \lambda x.t & \text{(function)}
\end{aligned}
$$

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2} \quad \text{(Con1)}$$

$$\frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'} \quad \text{(Con2)}$$

$$(\lambda x.t_3) v_3 \to [x \mapsto v_3]\, t_3 \quad \text{(Comp1)}$$

$$c\ v \to t_{cv} \quad \text{(Comp2)}$$

Figure 4.1: *Syntax and semantics of untyped lambda calculus.*

$$
\begin{aligned}
v ::= & & \text{(value)} \\
& x & \text{(parameter)} \\
& c \in \{0, 1, 2, \ldots, \} & \text{(constant)} \\
& \lambda \overline{x}.b & \text{(function)} \\
b ::= & & \text{(body)} \\
& v\ \overline{v} & \text{(application)} \\
& \texttt{fix}\ x = \lambda \overline{x}.b\ \texttt{in}\ b & \text{(fix)}
\end{aligned}
$$

$$(\lambda \overline{x}.b)\ \overline{v} \to \begin{bmatrix} x_1 \mapsto v_1 \\ \vdots \\ x_n \mapsto v_n \end{bmatrix} b \quad \text{(CPS1)}$$

$$\begin{aligned} &\texttt{fix}\ x_f = \lambda \overline{x}.b\ \texttt{in}\ b_f \to \\ &\to [x_f \mapsto \lambda \overline{x}.\texttt{fix}\ x_f = \lambda \overline{x}.b\ \texttt{in}\ b]\, b_f \end{aligned} \quad \text{(CPS2)}$$

$$c\ \overline{v} \to b_{c\overline{v}} \quad \text{(CPS3)}$$

Figure 4.2: *Syntax and semantics of Continuation Passing Style lambda calculus.*

The congruence rules Con1, Con2 are used to compute subexpressions of the top-level application term. Once all terms are reduced to values, the top level application can be computed through Comp1 or Comp2 rules.

The main computation rule Comp1 takes the body of the lambda t_3 and replaces all occurrences of the parameter x with the value v_3. We use the standard capture-avoiding substitution, with renaming of bound variables where necessary.

The Comp2 represent the case where an intrinsic function, represented as a constant, is invoked. It is up to that function to produce a new, correct t_{cv} term. Listing all possible intrinsic functions and their rules is beyond the scope of this formal definition.

Let us now compare the standard to CPS-based lambda calculus, given in Figure 4.2. Arguments must be variable names, constants, or lambdas, but cannot contain subexpressions. That is because syntactically, an application is no longer a value. An application may appear only as a body of a lambda. The body of the lambda is a single application, or — for convenience reasons — an explicit fixed-point combinator construct **`fix`**. Moreover, since currying is no longer possible, we allow lambdas to have multiple parameters, indicated by the $\overline{x}$ which denotes a list $x_1, x_2, ..., x_n$ for any natural n.

The semantics of CPS is arguably simpler: It does not have congruence rules. Those are not needed, because the top-level application – as any other – must

Basic syntax:

$\lambda\overline{x}.b$	`(x₁, x₂, ...){`b`}`	(function)
x	*typename*	(parameter)
$v\ \overline{v}$	`v . v₁ v₂ ...`	(application)
$\texttt{fix}\ x = v\ \texttt{in}\ b$	**fix** x v **in** b	(fix)

Syntactic sugar:

$(\lambda x.b)\ v$	**let** $x\ v\ b$	(let construct)
$v\ \overline{v}\ (\lambda\overline{x}.b)$	`v . v₁ v₂ ...` `(x₁, x₂, ...)`b	(function as last argument)
$(\lambda x.b)\, p \xrightarrow{\beta} (\lambda x.b)\, v$	p `. (x)` b	(non-CPS expression p)

Figure 4.3: *Syntax of lambda calculus and DeepCPS.*

be built from concrete values and not subexpressions. The main computation rule (CPS1) substitutes the tuple of lambda parameters, with the tuple of actual arguments given in the application. The arity of both tuples must match. Failing to do so is an error, but this constraint cannot be specified on the syntactic level. A type system would be the best place to enforce it.

For convenience and performance reasons, we include explicit semantics for the fix-point combinator. In Appendix C we show how a CPS version of a Y-combinator can be defined in plain CPS without **fix**. The fix construct defines a $\lambda\overline{x}.b$ under the name x_f. By using x_f in the body b one can specify recursion. When fix is evaluated through the CPS2 rule, a single recursion step is unrolled: x_f is replaced with another $\lambda\overline{x}$ containing the same fix construct with a replicated body b.

CPS3, similar to Comp2, is given for completeness, representing a call to an intrinsic function. The intrinsic function too must be in CPS form. That means, it is expected to take a continuation as one of the arguments and invoke it with the computed value. As a result, the call $c\overline{v}$ reduces to the body of such continuation $b_{c\overline{v}}$ with the result substituting the parameters of the continuation. The syntax and semantics of the language impose no further constraint on what c actually does. It may accept several continuations, or even somehow create a new one based on the arguments, in which case $b_{c\overline{v}}$ may not even appear initially in the source code.

4.1.2 Stageless DeepCPS

DeepCPS Syntax

As we have shown in Listing 4.1c, lambda calculus syntax is impractical to express CPS programs. As the program grows, the nesting of lambdas increases. Since most of our discussion in this work uses CPS, and ManyDSL requires CPS source to be written by a programmer, we decided that a more suitable syntax is needed.

The changes are summarized in the Figure 4.3. First, we redefine the use of parenthesis. Contrary to standard lambda calculus, expressions no longer need to

be disambiguated through parenthesis. Instead, we now *require* to put parenthesis around the lambda function parameter list. Not only does this allow to forgo the actual λ and dot symbols, it also emphasizes the arity of the function.

The body of the function can be a complex expression, thus it is put in a separate pair of parenthesis. To avoid confusion with the lambda header, we use curly braces instead, known from C-style languages. These can be skipped however if the whole lambda is used as the last argument of an application. Thus: `f . g (h)i j` is equivalent to `f . g (h){i j}`.

This particular use case is very frequent in CPS programming. Many functions take a lambda as the last parameter, acting as a continuation and representing the rest of the program. In our syntax, by avoiding the curly braces for the last argument, we significantly reduce the nesting compared to the standard lambda notations.

We have decided to use a dot in the application syntax to separate callee from its arguments, even though formally it is not needed. In practice we found that it adds clarity to the program structure. It also helps accurately localize syntactic errors in program sources. Note that the dot was not used in the original Dynamic Staging paper [32].

Apart from functions, we use the usual integer and floating-point values, Boolean values (`true`, and `false`), and string constants. We support aggregate values in a form of tuples `[...]`. Each value has a type. Basic types include **`bool`**, **`int`**, and **`float`**, aggregate type `tuple[...]` and function type **`fn`**`[...]`. We also permit a special type any when basic typing is impossible to specify. When the type can be easily deduced from the context or is irrelevant for the given example, we choose to omit it.

DeepCPS supports the following tuple operations:

- Creation, by simply listing all its elements: e.g. `[1, 2, 3]`. This is a value an may be used as an argument.
- Concatenation of two existing tuples, e.g.
  ```
  $cat . [1, 2] [3] (result) ...
  ```
  The produced `result` is `[1, 2, 3]`.
- Extracting a single element of a tuple (projection), e.g.
  ```
  $proj . 0 [1, 2, 3] (result) ...
  ```
  The `result` is 1 – the first element of the tuple, which has an index 0.
- Splitting a tuple into components, each passing as a separate argument to a function. This is done through a special syntax `!arg`. For example:
  ```
  let args [3, 72]
  power . !args (result)
  ```
  invokes the function `power` from Listing 4.1d which expects three arguments: base, exponent and the continuation. The tuple `[3, 72]` is split into two arguments, passing `3` as base, and `72` as the exponent.
- Aggregating excessive arguments into a tuple. This is done through a special syntax `!pars` appearing in the lambda header. For example:

```
fix power (float base, int exp, fn[float] cont) {
  exp==0 . (bool b)
  if . b () {
    cont . 1
  } ()
  exp mod 2 == 1 (bool odd)
  if . odd () {
    exp-1 . (em)
    power . base em (part)
    part*base . (result)
    cont . result
  } () {
    exp/2 . (eh)
    power . base eh (part)
    part*part . (result)
    cont . result
  }
} in ...
```

Listing 4.2: *An integer power function in DeepCPS.*

```
let create(!args, return) { return . !args }
create . 1 2 3 (result) ...
```

The function `create` accepts 1 or more arguments. The last argument is the returning continuation. All other arguments are aggregated into a single tuple `args`. In this example, `result` becomes `[1, 2, 3]`. There can be at most one aggregating tuple in each lambda.

Finally, for the sake of readability of the examples in this work, we permit non-CPS sections to appear. These are used where CPS does not convey any significant information and transformation into CPS is straightforward. Most frequently it is limited to arithmetic expressions. These fragments are represented in blue, and must reduce to a single value. The obtained value is then passed into an unary continuation lambda that follows it.

DeepCPS syntax puts no requirements on the use of whitespace. It helps however, when the code is written in a consistent style. We found that the code is most readable when every application starts a new line. This may be confusing at first, since the lambda header becomes separated from its body between two lines. In practice however the body has little to do with its header: The variables used in the body often refer to earlier headers, which are merely captured by the lambda the body belongs to. On the other hand, the parameter list is much more important in the context of the application where the lambda is actually being *used*.

Example Usage: The Power Function

Let us have a look on a DeepCPS implementation of the power function in Listing 4.2. As with many other CPS functions, `power` now takes three arguments: the base, the exponent, and the continuation `cont` which should be invoked with the final result.

First, we check whether the exponent is zero. To create a branch, we invoke an intrinsic function `if`, which is of type `fn[bool,fn[],fn[]]`. This functions

```
fix for (int from, int to, fn[int,fn[]] body, fn[] end) {
  from<to . (bool b)
  if . b () {
    body . from ()
    from+1 . (next)
    for . next to body end
  } end
} in ...
```

Listing 4.3: *Definition of the `for` looping function, with the help of built-in `if`.*

takes a Boolean value acting as a condition and `two` continuations. Depending on the value of the condition, either the first or second continuation is invoked. In our case, the first continuation is the short function `() {cont 1}`, the other is the rest of the power function.

When using `if` one does not specify where the branches converge. In fact the continuations may be completely independent from that point forward. Typically, however, at some point all branches invoke the same continuation, implicitly ending the divergent section. In our case, this is the invocation of `cont` that appears at the end of all branches.

If the exponent is nonzero, we then check whether it is odd or even. We use another `if` function to handle each case separately.

When the exponent is odd, we take the first continuation. We subtract 1 from the exponent and perform a recursive call to `power`. Subsequent `power` returns, by passing its partial result $\texttt{base}^{\texttt{exp}-1}$ through its continuation `(part)...`. Then, by multiplying it with `base`, we obtain the final result, which we return back by calling the continuation: `cont . result`.

When the exponent is even, we take the second continuation of the `if` call, but otherwise the executed code is very similar.

Control Flow Functions

In the power function example in Listing 4.2 we use the intrinsic function `if`, which creates a branch. All control flow structures in CPS programming can be represented as *functions*. Even atypical flow operations, such as switch, static gotos, or exceptions can be handled with functions. There is no need for additional built-in constructs to generate that control flow.

Moreover, `if` is the only built-in function in DeepCPS. Other control flow functions can be defined as an ordinary higher-order function with the help of `if`. For example Listing 4.3 is the simplest kind of a `for` loop.

The `for` looping function takes the iteration range `from .. to`, and for each integer in between invokes the `body` function. The body is expected to invoke its continuation at some point to return the control back to the `for` looping function. When the loop is complete, the for function return by invoking its final continuation `end`.

Note, however, that the author of the `body` may choose to break the loop, simply by *not* invoking the passed continuation.

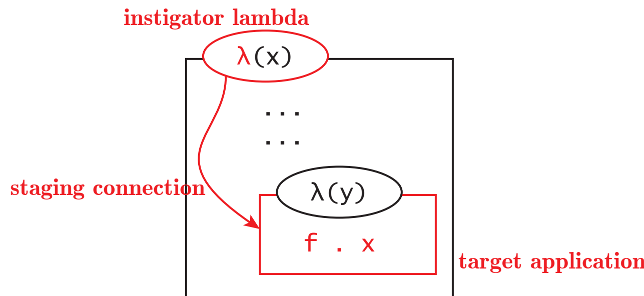


Figure 4.4: *The staging mechanism of Dynamic Staging: a staging connection between lambda header (instigator) and application (target) located somewhere within the body of the instigator. When the instigator is invoked, the target is scheduled for the execution in the next step of the program evaluation.*

In Section 6.1.1 we show how to define other, more generic flow functions. We also show how to have one or more values change within each iteration of the loop.

4.1.3 Definition of Dynamic Staging

In each execution step of a traditional CPS program, the beta reduction is applied on the top-most application. This causes the program to be evaluated in a natural order from top to bottom. We are interested in a mechanism that would allow to change this order, permitting reductions somewhere in the middle of the code.

In purely functional languages this can be achieved by redefining the semantics in a way that the execution order is arbitrary. Unfortunately, with such arbitrary execution order, no guarantees can be made on the execution complexity of almost any algorithm. It could happen, for example, that arbitrarily many reduction steps are performed in a branch that ultimately is not taken.

Other solutions involve enriching the language with staging. In in Section 2.3 we give examples how staging can be defined and what are the drawbacks of these definitions. Now, let us introduce *dynamic staging* – our novel approach to staging.

Intuitive Definition

Conceptually, as depicted in Figure 4.4, dynamic staging is specified by connecting lambda headers and parameter names on one end, and applications on the other. Connected headers and parameters are *instigators* of this *staging connection*. Connected application (or a fix construct) are *targets*. When a lambda is invoked, or a constant bound to a parameter, the instigator become *active*. An active instigator triggers all its targets, so that they are executed in the next step.

A target may be located deep within the body of another, yet uninvoked lambda. When that happens, the execution process jumps over a section of code, leaving it intact, and performs a reduction “under the lambda” of the targeted application.

$v ::=$	(value)
$\quad x$	(parameter)
$\quad y$	(staging parameter)
$\quad c \in \{0, 1, 2, \ldots\}$	(constant)
$\quad \lambda\texttt{[}y\texttt{]}\overline{x}.b$	(function)
$e ::=$	(staging expression)
$\quad \top$	(true)
$\quad \bot$	(false)
$\quad v$	(value)
$\quad e \texttt{ and } e$	(and expression)
$\quad e \texttt{ or } e$	(or expression)
$\quad \texttt{not } e$	(not expression)
$b ::=$	(body)
$\quad \texttt{[}e\texttt{]}\, v\ \overline{v}$	(application)
$\quad \texttt{[}e\texttt{]}\texttt{fix [}y\texttt{]}x = \lambda\texttt{[}y\texttt{]}\overline{x}.b \texttt{ in } b$	(fix)

Figure 4.5: *Syntax of staged CPS-based lambda calculus.*

Staging connections may appear anywhere in the code. The only requirement is that the targets are within the body of the instigator lambda, possibly deeply nested.

Syntax

Syntactically, dynamic staging is just a small change to CPS lambda calculus. It specifies the connection between instigators and their targets. We achieve that by introducing a new syntactic category *staging expressions* (see Figure 4.5).

A lambda value is augmented to include exactly one special staging parameter y. Throughout the body of the lambda, y can be used as an ordinary value. Parameters and staging parameters are separated syntactically only for clarity.

Changes to a body (application or fix construct) are more involved. Each body is now prefixed with a *staging expression*, which controls when the given target is invoked. A staging expression is an arbitrary Boolean expression over normal and staging values. When the staging expression evaluates to true, the body is meant to be executed. There are two staging constants: $\top$ equivalent to “true” and $\bot$ equivalent to “false”. Note that staging constants $\top$ and $\bot$ are separate, different kind of symbols than the regular Boolean values which belong to c.

Semantics

The semantics of dynamic staging is a bit more involved. Formal definition of executing “under the lambda” requires a bit more rules to dig into the nested body that was chosen to be executed. Moreover, the case when multiple bodies

become triggered at the same time has to be handled. To handle this we define two supporting relations: *Active* and *Waiting*.

Active. A staging expression is considered *active* when it evaluates to $\top$. For the purpose of staging expression evaluation, all bound values (constants and lambda functions) are considered $\top$ while unbound values are $\bot$.

$$\frac{}{A(\top)} \qquad \frac{}{A(c)} \qquad \frac{}{A(\lambda\,[y]\,\overline{x}.b)}$$

$$\frac{A(e_1) \quad A(e_2)}{A(e_1 \texttt{ and } e_2)} \qquad \frac{A(e_1) \vee A(e_2)}{A(e_1 \texttt{ or } e_2)} \qquad \frac{\neg A(e)}{A(\texttt{not } e)}$$

Expression that evaluates to $\bot$ is *inactive*, thus not a member of A relation. There are also no rules for x or y — these terms are considered inactive as long as they are not resolved. When x or y is substituted by a constant, lambda, or $\top$, the staging subexpression using it becomes active.

$$\frac{A(t)}{A([x \mapsto t]x)}$$

This is the main mechanism that triggers the execution of new applications.

In short, we say that an application or fix-construct is active, when its staging expression $[e]$ is active. However, formally the A relation is defined only on staging expressions.

Waiting. When multiple bodies are active at the same time we choose to execute the one, that does not contain any other active bodies. To check this property, we introduce a second relation *waiting*. A value or body is called *waiting* (W_v, W_b) when it does not contain any arbitrarily nested active staging expression.

$$\frac{}{W_v(c)} \qquad \frac{}{W_v(x)} \qquad \frac{}{W_v(y)} \qquad \frac{W_b(b)}{W_v(\lambda\,[y]\,\overline{x}.b)}$$

$$\frac{\neg A(e) \quad W_v(v) \quad \overline{W_v(v)}}{W_b(\,[e]\,v\;\overline{v})} \qquad \frac{\neg A(e) \quad W_b(b) \quad W_b(b_f)}{W_b(\,[e]\texttt{fix}\;[y_f]x_f = \lambda\,[y]\,\overline{x}.b \texttt{ in } b_f)}$$

Looking at the CPS program from the bottom of its expression tree, all leafs and their ancestors are considered waiting, until an active application or fix construct is found (that is: its staging expression is in A). From that point onward, up to the root of the expression tree, all bodies and lambdas are *non-waiting*.

Execution. With the help of active and waiting relations, we can now specify that only the terms that are active while all their subterms are waiting are allowed to execute. We call it the *active-waiting* requirement. A transformation that is an actual execution step is indicated by $\rightarrow_p$.

$$\frac{A(e) \quad W_v(v) \quad \overline{W_v(v)} \quad [e]\,v\;\overline{v} \to b'}{[e]\,v\;\overline{v} \to_p b'}$$

$$\frac{A(e) \quad W_b(b) \quad W_b(b_f) \quad [e]\texttt{fix}\;x_f = \lambda[y]\overline{x}.b\;\texttt{in}\;b_f \to b'}{[e]\texttt{fix}\;[y_f]x_f = \lambda[y]\overline{x}.b\;\texttt{in}\;b_f \to_p b'}$$

Rebuilding. When a nested body is executed, all terms on the path from the root to it need to be rebuilt to include the resulting body b' of the execution.

$$\frac{}{c \to_p c} \qquad \frac{}{x \to_p x} \qquad \frac{b \to_p b'}{\lambda[y]\overline{x}.b \to_p \lambda[y]\overline{x}.b'}$$

$$\frac{\neg(A(e) \wedge W_v(v) \wedge \overline{W_v(v)}) \quad v \to_p v' \quad \overline{v \to_p v'}}{[e]\,v\;\overline{v} \to_p [e]\,v'\;\overline{v'}}$$

$$\frac{\neg(A(e) \wedge W_b(b) \wedge W_b(b_f)) \quad b \to_p b' \quad b_f \to_p b'_f}{[e]\texttt{fix}\;[y_f]x_f = \lambda[y]\overline{x}.b\;\texttt{in}\;b_f \to_p [e]\texttt{fix}\;[y_f]x_f = \lambda[y]\overline{x}.b'\;\texttt{in}\;b'_f}$$

Evaluation. Finally, we define how the actual evaluation is performed. These rules correspond to those provided in standard CPS Figure 4.3. We enhance the original evaluation rules by including staging expressions. Same as before, when invoking a function through the application expression, the number of actual arguments must match the normal parameter count of the lambda function. There is no argument given for the staging parameter $[y]$. Instead, it is always replaced by $\top$, indicated that the lambda has been called. Consequently, in the context of staging expressions y becomes active. Since no explicit argument is given to replace y, we often refer to y as an *implicit staging parameter*.

$$[e]\;(\lambda[y]\overline{x}.b)\;\overline{v} \to \begin{bmatrix} x_1 \mapsto v_1 \\ \vdots \\ x_n \mapsto v_n \\ y \mapsto \top \end{bmatrix} b$$

$$[e]\;\texttt{fix}\;[y_f]x_f = \lambda[y]\overline{x}.b\;\texttt{in}\;b_f \to$$
$$\to \begin{bmatrix} x_f \mapsto \lambda[y']\overline{x}.[y']\;\texttt{fix}\;[y]x_f = \lambda[y]\overline{x}.b\;\texttt{in}\;b \\ y_f \mapsto \top \end{bmatrix} b_f$$

A bit of attention has to be paid to the fix-point combinator. Conceptually, every occurence of x_f within body b_f should be replaced by the function definition $\lambda[y]\overline{x}.b$. However, it is an error to simply write $x_f \mapsto \lambda[y]\overline{x}.b$, because the body of the definion most likely contains recursive uses of x_f. Such notation would lead to infinite unrolling of x_f.

Basic syntax:

$\lambda\,[y]\,\overline{x}.b$	`(x1, x2, ...)[y]{b}`	(function)
x	*type name*	(parameter)
$\top\ \bot$	`always never`	(staging constants)
and or not	`& | !`	(staging operators)
$[e]\,v\,\overline{v}$	`@e: v . v1 v2 ...`	(application)
$[e]\,\mathtt{fix}\ [y]x = v\ \mathtt{in}\ b$	`@e:` **fix** `[y] x` v **in** b	(fix)

Syntactic sugar:

$\lambda\,[y]\,\overline{x}.\,[y]\,v\overline{v}$	`(x1, x2, ...) { v . v1 v2 ... }`	(natural staging)
$[e]\ (\lambda\,[y]\,x.b)\ v$	`@e:` **let** `[y]` x v b	(let construct)
————————	`@e:` **fix** `[y]` $\mathtt{x}_1\ v_1\ \mathtt{x}_2\ v_2\ \mathtt{x}_3\ v_3 \ldots$ **in** b	(polyvariadic fix)
$[e]\,v\ \overline{v}\,(\lambda[y]x.b)$	`@e: v . v1 v2 ... (x1, x2, ...)[y]` b	(last argument)
$(\lambda[y]x.b)\,p \xrightarrow{\beta} (\lambda[y]x.b)\,v$	p `. (x)[y]` b	(non-CPS expression p)

Figure 4.6: *Syntax of DeepCPS with dynamic staging.*

What we want instead, is to expand x_f only once. All recursive calls should remain as a **fix** until they are actually reached by the program. Since **fix** itself is not a value, it must be put into its own lambda $\lambda\,[y']\,\overline{x}.\,[y']$. The body of that lambda is the recursive definition of the **fix**, with the original body of x_f placed in the **in** clause.

Also note that we introduce a completely new name y' for the staging parameter of the lambda replacing x_f. The original name y appears only later, as a staging parameter of the **fix**. This way, instructions staged upon y in the **in** clause may assume that x_f is already expanded and may be called.

4.1.4 Staging in DeepCPS

We need to update the syntax of DeepCPS, previously given in Figure 4.3. In order to maintain code readability, all staging expressions and implicit staging parameter are written in orange. The new syntax of DeepCPS is given in Figure 4.6.

We have chosen a slightly different syntax for staging expressions to differentiate them from the implicit staging parameter.

Formally, all applications and fix construct are annotated by a staging expression. In DeepCPS we allow the annotation to be omitted when the body is staged upon the lambda it is directly contained in.

Similarly, every lambda should define its own implicit staging parameter $[y]$. The parameter declaration can be skipped, however, if it is used only in the body of that lambda or if it is not used at all.

We call *natural staging* when the only target for a lambda is its body, and that body is staged only upon its lambda. This is because the execution at that

```
fix power (float base, int exp, fn[float] cont) {
  exp==0 . (bool b)
  if . b () {
    cont . 1
  } ()
  exp mod 2 == 1 . (bool odd)
  if . odd () {
    exp-1 . (em)
    power . base em (part)
    part*base . (result)
    cont . result
  } () {
    exp/2 . (eh)
    power . base eh (part)
    part*part . (result)
    cont . result
  }
} in
let power72 (float base, fn[float] cont) {
  power . base 72 (result)
  cont . result
} ...
```

Listing 4.4: *An integer power function and a special case where the input* `base` *value is raised to the power of 72. Without staging, there is no benefit of calling* `power72 . base` *over* `power . base 72`*.*

location matches normal execution of unstaged program. In fact, any unstaged code can be considered a special case of staged code where only natural staging is being used.

4.1.5 Using Dynamic Staging

With just the definition of the dynamic staging it may be unclear how one can actually use it. Let us use the running example of the power function again, and use dynamic staging to generate a specialized version of it. In the first step, we define all the functions without staging: The generic power function and a specialized version which raises the base to the constant exponent 72. Using the specialized version `power72`, as given in Listing 4.4, gives no performance benefit. This is because it is merely a wrapper around the generic version. When the specialized version is invoked, it just calls the generic one with the constant exponent 72.

Static Staging

Let us now assume that `powergen` names a function with the following properties. Same as `power`, it takes three arguments: the base, exponent and a returning continuation. However, when `base` is an unknown (unbound) value, it generates a simplified piece of code. The generated code computes the desired value of the power when the base is finally specified.

In the next subsection we will provide the definition of `powergen`, but for the moment let us assume that the function is already given. With it, we can now define a new version of `power72` as in Listing 4.5. Let us examine step by step what happens:

```
fix [def] powergen (float base, int exp, fn[float] cont) { ... } in
@def: let power72 (float base, fn[float] cont)[call] {
  @def: powergen . base 72 (result)
  @call cont . result
} ...
```

Listing 4.5: *Example use of static staging. The powergen call within power72 is invoked as soon as powergen is defined (`def`). When power72 is called (`call`) the body of the function will already contain the specialized code generated by powergen.*

1. The powergen is defined through the **`fix`** construct. When the initial binding is resolved, `def` becomes active.

2. There are two targets for `def`: **`let`** power72 and the invocation of powergen. The latter is nested deeper than the former, thus it is invoked first.

3. powergen is invoked. At this point base is unbound, since we are executing under the lambda. The value of power cannot be computed, but partial evaluation — with the known exponent — is possible. As a result, a specialized code is spliced into the body of power72.

4. The function powergen finishes its execution by invoking its continuation `(result){@call: cont . result}`. There is no application that is staged upon this lambda. The lambda body is staged on something else.

5. The only remaining active application is the **`let`** statement of power72. The already specialized version of the function becomes bound to that name.

6. Later on, when power72 is invoked, the staging variable `call` becomes active. The powergen is no longer invoked. The execution control jumps over, and follows directly to the continuation call, where now-concrete `result` can be returned.

The last step is not entirely accurate. When power72 is invoked, the partially evaluated code generated by powergen should be evaluated, *before* calling the returning continuation. This is an error which we introduced for the sake of simplicity of the example, and we are about to fix it.

Staging on Parameters

Let us now define the powergen function. The computational logic of the function does not differ from the original power function, but care has to be taken when instructions are executed. We expect that the exponent and the continuation are always known, but base may remain unknown at the time of the call. Consequently, all mathematical operations that directly or indirectly depend on base must be deferred. Observe that only the multiplication falls into that category. All conditions and the recursion itself can be resolved without base being known. By specifying this knowledge in the code we obtain powergen as in Listing 4.6a.

We now have a function that behaves very similarly to a normal power function, up to the point when multiplication is performed. At that point, the `[jmp]`

```
fix powergen (base, exp, cont) {
  exp==0 . (b)
  if . b () {
    cont . 1
  } ()
  exp mod 2 == 1 . (odd)
  if . odd () {
    exp-1 . (em)
    powergen . base em (part)[jmp]
    @base & part:
    part*base . (result)
    @jmp:
    cont . result
  } () {
    exp/2 . (eh)
    powergen . base eh (part)[jmp]
    @part:
    part*part . (result)
    @jmp:
    cont . result
  }
} in
@powergen:
let power72 (base, cont) {
  @powergen:
  powergen . base 72 (result)
  @result:
  cont . result
} ...
```

(a)

```
let power72 (base, cont) {
  @base & always:
  1*base . (result)
  @result:
  result*result . (result)
  @result:
  result*result . (result)
  @result:
  result*result . (result)
  @base & result:
  result*base . (result)
  @result:
  result*result . (result)
  @result:
  result*result . (result)
  @result:
  result*result . (result)
  @result:
  cont . result
}
...
```

(b)

Listing 4.6: *Staged power function in DeepCPS and its use to specialize* power72 ***(a)****. When* powergen *is defined, it is immediately invoked from within the body of* power72*. Code that depends on the unknown value* base *is spliced back into the context of* power72*, producing the result* ***(b)***

staging directs the execution to jump over the multiplication instruction, and immediately invoke the returning continuation. As a result, the function does not return any concrete value, but a symbol `result`. The symbol cannot be used for further mathematical operations, but can be used in further symbolic substitutions. In particular, when the recursion is unrolled, the symbol is representing the partial result and substitutes the `part` variables that appear in both branches of the conditional.

Ultimately, when `powergen` function finishes its execution, we obtain a new, specialized code for `power72` as shown in Listing 4.6b.

Finally, we fix the problematic `power72`. We no longer use the implicit staging parameter `[call]`. Instead, `result` is passed to `cont` only when it is actually a known value. While the implicit staging parameter associated with `power72` is never used, it does not mean that nothing happens when the function is called. When a concrete value substitutes `base`, it triggers the execution of the first instruction produced by `powergen`. This begins the chain of instructions that ultimately produce the final result.

4.1.6 Staging as a Graph

So far we have looked at CPS programming and dynamic staging from the perspective of lambda calculus. A less formal, but more intuitive approach is to view the CPS program as a form of a directed graph. The actual implementation, which we discuss in Section 5.1, also resembles the graph interpretation.

Graph Primitives

The primitives used to represent a CPS program as a graph are summarized in Figure 4.7. Consider a graph with two kind of nodes: values and bodies. A value node can be a constant, a name, or a lambda header. If a value node represents a lambda header, it has a single outgoing edge connecting it to the body node.

The body node can represent an application of a fix construct. An application can have multiple outgoing edges, each for an argument in an application. For the sake of simplicity however, values which are not actual lambdas are written as a part of the label of the application node in which they are used.

Formally a fix construct and a let construct are very different. Pragmatically

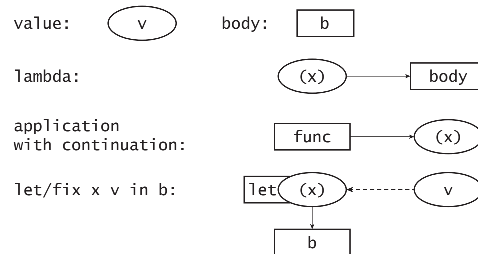


Figure 4.7: *Graph primitives for representing a CPS program.*

however, they do the same thing: bind a value to a new name which can be used in the following part of the program. For that reason, we represent those two construct in a similar way:

A **fix** or **let** is a body node with two lambda arguments. One argument is the *subject* value that gets a name. The other, is the *context* lambda where the name is going to be used. In the graph representation we connect the subject to the context through a dashed edge.

Example Program as a Graph

Let us now show how a stageless DeepCPS program can be represented with the primitives sketched above. In Figure 4.8a we take the power/power72 function example given in Listing 4.2 and represent its structure as a graph. By representing lambda headers and their bodies as separate nodes, the CPS program no longer looks as a monolithic, deeply nested structure of entities. Instead, the code becomes a path in the graph. Occasional branches appear only at **fix**/**let** constructs, or when a function takes multiple continuations (such as `if`).

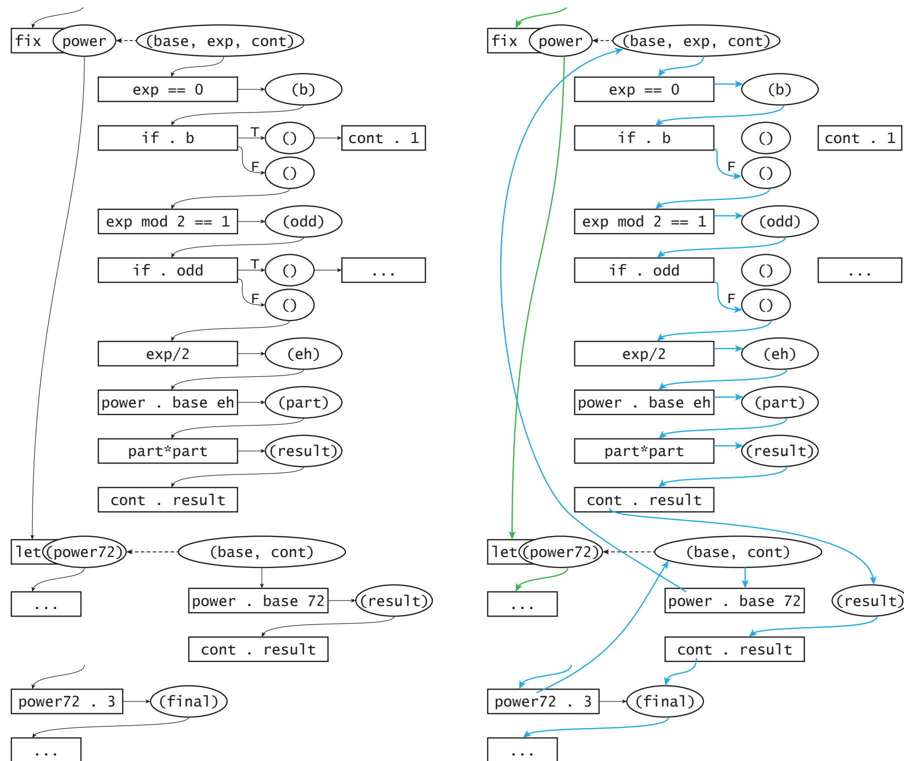


Figure 4.8: *The structure graph (a) and run paths (b) of the power and power72 functions without staging. Green edges are followed when defining the functions, and blue edges, when the function is invoked. Most of the work happens at call-time.*

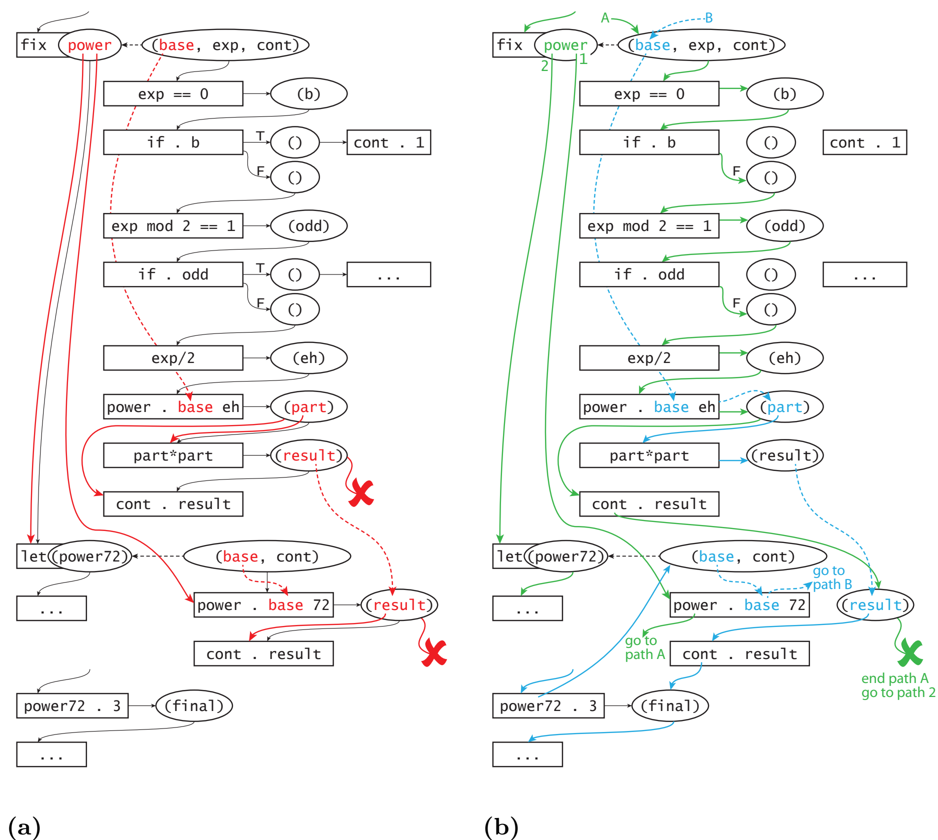
When one connects the nodes in the order in which they are executed, one obtains the *run path*, like the one shown in Figure 4.8b. In our example we highlight two run paths: The green path is taken when functions are being defined, and the blue path occurs when `power72` is being invoked.

The run paths are actually similar to the structure graph of the program. This is because the code is executed by following the edges in the structure graph. Jumps appear only when dereferencing a name that maps to a lambda.

Altering the Run Paths

We now introduce a new kind of edges: *staging connection* – the same it was done intuitively in Section 4.1.3. These kind of edges connect from lambda nodes (instigators) to arbitrarily nested body nodes (targets). Staging connections can also originate from particular parameters of the lambda.

In order to be strictly formal, one would put each lambda parameter as a separate,



(a) **(b)**

Figure 4.9: *The structure graph (a) and run paths (b) of the* `power` *and* `power72` *functions with staging. Staging connections are colored in red. Staging connections corresponding to natural staging are skipped. Green edges are followed when defining the functions, but because of the staging, a major part of the power function is executed at that time. At call-time there is significantly less work remaining, indicated by the blue path.*

new kind of node so that edges can originate from them. For the sake of brevity we keep parameters as sub-labels in a single lambda node.

When an instigator lambda is reached during execution, the staging connections are followed instead of the normal structural edges. This way we can create new run paths, while keeping the original program structure.

Furthermore, new staging connections may appear as a result of execution, reflecting the dynamic nature of our solution. Consider a staging parameter `s` that is used in a staging expression. Suppose an argument `t` substitutes `s` ($[s \mapsto t]$) as a result of an invocation. When that happens, the new parameter `t` *inherits* all the targets of the old one (`s`). In other words, all staging connections originating from parameter `s` are reconnected so that they originate from `t` instead.

As an example, let us again represent the `power`/`power72` functions, but this time the version using dynamic staging (Listing 4.6a). We obtain a structure graph as shown in Figure 4.9a. The solid red edges represent all staging connections that appear in the code. The dashed red edges mark connections that appear at some point during the execution of the program, due to staging inheritance that happens during substitution.

The run paths of the staged program are now different (Figure 4.9b). A major part of the computation within the `power` function occurs early, in the (green) definition path. The call-time path (blue) now contains only the multiplication operations which require the `base` value to be known.

In our opinion, this graph representation of code and staging compared to the formal definition provides better intuition on how our method works and how it can be used. As we have shown, staging changes the paths of execution acting as a railroad switch on tracks defined by the program structure.

4.1.7 Programing Patterns

We have formally defined and gave basic usage example of dynamic staging. Staging is realized by a simple construct connecting instigators with their targets. The construct can be used as a basis to define bigger, more specialized staging patterns. Let us now show those patterns and relate them to constructs available in other languages.

Early and Late Blocks

The most basic pattern is to select a block of code to be executed at a different time: Either earlier or later than the surrounding code. This is similar to staging through quotation, such as one in MetaML discussed in Section 2.3.5.

Assume that we have two stages represented by variables p and q. We are given a piece of code that should execute at p. However, there is a block within that code, starting from the instruction B and ending with E, that should be staged upon q.

```
let foo(p, q) {
  @p:
      ...              λB
      ... ()[p1]
  @q: B ...
      ...              λE
      ... ()[q1]
  @p1: E ...
      ...
}
```

(a)

```
let foo(q)[p] {
  @p:
      ...
      ... ()[p1]
  @q: B ...
      ...
      ... ()[q1]
  @p1: E ...
      ...
}
```

(b)

```
... ()[q]
let foo()[p] {
  @p: ...
      ... ()[p1]
  @q: B ...
      ...
      ... ()[q1]
  @p1: E ...
      ...
}
```

(c)

Listing 4.7: *Selecting a fragment of code to be executed at a different stage.* ***(a)*** *a generic staging on two stage variables* `p` *and* `q`*. Either of them may be executed first.* ***(b)*** *late execution stage. Surrounding code is executed when* `foo` *is invoked, but fragment staged upon* `q` *is deferred.* ***(c)*** *early execution stage. The fragment staged upon* `q` *is executed early, before the lambda is even bound to* `foo`*.*

```
() {
  ...             p
  ... ()
  B ...
  ...             q
  ... ()
  E ...           p
  ...
}
```

We implement such scenario in DeepCPS in Listing 4.7. Naturally, we start by staging the beginning of the code upon p. However, upon reaching an instruction B the execution should "jump over" and resume at a further instruction E. Therefore, when stage p is triggered, the fragment of the code between instructions B and E remains intact.

Consider a lambda λ_B for which B is its whole body. In CPS programming such λ_B always exists. We take the implicit staging parameter of λ_B, naming it p_1 and stage the instruction E upon it. Then, we stage B on the new stage q.

When stage p is triggered, all code up to λ_B is executed. Then, p_1 is triggered, which directs the execution to jump to the instruction E, leaving the contents between B and E intact. On the other hand, when q is triggered, the execution jumps into B and executes everything until λ_E. Then, the q stage ends because the body E is not targeted by the implicit staging parameter of λ_E.

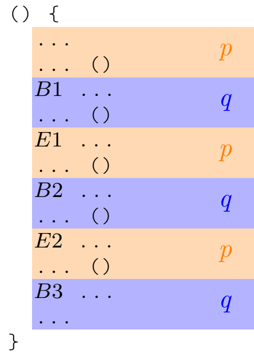
This way we obtain a block that is executed at a different time than the surrounding code. The pattern however does not specify the actual order of execution. Either `p` or `q` can be triggered first, causing the block to be executed either late or early.

Staging Chain

The pattern of having a block executed at a different stage can be repeated arbitrarily many times. Effectively, a program can consists of interleaved instructions to be executed in 2 or more stages: When a given stage becomes active

we expect instructions associated with that stage to be executed in sequence, jumping over parts which are scheduled with another stage.

Consider, for example, a code consisting of interleaved blocks to be executed in stage p and q Each q block starts with an instruction B_n and ends with instruction E_n, where $n \in \mathbb{N}$. The p blocks are in between, starting with E_{n-1} and ending with B_n.



A naive approach would be to stage each B_n upon q and each E_n upon p. However, this will cause blocks to execute in the reverse order, bottom-up. When q becomes active, *all* instructions B_n become scheduled for execution at the same time, and the deepest one, which starts the last block, is executed first.

Instead, we use repeatedly the early/late block pattern. As in Listing 4.8 we define a series of staging parameters: p_n and q_n, with p_0 and q_0 being equivalent to p and q respectfully. The n-th p block is staged upon p_n and the last lambda of the block defines the next staging parameter p_{n+1} to be used in the next p block. The same pattern is used for q blocks.

This effectively connects, through staging, blocks of the same type in a chain. Staging formed by the sequence s_i used in such pattern we call a staging chain s. We refer to staging variables forming the pattern as *chain staging variables*. Instructions in a staging chain s are executed in order, when s_0 becomes active. The staging chain pattern is the primary implementation of traditional execution

```
let foo(p, q) {
  @p: ...
  ... ()[p1]
  @q: B1 ...
  ... ()[q1]
  @p1: E1 ...
  ... ()[p2]
  @q1: B2 ...
  ... ()[q2]
  @p2: E2 ...
  ... ()[p3]
  @q2: B3 ...
  ...
}
```

Listing 4.8: *Interleaved stage chains* p *and* q.

phases, such as “compile-time” and “run-time” phases.

Fragment Chain Pattern

In the previous example of the staging chain, the pattern is used within a single sequence of instructions. However, since staging variables can be passed as arguments, the pattern can be used across function calls. Typically, a chain stage variable is passed into the function, and its further instance is returned through the continuation. A call to a stageless function of the form

```
F . args (ret)...
```

becomes

```
F . s_i args (s_j, ret)...
```

Within `F` the staging argument `s_i` can be used to chain arbitrary many blocks. The chaining stage variable from the last block is returned through the continuation as `s_j`. Naturally, within `F` the `s_i` or following chain stage variables can be also used for nested function calls, or in other ways beyond the limits of the pattern.

Using chains in this way is particularly useful in the context of building code. Chaining through functions gives DeepCPS an ability to connect two initially unrelated blocks of code, so that in the next staging chain they are executed one after another.

Consider a set of functions `f1`, `f2`, ..., `fn`, each taking and returning (through a continuation) a chain staging variable. Each function contains a code fragment staged with respect to its **`stage`** parameter, creating the *late block* pattern as described before. Therefore, we call such functions *container functions*.

We then define a master function `fM` that invokes all the container functions in any order of our choosing. We use the staging chain pattern to connect the late blocks of the container functions. When `fM` is invoked, all the containers are executed but their late blocks remain intact. These late blocks become connected together, as if they were written one after another.

For example, the master function in Listing 4.9 (a) invokes functions `f1`, ..., `fn` in that order. When `fM` is invoked, all the `f`*i* are executed whereby the staging parameters get resolved. As a result, all code fragments in `f`*i* become connected through staging as shown in Listing 4.9 (b).

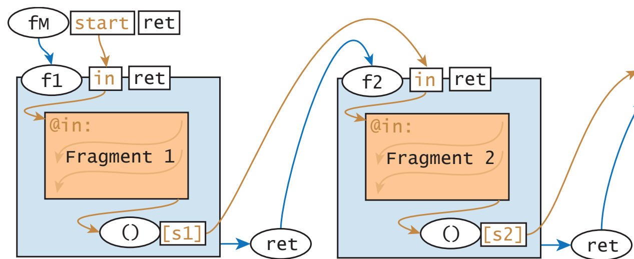
We refer to such pattern as *fragment chaining*, as it allows fragments of code to be specified in independent container functions. When staging is resolved these fragments become connected forming a new program. Naturally, container functions can be reused multiple times, in which case the late blocks become replicated. Such pattern can be used, for example, in recursion and loop unrolling as we show in Section 6.2.

```
let f1(stage sin, fn[stage] cont) {
  ... normal code ... [jmp]
  @sin: ... fragment 1 code ... [sout]
  @jmp: cont . sout
}
let f2(stage sin, fn[stage] cont) {
  ... normal code ... [jmp]
  @sin: ... fragment 2 code ... [sout]
  @jmp: cont . sout
}
let f3 ...
let f4 ...
let fM(stage start, fn[stage] ret) {
  f1 . start (s1)
  f2 . s1 (s2)
  f3 . s2 (s3)
  f4 . s3 (s4)
  ret . s4
}
```

(a) *Unconnected code fragments.*

```
@start: ... fragment 1 code ... [s1]
@s1: ... fragment 2 code ... [s2]
@s2: ... fragment 3 code ... [s3]
@s3: ... fragment 4 code ... [s4]
...
```

(b) *Generated code after fragment chaining.*



(c) *Graphical representation.*

Listing 4.9: *Fragment chaining pattern:*
(a) *Function* `f1` *and* `f2` *contain a code fragment that is staged upon the* **`stage`** *parameter* **`in`** *and the stage of the fragment's last instruction is returned. These functions are invoked in a sequence, chaining the code fragments by passing the result of one function as an input for the next one.*
(b) *The result of the invocation of the master function* `fM`.
(c) *Graphical representation of the fragment chaining pattern. Blue blocks represent the code being executed immediately, orange blocks are the deferred fragments chained together.*

```
... ()[p]
@p:
()[q] {
  ... rest of the program ...
  @p: ...
  @q: ...
} .
```

(a)

```
... ()[p]
@p:
let [q];
... rest of the program ...
@p: ...
@q: ...
```

(b)

Listing 4.10: *Stage Branch pattern: A let-like construct naming an implicit staging parameter. Note the invocation dot at the very end of **(a)**. Within the body of let, instructions staged directly or indirectly upon p are executed before stage q becomes active. Example **(b)** is a syntactic sugar of the same construct.*

Stage Branch

A counterpart of a stage chain is a *stage branch*. It is a small function where we create two or more stage variables, marking the beginning of multiple chains. Stage branches are particularly useful when an endpoint of a chain is difficult or impossible to specify.

The Stage Branch pattern is a parameter-less anonymous function, containing the remaining program, as depicted in Listing 4.10a. The function is invoked in some stage p and defines a new implicit staging variable q. Within the body of the function, both p and q can be used.

Recall, that an action can be executed only when no nested actions are active – a property formally explained in Section 4.1.3. That is why, a prerequisite for the construct to be executed is that no other action is staged upon p. This means, stage q becomes active only when all instructions staged directly or indirectly upon p are already completed.

In other words, when such construct is encountered the DeepCPS interpreter executes everything under stage p first, and then proceeds with the stage q.

The stage branch function is very similar to a let construct: A **`let`** is a lambda containing the remainder of the program, invoked immediately with a single argument. The argument is given a name within the let lambda, and can be referred through that name within the program. A stage branch takes no arguments, but gives a name to an implicit stage parameter.

In order to reflect this similarity and avoid braces encapsulating a reminder of the program we support a special **`let`** construct, as shown in Listing 4.10b. A typical **`let`** binds a value to a normal parameter. However, if the **`let`** is cut short by a semicolon, it is simply a parameter-less let lambda. Both normal let constructs or parameter-less let lambda can explicitly name its implicit staging parameter. This way `let [s];`, with the semicolon at the end, defines the stage branch pattern in a concise way.

4.2 Language Creation

In Section 2.6 we explained two main approaches for language manipulation. The simpler and more common approach is through macro processing. A less explored route is to provide the programmer direct access to the parser. ManyDSL uses the second approach, providing unprecedented flexibility.

As we briefly explain in Section 3.5, source code is being parsed in fragments by a mutable parser, initially in DeepCPS. The produced Target Representation (TR) code is then interpreted or compiled to LLVM. Interpreted code may affect the parser, changing how the next fragment of the source file is processed.

4.2.1 Interleaved Parsing

A typical interpreter pipeline consists of 2 sequential steps: First, the source code is parsed, forming some internal representation of the code. In the second step, the code is executed. ManyDSL cannot follow this typical setup because we want the second step to affect the first — the code may modify the parsing process.

Our solution is to interleave parsing and interpretation. ManyDSL is able to parse only a portion of the code into TR, up to what we call a *halt marker* and then freeze the parsing state. The available TR is interpreted, possibly affecting the parser state. Finally when parsing resumes it may already work in a different setup or with different language.

There are numerous design considerations to be made in order to ensure that interleaved parsing is possible.

First, it is important to ensure that when the halt marker is reached, all the code preceding it is actually processed to produce TR. For a bottom-up parser this is a major problem, especially in the context of continuation-passing style where the parse tree is deeply nested. A single incomplete node, caused by the halt marker, prevents all nodes in the path to the root to be built. With such an incomplete parse tree and missing nodes in the TR structure, the interpreter cannot run. A top-down parser on the other hand can create TR nodes *before* descending it their children. All program structure up to the halt marker can be complete at the moment when the parser is interrupted.

Another design consideration is the amount of lookahead of the parser. Since the grammar and lexing tokens can change during execution phase, the parsing cannot rely on tokens that appear after the halt marker. Moreover, as we will show shortly, any custom language semantic may involve altering the parser in a similar way, further restricting the permitted lookahead.

Taking these constraints into the account, we have chosen to limit ourselves to LL(1) grammars. Note, that languages can still be made powerful and expressive in this grammar class. Most notably, Python is explicitly restricted to LL(1) for

```
Program ::= Expr { print Expr.val }
Expr    ::= Diff { Expr.val = Diff.val }
Diff    ::= Diff₁ "-" Quot { Diff.val = Diff₁.val - Quot.val }
Diff    ::= Quot { Diff.val = Quot.val }
Quot    ::= Quot₁ "/" Value { Quot.val = Quot₁.val / Value.val }
Value   ::= "(" Expr ")" { Value.val = Expr.val }
Value   ::= Number { Value.val = Number.val }
```

Listing 4.11: *A grammar for left-associative subtract-and-divide expression using the SDT scheme, taking operator precedence into account.*

simplicity sake[1].

4.2.2 Syntax-Directed Execution

A syntax-directed translation (SDT) scheme [88, 104] is a method for specifying parsers that read the source code and translate it to some other representation. The parser is specified by a context-free grammar augmented with *semantic actions*. The semantic actions are arbitrary pieces of code that may appear at any position within the body of a production. In addition, each symbol within the productions can have arbitrarily many associated attributes. The actions read and modify these attributes. This way actions can communicate and form meaningful programs.

Parsing using a SDT scheme, in the most general approach, is performed in two steps. First, the source is parsed forming a parse tree, containing the actions as separate nodes. Attributes are stored as mutable record fields within each node. Then, the tree is traversed in a left-to-right, depth-first order, executing all the action nodes in the order that they are encountered. A more common approach is to execute the semantic actions on the fly, during parsing, without ever creating a parse tree.

Consider an example grammar of binary mathematical operations: subtraction and division. Such "MinusDiv" grammar which we will use throughout this chapter, uses two left-associative operators. In addition, care has to be taken for the operator precedence. A simple solution in SDT scheme, is given in Listing 4.11. This version of the grammar is left-recursive and cannot be used as-is in an LL(1) parser. We address this problem briefly, near the end of this section.

In the example Listing 4.11, the expression is evaluated and the parser prints a single value at the end. In practice however, SDT are designed with *translation* in mind – the semantic actions create a custom representation of code based on the input syntax. This usually is an explicitly built AST resembling the parse tree.

We propose a *Syntax-Directed Execution* (SDE) scheme. Formally, it is equivalent to the SDT scheme explained above. SDE however puts emphasis on the execution of the code, rather than translation to another representation. Productions are treated as syntax-directed functions, containing local variables, further function calls, and snippets of code. We replace attributes with an equivalent notion

[1] https://www.python.org/dev/peps/pep-3099/, retrieved on 26.02.2017

of production-local variables, which are passed into and out of the terms as arguments.

More formally, a language in SDE scheme is defined by an L-attributed LL(1) grammar [89] augmented by *semantic actions* that may appear at any position in the production body. In other words, the grammar is a set of productions of the form $\mathbf{N} \rightarrow (\mathbf{\Sigma} \cup \mathbf{N} \cup \mathbf{A})*$, where:

- $\mathbf{\Sigma}$ is a set of terminals.
- $\mathbf{N}$ is a set of nonterminals.
- $\mathbf{A}$ is a set of all possible actions, which are arbitrary DeepCPS functions.

Each nonterminal $N \in \mathbf{N}$ can take an arbitrary number of input parameters i^N and return any number of values o^N. Whenever N is used as a term t within a body of a production, it takes input arguments i^t which must match the parameters i^N. Similarly, term t output parameters, o^t, must match the returned values o^N. We represent each attributed term as $(i^t) \rightarrow t \rightarrow (o^t)$ to indicate the flow of the data — input arguments (i^t) are passed into the term t and the results are returned into (o^t). A complete parametrized production of the form $N ::= t_1 t_2 t_3$ looks as:

$$
\begin{aligned}
(i^N) \rightarrow N \rightarrow (o^N) ::= \; & (i^{t_1}) \rightarrow t_1 \rightarrow (o^{t_1}) \\
& (i^{t_2}) \rightarrow t_2 \rightarrow (o^{t_2}) \\
& (i^{t_3}) \rightarrow t_3 \rightarrow (o^{t_3})
\end{aligned}
$$

The (red) parameters are lists of names defining new local variables and can be used at any later position. If the same name is used multiple times, the new variable hides the previous one. The (green) values are lists of arguments which refer to previously declared parameters. The final output values (o^N in the example) can refer to all parameters in the production, as if it was written at the very end. Unlike in SDT scheme, references are plane names without naming the term that it originates from.

Terminals take no arguments and may return a single string value representing the actual token that has been read. This is useful when a token is not a single string constant but is given in a form of regular expression.

Semantic actions in SDE are functions with their own, independent scope. Production local variables have to be explicitly passed and returned from the action.

In SDE scheme a grammar must be L-attributed. That means, that the hypothetical parse tree can be evaluated with a single depth-first left-to-right traversal (Figure 4.10). This put restrictions on the values that can be used as arguments:

- Attributes i^{t_x} may only depend on i^N as well as o^{t_y} for $y < x$.
- Attributes within the tuple o^N may depend on i^N as well as any o^{t_y}.

```
()->Program->()  ::= ()->Expr->(val) (val)->{ print val }->();
()->Expr->(val)  ::= ()->Diff->(val);
()->Diff->(val)  ::= ()->Diff->(left) "-" ()->Quot->(right)
                     (left,right)->{ return left-right }->(val);
()->Diff->(val)  ::= ()->Quot->(val);
()->Quot->(val)  ::= ()->Quot->(left) "/" ()->Value->(right)
                     (left,right)->{ return left/right }->(val);
()->Value->(val) ::= "(" ()->Expr->(val) ")";
()->Value->(val) ::= Number->(str)
                     (str)->{ return str2int(str) }->(val);
```

Listing 4.12: *A grammar for left-associative subtract-and-divide expression using the SDE syntax, taking operator precedence into account. This grammar however cannot be parsed by a predictive LL(1) parser due to left recursion.*

The L-attributed restriction is what enables us to treat attributes as local variables.

L-attributed LL parsers do not need an explicit parse tree. Instead, a parser can behave as an interpreter of a low-order functional language. Nonterminals can be viewed as *functions* which take input arguments and return output values. In that view, a production $p \to t_1 t_2 ... t_n$ becomes a definition of a function, with the terms t_x forming a sequence of instructions to be performed when p is invoked.

The direct translation of the SDT-based “MinusDiv” grammar from Listing 4.11 into SDE is givesn in Listing 4.12. The grammar, however, contains left recursion in both `Diff` and `Quot` rules. As such it is not an LL(1) grammar and does not satisfy the SDE requirements. A well-known solution to left recursion is to take the offending production:

$$A ::= A\alpha$$
$$A ::= \beta$$

and replace it with right recursive grammar, such as:

$$A ::= \beta C$$
$$C ::= \epsilon$$
$$C ::= \alpha C$$

The new grammar can parse the same input, but is LL(1). The new grammar produces a different parse tree, but the SDE does not rely on that. The position

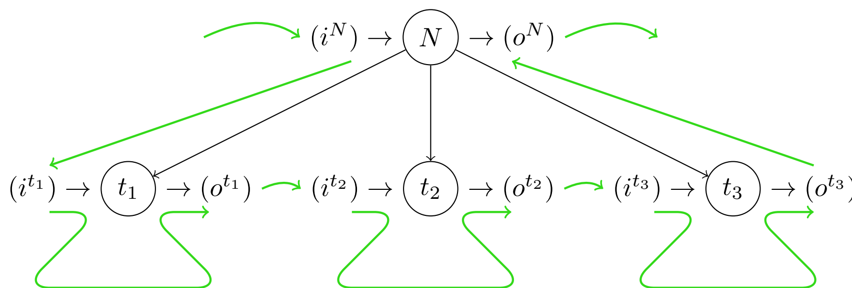


Figure 4.10: *A fragment of attributed parse tree for a production $N \to t_1 t_2 t_3$. An L-attributed tree can be traversed in a depth-first left-to-right fashion (green path). Each attribute along the path can be evaluated based on the values encountered earlier.*

```
()->Left->(val)           ::= ()->Number->(val) (val)->LeftCont->(val);
(val)->LeftCont->(val) ::= epsilon;
(val)->LeftCont->(val) ::= Operator ()->Number->(val2)
                                (val,val2)->action->(val)
                                (val)->LeftCont->(val);

()->Right->(val)          ::=()->Number->(val) (val)->RightCont->(val);
(val)->RightCont->(val)::= epsilon;
(val)->RightCont->(val)::= Operator ()->Number->(val2)
                                (val2)->RightCont->(val2)
                                (val,val2)->action->(val);
```

Listing 4.13: *Left and right associative binary operators. The* `action` *takes 2 values and performs the intended operation. The action of the left-associative operator is performed before the recursion, while for the right-associative operator it is performed after the recursion.*

of semantic actions is what matters and differentiates left and right-recursive languages.

Let us consider an implementation of left and right associative infix binary operators as an example of left and right recursion. In SDE these binary operators look the same, with an exception on the position of the semantic actions, as shown in Listing 4.13. For the left-associative operator, the action is performed before the recursion. The value `val` is passed recursively and combined with consecutive values `val2` in the order they appear in the input. In the right-associative operator, the action is performed after the recursion. Consequently, a recursive subexpression `RightCont` is reduced to a single value `val2` before it is combined with the left value `val`.

4.2.3 SDE: Formal Definition

Syntax

We define the complete Syntax-Directed Execution scheme in two parts. First, we specify how the language syntax is defined, i.e. the syntax of language specification. Then, in a method similar to small-step semantics we define how the parser operates.

A syntax of the language specification in SDE scheme is given in Figure 4.11. A complete language is a set of productions p.

Each production uses a nonterminal n as its (not unique) name. A production can take any number of input ($\overline{x}$) and output ($\overline{z}$) parameters. If two productions use the same nonterminal as their name, they must match their signatures. That means, the number of input and output parameters must match between the productions.

The production body is a sequence of attributed terms. Each term refers to either a terminal, nonterminal, or a DeepCPS function. A terminal reference has a single output parameter. Nonterminal and function references can take multiple input and output parameters.

$$
\begin{array}{rll}
n \in & N & \text{(nonterminal)} \\
\sigma \in & \Sigma & \text{(terminal)} \\
L ::= & \{\overline{p}\} & \text{(language)} \\
p ::= & (\overline{x})n(\overline{z}) ::= b & \text{(production)} \\
b ::= & \overline{a} & \text{(production body)} \\
a ::= & & \text{(attributed term)} \\
 & \sigma(z) & \text{(attributed terminal)} \\
 & (\overline{x})n(\overline{z}) & \text{(attributed nonterminal)} \\
 & (\overline{x})\{\lambda\overline{y}.b\}(\overline{z}) & \text{(attributed action)} \\
x, y, z & & \text{(parameter)} \\
\overline{x} & & \text{(list of parameters)}
\end{array}
$$

Figure 4.11: *Syntax of language specification in SDE scheme. A language is a set of productions, each defining a transformation of a nonterminal n to a sequence of attributed terms. Each attributed term refers to terminal, nonterminal, or a DeepCPS function, providing a set of input and output parameters.*

Support Function

Before we define the formal semantics, let us define a support function: FIRST. The function computes the set of terminals that may begin a given sequence of terms after they have been expanded. This is the function one typically computes when creating a normal LL(1) parser[3, Chapter 4.4.2]. We define the function recursively as follows:

$$
\begin{aligned}
\mathit{FIRST}(\sigma) &= \{\sigma\} \\
\mathit{FIRST}(\lambda x.b) &= \{\epsilon\} \\
\mathit{FIRST}(n) &= \bigcup \{\mathit{FIRST}(p) : p \in L, p = (n ::= b)\} \\
\mathit{FIRST}(p) &= \mathit{FIRST}(n ::= b) = \mathit{FIRST}(b) = \mathit{FIRST}(\overline{a}) \\
\mathit{FIRST}(\overline{a}) &= \mathit{FIRST}(a_1\overline{a_r}) = \\
&= \begin{cases} \{\epsilon\} & \overline{a} = \epsilon \\ \mathit{FIRST}(a_1) & \epsilon \notin \mathit{FIRST}(a_1) \\ (\mathit{FIRST}(a_1) \setminus \{\epsilon\}) \cup \mathit{FIRST}(\overline{a_r}) & \epsilon \in \mathit{FIRST}(a_1) \end{cases}
\end{aligned}
$$

Typical LL(1) parsers also use a FOLLOW function to compute which terminals can appear after the epsilon rules. We do not do this as it may work incorrectly for a source containing multiple languages – a problem we discuss further in Section 5.2. Instead, we treat ϵ as a *wildcard production*, which is taken when all other productions fail.

All languages recognized by the traditional LL(1) can be parsed by ours as well, but some languages that could be considered incorrect for LL(1) parsing are

accepted by ours. Our parser silently gives non-epsilon productions a higher priority. In case of a conflict between epsilon and non-epsilon production, the latter is taken.

Finally, similarly to LL(1) we define the *PREDICT* function. For a given terminal σ and nonterminal n, the *PREDICT* function finds the production p that expands from n and $\sigma \in \mathit{FIRST}(p)$. If no p satisfies the constrain, *PREDICT* chooses an epsilon rule of n. Finally, if no such rule exists, it returns a special error value E, indicating an error in parsing. During parsing the *PREDICT* function helps find the right production to expand to, based on the input.

In the formal definition of *PREDICT* we use a helper function $N(p)$ which gives the nonterminal n that the given production expands for.

$$\mathit{PREDICT}(\sigma, n) = \begin{cases} p : \sigma \in \mathit{FIRST}(p) \land N(p) = n & \sigma \in \mathit{FIRST}(n) \\ p : \epsilon \in \mathit{FIRST}(p) \land N(p) = n & \sigma \notin \mathit{FIRST}(n) \land \epsilon \in \mathit{FIRST}(n) \\ E & \sigma \notin \mathit{FIRST}(n) \land \epsilon \notin \mathit{FIRST}(n) \end{cases}$$

Semantics

Let us now define the semantics of the Syntax-Directed Execution (SDE) scheme. In SDE the language specification acts like a program. However, the program is not executed alone, in the void. Instead, as it executes, it consumes an input sequence of terminals $I ::= \overline{\sigma}$. For that reason, the execution state is represented by a pair: $\langle I, P \rangle$, where:

- I is the input string of the terminals $I = \overline{\sigma}$.
- P is the parser state, given as a sequence of attributed terms. At the beginning, the parser is set to a single starting nonterminal S, taking and returning no arguments.

In each step, the state transforms according to the rules given in Figure 4.12.

- The (Term) rule accepts and consumes a terminal when it appears both in the input and in the program.
- The (NTerm) rule is used when a nonterminal n is encountered in the program. We use previously defined *PREDICT* function to find a production that contains the next input symbol σ_1 in its *FIRST* set, or – if no production can satisfy the constraint – the wildcard epsilon production.
- The (ActCall) rule is used when an action, represented as a DeepCPS function, is encountered. The input parameters are passed into the DeepCPS lambda, together with a special returning continuation τ.
- With the help of (ActRun), the underlying DeepCPS program is allowed to execute freely, without any interference from the parser.

$$\frac{\sigma_1 = \sigma_t}{\langle \sigma_1 \overline{\sigma}, \sigma_t(z)\overline{a} \rangle \to \langle \overline{\sigma}, [z \mapsto str(\sigma_1)]\, \overline{a} \rangle} \qquad \text{(Term)}$$

$$\frac{PREDICT(\sigma_1, n) = p = ((\overline{x'})n(\overline{z'}) ::= b_p)}{\langle \sigma_1 \overline{\sigma}, (x)n(z)\overline{a} \rangle \to \langle \sigma_1 \overline{\sigma}, (\left[\overline{x'} \mapsto \overline{x}\right] b_p)\, \left[\overline{z} \mapsto \overline{z'}\right] \overline{a} \rangle} \qquad \text{(NTerm)}$$

$$\langle \overline{\sigma}, (\overline{x})\{\lambda \overline{y}.b\}(\overline{z})\overline{a} \rangle \to \left\langle \overline{\sigma}, \{ \begin{bmatrix} y_1 \mapsto x_1 \\ \vdots \\ y_n \mapsto x_n \\ y_{n+1} \mapsto \tau \end{bmatrix} b\}(\overline{z})\overline{a} \right\rangle \qquad \text{(ActCall)}$$

$$\frac{b \to b'}{\langle \overline{\sigma}, \{b\}(\overline{z})\overline{a} \rangle \to \langle \overline{\sigma}, \{b'\}(\overline{z})\overline{a} \rangle} \qquad \text{(ActRun)}$$

$$\langle \overline{\sigma}, \{\tau \overline{z'}\}(\overline{z})\overline{a} \rangle \to \langle \overline{\sigma}, \left[\overline{z} \mapsto \overline{z'}\right] \overline{a} \rangle \qquad \text{(ActRet)}$$

Figure 4.12: *The semantics of SDE*

- Finally, by the rule (ActRet), when τ is invoked from within the DeepCPS program, control is returned back to the parser. The arguments received by τ are forwarded to the output parameters of the initial action call.

4.2.4 LangDSL

We have realized the SDE concept in a new language `LangDSL` as a part of ManyDSL. The `LangDSL` itself is built on top of DeepCPS, using the mechanism that we are describing right now.

At the top level of LangDSL a language name is defined, followed by its contents within curly braces. The language definition uses the syntax similar to the SDE scheme, summarized in Figure 4.13. It also includes lexing rules such as keywords or regular expression lexemes.

Semantic actions are DeepCPS functions. When a parser reaches a semantic action, the execution control is passed temporarily to the DeepCPS interpreter, employing the interleaved parsing explained in Section 4.2.1. The execution control is released back to the parser when the special return continuation τ is invoked.

The input and output action arguments must match the parameters of the DeepCPS function. Consider an action of the form $(i^A) \to A \to (o^A)$ containing n input and m output arguments. A DeepCPS function used in A must have $n+1$ parameters: $p_1, p_2, ..., p_{n+1}$. When A is invoked, the arguments are mapped as follows

- For $x \leq n$, o^A_x is mapped to p_x.

Parser rules:

$L: \overline{p}$	`grammar L { p1 p2 ... }`	(language)
$p: (\overline{x})n(\overline{z}) ::= b$	`(x1,x2,...)->` `n->(z1,z2,...)::=`b`;`	(production)
$b: \overline{a}$	`a1 a2 ...`	(production body)
$a: \sigma(z)$	σ`->(z)`	(attributed terminal)
$a: (\overline{x})n(\overline{z})$	`(x1,x2,...)->n->(z1,z2,...)`	(attributed nonterminal)
$a: (\overline{x})\{\lambda\overline{y}.b\}(\overline{z})$	`(x1,x2,...)->(z1,z2,...)` `{ (y1,y2,...)` b `}`	(attributed action)

Terminals:

σ:	**token** σ *regexp*	(regular expression terminal)
σ:	**keyword** σ *identifier*	(keyword)
σ:	**symbol** σ *str*	(symbol)

Figure 4.13: *Syntax of LangDSL in relation to SDE scheme. LangDSL also defines the actual terminals through* **token** *(any regular expression),* **keyword** *(alphanumeric identifier, recognized only as a whole word) and* **symbol** *(any sequence of characters matched in verbatim).*

- An implicitly defined return continuation τ is mapped to p_{x+1}.
- The return continuation τ expects m arguments $r_1, r_2, ..., r_m$. When invoked, the values r_x are mapped to o^A_x and the execution control is released back to the parser.

For example, an action with 3 input and 2 output parameters would look as:

```
(i1, i2, i3)->(o1,o2)  {
  (p1, p2, p3, return)
    ... DeepCPS code ...
    return . r1 r2
}
```

We renamed the parameter p4 as `return` to better indicate its functionality. It should be pointed however that `return` is a parameter name, not a keyword.

For convenience additional syntactic sugars are available, summaried in Figure 4.14:

1. We omit the header of a lambda which becomes the semantic action. We assume that the input names p_x have the same name as the arguments i_x. The returning continuation is always named `return`.
2. When no arguments are passed in or out from a term, the empty tuple indicating it can be completely omitted.
3. Simple renaming can be achieved with input-output argument syntax without any action body.
4. If a sequence of last input or output values have the same name as the corresponding parameters in the nonterminal definition, then they can be omitted as well.
5. A free variable within a production body is treated as if it was the input parameter of the production.

Omitting the lambda header:

```
(i1, i2, ...)->(o1, o2, ...) {
(i1, i2, ..., return)
...DeepCPS function body...
}
```

```
(i1, i2, ...)->(o1, o2, ...) {
...DeepCPS function body...
}
```

Omitting empty argument lists:

```
()->N->(o)
```

```
N->(o)
```

Renaming:

```
(in)->(out) { return . in }
```

```
(in)->(out)
```

Default arguments:

```
Left->(val) ::= Number->(val)
    (val)->LeftCont->(val);
(val)->LeftCont->(val) ::= ...
```

```
Left->(val) ::= Number->(val)
    LeftCont;
(val)->LeftCont->(val) ::= ...
```

Free variable as input parameter:

```
(val)->RightCont->(val) ::=
    Operator Number->(val2)
    (val2)->RightCont->(val2)
    (val,val2)->(val) {...};
```

```
RightCont->(val) ::= Operator
    Number->(val2)
    (val2)->RightCont->(val2)
    (val,val2)->(val) {...};
```

Figure 4.14: *A summary of syntactic sugars used in LangDSL. Each pair is equivalent. On the left is the canonic LangDSL and the right is the version with syntactic sugar used.*

The last two syntactic sugars are the most involved and require an explanation. The automatic insertion of additional parameters and arguments is not performed during parsing, but rather when a grammar definition is complete and the parser is being compiled. It is an iterative process: adding default arguments within a production body may introduce free variable names, causing an addition to the parameter list of the production header. The process is maintained until a fixed point is reached.

This mechanism is useful when a deeply nested grammar subexpression requires a specific value which originates from a much higher level rule. Let us consider the `MinusDiv` grammar, given in Listing 4.14. Let us assume that the `Value` rule may be an identifier. It uses a lookup map `env` to convert a name to an actual value (we discuss the lookup map usage in detail in Section 6.4.3):

```
(env)->Value->(val) ::= Id->(str)
                        (str,env)->(val) {
                            "env.lookup(str)" . (val)
                            return . val
                        };
```

With such a change, the top level `Expr` must take the `env` as an argument and propagate it into all rules which may ultimately use `Value`. However, this process can be automated by using the the default arguments and parameters. It suffices to just pass `env` to `Expr`, without changing any of the nonterminals on the path between `Expr` and `Value`.

```
grammar MinusDiv {
token Number [[:digit:]]+;
Program       ::= Expr (val)->() { print . val return };
Expr->(val)   ::= Diff;
Diff->(val)   ::= Quot->(left) DiffCont;
(left)->DiffCont->(val) ::= epsilon (left)->(val);
(left)->DiffCont->(val) ::= "-" Quot->(right)
                           (left,right)->(val) {
                             left-right . (val) return . val
                           }
                           (val)->DiffCont->(val);

Quot->(val)   ::= Value->(left) QuotCont;
(left)->QuotCont->(val) ::= epsilon (left)->(val);
(left)->QuotCont->(val) ::= "/" Value->(right)
                           (left,right)->(val) {
                             left/right . (val) return . val
                           }
                           (val)->QuotCont->(val);

Value->(val)  ::= "(" Expr ")";
Value->(val)  ::= Number->(str)
                     (str)->(val) {
                         str2int . str return
                     };
}
```

Listing 4.14: *A grammar for left-associative subtract-and-divide expression using LangDSL syntax with syntactic sugars. The grammar replaces left recursion with right recursion while the operators are still left-associative.*

Grammar function definition:
function f<$\overline{x}$> { $\overline{p}$ return n; }
$\overline{x}$ – list of parameters
$\overline{p}$ – sequence of productions
n – nonterminal (appearing in $\overline{p}$)

Grammar function usage:
f<$\overline{a}$>
$\overline{a}$ – list of attributed terms

Invocation semantics:

$$f\texttt{<}\overline{a}\texttt{>} \rightarrow \left(\begin{bmatrix} x_1 \mapsto a_1 \\ x_2 \mapsto a_2 \\ \dots \end{bmatrix} \overline{p} \right) .n$$

where $\overline{p}.n$ denotes nonterminal n defined by the set of productions $\overline{p}$.

Figure 4.15: *The syntax and semantics of grammar function definition and invocation. A grammar function defines a portion of grammar $\overline{p}$ with parameters $\overline{x}$ acting as placeholders. When invoked, the function returns a single nonterminal n which can be used in the same context as a regular nonterminal, i.e. within the production definitions. Rules for other nonterminals defined within the function are private and cannot be referenced from the outside.*

4.2.5 Grammar Composition

Some grammar constructs can be repetitive, appearing in many languages. Some can even reappear in different contexts within the same language. For example, comma-separated parameter lists, infix binary expressions, parenthesized instruction blocks, etc. The removal of the left recursion, and the left-associative

```
function lassoc<elem, op, action> {
  N->(val) ::= elem->(val) (val)->NCont->(val);
  (val)->NCont->(val) ::= epsilon;
  (val)->NCont->(val) ::= op elem->(right)
      (val,right)->action->(val) (val)->NCont->(val);
  return N;
}
function rassoc<elem, op, action> {
  N->(val) ::= elem->(val) (val)->NCont->(val);
  (val)->NCont->(val) ::= epsilon;
  (val)->NCont->(val) ::= op elem->(right) (right)->NCont->(right)
      (val,right)->action->(val);
  return N
}
grammar MinusDiv {
token Number [[:digit:]]+;
Program       ::= Expr (val)->() { print . val return };
Expr->(val)   ::= Diff;
Diff->(val)   ::= lassoc<Quot,"-", (left,right)->(val) {
                            left-right . (val) return . val
                          }>;
Quot->(val)   ::= lassoc<Value,"/", (left,right)->(val) {
                            left/right . (val) return . val
                          }>;
Value->(val) ::= "(" Expr ")";
Value->(val) ::= Number->(str)
                    (str)->(val) {
                        str2int . str return
                    };
}
```

Listing 4.15: *A left- and right-associative binary operator abstractions in LangDSL. The following* `MinusDiv` *grammar takes advantage over the generic* `lassoc`*.*

binary operator are also examples of repetitive grammar constructs.

LangDSL permit abstractions over portions of grammars, as shown in Figure 4.15. Such a grammar function takes an arbitrary number of values, lexemes, and nonterminals and creates a grammar fragment represented as a set of productions as a result. These productions can be used as a part of a concrete grammar definition. The production names within the function have only a local scope and cannot be referenced from outside. However, nonterminals which are returned from such function can be bound to names at the call site, and used in other parts of the grammar.

Consider an example in Listing 4.15, which defines the same “MinusDiv” grammar as in Listing 4.14. This time however, the left-recursion construct is abstracted out within the functions `lassoc` and `rassoc` for left- and right-associative operators. The abstractions can be used multiple times within a grammar without interference. They can also be used in multiple languages. In the LangDSL implementation, we provide grammar libraries, including the `lassoc` and `rassoc` functions, allowing the user to immediately use them to define their own languages.

4.2.6 Abstraction over Parameters

The parameters of a regular function are of certain types (integers, reals, complex, etc...). A statically-typed higher-order function accepts or returns functions of certain type. The arguments which are functions themselves must have a fixed set of parameters. Grammar functions on the other hand may take nonterminals with unspecified number of input and output parameters. This capability is essential, because the same grammar constructs are often needed with different kind of terms.

Consider the `MinusDiv` grammar handling values in a different ways:

- A number may be a quotient represented by a pair of integers. A `Value` would return a pair: `->(num, denom)`.
- A number may be actually an identifier naming a value defined earlier. A `Value` would require an environment map `env` which maps identifiers to values in order to obtain a literal constant. We have `(env)->Value->(val)`.
- Expression values may be kept in a single stack structure `S` rather than represented as explicit local grammar parameters. We would then have a production `(S)->Value->(S')`, where the new stack `S'` contains the parsed value on top of the old `S`.

In all these scenarios we would like to be able to reuse the same `lassoc` and `rassoc` abstractions to define the `MinusDiv` grammar.

LangDSL abstractions help achieve that kind of flexibility in three ways.

First, the default arguments and parameters that we described in Section 4.2.4 help define abstractions polymorphic with respect to the nonterminal parameters. For example, if `Value` nonterminal expects `env` as an input, this does not need to be spelled out within the function. It would be added as the default argument once the function is invoked.

Secondly, the function syntax has been extended to support a *name list* object as an argument. A *name list* is a parenthesised tuple of identifiers. The parameter accepting a name list n can appear within the function body in the context of any regular parameter/argument list l. When the function is invoked, the contents of n are spliced into l. To avoid potential name conflicts, each use of the name list can be given a prefix name. To clarify, consider a simple example:

```
function invoke<N, ins, outs> {
  (ins)->a->(pref.outs) ::=
          (ins)->N->(pref.outs)
          (pref.outs)->(outs);
  return a;
}
(env)->Expr->(num, denom) ::= invoke<Value,(env),(num, denom)>;
```

causes substitution `ins` $\mapsto$ `env` and `outs` $\mapsto$ `num, denom` within all parameter/argument lists appearing within the grammar function body. This example reduces to:

```
function lassoc<elem, val, op, action> {
  N->(val) ::= elem->(val) (val)->NCont->(val);
  (val)->NCont->(val) ::= epsilon;
  (val)->NCont->(val) ::= op
    elem->(right.val)
    (val,right.val)->action->(val)
    (val)->NCont->(val);
  return N;
}
lassoc<Value, (num,denom), "/", action > =
  N->(num,denom) ::= elem->(num,denom)
      (num,denom)->NCont->(num,denom);
  (num,denom)->NCont->(num,denom) ::= epsilon;
  (num,denom)->NCont->(num,denom) ::= op
    elem->(right.num, right.denom)
    (num,denom,right.num,right.denom)->action->(num,denom)
    (num,denom)->NCont->(num,denom);
  return N;
}
```

Listing 4.16: *LangDSL abstraction taking a nonterminal argument list as an additional parameter* `val`*. The contents of the parameter* `val` *are spliced every time it is used in the parameter list. For example, by passing a pair* `num,denum` *we obtain a set of productions exchanging such a pair in between them.*

```
(env)->Expr->(num, denom) ::=
        (env)->Value->(pref.num, pref.denom)
        (pref.num, pref.denom)->(num, denom);
```

In the example Listing 4.16 the `lassoc` grammar function uses a name list object `val`. In the last production, in order to differentiate between the operands, the right value is given a separate name `right` which becomes the prefix for all `val` elements. The elements of `val` are still spliced into the argument list and do not form a separate tuple. Doing otherwise would be problematic as argument count would not match the parameters.

Finally, to avoid excessive and repetitive name list passing, input and output parameter lists can be extracted for every argument term. If a parameter `t` receives an argument of the form `(i)->n->(o)` a special value `t:in` gives the `(i)` tuple, and `t:out` gives the `(o)` tuple. This allows abstractions to be polymorphic with respect to term argument lists without explicitly requiring the caller to specify these lists. For example, the same `invoke` function can be rewritten as:

```
function invoke<N> {
  (N:in)->a->(pref.N:out) ::=
          (N:in)->N->(pref.N:out)
          (pref.N:out)->(N:out);
  return a;
}
(env)->Expr->(num, denom) ::= invoke<Value>;
```

When invoked, this example reduces to the same result as before, assuming that the nonterminal `Value` takes `(in)` as input and `(num, denom)` as output.

As an additional syntactic sugar, any term or expression can be bound to a new name using the **`alias`** keyword. With it the example `lassoc` can be made polymorphic with only a single additional line, as shown in Listing 4.17.

```
function lassoc<elem, op, action> {
  alias val = elem:out;
  N->(val) ::= elem->(val) (val)->NCont->(val);
  (val)->NCont->(val) ::= epsilon;
  (val)->NCont->(val) ::= op
    elem->(right.val)
    (val,right.val)->action->(val)
    (val)->NCont->(val);
  return N;
}
```

Listing 4.17: *LangDSL abstraction taking a polymorphic `elem` nonterminal. All of its output arguments `elem:out` are used by the productions in the abstraction*

A different approach to parameter passing could involve declaring variables having a scope over multiple productions. We have decided, however, not to take that route. Such design of captured, mutable variables known from imperative languages has proven confusing in the past. It creates a parser state which can be hard to track by the programmer.

4.3 Parser Actions

In the Section 4.2.4 we have shown how Syntax Directed Execution scheme is realized in LangDSL, which is a mixture of grammar specification and DeepCPS code. We have then shown how grammar fragments can be abstracted and combined. In this chapter we focus on the semantic actions: Explain how exactly the execution process works. We give examples how DeepCPS, and dynamic staging in particular, can be used not only to execute code during parsing, but also to produce refined code to be executed at a later time.

What is unique for SDE scheme and DeepCPS staging, is that no additional intermediate representation is used. We do not create node objects that would need to be explicitly visited later on to produce code. Instead, an action is invoked as if it was an ordinary DeepCPS function. The function may, through dynamic staging, produce code, but it is not obliged to do so.

In Section 4.2 we included examples with only the simplest semantic actions which perform the actual computation immediately during parsing. However, since DeepCPS is a higher-order language, a semantic action can produce a function holding a piece of code. Dynamic staging can be then used to resolve any kind of overhead resulting from that kind of functional encapsulation.

Throughout this chapter we will refer to the function that is being generated as *program* or simply as a P function. This is to distinguish the code we build, from other functions and code pieces that are executed during parsing.

4.3.1 Building Code

DeepCPS with staging can be used to build a program piece-by-piece without any overhead in the generated code. Each piece is represented in the most generic way as a regular function, taking arbitrary many arguments, performing some

operation, and returning through a continuation:

```
let code
  (!args1, return) {
    ...do something...
    return . !args2
  }
```

Recall from Section 4.1.2, `!arg1` is tuple aggregation, permitting `code` function to accept any number of arguments. At the same time `!args2` is tuple separation, splitting its contents as separate arguments to `return` continuation function. Pieces of code of that form can be connected together in a chain:

```
code1 . !args1 (!args2)
code2 . !args2 (!args3)
code3 . !args3 (!args4) ...
```

This piece of code effectively performs all the instructions within the `code` functions. The actual body of each `code` function is provided by the DSL creator. They can freeerly access the contents of the `args1` tuple and form any `args2` tuple of their choosing. It is up to them to ensure that the code pieces are assembled in such a way that the arguments are correctly passed between them, i.e. arity of `code2` corresponds to what `code1` is producing.

It remains challenging to:

- Automate the process of creating and connecting code.
- Allow code to be incrementally glued together, producing bigger code pieces.
- Ensure that the above pattern introduces no overhead.

This is achieved with what we name *builder functions*. The most basic builder, without staging for now, is given in Listing 4.18. It consists of 3 layers of functions:

- The blue is a *builder function*. It accepts arbitrary code, represented in a function `code` and creates a green *fragment function*. The fragment is immediately returned through `return` continuation.
- The green *fragment function* represents a fragment of the program we constructed with a yet to be specified continuation `next`.

```
let build
  (code, return) { return .                    builder
    (next, return) { return .              fragment
      (!args)                              body
        code . !args (!args2) next . !args2
    }
  }
```

Listing 4.18: *The basic builder function (blue) without staging. When invoked with a specified user code, it returns the code encapsulated in a fragment function (green). The fragment awaits for a continuation* `next` *to continue the user code (brown)*

- The brown *body function* forms a fragment of a program. It contains the code specified by the DSL developer, followed by the continuation `next`.

The builder calls its continuation `return` with a newly created fragment function. By invoking the fragment with a specified continuation `next` we create a body. The body takes arbitrary many parameters and passes them to user-defined `code`. Therefore, in simple terms, the complete user program is created by combining:

$$\text{program} = \text{builder(code)} + \text{fragment(next)} + \text{body(args)}$$

Builder functions can be invoked directly by the user, or as a part of a semantic action. This however just creates a series of fragments without any flow or coherence. Two or more fragment functions can be combined together by *glue functions* forming a single, bigger fragment.

The simplest glue example, in Listing 4.19, puts two fragment functions, one after another: It takes two fragments `Fprev` and `Fnext`, concatenates them within its new body and produces a new single fragment function. Fragments are invoked in the reverse order: `Fnext` is invoked first, with an unknown continuation `next`, and producing the body `body_next`. The `body_next` is then treated as a continuation for the previous fragment `Fprev`. This way the produced `body_glued` contains the code of `Fprev` and `Fnext`, in that order.

The basic builders and glue functions as presented here assume a single continuation chain. Sometimes however, there are different case needed. The code may require no continuation when it represents an exit or a continuation call on its own. The code may require multiple continuations or follow paths when it represents a branching construct. Finally, multiple code fragments may continue into the same followup, for example when representing a branch convergence point. Such convergence is implicit in CPS programming, but it may be convenient to distinguish such a case when using builders to describe a more traditional non-CPS language.

As branching is a bit more involved topic we discuss it later in Section 4.3.3. Building a fragment function without continuation, and gluing it to a previous fragment is easier, and can be achieved with functions in Listing 4.20.

With these tools we are now able to present a simple use case. Consider the `MinusDiv` language we were using in Section 4.2 that we repeat in Listing 4.21.

```
let glue
  (Fprev, Fnext, return) { return .
    (next, return) {
      Fnext . next (body_next)
      Fprev . body_next (body_glued)
      return . body_glued
    }
  }
```

Listing 4.19: *The basic glue function without staging. It invokes fragment functions `Fnext` and `Fprev`, putting the body of the former after the latter. The result is encapsulated within a new fragment function.*

```
let buildEnd
  (code, return) { return .
    (return) { return .
      (!args)
        code . !args
    }
  }
let glueEnd
  (Fprev, Fnext, return) { return .
    (return) {
      Fnext . (body_next)
      Fprev . body_next (body_glued)
      return . body_glued
    }
  }
```

Listing 4.20: *Builder and glue function for ending a sequence of fragments. Fragment functions produced by these do not expect any further continuations.*

```
grammar MinusDiv {
token Number [[:digit:]]+;
Program       ::= Starting Expr Ending Finalize;
Expr->(val)   ::= Diff;
Diff->(val)   ::= lassoc<Quot,"-", Operator(-)>;
Quot->(val)   ::= lassoc<Value,"/", Operator(/)>;
Value->(val)  ::= "(" Expr ")";
Value->(val)  ::= Number->(str) Number;
}
```

Listing 4.21: *MinusDiv language that we defined in Listing 4.15. We now focus on supplying actions for this grammar. The actions are highlighted in blue. These are not part of LangDSL, but will be replaced by an actual code in this section.*

We want the language to parse an expression and build a program P piece by piece. When P is invoked, it should perform the arithmetic operations and return a single integer as a result.

We create P from pieces, each being a lambda function of the form `(numbers, end, cont){ ... }`, where

- `numbers` is a list of numbers that are parsed. We represent the list as a recursive tuple, that is:
  - `[]` — an empty tuple, representing an empty list.
  - `[N,numbers]` — a pair containing a single number N and another tuple representing the remainder of the list.

  It is a common way of representing lists in functional languages.
- `end` is an actual parameter of P. It is an ending continuation, telling the program what to do after P finishes its execution.
- `cont` is a parameter accepting a continuation `next` provided by the fragment function, which becomes the next piece of our program.

We create P with five semantic actions:

Starting When starting an expression, we create a new P function represented as a lambda `(end, cont)`:

```
build .
  (end, cont) {
    cont . [] end
  }
(F) ...
```

The lambda function `(end, cont)...` is bound to the argument `code` and is invoked within the body in Listing 4.18. This substitutes:

$$\texttt{end} \mapsto \texttt{!args}$$
$$\texttt{cont} \mapsto \texttt{(!args2)\{next . !args2\}}$$

The lambda `(end, cont)...` at the moment does nothing besides calling its continuation with an empty numbers list and `end`.

Number When a number N is read, it is added to the `numbers` tuple on the left, at position 0:

```
build .
  (numbers, end, cont) {
    cont . [N,numbers] end
  }
(Fnext)
glue. F Fnext (F)
```

The `numbers` list arity increases by one.

Operator When a minus or divide operator is read and its right argument is accepted the computation action should be performed. We take two first elements of `numbers`, in the reversed order, and perform the operation on them:

```
build .
  (numbers, end, cont) {  // numbers = [R, [L, rest]]
    numbers[0] . (R)
    numbers[1] . (second)
    second[0] . (L)
    second[1] . (rest)
    L-R . (result)
    cont . [result,rest] end result
  }
(Fnext)
glue. F Fnext (F)
```

Ending Finally, when the whole expression is computed, we finish the function:

```
buildEnd .
  (numbers, end) {
    numbers[0] . (result)
    end . result
  }
(Fend)
glueEnd . F Fend (F)
```

At that point the list must be reduced to a single last value. The parameter `end` is the ending continuation provided by the DSL developer that was defined in the starting fragment and has been carried through all fragments in between.

Finalize After gluing the end, we obtain the fragment `F` that encapsulates a complete program `P`. In order to peel off the encapsulation, one simply invokes `F` and obtains `P` in the returning continuation:

```
F . (P) ...
```

To see how these actions interact, consider an example input `"1-4/2"`. Given the position of actions in the grammar, particular tokens are processed in the postfix order: `S 1 4 2 / - E F`, with `S` indicating the starting and `E`, `F` ending/finalize actions. Parsing such an input executes code equivalent to the one shown in Listing 4.22. Note however, that each build call is actually originating from different locations within the grammar.

In the above example building is interleaved with gluing. This pattern is not a must: Gluing can happen at any later moment. This allows fragments to be rearranged, or used multiple times if needed.

By executing the builders in the example, we obtain a program P that is equivalent to:

```
(end, cont) {
  4/2 . (result1)
  1-result1 . (result2)
  end . result2
}
```

We say *equivalent* because what we actually obtain is a tree of fragment function calls. Only by performing a beta reduction on all the fragment function calls, and performing partial evaluation of their bodies, the above code for P is obtained. In the next section, we show how to achieve that through dynamic staging, without any overhead coming from gluing and building.

4.3.2 Removing the Overhead

The presented solution shows how the code can be built piece by piece, but the produced program is not very efficient. The program contains bodies of all builder functions and the calls to `code` functions specified by the DSL developer. Transitioning from one piece to another and managing the `!args` argument is performed at run time, when the generated program is executed. With the help of dynamic staging, as presented in Section 4.1, we can force the interpreter to perform the necessary reductions early.

For the purpose of our discussion we define two staging chains (defined in Section 4.1.7).

- The build-time staging chain represents work that should be done during building. It is represented by the `bt` stage variables. The chain is triggered at the final step of the build process. In this chain, all calls to `code` are resolved, packing and unpacking the `!args` parameters. Afterwards, values coming from one `code` function are used directly in other `code` functions.
- The function-time staging chain represents the moment when the produced program P is actually called. It is controlled by `ft` stage variables.

```
build .                                                    starting
  (end, cont) {
    cont . [] end
  }
  (F1)
build .                                                    number
  (numbers, end, cont) {    //numbers = []
    cont . [1, numbers] end
  }
  (Fnext)
glue . F1 Fnext (F2)
build .                                                    number
  (numbers, end, cont) {    //numbers = [1, []]
    cont . [4, numbers] end
  }
  (Fnext)
glue . F2 Fnext (F3)
build .                                                    number
  (numbers, end, cont) {    //numbers = [4, [1, []]]
    cont . [2, numbers] end
  }
  (Fnext)
glue . F3 Fnext (F4)
build .                                                    operator
  (numbers, end, cont) {    //numbers = [2, [4, [1, []]]]
    numbers[0] . (R)        //R = 2
    numbers[1] . (second)
    second[0] . (L)         //L = 4
    second[1] . (rest)
    L/R . (result)
    cont . [result,rest] end result
  }
  (Fnext)
glue . F4 Fnext (F5)
build .                                                    operator
  (numbers, end, cont) {    //numbers = [2, [1, []]]
    numbers[0] . (R)        //R = 2
    numbers[1] . (second)
    second[0] . (L)         //L = 1
    second[1] . (rest)
    L-R . (result)
    cont . [result,rest] end result
  }
  (Fnext)
glue . F5 Fnext (F6)
buildEnd .                                                 ending
  (numbers, end) {          //numbers = [-1, []]
    numbers[0] . (result)
    end . result
  }
  (Fend)
glueEnd . F6 Fend (F7)
F7 . (P)
```

Listing 4.22: *All actions that are executed when evaluating the input* "1-4/2".

Both build-time and function-time staging chains are available to the DSL developer when they specify the `code` functions. It is up to them to decide which part of their code should be executed when.

Consider an updated builder function in Listing 4.23. The functionality remains the same: The builder encapsulates a fragment, which encapsulates the body function containing user code. This time however, the body function uses the fragment chaining pattern twice for the build-time and function-time staging chains. In addition we stage the call to the continuation `next` upon the parameter `next` itself. This means, that as soon as `next` is a concrete function rather than a symbolic name, it is invoked.

The glue function in Listing 4.24 is updated as well. As soon as `glue` is invoked with concrete `Fnext` and `Fprev` fragment functions, these are invoked. This peels off the fragment function encapsulation around the body functions. Bodies are connected and placed in the context of the `(next, return)` lambda function. When completed, glue returns a new fragment function with no nested fragments inside — just the body functions staged upon `bt`.

In addition to `build` and `glue` we also need a `finalize` function, as the one in Listing 4.25. This is to be called on a fragment containing a complete program. It triggers the build-time chain, calling all the `code` functions in sequence, and leaving only those pieces that are actually staged upon the `ft` chain.

Notice that in this discussion we avoid the terms “compile-time” and “run-time” on purpose. The mechanism we describe does not put any requirements on when the generated program is actually called. The program may be used at compile-time, parse-time, as a part of another, more complex builder or at any other moment of interpretation. The mechanism also does not prevent the DSL designer from incorporating more staging variables within its program. These may be hidden within `!args` or be captured in some other way.

In order to see how to use the staged builders, let us revisit the MinusDiv actions again. As before, we use 5 semantic actions:

Starting As before, when starting an expression, we create a new P function. This time however, we add the additional function-time staging variable `ft`:

```
build .
  (ft, end, cont) {
```

```
let build
  (code, return) { return .                                   builder
    (next, return) { return .                            fragment
      (bt, ft, !args)                                    body
        @bt: code . ft !args (ft, !args2)[bt]
        @next: next . bt ft !args2
    }
  }
```

Listing 4.23: *The basic builder function (blue) with staging. At build-time (bt)* `!args` *are properly expanded and calls to user code are resolved. Only pieces within user code staged on function-time (ft) remain, leaving no trace of this building process.*

```
let glue
  (Fprev, Fnext, return)[g] { return .
    (next, return) {
      @g: Fnext . next (body_next)
          Fprev . body_next (body_glued)
      @return: return body_glued
    }
  }
```

Listing 4.24: *The basic glue function with staging. Fragment functions* `Fnext` *and* `Fprev` *are invoked immediately, under the* `(next, return)` *lambda. The result of the reduction is encapsulated within a new fragment function.*

```
let finalize
  (F, return) {
    F . (P)[bt]
    return .
      (!args)[ft] {
        @bt: P . bt ft !args
      }
  }
```

Listing 4.25: *The* `finalize` *function, where build-time and function-time chains begin. The program P is invoked early, with yet-unspecified* `ft` *(equivalent to $\perp$) and unknown* `!args`*. All instructions within P that are staged upon* `ft` *remain as code and are spliced together as the body of the lambda* `(!args)[ft]`*....*

```
    cont . ft [] end
  }
(F)
```

This additional parameter is removed at the very last step of the building process, in the Finalize action.

Number A number N is added to the `numbers` tuple. No instruction is staged upon `ft`, thus this code piece produces no code at function-time.

```
build .
  (ft, numbers, end, cont) {
    cont . ft [N,numbers] end
  }
(Fnext)
glue. F Fnext (F)
```

Operator As before, when an operator is encountered, the operands are extracted from the `numbers` list. All tuple operations are performed when the code is invoked from within the fragment function — that is, at build-time `bt` (the code invocation in Listing 4.23). Only the actual operation is performed at function-time.

```
build .
  (ft, numbers, end, cont) {  // numbers = [R, [L, rest]]
    numbers[0] . (R)
    numbers[1] . (second)
    second[0] . (L)
    second[1] . (rest)[bt]
    @ft: L-R . (result)[ft]
    @bt: cont . ft [result,rest] end result
  }
(Fnext)
glue. F Fnext (F)
```

Ending When the whole expression is computed, the function finishes at function-time:

```
buildEnd .
  (ft, numbers, end) {
    numbers[0] . (result)
    @ft: end . result
  }
(Fend)
glueEnd . F Fend (F)
```

Finalize After gluing the ending, we obtain the fragment `F` that encapsulates a complete program. We use the `finalize` function to obtain an actual program, which no longer contains any overhead.

```
finalize . F (P)
```

Using the above semantic actions with the input `"1-4/2"` gives us exactly the desired result:

```
(end, cont)[ft] {
  @ft: 4/2 . (result1)[ft]
  @ft: 1-result1 . (result2)[ft]
  @ft: end . result2
}
```

A detailed simulation of all the interpreter steps leading to this result is given in the Appendix A.

4.3.3 Branching

So far we focused on the simplest fragment functions where the code uses only a single continuation. In short, we say that these fragment functions are of arity 1. However, if more continuations are needed in the code we need to adjust the builder as well. For example, if we were using a branch instruction in the code we build, we would be needing two continuations.

A naive approach would be to create a new builder suited for branching, as shown in Listing 4.26. The user `code` accepts not one but two continuations and uses both in some way.

The produced fragment however is different: it expects not one but two `Fnext` functions, i.e. it is of arity 2. This makes the fragment incompatible with the simple glue function we used so far. As we incrementally build our code, we need to attach other fragment functions to this one. A straightforward approach

```
let build2
  (code, return) { return .
    (next1, next2, return) { return .
      (bt, ft, !args)
        @bt: code . ft !args
          (ft, !args2)[bt] { @next1: next1 . bt ft !args2 }
          (ft, !args2)[bt] { @next2: next2 . bt ft !args2 }
    }
  }
```

Listing 4.26: *A builder for a code which takes two continuations.*

```
let glue2_1
  (Fprev2, Fnext1, idx, return)[g] { return .
    (next1, next2, return) {
      @g: idx==0 . (first)
          ifelse . first
            (endif) { Fnext1 . next1 (body1)
                      Fprev2 . body1 next2 endif  }
            (endif) { Fnext1 . next2 (body2)
                      Fprev2 . next1 body2 endif  }
      (body)
      @return: return . body
    }
  }
let glue1_2
  (Fprev1, Fnext2, return)[g] { return .
    (next1, next2, return) {
      @g: Fnext2 . next1 next2 (body1)
          Fprev1 . body1 (body)
      @return: return . body
    }
  }
let glue2_0
  (Fprev2, Fnext0, idx, return)[g] { return .
    (next, return) {
      @g: Fnext0 . (body)
          idx==0 . (first)
          ifelse . first
            (endif) { Fprev2 . body next endif }
            (endif) { Fprev2 . next body endif }
          (body)
      @return: return . body
    }
  }
```

Listing 4.27: *A naive approach to gluing: a set of glue functions needed for connecting fragment functions of different arity. The number after the names* Fprev *and* Fnext *indicates the expected arity of the argument fragment function.*

```
let build
  (arity, code, return) { return .
    (neli, return) {
      for . neli [] 0 arity
        (neli, lambda_tuple, i, break, continue)[g] {
          @neli: unfold(neli) . (next, neli_remain)
          @g:
          let lambda_i = (ft, !args2)[bt] {
            @next: next . bt ft !args2
          }
          concatenate(lambda_tuple, [lambda_i]) . (lambda_tuple)
          continue . neli_remain lambda_tuple
        }
      (neli_remain, lambda_tuple) return .
        [ (bt, ft, !args) { @bt: code . ft !args !lambda_tuple }
          , neli_remain ]
    }
  }
let glue
  (Fprev, Fnext, return)[g] { return .
    (neli1, return) {
      @g: Fnext . neli1 (neli2)
          Fprev . neli2 (neli3)
      @return: return . neli3
    }
  }
```

Listing 4.28: *A generic* `build` *and* `glue`

would be to create a set of `glue` functions to connect our arity-2 fragment in various combinations. For example, in Listing 4.27 we include:

- `glue2_1` allows to connect an arity-1 fragment to either of the branches of our arity-2 fragment function. The parameter `idx` signifies which branch we connect to. The result is a new arity-2 fragment.
- `glue1_2` allows to put our arity-2 fragment as a follow-up of another arity-1 fragment, producing a new arity-2 fragment function.
- `glue2_0` sets an arity-0 fragment as a continuation of one of the branches of arity-2 fragment. When an arity-0 fragment is attached to one of the branches, it ends this branch. The result is a new fragment with arity 1.

The situation gets more complicated when the number of branches increases. When code needs even more continuations or when the branches are nested, we may produce fragments with arbitrarily high arity. For that reason we need a more generic solution to builders, so that the DSL developer is not confused by many different versions of those functions.

We want a single generic `glue` function that can handle any number of `next` continuations. To that end, we put all these continuations in a single structure and pass that between the builders. Each fragment function can consume as many `next` continuations as needed and provide any number back into the struct as well. Only the builders need to explicitly specify the arity.

In our solution, we keep the `next` continuations in a list `neli` (**ne**xt **li**st). This

is similar to how we kept numbers in our MinusDiv language. The list is defined in a typical functional way, with each element being either an empty tuple [] signifying the end of the list, or a pair containing a value and the remainder of the list [value, list].

The build function takes an integer argument arity that must be provided by the DSL developer. The value specifies the number of continuations needed by the user code that are being constructed incrementally with the fragments. The builder then unfolds arity number of elements from the list and builds a regular lambda_tuple tuple containing all the continuations. With it, the call to user-defined code is possible. The new resulting code fragment is put on top of the remaining neli and returned.

With this setting, the glue function is as simple as its original arity-1 version in Listing 4.24. The only difference is that it now passes lists between the fragment functions, and not a singular next/body function. In this version we do not give the parameter idx to let the user specify to which branch to attach. Such index is hard to track in a multi-branched scenario. Instead, we require that the follow-up fragments are provided in the left-to-right order. A fragment can be glued as a part of the second branch only when the first branch is finished (i.e. is sealed with an arity-0 fragment).

We do not think that this limitation is a problem. It keeps the function usage simple. In most scenarios, parsers read the code in a depth-first left-to-right order and that specific order can be easily incorporated using these generalized builders. If this sequence cannot be maintained, fragment functions that build up each branch can be glued separately, before attaching them to the high-arity fragment where the branching occurs. The DSL designer may choose to glue pieces to construct a complete branch, expressing it as a single fragment and only then glue it to the code that actually creates the branch.

The generalized builder also needs an explanation of its staging. In general, the whole fragment is executed when invoked in a glue function. The arity for loop is unrolled and the continuation lambda_i functions are created. The only exception is the the unfolding of the list, that is delayed until the value neli is known. It is important to note that this does not mean that we need the *whole* list to be known. The contents of the known neli pair may again be symbolic. All needed next values can be retrieved as soon as they are available, even when working on an incomplete list.

It suffices that the neli pair is provided, even if its elements next and neli_remain remain as symbolic values. This means, all next operations can be resolved even when operating on the incomplete list.

In fact, neither build nor glue ever specify the list end. This reflects the fact, that any fragment, even the one representing the whole program, may be used as a piece to build something bigger. The end is provided only in the finalize function (Listing 4.29) which seals the list with an empty tuple, and makes the program ready to be executed.

In the later examples whenever we use build, glue, and finalize we refer to these generalized versions of functions. Their behavior for a non-branched

```
let finalize
  (F, return) {
    F . [] (list)
    unfold(list) . (P, empty)[bt]  //empty is []
    return .
      (!args)[ft] {
        @bt: P . bt ft !args
      }
  }
```

Listing 4.29: *The `finalize` function adjusted to work with the generalized `build` and `glue`. This is the only place where we specify the end of the `neli` list.*

fragments is exactly the same as their simple counterparts we introduced earlier. These three functions: `build`, `glue`, and `finalize` suffice to construct a complete program.

4.3.4 Converging

As we discuss in Section 4.1.2 and Section 6.1.1, branches in CPS do not converge explicitly. Typically, such convergence is achieved implicitly by having both branches call the same continuation at some point. For example, consider a DeepCPS definition of `ifelse` given in Listing 6.1, which we repeat here:

```
let ifelse(cond, tb, fb, endif) {
  if . cond
    () { tb . endif }
    () { fb . endif }
}
```

The intrinsic `if` creates a branch that by default never converges. However, both `tb` and `fb` branches take the same `endif` continuation as an argument. By calling it, both branches converge on the same path.

When building branching code, one may try the similar solution. One would create a fragment function `endif` representing the rest of the program. Then, one would provide the `endif` twice, as a continuation of both branches.

Recall however, that fragments are invoked during gluing, at construction time. As a result, `endif` fragment would be invoked twice, replicating the code that is produced. Overall, the created program behaves as intended, but the code becomes exponentially large with the number of branches. If a loop was created, this strategy could infinitely unroll the loop at code building time. Such behavior is required formally (Section 4.1.1) but it is also the actual behavior of the interpreter as described in Section 5.1.2

The key observation is that the converging continuation `endif` is effectively invoked at build-time, before the function-time condition is resolved. Even when using dynamic staging without builders at all, such a thing should be avoided unless the code replication is the intended behavior, e.g. when unrolling a loop.

In our case we want the actual call to the converging continuation to be performed when executing the program that we produce. It can be achieved in the following way:

```
build . 3
  (ft, end, tb, fb, converge)[bt] {
    @ft: ifelse . <condition>
      (endif)[ft] { @bt: tb . ft endif }
      (endif)[ft] { @bt: fb . ft endif }
      ()[ft] { @bt: converge . ft end }
  }
  (F_tb_fb_conv)
build . 0
  (ft, endif) {
    <true branch body> . ()
    @ft: endif .
  }
  (F_tb)
glue . F_tb_fb_conv F_tb (F_fb_conv)
build . 0
  (@ft: endif) {
    <false branch body> . ()
    @ft: endif .
  }
  (F_fb)
glue . F_fb_conv F_fb (F_conv)
... //(F_conv) is now a normal, non-branched fragment
```

Listing 4.30: *Building a branch that converges using the generalized* `build` *and* `glue`

- Specify the converging continuation as a separate, *third* continuation of the branching fragment.
- Pass the converging continuation as an actual parameter into the true- and false- branches.
- From the perspective of each branch, treat the converging continuation as an `end` parameter that ends the branch.
- Finish building each branch with an arity-0 fragment and its body calling the `end`.

The need for a third continuation may be surprising at first. Notice however that we are building a construct more similar to `ifelse` given above rather than the non-converging, generic `if`. The `ifelse` takes *three* continuations instead of two, and that is the arity of branching that we need.

In the example Listing 4.30 we show this solution. The first fragment we build, `F_tb_fb_conv)` contains the `ifelse` call within the code we produce. The fragment requires three continuations: `tb`, `fb` and `converge`. Each of the continuations is called at the build-time stage, before the condition is actually resolved. This is needed in order to actually build the code.

The branches `tb` and `fb` are created separately in the following `build` calls. In our example, a single build is used for a complete branch, but it could be created incrementally from multiple fragments, as if a separate program was constructed. Each branch ends with a function-time call to `endif`. Notice, that `endif` is a regular code parameter originating from `ifelse`, and *not* a code reference produced by a fragment function.

```
let code
  (!args_1, loop_body) {
    fix rec (!args_n) {
      loop_body . !args_n rec
    }
    in
      rec . !args_1
  }
```

Listing 4.31: *A code for a simple loop to be used with stageless builders*

```
let code (ft, !args_1, loop_body) {
  rec-static(args_1) (sargs_1)
  rec-dynamic(args_1) (dargs_1)[bt]
  @ft:
  fix rec (!dargs_n)[ft] {
    @bt:
    concatenate(sargs_1,dargs_n,[rec]) . (dargs_n)
    loop_body . ft !dargs_n
  }
  in
    rec . !dargs_1
  }
```

Listing 4.32: *A code for a simple loop to be used with builders with staging.*

As soon as each branch is complete, it is glued to the `ifelse` branching fragment. Each time the arity of the fragment is reduced. At the end of the snippet, the `F_conv` function can be used as a simple fragment of arity 1. New fragments can be connected to the front or back of it, producing a bigger construct.

4.3.5 Loops and Recursion

Loops and recursion pose an additional challenge in DeepCPS. They require a fix-point combinator. This, however, can be provided within the user code, and requires no change in the builders. The simplest, potentially infinite loop without staging is given by Listing 4.31. This `code` function is designed to be used with an unstaged `build`. All we do within its body is to invoke a recursive function `rec`, which in turn invokes the user-defined `loop_body`. The last parameter provided to `loop_body` is again the `rec`, allowing the user-defined function to loop. The `code` provides no direct means to break the loop, but a breaking continuation may be available among `!args_1` parameters.

When introducing staging or building, attention has to be made to recursive and looping functions. As we explained in last section, these should not be called unconditionally at build-time as it may lead to infinite unroll. Instead, in most cases both the definition and call should appear at function-time.

The change in Listing 4.32 is a little bit more involved though. It must be decided which arguments remain static with respect to the loop, and which change with each iteration. For the purpose of the example we assume that the recurring argument `!args` contains both and that it can be split into a static (`sargs`) and a dynamic (`dargs`) part. The static part is directly captured by the recursive function and does not change with each loop iteration. The dynamic part is passed as a tuple argument `dargs`.

```
let code (ft, !args, f1_body, f2_body, in_body) {
  rec-static(args) (sargs)[bt]
  @ft:
  fix
    f1 = (!fargs)[ft] {
      @bt:
      cont1 . ft !sargs !fargs f1 f2
    }
    f2 = (!fargs)[ft] {
      @bt:
      cont2 . ft !sargs !fargs f1 f2
    }
  in
    let [ft];
    @bt:
    in_body . ft !args f1 f2
}
```

Listing 4.33: *Defining two mutually recursive functions with builders and staging.*

Consider the following, more practical example: In Listing 4.33 we have a series of mutually-recursive functions. Assuming the DSL developer knows upfront how many functions are there, this can be built using the polyvariadic **`fix`** of DeepCPS.

As before, the arguments `!args` provided to the initial `code` are split into static and dynamic parts. Within the fix we define two functions, named `f1` and `f2`. Each has an unspecified body represented as a continuation `f1_body` and `f2_body`, which are provided later on through the generic `glue` function.

Both functions take a generic `!fargs` tuple as an argument list, but if the actual arguments are known to the DSL developer those can be specified explicitly. Function arguments are passed to the continuation, together with the static part of `!args`. Unlike the previous example, the dynamic part of `!args` does not have to reach the functions `f1`, `f2`. The functions can have different parameters.

In addition to `!fargs` and `!sargs` the continuations take the functions themselves as arguments. This way, from within the bodies of these functions, `f1` and `f2` can be called recursively.

Finally, at the very end of the fix construct, an additional continuation `in_body` is used. This represents the normal code that follows up the function declarations, within which those functions can be referenced for the first call.

Unfortunately, the above piece of code cannot be abstracted out easily with the respect of the number of functions. This is because the DeepCPS fix construct is not flexible in that respect. It cannot be built in pieces or created in a generic way. In practice, it is still possible to achieve that through careful manipulation of ManyDSL nodes, which we show in Section 6.4.3.

4.4 Conclusion

In this chapter we have shown the novel, dynamic approach to staging. We have explained how new language grammars are defined in LangDSL. Finally,

we explained how actions, written in staged DeepCPS, can be used to construct new code free from any overhead coming from this building process.

When explaining the language construction we had a running example of a small `MinusDiv` language. Let us, once again, bring that example, in its most complete form in Listing 4.34.

For the readability sake, we have defined all the actions before the grammar, using only their names within the language definition. The logic of the produced code has been explained in Section 4.3.1: Whenever we parse a number, we add it to `numbers` list, while each operator takes its two arguments and puts the result there.

As explained in Section 4.3.2 we use two staging chains: build-time (`bt`) and function-time (`ft`). Code staged upon build-time is executed during the construction. Function-time chain begins in `finalize` function (Listing 4.25) and represents all computation that should be performed when the function is invoked. In our case, the only function-time computation is the actual subraction and division. Everything else is resolved during building.

The order of action invocation is controlled by the grammar of the language. As explained in Section 4.2.5 we use a grammar function `lassoc` to define a left-associative operators `-` and `/`.

Because throughout the language we operate on a single fragment function `F`, incrementally adding code to it, that is the only parameter that is passed around between the terms of the grammar.

The `MinusDiv` language is intentionally minimal, explaining the underlying mechanisms. In Section 6.4 we move upwards, showing how more complex structures and domain-specific optimization can be defined. Finally, in Section 6.5 we show a more complex array-processing DSL implemented using the techniques shown here. In those examples we focus only on the work of a DSL developer – we no longer inspect or alter LangDSL or the builder functions explained in this section.

```
let starting(return) {
  build . 1 (ft, end, cont) {
              cont . ft [] end
            } return
}
let number(F, str, return) {
  str2int . str (N)
  build . 1 (ft, numbers, end, cont) {
              cont . ft [N, numbers] end
            } (Fnext)
  glue . F Fnext return
}
let operatorDiff(F, return) {
  build . 1 (ft, numbers, end, cont) {
              $proj . 0 numbers (R) $proj . 1 numbers (second)
              $proj . 0 second (L) $proj . 1 second (rest)[bt]
              @ft: L-R . (result)[ft]
              @bt: cont . ft [result, rest] end
            } (Fnext)
  glue . F Fnext return
}
let operatorQuot(F, return) {
  build . 1 (ft, numbers, end, cont) {
              $proj . 0 numbers (R) $proj . 1 numbers (second)
              $proj . 0 second (L) $proj . 1 second (rest)[bt]
              @ft: L/R . (result)[ft]
              @bt: cont . ft [result, rest] end
            } (Fnext)
  glue . F Fnext return
}
let ending(F, return) {
  build . 0 (ft, numbers, end) {
              $proj . 0 numers (result)
              @ft: end . result
            } (Fnext)
  glue . F Fnext return
}
function lassoc<elem, op, action> {
  (F)->N->(F) ::= (F)->elem->(F) (F)->NCont->(F);
  (F)->NCont->(F) ::= epsilon;
  (F)->NCont->(F) ::= op (F)->elem->(F) (F)->action->(F)
      (F)->NCont->(F);
  return N;
}
grammar MinusDiv {
token Number [[:digit:]]+;
Program->(P)  ::= starting->(F) Expr (F)->ending->(F)
    (F)->finalize->(P);
(F)->Expr->(F)  ::= Diff;
(F)->Diff->(F)  ::= lassoc<Quot,"-", (F)->operatorDiff->(F)>;
(F)->Quot->(F)  ::= lassoc<Quot,"-", (F)->operatorQuot->(F)>;
(F)->Value->(F) ::= "(" Expr ")";
(F)->Value->(F) ::= Number->(str) (F,str)->number->(F) ;
}
```

Listing 4.34: *The complete version of `MinusDiv` language, which for given input produces a program that – when invoked – computes the result.*

Chapter 5

Implementation

In this chapter we focus on the implementation details of ManyDSL. We explain the algorithms we use, practical obstacles that were encountered and how we have chosen to resolve them.

The whole project is divided into 5 main modules, each depending on the previous ones:

ManyDSL Core — implements the ManyDSL Target Representation and the interpreter operating on it.

Lang module handles language specification and parsing

DeepCPS module specifies the DeepCPS language using Lang.

Executor module provides top level control on file reading and switching between different components of ManyDSL.

Compilator module translates the ManyDSL Target Representation into Thorin and later LLVM.

Finally, we provide a thin Application layer which allows the ManyDSL system to be run by the user.

5.1 ManyDSL Core

The ManyDSL core defines the representation of ManyDSL programs, and provides functionality necessary to interpret and transform it according to CPS and dynamic staging rules. For the purpose of this section let us assume that the whole code is given at once. In Section 5.7 we describe how the interpreter behaves when executing an incomplete program.

5.1.1 Target Representation

All ManyDSL programs are represented in *Target Representation* (TR). It is the representation produced by the parsers and is used directly for interpretation of the code. There is no additional intermediate representation used for other actions, such as code optimization.

The TR is implemented entirely in C++. The main class representing the code structure is `WindNode`. The name "wind" is the internal name for ManyDSL interpreted code.

WindNode

ManyDSL programs in TR are represented as a graph. All graph nodes are objects of the `WindNode` class.

Graph edges are represented as pointers within nodes. There are two categories of edges: primary and secondary. Primary edges form a tree and define the structure of the program, i.e. which program terms are children of other terms. Secondary edges can connect any nodes without constraints, defining a reference for variables and staging.

Throughout this chapter, whenever we refer to a `child` or `parent` of a given node, we follow the primary edge. Secondary edges are denoted as `references`.

All outgoing edges, both primary and secondary are kept in a single array within the `WindNode` object. The `WindNode` class provides methods to attach new edges, reconnect elements between two branches of the program tree, or even clone a whole subtree of a given node if necessary. The class also handles garbage collection, which we discuss later in this section.

Every edge connecting two nodes $X \to Y$, has a corresponding `backedge` connecting $Y \to X$. As the actual edge is stored in the source node X, the backedge is stored in the target Y. This way, every node "knows" where it is being used. A backedge of a primary edge is referred as *parent edge* and points to the parent node. Since primary edges form a tree, each node has at most one parent. It is possible, however, for a node to be *dangling*, without a parent.

A plain `WindNode` object holds no semantic meaning; semantics is provided by classes that derive from `WindNode`.

Simple Values

A `Value`, derivative of `WindNode`, is any term that is the first-class citizen of the language. ManyDSL supports, as derivatives of `Value`:

- Simple values, stored in `PrimitiveValue`. It is a single value, which is an integer (32-bit or 64-bit, signed or unsigned), a floating-point value (32-bit or 64-bit) a Boolean or a null-terminated string.
- Tuples, consisting of arbitrary number of member values.

- Raw pointers and strong pointers. The former can point to any kind of data. Strong pointers can be used when DeepCPS code itself operates on ManyDSL nodes, not necessarily created as a part of the current program code.
- Type values (`Kind` class and derivatives). While ManyDSL provides no type checking, it implements a basic type annotations allowing the user program to use them.
- Intrinsic values (functions). These provide interface to functionality natively supported by ManyDSL. This includes arithmetic and Boolean operations, comparison, string manipulation, conversion, importing C libraries, exiting and halting a program. It also provides simple debugging tools, such as printing values in the console or printing the underlying ManyDSL structure.

Closure and Parameters

The `Closure` class (derivative of a `Value`) represents a lambda function. Since references within the function may point outside of its scope, such function actually acts as a closure: free variables may be bound and act as an environment for the closure.

A `Closure` has at least one child: its body. It may also have a staging *payload* (which we explain below), and arbitrary many parameters. The `Payload` object can be omitted only when using natural staging.

Each `Parameter`, also derivative of `Value` is a simple node acting as a proxy. Initially it contains merely the parameter name for debugging purposes. Similarly to a closure, it may also have a staging payload. Parameter also holds a reference to a value when the parameter is bound. We discuss value-parameter binding in detail in Section 5.1.2.

Parameters keep track of all uses of it, and know its use count. Every reference to a given parameter causes the parameter to hold a *backedge* to the using node.

Action

The `Action` class is a `WindNode` that is not a `Value` and represents a single computation. Most commonly, it is an application: Calling a function with a list of arguments. However, it may also represent a fix node, branch, memory operation, or program termination. Some of these representations do not require any arguments or have no function to be called.

The function and its arguments can be either actual children of the action or mere parameter references. If an argument (or callee function) is an actual child, it means that the value is `inlined` at the point of its usage.

In addition, each `Action` may have a `StageEvaluator` child. If present, it represents the staging expression that formally precedes every application and facilitates its evaluation. The `StageEvaluator` object can be omitted, for optimization reasons, only when using natural staging.

A DeepCPS **let** x y construct is translated to an `Action` node. Such node takes one argument: y and invokes an anonymous `Closure` λ. The λ has one parameter x and its body contains the remaining part of the program.

A DeepCPS **fix** construct looks mostly the same as a **let** described above. The only differences are:

- The `Action` node is flagged as `FIX`, but it has no actual impact on the interpreter.
- The λ can have multiple parameters. Their count and name matches the names defined in the **fix** construct in DeepCPS.
- The parameters of λ can be referred by the arguments of the `FIX` node, which are not in the λ subtree. This unique behavior allows to define recursion.

Staging and Payload

An object of type `StageEvaluator` is a child of every action that has nontrivial staging. It holds the whole boolean expression of the staging and may be evaluated to check if it has become active.

Suppose a term x is used in a staging expression. When that happens, the node x is given an additional child of type `Payload`. Whenever x is used in a staging expression, the `StageEvaluator` object references thy `Payload` of x. Backedges of the `Payload` are used to find all actions that may become active when x becomes equivalent to $\top$. Recall from Section 4.1.3, this may happen if:

- x is a `Closure` object that is invoked.
- x is a `Parameter` object, and a known constant has been asigned to it.

Garbage Collection

ManyDSL performs garbage collection to reclaim memory owned by nodes that are no longer reachable. A template class `sp<Node>` acts as a *strong pointer* to any ManyDSL node of type `Node`. Nodes referenced in this way are reference-counted. The reference count is stored within the `WindNode` class, i.e. implementing the intrusive pointer concept. Weak pointers are synonymous to raw pointers.

The strong pointers are intended to be used in code outside of ManyDSL core control. These act as root nodes for a mark-and-sweep garbage collection. During the garbage collection, the program structure is inspected. All nodes that are reachable from the roots, following both primary and secondary edges, are marked and remain intact. Backedges are ignored in this reachability test. Unreachable nodes are then deleted.

Example

In Figure 5.1 we give a short example how the Target Representation structure actually looks like. This is a part of the staged `pow` function examples we used

Section 4.1.5, namely:

```
fix pow (base, exp, return) ... in
@pow:
let pow72 (base, return) {
  @pow: pow . base 72 (result)
  @result: return . result
}
...
```

Let us explain the structure, step by step. To help navigate, the numbers of our list refer to the objects marked by those numbers.

1. At the top, we have an `Action` representing the `FIX` node. The right subtree is a lambda that we bind to `pow`. The left subtree is a lambda representing the rest of the program, appearing after the **`in`** clause.

2. The **`in`** clause is represented as a lambda (diamond shape) having a single parameter `pow` — the only value defined by this **`fix`**. The body of the lambda is the left child. It is the first action appearing after the **`in`** clause.

3. The first action after the **`fix`** `...` **`in`** is a regular `Action` (square). It represents the statement `@pow:` **`let`** `pow72 (...)... ...` with three children:

   - The first child is the `StageEvaluator` depicted as a hexagon. It holds the Boolean expression of the staging for the parent action. In this particular case, the action is staged upon a single value reference `@pow:`, thus the `StageEvaluator` references the `Payload` (triangle) of the `pow` parameter object.

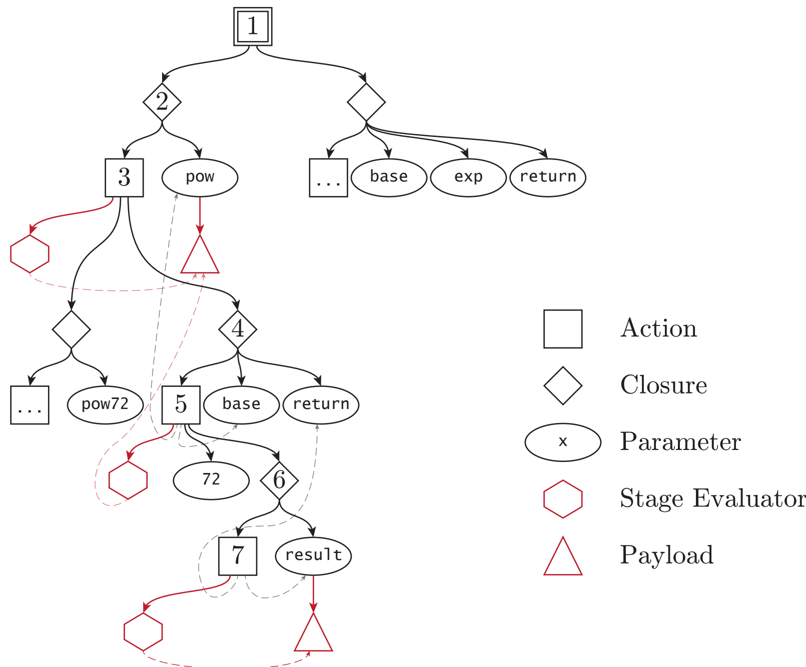


Figure 5.1: *Graph primitives for representing a CPS program.*

- The second child is the **`let`** lambda, containing the rest of the program and the parameter `pow72` that is being defined.
- The third child is the value that `pow72` is being set to.

4. The value that we bind to `pow72` is a `Closure` node with 3 children: a body `Action` and parameters `base` and `return`.

5. The body of the lambda that we bind to `pow72` is an `Action` with 5 children, given in this order:

    - The `StageEvaluator` controlling the staging. As before, it merely references the `Payload` of `pow` parameter.
    - The function that is being invoked. At this moment, it is the symbolic value `pow`. Thus, a secondary edge references the `pow` parameter.
    - The first argument, which is the parameter `base`, referenced through a secondary edge.
    - The second argument, a constant `72`, referenced directly through the primary edge.
    - Third argument — the returning continuation passed to `pow`, given as a lambda and referenced through the primary edge.

6. The returning continuation is a `Closure` object, containing a body and a single parameter `result`.

7. The body of the returning continuation is an action with three children:

    - Since the action is staged, the `StageEvaluator` is provided, which refers the `Payload` of `result` parameter object.
    - The callee is the `return` parameter object.
    - The only argument passed to `return` is the `result` parameter object.

5.1.2 Interpretation

So far, we have explained how the program representation looks like. We now focus on how the representation changes during the program execution. We use the word *interpretation* because the ManyDSL representation explained above is used as-is to perform computation, as opposed to compiling it to some other form prior to execution. However, unlike typical interpreters, ours is *destructive*, i.e. changes the code as it executes. The execution steps closely resemble the actual formal semantic rules explained in Section 4.1.3. For example, calling a lambda function causes the application to be actually removed from the code and replaced by the function body.

Stageless Execution Step

Let us assume for the moment that the interpreter is executing a portion of the code that is not using staging, i.e. only natural staging is used. We have an `Action` A selected to be executed in the current interpretation step. The algorithm does three things, as shown on Figure 5.2:

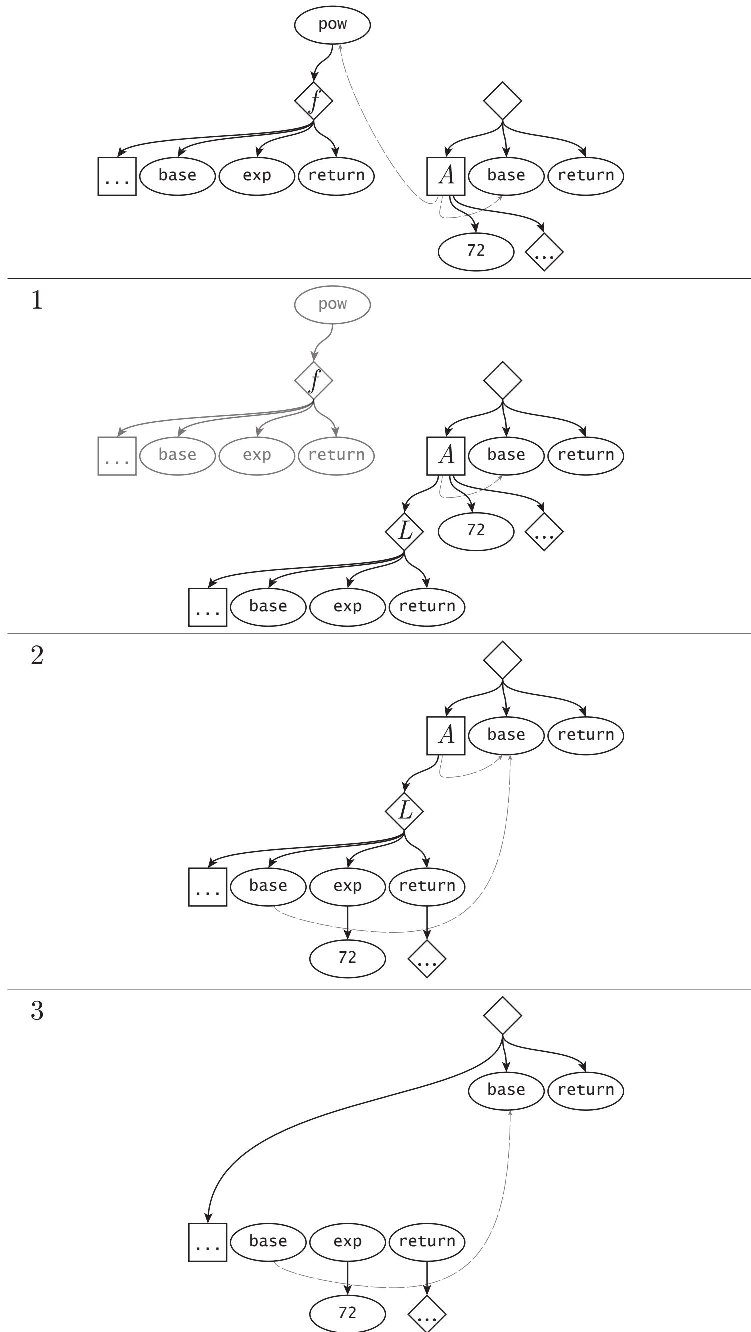


Figure 5.2: *A single interpretation step of an action A. The callee is cloned, arguments are bound to parameters and finally the caller action is replaced by the body of the callee.*

1. If the called function f is referenced through a parameter and is not an actual child of A, then it is cloned: A new lambda function L (`Closure` object) is created as a child of A, replacing the reference. The L is a deep copy of f, all nodes reachable from f through primary edges are copied. Secondary edges within L are updated accordingly: pointing to new nodes if they were cloned in the process, or referencing originals when they are outside of the scope of f.

   If the called function is already a lambda L, an actual child of A, nothing happens in this step.

2. The arguments v_i of the action A are bound to the parameters x_i of the called lambda function L. For each argument v_i we check the type of the edge that connects it with A. If v_i is connected through a secondary edge, then a new reference edge is added, connecting the `Parameter` x_i to v_i. On the other hand, if v_i is connected through the primary edge to A, then the said argument is "stolen" in favor of x_i; that means v_i is disconnected from A and `Parameter` x_i becomes its new parent.

   There is no actual substitution at the use site of the parameters. All references to the `Parameter` x_i still point there, instead of the newly acquired value v_i. Sill, v_i is now the child of x_i making it easily reachable from all of its uses.

3. The action A is removed from the code, and replaced with the body of the called function L. That means, the parent of A — which must be a `Closure` object — points to the body of L as its new body. The body, which is another `Action` object, is scheduled for the next execution step.

Compared to a formal definition of application semantic (CPS1):

$$(\lambda\overline{x}.b)\ \overline{v} \rightarrow \begin{bmatrix} x_1 \mapsto v_1 \\ \vdots \\ x_n \mapsto v_n \end{bmatrix} b$$

the only difference is that parameters within the called function are not immediately replaced by the arguments. The substitution is delayed until the actual value is needed.

Naturally, it may happen that the `Action` node is not an actual lambda function call. It may be a call to some built-in functionality or a foreign C function call. In those cases, the arguments are evaluated and passed to the corresponding C++ module handling that particular function. Typically, one or more arguments acts as a returning continuation c. In that case the operations are similar to those above:

- The returning values $\overline{r}$ are computed.
- If c is referenced through a parameter and is not an actual child of A, then it is cloned as a new lambda L.
- The result $\overline{r}$ is bound to the L parameters.

- The action A is replaced by the body of L.

If more than one continuation exists, usually only one is chosen. This is particularly true when branching. The branch taken acts as c in the above algorithm. The other continuations are discarded — they are explicitly removed from the code.

Staging

Non-trivial staging adds complexity to the above algorithm. In a single step, multiple actions can be queued for execution at the same time. Still, only one is executed at a time.

To handle this, the interpreter maintains an execution queue Q referencing all actions within the program tree that are scheduled for execution. The queue Q is partially sorted, such that $\forall i, j : j > i$ the action Q_i is not in the subtree of Q_j. In short, we write $A \supset B$ when A is an ancestor of B (thus, the subtree of A contains B). Thus the queue Q has the property $\forall i, j : j > i, Q_j \not\supset Q_i$.

With the queue, each step of interpretation of a program with staging looks as follows:

- An action A, which is last in the queue Q, at position n, is selected for execution and removed from that queue. Because the queue is partially sorted, none of the other actions $Q_{1..n-1}$ can be in a subtree of A, satisfying the active-waiting requirement, formally specified in Section 4.1.3. The size of the queue is reduced to $n - 1$.

- If the called function f within A is referenced through a parameter it is cloned, producing an inlined lambda function L — same as in the case on non-staged program.

- The arguments v_i of the action A are bound to the parameters x_i of L.

  If v_i is a concrete value and x_i has a payload, it is *consumed*. This means, all actions referencing the payload are put into a special *candidate queue* $\widetilde{Q}$, to be potentially selected for the next execution.

  On the other hand if v_i is a symbolic value, i.e. another parameter value, the payload of x_i is transferred to the payload of v_i instead.

  Finally, if L has a payload, it is consumed into $\widetilde{Q}$ as well. As a special case, L may not feature a payload, but be flagged for natural staging. In that case the body of L is put into $\widetilde{Q}$.

- The action A is removed from the code, same as in the unstaged version.

- At this point $\widetilde{Q}$ contains all actions for which the staging expression value may have changed. For every action $\widetilde{A}$ we reevaluate the expression held in its `StageEvaluator` object. If the staging expression yields $\top$, the $\widetilde{A}$ is moved from $\widetilde{Q}$ into the actual execution queue Q. If the result is still $\bot$ then $\widetilde{A}$ is simply removed from $\widetilde{Q}$.

The interpreter takes advantage of the fact that the stage values are used only within L and A is their ancestor. As such every new $\widetilde{A}$ is a child of A, thus cannot contain a child among $Q_{1..n-1}$:

$$\left.\begin{array}{c}\forall i < n \; A \not\supset Q_i \\ A \supset \widetilde{A}\end{array}\right\} \Rightarrow \forall i < n \; \widetilde{A} \not\supset Q_i$$

Consequently, elements of $\widetilde{Q}$ are compared only against each other when adding to Q. We do this incrementally, through a simple insertion-sort, by checking manually if $\widetilde{A_1} \not\supset \widetilde{A_2}$ or $\widetilde{A_2} \not\supset \widetilde{A_1}$.

After the last point, Q is partially sorted again and $\widetilde{Q}$ is empty. The algorithm can start the next step of the interpretation.

5.1.3 Performance Concerns

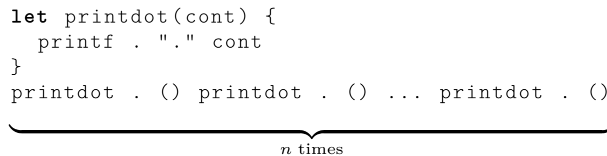
The amount of deep cloning of functions in the algorithms explained in Section 5.1.2 raises concern if the implementation is viable for anything but the smallest examples. The cloning problem is particularly painful for CPS programming, where functions return by invoking another function representing the rest of the program.

Consider a completely linear program, such as the one given in Listing 5.1. The same function is invoked in sequence n times. At call i the continuation assigned to `cont` contains a chain with the remaining $n - i$ calls. When `cont` is invoked, $O(n - i)$ nodes are cloned. If we sum up all the calls in the course of the whole program execution we obtain a value in the order of $O(n^2)$.

In theory, it is possible to hide long continuation chains under symbolic values to prevent excessive cloning. However, in practice, we cannot expect programs to avoid such chains.

For that reason we implemented a *single-use optimization*: cloning is avoided if the parameter is referenced only once. Instead, in the inlining step, the subtree representing the value is *moved* to the use site. After that, the parameter has no more references and can be discarded.

Note, that the reference count is checked only during the inlining step and not earlier. It is possible for the code to initially use a given symbol `f` multiple times in the code. However, by the time the call to `f` is reached, other uses of `f` may be already removed from the code. This is particularly true when branching instructions are used which remove branches which are not taken.



Listing 5.1: *A simple linear chain of function calls in DeepCPS. With every function call cloned this leads to $O(n^2)$ memory and time complexity*

For example, in the power function Listing 4.2, the continuation `cont` is used in two branches. Which one is called depends on whether the exponent is even or odd. However, by the time `cont` is invoked, the condition is already resolved and the `if` instruction is removed from the code. The whole branch that is not taken is removed as well. At that time, `cont` is used in one place only and can benefit from the move optimization.

Even in a program with complex staging it is rare for a continuation to be cloned. It can happen when the programmer explicitly requires a condition to be delayed in execution, but its converging continuation is executed in an early stage.

5.1.4 Binding with C/C++

ManyDSL programs can connect with existing code in two ways.

First, any C function can be imported. The function `$import` takes 5 arguments:

1. The string name of shared library to open.
2. The string name of C function to load from that library.
3. A tuple of types naming the types to be passed.
4. A type of the returning value
5. The continuation with a single parameter, under which the bound C function is stored.

For example:

```
$import . "libc.so" "fopen" [string,string] *[] (fopen) ...
```

loads the `fopen` function from the standard C library. The value stored under the symbol `fopen` acts as a regular function. The function takes the arguments specified by the third argument, plus the returning continuation where it stores the result upon completion.

This way DeepCPS can easily be made to open files. Similarly other standard and custom libraries which feature a C API can be used.

The other way of invoking existing code is through *ManyDSL-aware* functions. These are special kind of C++ functions that use the C++ interface of ManyDSL. It not only allows to skip marshalling, but also allows the function to take arbitrary DeepCPS arguments – even symbolic values or multiple lambdas are permitted.

ManyDSL-aware functions are imported through `$wimport` with arguments similar to `$import`:

1. The string name of shared library to open.
2. The string name of the C++ function to load from that library. In all our examples we use C linkage to avoid name mangling.

```
extern "C"
void ifSumPositive(Interpreter* interpreter, Action* action) {
    anydsl_assert(action->getArgCount() == 4,
      "ifSumPositive requires 2 arguments + 2 continuations");
    int v1 = dig<int>(action->getArg(0));
    int v2 = dig<int>(action->getArg(1));
    int sum = v1+v2;
    printf("Sum is %d\n",s);
    Closure* cont;
    if (s>0)
        cont = action->getInlineArg(2)->as<Closure>();
    else
        cont = action->getInlineArg(3)->as<Closure>();
    interpreter->bindArgument(cont, 0, make(sum));
    interpreter->invoke(cont);
    action->substituteWithFollowup(cont->body());
}
```

Listing 5.2: *A simple higher-order ManyDSL-aware function in C++. The function expects two* `Value` *objects holding an integer. If their sum is positive, it calls the first continuation, otherwise the second. In either case, the sum is passed as a result into the called continuation.*

3. A type of the whole function to import
4. The continuation with a single parameter, under which the bound ManyDSL-aware function is stored.

For example:

```
$wimport . "libmy.so" "ifSumPositive" fn[int,int,fn[int],fn[int]]
    (ifsp) ...
```

imports a higher-order function taking two integers and two continuations as arguments. Upon success, the obtained function (`ifsp` in the example) can be used similarly to any other intrinsic ManyDSL function.

ManyDSL-aware functions need more attention from the programmer on the C++ side. Consider a simple example in Listing 5.2. All ManyDSL-aware functions hold the same signature, by taking two arguments:

- The pointer to the interpreter that is used to executed the ManyDSL code.
- The pointer to the `Action` object in the TR representation that was used to make this call.

Typically, at the beginning of the ManyDSL-aware function, the arguments of the action are inspected. The method `action->getArg(`n`)` retrieves the object attached directly as the n-th argument. In order to retrieve the underlying C value, a series of TR node operations need to be performed (e.g. dereferencing, node type checking). For convenience ManyDSL provides the `dig` template function to obtain the value in one go, or report an error if that was impossible.

Higher-order values are always objects of type `Closure`. These can be retrieved through: `action->getArgDeref(`n`)->as<Closure>()`. However, if the closure is the continuation that we ultimately invoke, it must be accessed through

`action->getInlineArg(`n`)` instead. This causes inlining of the argument as discussed in Section 5.1.2.

Once the arguments are read, the author may use them to perform any desired operation in C++. Produced results are usually passed to one of the continuations in three steps:

1. The values are bound to the parameters of the continuation, using the `interpreter`.
2. The continuation is invoked with the `interpreter`.
3. The current action object is removed from the code.

The `interpreter` plays an active role when binding arguments to parameters. This causes all instructions targetted by the instigators to be put into the candidate queue. Neither binding nor `invoke` does not actually cause execution of the follow-up TR code. Instead, they merely set up the interpreter to perform the next steps of the interpretation.

Once all parameters are set, the ManyDSL-aware function may end, releasing control back to the interpreter.

5.1.5 Pitfalls

In the above algorithm we silently skipped few less obvious scenarios that need attention.

Payload Changes after Cloning

Attention has to be made to `Payload` objects when cloning a subtree. A payload outside of the subtree may gain additional references from the clone. This means, that when the payload is consumed, more actions may be activated than it would seem when inspecting the original code.

Consider a lambda function L and a staged action A nested somewhere within the body of L (thus $L \supset A$). The action A is staged upon P which is not within the scope of L ($L \not\supset P$). The function L is bound to some name f and then cloned when f is invoked.

As a result of such clone, the subtree of L is copied, including A, but not P. Consequently, both the action A and its copy A' are staged upon the same payload P.

In our first implementation, `Payload` objects were referencing all actions that were staged upon it. However, the above case complicated the cloning process. Each staged action clone A' had to check if its referencing payload was cloned and act accordingly.

Since then, we decided that it is better to reverse the edge, so that the action references the payload instead. The payload consumption relies on *backedges*

```
(return)[builder] { return .
  (x, end)[P] {
    @builder:
    let sqr(return)[call] {
      @P: x*x (xsqr)
      @call:
      print . xsqr return
    }
    @P:
    sqr . ()
    sqr . ()
    end .
  }
} . (P)
...
```

Listing 5.3

which creation is handled by edge creation routine, simplifying the cloning implementation. This way the routine cloning `Payload` references does not differ from regular references.

Stages to Dead Code

In the previous scenario we observed that additional work can be added to a payload in the course of interpretation. The reverse is also possible: Whenever an `if` is resolved, the branch that is not taken is removed completely from the code. If such branch contained payload references, those are removed too.

Consider a different behavior, where a skipped branch B is simply detached from the code, without being removed. There are no edges or references from outside B, and the whole branch may be removed by the garbage collector at any moment. Let us assume that action $A \subset B$ is staged upon some `Payload` P which is not within B. This makes A reachable from P through the *backedge* – that garbage collector ignores in its reachability test. If at that time P is consumed, the action A may be scheduled for execution, although it is no longer part of the code. We say that A is a *phantom* use of P.

In order to remove phantom uses, the semantic of `if` is to explicitly iterate over all descendants of the skipped branch and remove the nodes. As a part of the process the reference edges are removed, and so are the backedges from the code into the dead section. The removal of phantom uses is also important for the optimization described in Section 5.1.3.

Stages to Bound Values

Another interesting scenario is when a staged action A is within a lambda L bound to another parameter f. Previously, we considered what happens when f is invoked. It may happen, however, that A may be potentially cloned when the whole binding to f is cloned instead.

Consider an example code in Listing 5.3. It is a lambda builder which generates a program `P`. The program `P` takes an integer argument `x` and prints its squared value twice.

However, we use staging twice to alter the normal execution flow of the program:

- The binding of lambda (return)[call]... to the name sqr is done as soon as the builder is called.
- The squaring of the integer x done within the function sqr is done as soon as the program P is invoked.

If the interpreter was strictly conforming to the formal definition in Section 4.1.3, the binding to sqr would immediately cause all uses of that name to be replaced by the lambda (return)[call]. After evaluating the **let** we would obtain:

```
(x, end)[P] {
  @P:
  (return)[call] {
    @P: x*x (xsqr)
    @call:
    print . xsqr return
  } . ()
  (return)[call] {
    @P: x*x (xsqr)
    @call:
    print . xsqr return
  } . ()
  end .
}
```

The function is replicated twice, making three instructions staged upon P.

Our interpreter however delays the substitution. The code is a merely:

```
(x, end)[P] {
  sqr . ()
  sqr . ()
  end .
}
```

The lambda (return)[call] is no longer a direct part of the code, although it is still referenced through the sqr Parameter object. The multiplication instruction is now the phantom use of the payload of P. It is still potentially reachable however, when sqr is substituted. For that reason we cannot remove this use of P.

Let us now assume that function P is invoked. There are two actions in the payload of P: The phantom action $x*x$ (xsqr) and the call to sqr. Neither is a child of another thus, in theory, they can be evaluated in any order. In practice we use a heuristic to ensure that phantom actions execute first. The solution is not ideal and if multiple phantom uses are present may lead to a suboptimal order of execution.

Secondly, we must consider that the function P is cloned to P'. What happens to the lambda that is bound to sqr? The correct behavior is to clone it as well, since the value of x may be different between P and P'. To ensure this, the original lambda function (return)[call] must retain its context within the code. It is achieved by maintaining *phantom edges* when an action is removed from the code after execution. These phantom edges replace only the primary edges and are used only during cloning to capture all the necessary nodes requiring a copy.

The current implementation of these corner cases is suboptimal. For example, many of the nodes referenced by the phantom edges do not require a copy if all its free variables provided by the context are not copied. However, checking that during every clone would be very inefficient. We continue to search for a better algorithm to handle this case.

5.1.6 Hardware Reflection

The ManyDSL core provides a basic information on the environment and the hardware that the code is running on, through the built-in objects `$os` and `$arch`. `$os` informs about the operating system. `$arch` provides the name, version, instruction set of the CPU. It also gives more detailed information, such as the supported SIMD width or the cache structure: The number of levels, their size, the cache line width, etc.

A fine-tuned program may take these values into an account in order to best utilize the hardware. With the help of staging, functions may specialize for very specific configuration, without producing any run-time overhead.

When compiling, the user may substitute the default values `$os` and `$arch`, with their own values. By doing so, the underlying compiler may be directed to generate code for a different architecture than the one ManyDSL is running on.

We do not provide any library or database of values for `$os` and `$arch` based on the commonly accessible hardware. Creating and maintaining such a catalog would be needed when ManyDSL becomes a commercial product. At the current stage of the project, we decided to focus in other areas.

5.2 Lang Module

While the ManyDSL Core handles execution of the code given in the Target Representation, the Lang module is responsible for reading the source code and generating that representation. The module splits the work in a traditional way between a lexer and a parser: The lexer recognizes tokes in the input character stream. The tokens are then sent to the ManyDSL parser.

Both lexer and parser are mutable and the language grammar they operate on may be specified by the user code. Moreover, every parser may also interact with the ManyDSL Core as a part of the parsing process. For that reason ManyDSL features a built-in parser generator as well as its own parsing engine.

The semantics of any custom language is given through semantic actions (see Section 4.3). These actions are given as TR functions and are executed during parsing. The mechanism for interleaved execution and parsing caused by this is explained in detail in Section 5.7.2.

5.2.1 Components

Lexer

The lexer of ManyDSL is built on top of Xpressive library, included in the boost C++ libraries [1] . The lexing is streamed, reading at most one additional character beyond the token that is needed by the parser.

A lexer can be defined by the user, either in C++ or as a part of a language specification in ManyDSL. It is done in a declarative way by defining lexing rules. A general rule is any regular expression term. For convenience, ManyDSL also provides functions for defining:

- keywords rules — any alphanumeric word. The word is case-sensitive and is recognized by the lexer only if it is the whole word, separated by other, non-alphanumeric characters.
- literal rules — any sequence of characters that is read verbatim, ignoring the regular expression syntax.

The lexer provides tokens to the parser, represented by identification integers. It is accompanied by a string that was read to produce the token.

Parser Building

Before the lexemes can be processed by the parser, a language for that parser must be specified. When the user creates the language, ManyDSL needs to process its grammar. In this phase, ManyDSL computes the *FIRST* sets for every production, as defined in Section 4.2.3. With it, for each nonterminal n it creates a decision table T_n indexed by the lexemes of the language Σ. For a given terminal σ the table points to the production of n such that σ is in its *FIRST* set. As we will show shortly, the parser uses the table to make decision which production to take when expanding the nonterminal n.

$$T_n : \Sigma \to P; \quad T_n(\sigma) = p : \begin{cases} p = n ::= \overline{a} \\ \sigma \in \mathit{FIRST}(p) \end{cases}$$

If two or more productions satisfy the constraints given above, we face a *conflict*. The same terminal begins two productions and the parser cannot make a decision which one is correct. This means that the grammar is not LL(1). It is an error that must be fixed by the language developer.

Nonterminals in a language definition are referred to by their actual names. This allows single-pass definition of a language, with terms referring to nonterminals that do not yet exist. It is also necessary, when referencing nonterminals from different languages, as in the example given in Section 6.4.5.

[1] http://www.boost.org/doc/libs/1_60_0/doc/html/xpressive.html, retrieved on 17.03.2016

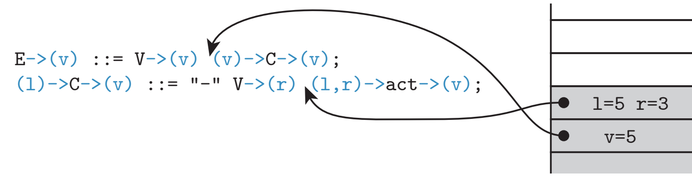


Figure 5.3: *Stack of the recursive descent parser.*

Parser Execution

With T_n tables ready, the language can be used for parsing. The tokens provided by a lexer are sent to the ManyDSL parser engine. The parser follows the user-defined grammar, handles variable passing between the rules and invokes the underlying ManyDSL interpreter when a DeepCPS action is encountered, following the semantics given formally in Section 4.2.3.

Parsing is performed by a predictive recursive descend parser. The state of the parser is defined by a Parse Stack (PS) (Figure 5.3). The PS is independent from the ManyDSL Core interpreter. Each element of PS refers to a single parser position (PP) within a body of one of the productions in the grammar. In addition, each element of the stack holds values of all local variables within the production.

In each parser step the behavior of the parser depends on the next token, following the top PP.

Encountering a terminal requires the input stream to contain a corresponding lexeme unconditionally. If the terminal returns an argument, the matched string is returned. Finally, the PP is advanced.

When a C or ManyDSL function call is encountered, the input stream remains unaffected. The C function is invoked as is, and the parser work is limited to preparing the arguments and then reading the result. In case of a ManyDSL function, in addition to the language-defined arguments, the parser constructs a special return continuation. The core executes the function until the continuation is invoked. When that happens, control is returned back to the parser. The mechanism of halding the interpreter is discussed in detail in Section 5.7.

Finally, when a nonterminal n is encountered, the parser compares the look-ahead lexeme σ with the decision table T_n. If $T_n(\sigma)$ points to a valid production, the production is invoked similarly to a function call: A new entry is added to PS, and the values of the arguments are bound to the parameter names.

It may happen that no production satisfies the requirements and $T_n(\sigma)$ does not give an answer. In that case, the epsilon production for n is invoked, acting as a wildcard. Only when no such production is found, we raise an error.

As we discussed in Section 4.2.3, this behavior is different when compared to typical LL(1) parsers. In a typical LL(1) parser, the epsilon rules are already taken into account in T_n tables by computing the *FOLLOW* sets. Because we do it differently, languages that have a shift-reduce conflict originating from the epsilon rule are incorrect for LL(1) but are accepted by ours (giving shift a

```
Block ::= Element BlockCont;
BlockCont ::= epsilon;
BlockCont ::= Separator Element BlockCont;
```

Listing 5.4: *A typical* `Block` *rule for a sequence of* `Elements` *separated by some* `Separator`*. The last production for* `BlockCont` *is right-recursive and may be optimized by the tail rule optimization.*

higher priority). Moreover, in case of an actually incorrect input, epsilon rules are greedily taken before an error is actually raised.

The difference may seem small, but it is a crucial property in a multi-language environment.

Firstly, it may happen that the following token belongs to a different language. In that case, any reliance on the $FOLLOW$ set would be misleading.

Secondly, by continuing parsing epsilon rules, we have a chance to encounter a user-defined action which manually changes the state of the parser, avoiding the error. Most typically, such action changes the language in which parsing occurs, making the next token a valid one.

For example, notice from the LangDSL specification given in Section 4.2.4 that a DeepCPS section is put in extra curly braces that are not part of DeepCPS syntax. If the trailing `}` was read by the DeepCPS parser in isolation, it would raise an error. It does not happen however: The DeepCPS parser has a change to unwind its parse stack gracefully, reduce the whole code that has been read to a single value, and execute an action that switches parsing back to LangDSL.

Tail Rule Optimization

Every practical language contains recursions in its definition. LL languages cannot contain left-recursion, because it leads to parse conflicts. Other types of recursion however are allowed, and right-recursion is the most common one.

One such use case is when iterating over a sequence of terms. Language constructs such as: Listing members of a tuple, providing an argument list for a function call, enumerating members of a structure, defining a block of instructions – all these are typically defined through a recursive productions of the form given in Listing 5.4.

From the execution point of view, where rules are treated as functions, right-recursion is equivalent to tail recursion. We identified that such constructs needlessly increase the stack of the parser. As a remedy to this problem the ManyDSL parser incorporates the following optimization. Assume that we execute a rule $X \to \alpha$ and the following is true:

- A nonterminal N is invoked as the last term of the production
- The values returned from N are exactly the values that are also returned from X.

If the above is true, then upon calling N, the call to X is removed from the

```
Tuple->(t) ::=
        "["
        ()->(t) {return . []}
        (t)->OptTupleElements->(t)
        "]";
(t)->OptTupleElements->(t) ::= epsilon;
(t)->OptTupleElements->(t) ::= (t)->TupleElements->(t);
(t)->MoreTupleElements->(t) ::= epsilon;
(t)->MoreTupleElements->(t) ::= "," (t)->TupleElements->(t);
(t)->TupleElements->(t) ::=
        Element->(v)
        (v,t)->(t) { concatenate(t,[v]) . return }
        (t)->MoreTupleElements->(t);
```

Listing 5.5: *A concrete example of right-recursion used for defining a tuple of the form* `[a, b, c, ... ]`

parse stack. Consequently, the values returned from N are passed directly into the context where X was invoked.

Consider a concrete language example given in Listing 5.5. The nonterminal `MoreTupleElements` is used exactly once, as the last term of the `TupleElements` production. Similarly, the nonterminal `TupleElements` is used twice, always as the last term of a production. In all these cases, the nonterminal returns a value `t` that is immediately returned further back by the enclosing production. In this scenario the tail recursion optimization triggers and the parse stack does not grow with the length of the tuple.

5.3 Executor

An *Executor* is a small control module of ManyDSL, which connects the core and the parser. Depending on its state, it performs either interpreter steps or parser steps. It provides mechanisms for communication between the former and the latter. The executor also maintains the stack of files opened for parsing and handles inclusion of new files.

The executor object can be accessed and to a certain degree be modified from the ManyDSL code. Through it, the user can modify the state of the parser or even the interpreter that is used to process the very code that is making the change. Typically, such low-level manipulation is needed in special events such as including an additional source file for parsing or changing the language used for parsing. It can potentially be used for other scenarios that we did not explore yet, such as manipulating the parse stack in a way that is beyond the standard LL(1) parsing.

5.4 Parsers

So far we explained the parts of ManyDSL that execute and generate code (see page 129). However, the modules themselves provide no starting language. All languages, including DeepCPS itself, are defined on top of the Lang module.

5.4.1 The DeepCPS Parser

DeepCPS is the first language defined in ManyDSL and is always available to the user. The parser is written in C++ but uses the very same Lang functionality that is availabe for other languages. DeepCPS is defined as an LL(1) grammar, enriched with actions which are C function calls. The actions build the TR structure based on the input.

Since DeepCPS is defined using the Lang module, it can be used the same way as any other user-defined language. In particular, when switching between languages, specific DeepCPS rules can be referenced. For example, LangDSL parser switches into DeepCPS lambda body nonterminal for the action definition. Examples on how to perform such a switch is given in Section 6.4.5.

The DeepCPS language is unique with respect of its actions: It produces the underlying ManyDSL structure by directly accessing TR classes in C++. As a result, if the parser was suddenly interrupted in an unexpected moment the structure may be incomplete and unsuitable for interpretation. This is in contrast to languages created with LangDSL — where actions are always complete pieces of DeepCPS code.

The DeepCPS parser is accompanied with an environment, naming all intrinsic functions and values available to the programmer. This includes:

- Integer types (signed and unsigned): `i8`, `i16`, `i32`, `i64`, `u8`, `u16`, `u32`, `i64`
- Floating-point types of 32 and 64 bit length: `f32`, `f64`
- Other basic types: `Boolean`, `string`, **`stage`**, `kind` (type of a type), `void`.
- The generic type: `any`
- Arithmetic and logic functions, such as `$add`, `$sub`, ... `$shl`, `$shr`, `$and`, `$or`, `$xor`, `$eq`, `$neq`, ... All these functions actually require 4 arguments:
  - A `kind` value representing the type T of the arithmetic values.
  - The first value of type T
  - The second value of T
  - Returning continuation

  Note that in the current version, the ManyDSL typing does not support parametric polymorphism and we use any as a fallback. These intrinsic functions have a type `fn[kind,any,any,fn[any]]` or similar.
- The reference to the `$executor` used for parsing and interpreting.
- The built-in DeepCPS language object is available as `$deepcps`.
- The reflection of target operating system `$os` and target architecture `$arch` as discussed in Section 5.1.6.
- An `$include` function to add additional files to the input parsing stream.

- A function `$current_environment` that returns the environment held by the parser at the time when the last halt node was encountered (Section 5.7).

- An `$exit` function, that terminates the program.

5.4.2 The LangDSL Parser

The Lang Module provides necessary C and ManyDSL-aware functions to build a new language both from C or DeepCPS level. Each element of the grammar: Nonterminals, terminals, productions with their arguments, and sequences of actions are built incrementally using those functions. Unfortunately, using them directly is inconvenient.

For that reason we provide the LangDSL parser that simplifies the task. The LangDSL is loaded by ManyDSL as a DeepCPS source library.

The actual implementation of LangDSL is done in itself. A special converter transforms ManyDSL TR back into DeepCPS source code, allowing us to bootstrap the library so that it can be loaded directly in the following runs of ManyDSL.

This approach allows us to incrementally upgrade LangDSL without directly operating on the low-level Lang module API. It also acts as a test, ensuring that LangDSL actually meets the goals we have set for a DSL-defining language.

5.5 Compiler

As we have described in the overview in Section 3.5, ManyDSL allows not only for interpretation but also for compilation into machine code. We use Thorin as the backend for our CPS programs. During the translation all remaining staging information is stripped off as it is not supported by Thorin. Then, Thorin transforms the code through its technique – lambda mangling [84] – in order to remove higher-order CPS functions, or translate them into jumps, such that they can be translated into assembly code.

With the code simplified, Thorin is translated into LLVM and subsequent low-level optimization passes are triggered. The compiler produces an LLVM module, which optionally can be compiled into actual machine code as well.

The translation is started through a ManyDSL-aware function that initializes the process, producing a Thorin code object. Further steps are available through a C-API of the respective tools. For convenience, we also include a few functions to operate on Thorin and LLVM in the most typical way, e.g. to trigger optimizations or emit LLVM bytecode into a file.

5.6 Application

Finally, the whole ManyDSL system with its modules is accompanied with a tiny application layer. The application takes a file as an argument and opens it as the first DeepCPS source. With an optional parameter `-r, --run` it forces interpretation after parsing is finished. However, that command is not needed to execute if the source is interleaving parsing and execution on its own, as detailed in Section 5.7.

When multiple languages are used, successful parsing requires execution. For that reason there is no “dry-run” option which would simply parse the code without execution.

5.7 Interleaved Parsing

We explained all modules of ManyDSL but we merely mention an important property of the system: The ability for interleaved parsing and execution. The ability to do so allows the user program to modify the parser on the fly. It is the central property of ManyDSL, but also one that strongly influences the implementation of all modules.

- The interpreter must be capable of running code that is incomplete. The possibility of new code appearing at a later time opens potentially dangerous scenarios, such as adding work to already consumed payloads, or adding new references to parameters that already triggered single-use optimization (Section 5.1.3).
- The parser must create a meaningful code structure even though it is incomplete. The parser and DeepCPS, in general cannot assume that when their work is resumed, their state remains unaltered. On the contrary: The user, through the interpreted code, may alter the parse stack or change the language.
- The interpreter and parser must agree on how locations where parsing was interrupted are marked.
- DeepCPS needs an explicit construct to halt parsing.
- Finally, the executor must provide mechanism for switching between parsing and interpreting.

5.7.1 Halting in DeepCPS

In DeepCPS, a special, parse-time operation is indicated through # and it comes in two flavors as described below.

Parse-time Function

In order to execute a given function at a given time during parsing, that function is put within braces: `#< ... >`. The code within them should return with its own `$exit` function at which point the control is returned to the parser. The most common use of such function is to include files, for example:

```
#<$include . "library.cps" () $exit . >
```

The code as above is completely parsed, but upon reading `>`, the process is halted and the contents are executed. In the above example, we add a new source file named `library.cps` into the lexer stack of opened files. Afterwards, the parser-time function exists through `$exit`, and parsing resumes. However, at that point the new file is on top of the source stack and the next token is read from that file. Text that follows `>` is parsed only after the parser reached end of `library.cps` file.

Since including files is the most common action for parser-time functions, DeepCPS comes with a simplified syntax:

```
#include<"library.cps">
```

It should be stressed however, that this is merely syntactic sugar. All parse-time actions are handled by the same language and use the same pipeline as the actual code that is being compiled or interpreted. There is no separate preprocessing step.

Halt Marker

The second form of parse-time operation is indicated through a *halt marker* `##`. This symbol may appear only in place of a lambda body. When the DeepCPS parser encounters this symbol, a special Halt node is put into the produced ManyDSL core code. The `Halt` class is a special `Action` and is inserted as a body of a `Closure` node. Afterwards, parsing is interrupted and the code that has been parsed so far is executed.

The interpreter works until it executes the `Halt` node. By executing the `Halt`, interpreter releases control back to the parser. The parser continues by parsing the actual body of a lambda in which the halt occurred. The `Action` object representing the actual body of the lambda becomes a special "replacement child" of the `Halt`.

When parsing is interrupted for the first time, the interpreter starts from the first instruction in the source code. At later interruptions — interpreter resumes execution from the replacement child of the `Halt` node it encountered last.

From the interpreter point of view, halt markers are transparent and have no impact on the execution. In most cases, adding halt nodes into DeepCPS code has no impact on the execution. There are a few corner cases however, where full transparency cannot be achieved. We discuss that in detail in Section 5.7.3.

Halt markers are typically used for big sections of code, or fragments that produce

```
proj . $arch 2 (ptrtype:kind)
$typeeq . ptrtype u32 (bit32:Boolean)
ifelse  . bit32
  (endif) { endif . "lib32.cps" }
  (endif) { endif . "lib64.cps" }
(libsrc)
##
#<$include . libsrc () $exit . >
```

Listing 5.6: *Using DeepCPS halting to compute the name of a file to include. At the time of the call to `$include`, the value of `libsrc` is already known.*

```
ifelse  . bit32
  (endif) { ##
    endif . "lib32.cps" }
  (endif) { ##
    endif . "lib64.cps" }
(libsrc)
```

```
fix power (base, exp, cont) {
  ##
...
```

Listing 5.7: *Invalid uses of halt marker. The marker may appear only in the trailing continuation, passed as the last argument to all functions that precede it. Otherwise, the interpreter will try to execute a call without all arguments parsed for it or jump over the marker into a code that is not yet specified.*

results that should be visible by the parser or the executor. In particular, a halt marker is needed after a new language is specified.

When a parser reads a language specification, either in LangDSL or plain DeepCPS, a TR representation of that specification is constructed. Only by executing it, the lang module creates an actual language object that can later be used for parsing in that new language.

Halt markers can be useful in other situations as well. For example in Listing 5.6 we compute a name of a file to inculde. All named values known at the time when the marker is reached can be used afterwards at parse time, such as in the parse-time call to `$include`.

Position of Parse-Time Constructs

The parser expects the `#` and `##` symbols only at the beginning of a lambda body. These symbols are part of the DeepCPS language and not a preprocessor construct. Trying to insert parser-time construct at different location leads to a parser error.

Moreover, when introducing the halt marker `##` all DeepCPS constructs preceding it must be complete, with an exception for the innermost lambda. That means, `##` should appear only in the continuations given as the last argument to any application. Naturally, it can also appear after **`let`** and **`fix`** constructs, but not within them.

The example Listing 5.6 satisfies the constraint: The halt marker is put within a lambda `(libsrc)...` which is the last argument of `ifelse`. It is also part of the last argument to all continuations appearing earlier.

However, in the examples in Listing 5.7 the halt marker is used incorrectly. If `##`

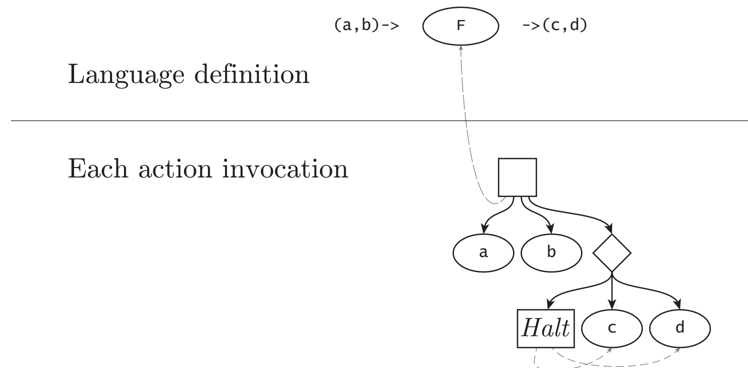


Figure 5.4: *ManyDSL node structure that is generated each time an action with user-defined ManyDSL function F is invoked. The structure is adapted to match the number of input and output arguments defined in the language grammar.*

appears in one of the branches of `ifelse` the interpreter would try to execute the call to it before the remaining arguments are even parsed. In case of a branch, a given halt marker may not even be reached. Moreover, the instruction of calling `ifelse` is removed from the code after being executed. Further parsing would attach newly created nodes to an `Action` that is no longer part of the code.

Similar problems happen when `##` is found within a **`let`** and **`fix`** nodes. The semantics of those is to bind a value to its name and immediately jump to code appearing after it. However, with the parser interrupted, there would be no follow-up code and the interpreter would have nowhere to jump to. Moreover, the halt marker would be within the value and would not be executed, preventing the parser from resuming its work.

5.7.2 Action Halting

Last method of triggering interleaved execution and parsing happens when a language action is triggered that calls a function already represented in TR. The function is guaranteed to be complete as it was provided when the language was defined. Still, the executor must switch from parsing mode into interpretation mode and at the end — switch back to parsing again.

Each time a semantic action with ManyDSL code is triggered, the executor creates a piece of surrounding code as shown in Figure 5.4. The whole piece is put right after the halt marker where interpretation was last interrupted, and then execution is resumed. Based on the number of the input and output arguments, a different surrounding code is constructed. Input arguments are passed to F, together with a returning continuation. The continuation takes the resulting values as arguments and passes them into the halt node. The executor then inspects the received values and passes them back to the lang module.

5.7.3 Halt and Interpretation

The existence of the halt marker has numerous consequences on how interpretation is being done. Most prominently, it affects the cloning process as well as staging.

Cloning

Halt markers should never be cloned.

If a halt marker was cloned it would incur problems affecting both the performance and the validity of the underlying algorithms. In terms of performance, a cloned halt marker would imply that the remainder of the program – as it appears within the subtree of the halt node – is cloned as well. The single-use optimization, which we explained in Section 5.1.3, is designed to prevent such case.

Consider a halt node created for a parser action. The executor fetches the arguments returned through it. However, if the halt node was cloned, it would be possible for the computed values to be stored in any of its copies. Every halt clone would need to be tracked by the executor in order to find the actual result.

A similar issue happens when capturing values by parse-time functions. The parse-time functions are not cloned. Doing otherwise would imply executing them multiple times, depending of the numbers of clones.

However, parse-time functions are parsed in the context of actual DeepCPS functions and may refer to parameters defined by them. For example, in Listing 5.6, we refer to `libsrc` defined by the lambda preceding `##`. If the lambda `(libsrc)...` was cloned, leading to the cloning of `##`, it would be unclear which copy of `libsrc` is referenced within the parse-time function. The most intuitive solution would be to refer to the original, but that one is most likely not going to have its value assigned and `$include` would fail. That is because usually when we clone, we invoke the copy, not the original. Tracking the last copy is an uncertain solution as well because cloning in general is not necessarily linear.

All these problems are resolved with the single-use optimization. Halt markers appear only in continuations which are invoked only once throughout the code. An important observation has to be made here: In the context of halting, the single-use optimization is no longer an optional improvement of the algorithm, but a requirement for halting to work correctly.

Use Count

When a halt node appears within a body of a lambda function, it is unclear how many times these lambda parameters are being used. It may seem that a given parameter p is used once, or not at all in the code that is already parsed, appears at some later time when parsing is resumed. For that reason, all such parameters must be exempted from the single-use optimization.

We achieve that by registering each halt node in all its ancestors. The `useCount`

function for a `Parameter` checks if a halt node is registered in its parent `Closure` object. If it is, `useCount` returns infinity.

Staging over Halt Nodes

It is possible for the user to try to stage an action behind the halt node on a lambda or parameter preceding it. Unsurprisingly however, an action cannot be executed before it is parsed, even if the stage is triggered. In such scenario, the halt marker cannot be made completely transparent. It must execute before any action that follows it, regardless of staging.

This scenario however requires attention from the interpreter. To handle it gracefully, whenever a `Payload` is consumed by another object C (the interpreter or another `Payload`), the consumed object stores a reference to C. When a parser adds additional work to an already consumed `Payload` it is put into the referenced object C instead. If C is the interpreter, the action is put into its candidate queue $\widetilde{Q}$ (see Section 5.1.2).

When interpretation resumes after such scenario, there is usually some work in $\widetilde{Q}$ that needs to be processed first. After that, the deepest instruction is executed first, following the normal rules of dynamic staging.

5.8 Roads not Taken

In order for ManyDSL to be practical, it must include an efficient interpreter. It is used not only for execution of the actual final program, but also for parsing, building, and specializing. For that reason we have spent considerable time designing and implementing it efficiently and trying different approaches.

In Section 5.1 we explained that every function that is being called has its body cloned. This does not seem as a good solution for an efficient interpreter. In fact, in our first approach we tried to avoid that.

Code Structure in a Context

Our first version of the interpreter, designed for a stageless DeepCPS, uses the traditional approach: The code is immutable, and the run-time values are stored in a binding map. The map binds symbolic parameters to actual values. Each time a function is called, new entries are added into the map for every function parameter. Each time a parameter is read to obtain its value, the map is used to look it up. We refer to such map as a *context* within which the immutable code is being interpreted.

Since control flow can return to the context of the previous function, e.g. through a closure or a continuation lambda, a previous map cannot be automatically removed. Instead, a technique similar to garbage collection is used. The interpreter periodically checks which contexts are reachable and which are already abandoned. In some cases, a heuristic is used to reclaim the memory immediately, e.g. when the CPS function call resembles the normal call —

for example, when returning through a continuation without any higher-order arguments.

In CPS programming, the same parameter may be bound multiple times. The interpreter must maintain information which context is being used in each case. The context can be identified simply by noting which function call instance was used to create it.

Unfortunately, the context management becomes much more complicated when doing staging. Each call instance may differ not only with respect to values, but also which instructions were actually executed. In order to support that, we extend the context such that it is able to hold ManyDSL core edges which override those given in the original program. However, unlike value binding, these edge overrides are mutable. The same change edge may be changed multiple times in the same context.

Consider a function F that is partially evaluated through staging. Suppose that F contains a loop with a body b that is unrolled n times. Each iteration of the loop produces a piece of code. Since the code is not cloned, each piece actually refers to exactly the same core structure. In order to distinguish one from the other we use the context information. The context of each iteration can be identified as $\langle b_i \rangle$ which marks the i-th invocation of the body function b.

Let us assume that afterwards the partially evaluated function F is invoked several times throughout the code. Each invocation creates a new context $\langle F_j \rangle$ but such identification is insufficient for the unrolled piece of code. Each iteration must remain distinguishable from other iterations of the same loop, but also remain distinguishable from other calls to F. Therefore, we identify it as $\langle F_j b_i \rangle$. This reads: i-th iteration of the body function b as part of the invocation F_j.

At this point, the string identification of the context can expand further in all directions:

- If the call to F_j is part of another function specialization G, then calling G will create $\langle G_k F_j b_i \rangle$.
- If within F_j the unrolled loop is passed as a lambda to another function G, then calling it will create $\langle F_j G_k b_i \rangle$.
- Finally, if within given iteration b_i a function G is called we create a new context $\langle F_j b_i G_k \rangle$

All these are possible because of staging.

At this point we realized that identifying the right context and transitioning from one to another is not a trivial task. Moreover, since in CPS every instruction is a function call, such identification of context would be very inefficient. Since the code relies heavily on tail recursion the identification strings end up being very long. The management of the names would have to be further optimized in some way.

For that reason we decided to abandon this path and search for a different,

simpler solution.

Lazy Cloning

With the assumption that cloning is necessary, one can still try to avoid it through lazy cloning. Only nodes that are directly accessed in the clone are being generated.

However, because of staging, nodes need not be accessed in the order of their appearance. Consequently, sections of code that are and are not cloned can be interleaved. It is necessary to memorize the cloning history so that when a new node is cloned, the edges can be recreated accordingly. This memorization becomes even more convoluted when the same code is cloned multiple times.

The same example as we previously used for string identification can become problematic to properly maintain the core structure and all its partially cloned copies. Ultimately we were unable to find a solution that would be correct and efficient.

For example, in one of our solution we implemented a *clone map*. The map contained node pairs (n, n') wherever n' was a clone of n. However, in many cases the cost of maintaining such a map was greater than doing an actual aggressive cloning operation.

In Conclusion

We do not deny the possible existence of a better solution. Indeed, further investigation of the problem may greatly benefit the efficiency of the whole ManyDSL approach. However, in order to continue the project, we opted for the simplest approach: Cloning everything upon a function call, as explained in Section 5.1.

Chapter 6

Usage Examples

In this chapter we focus on more practical examples and use cases.

6.1 Stageless DeepCPS

One of the highlights of CPS, as we explain in Section 4.1.1, is that there is only one type of instruction: An application. Other operational constructs, such as branching and loops can be expressed as functions. This has numerous practical consequences.

6.1.1 Defining Control Flow Structures

The most straightforward consequence is that the loop functions can be defined by the user and can be passed as arguments. Representing control flow in CPS program as functions is by no means novel, but it is an important property, and use case for later examples. It becomes particularly significant when defining new languages.

In DeepCPS there is only one build-in control flow function — the branching function:

```
fn[bool,fn[],fn[]] if
```

The `if` function takes a Boolean argument and two parameter-less continuations. If the Boolean argument is `true`, the `if` function calls the first continuation, otherwise the second one is invoked.

There is no explicit converging instruction to finish the two branches of the build-in `if`. Implicitly, this happens when the same continuation is invoked by these branches. This can be done in-line or be abstracted in a separate function, as shown in Listing 6.1.

The explicitly-converging branch function `ifelse` takes the Boolean condition and *three* continuations: `tb`, `fb`, and `endif`. The first two are the regular branches

```
(endif) {
  if . cond () {
    ... true branch ...
    endif .
  } () {
    ... false branch ...
    endif .
  }
} . ()
... rest of the program ...
```

```
let ifelse
  (bool cond, fn[any] tb,
   fn[any] fb, any endif) {
    if . cond () {
      tb . endif
    } () {
      fb . endif
    }
  }
```

Listing 6.1: *The inline version of if-then-else with converging continuation* endif*, and an abstracted version of the same construct.*

```
(endwhile) {
  fix loop = () {
    ... condition computation ...
        (cond)
  if . cond () {
    ... loop body ...
    while .
  } () {
    endwhile .
  }
  in
  while .
} . ()
... rest of the program ...
```

```
let while
  (!initargs, cond, body,
      endwhile) {
    fix loop = (!args) {
      cond . !args (cond)
      if . cond () {
        body . !args (!args2)
        while . !args2
      } () {
        endwhile . !args
      }
    }
    in
    while . !initargs
  }
```

Listing 6.2: *The inline version of a while loop, and an abstracted version of the same construct.*

and the third is the continuation representing the code after the convergence point. The `endif` is passed into both `tb` and `fb` and is called from within them to end the branch. The type of `endif` is not specified: It is a function taking arbitrary many arguments.

The type ambiguity of `endif` can be avoided when using the inlined version of ifelse. In theory it could also be addressed by a dependent-type function:

```
let ifelse (type T, bool cond, fn[T] tb, fn[T] fb, T end) ...
```

but such constructs are currently not supported in DeepCPS.

Since the built-in branch does not specify how it converges, it can be used for more complex control flow structures — for example, to specify a while loop with a termination criteria given as a function. In the example Listing 6.2 we define such a loop as a function containing condition computation and a body. The condition computation function returns, through continuation, a Boolean value `cond` upon which we decide if we continue with the loop or terminate it with the continuation `endwhile`.

Most loops, especially in a purely functional setting, take and return arguments, changing them with each iteration. That is why the `while` abstraction in the example uses a DeepCPS extension `!`. The function can take arbitrary many arguments and the excessive values are put all together into a tuple `initargs`. These are passed into the first iteration of the loop, and then into both `cond`

```
fix for = (!args, from, to, body, endfor) {
  from<to . (cond)
  if . cond () {
    body . !args from endfor (!args2)
    from+1 . (next)
    for . !args2 next to body endfor
  } () {
    endfor . !args
  }
} in ...
```

Listing 6.3: *A* `for` *loop iterating from* `from` *to* `to`*, incrementing the value by 1.*

and `body` functions. The `body` produces a new set of arguments which is passed into the following loop iteration. The general `while` suffers the same typing problem as `ifelse` but otherwise behaves as desired. In the inlined version, all additional arguments can be specified in-place.

For Loop

Naturally, more specialized loops can be defined by the programmer or built on top of others. For example, in Listing 6.3 we define a `for` loop. For every integer in the range `[from .. to)`, iterating by 1 from the bottom we invoke the `body` function. A generic argument list `!args` is passed into the body. The body returns, through a continuation, a new set of arguments `!args2` which is used in the next iteration of the loop. Finally, when `to` is reached, the ending continuation `endfor` is called.

When the `body` function is invoked, in addition to `!args` more arguments are passed (see `body` invocation in Listing 6.3):

- `idx` – the current iteration index.
- `break` – the `endfor` continuation, permitting the code within the `body` to exit the loop immediately.
- `continue` – the continuation leading to the next iteration of the loop.

The author of the `body` may choose to actually disregard the provided functional arguments and continue in a completely different direction. That way, a multi-level breaks or different jumps can be executed from within the loop.

Based on `for`, in Listing 6.4 we define a `foreach` loop that iterates over all elements of a given tuple. For each index `idx` we extract the `idx`-th element of the tuple and pass it into the `body`, together with `break` and `continue` continuations given from the generic `for` loop function.

Multi-dimensional For Loop

The functions `for` and `foreach` can be combined further to support iterating in multiple dimensions, as shown in Listing 6.5. In the `formd` function we form a sequence of functions F, each for one dimension:

```
let foreach (!args, tuple, body, endfor) {
  arity(tuple) . (size)
  for . !args 0 size (!args, idx, break, continue) {
    tuple[idx] . (element)
    body . !args element idx break continue
  }
}
```

Listing 6.4: *A* `foreach` *loop iterating over all elements of a tuple.*

```
//for multiple dimensions
let formd (!args, from, to, body, endfor) {
  let F0(!args_and_idxs, continueF) {
    body . !args_and_idxs endfor continueF
  }
  foreach . !args F0 from (Fim1, fromv, idx, break, continue) {
    to[idx] . (tov)
    let Fi(!args_and_idxs, continueF) {
      for . !args_and_idxs fromv tov
        (!args_and_idxs, idx, break, continue) {
        Fim1 . !args_and_idx idx continue
      } continueF
    }
    continue . Fip1
  } (Fd)

  Fd . !args endfor
}
```

Listing 6.5: *A multi-dimensional for loop. The arguments* `from` *and* `to` *are tuples of the same arity, and every value in the volume between these end points is accessed once. For each dimension* `i` *a function* `Fi` *is built, calling* `Fim1` *(F i minus 1) in a loop. When the final* `Fd` *is constructed, it is invoked.*

- `F0` packs together the user defined arguments `!args` and m-dimensional coordinate that the current `formd` iteration represents. It passes these arguments into the user-defined `body`.
- `Fi` handles the i-th dimension. It performs one-dimensional iteration between values `from[i]` and `to[i]`. In each step it invokes `Fim1` (F i minus 1, F_{i-1}), which — by induction — handles all dimensions lower than i.

We conclude, by induction, that F_d, where d is the dimensionality, handles the whole volume between the corners `from` and `to`. It should be noted, that F_{i-1} takes one more argument than F_i: The coordinate in i-th dimension is added to the `!args_and_idxs` when invoking `Fim1`.

Switch Statement

Another practical construct is a switch statement, which in DeepCPS can be defined as a function as well, for example as in Listing 6.6. In a switch statement, a control value is compared to a series of case values. The `cases` argument is a tuple, with each element being a pair: `casevalue` and `casefunction`. If `casevalue` is equal to `control`, the `casefunction` continuation is invoked. If none of the cases matches the control variable, the `default` continuation is taken.

```
let switch (!args, control, cases, default, endswitch) {
  foreach . !args cases
  (!args, case, trynext) {     //foreach body
    case[0] . (casevalue)
    case[1] . (casefunction)
    casevalue==control . (equal)
    if . equal
    () {
      casefunction . !args case endswitch
    } () {
      trynext . !args
    }
  } (!args) {                     //foreach end
    default . !args endswitch
  }
}
```

Example usage:

```
let int2apples(int value, fn[string] return) {
  switch . value
    [                                 //cases
      [0, (v, endsw) { endsw . "zero" } ],
      [1, (v, endsw) { return . "one apple" } ],
      [2, (v, endsw) { endsw . "two" } ],
      [3, (v, endsw) { endsw . "three" } ]
    ]
    (v, endsw) { endsw . "many" }     //default
    (strnum)                          //endswitch
  strnum+" apples" . (completestr)
  return . completestr
}
```

Listing 6.6: *A `switch` statement implementation iterating over all cases in a sequence. Example usage of such switch statement*

The `endswitch` continuation is passed to all `casefunction`-s and the `default`. It specifies the final convergence after the switch statement. However, similarly to all other branches, the case functions do not have to continue the normal way using the ending continuation they are given.

It should be noted that for the purpose of this example, each case function is independent. With this implementation, the execution cannot "fall-through" from one case statement to another. Supporting such behavior is naturally possible: The follow-up cases would have to be passed as continuations into the previous case function. We skip the precise definition for the sake of brevity.

6.1.2 Extending DeepCPS

We have shown how advanced control flow structures can be defined in DeepCPS, using only the simplest branch instruction as a built-in function. Some architectures may provide a more efficient implementation which is beyond the reach of native DeepCPS. For example:

- The programmer may want a parallel for loop, to take advantage of inherent parallelism of the device.
- The `switch` statement is often better handled through an associative jump

table, rather than a chain of conditionals.

The precise implementation of these are hardware specific and cannot be defined in DeepCPS alone.

Fortunately, since every language construct in DeepCPS is a function, extending DeepCPS is easy. It boils down to defining a new built-in function and letting the interpreter know about it. No change in the language itself is required.

In Section 5.1.4 we show how arbitrary C function can be imported to DeepCPS. The key function is the built-in `$import` function which extracts a C function from a library file, and encapsulates it such that it can be used in continuation-passing-style.

This mechanism allows to quickly expand the DeepCPS functionality without creating large sets of built-in functions in ManyDSL itself. For example, in Listing 6.7 we load the necessary functions to operate on files. We use these to load a data matrix stored in a textual form in a file. The file describes the dimensions x and y of the matrix, followed by $x \cdot y$ floating-point entries specifying the data.

The converted functions can pass and receive all numeric values or tuples. A C ellipsis argument (the third argument of `fscanf`) is represented by a special type `any` indicating that any type, or number of types, is accepted. Imported functions also handle conversion from C `char*` to `string` values.

However, non-string values passed by pointers have to be treated in a special way. Pointer values must refer to explicitly allocated memory, with data stored in a native format. For that reason, for example in the `loadInt` and `loadFloat` functions we explicitly allocate a single memory cell through `alloc` and free it at the end.

The same functionality can be implemented using a special class of C/C++ ManyDSL-aware functions, which we also introduce in Section 5.1.4. While the flexibility of ManyDSL-aware functions is not necessary in the file reading example, it may significantly improve the performance of the communication between C/C++ and DeepCPS code. In the example Listing 6.8 a single function is given which reads the whole matrix and creates necessary `Value` nodes right in the C++.

The base structure of this example does not differ much from Listing 5.2, but is a more practical example of using this functionality. It follows the same three-step pattern:

- We load the data from the arguments through `getArg` and `dig` helper functions.
- We perform regular C/C++ operations: We open the file, read a matrix, and close it.
- We represent the result as a TR `Value` and pass it to the continuation.

```
let void []           //void type
let voidp void*       //void pointer type
$import . "libc.so" "fopen" [string,string] voidp (fopen)
$import . "libc.so" "fclose" [voidp] int (fclose)
$import . "libc.so" "fscanf" [voidp,string,any] int (fscanf)

let loadInt(file, return) {
  alloc . float 1 (int* m)
  fscanf . file "%d" m (readcnt)
  load . m (v)
  free . m ()
  return . v
}

let loadFloat(file, return) {
  alloc . float 1 (float* m)
  fscanf . file "%f" m (readcnt)
  load . m (v)
  free . m ()
  return . v
}

let allocMatrix(dimX, dimY, return) {
  dimX+dimY . (size)
  alloc . float size (float* matrix)
  return . matrix
}

let loadMatrix(filename, return) {
  fopen . filename r (file)
  loadInt . file (dimX)
  loadInt . file (dimY)
  allocMatrix . dimX dimY (matrix)
  formd . [0,0] [dimX,dimY] (x, y, break, continue) {
    loadFloat . file (value)
    matrix+(y*dimX+x) . (addr)
    store . addr value
    continue .
  } ()
  fclose . file (succ)
  return . dimX dimY matrix
}
```

Listing 6.7: *Using C library functions to load a data matrix stored in a file. The library functions are imported using `$import`. When C functions are used, all arguments must be marshalled and memory pointers must refer to native data, rather than TR objects. For that reason we use pointers (`int*`) and memory loading/storing (`load`, `store`).*

DeepCPS:

```
$cpsimport . "mylib" "loadMatrix" (loadMatrix)
```

C++:

```
extern "C" void loadMatrix(Interpreter* interpreter, Action*
    action) {
  const char* filename = digString(action->getArg(0));
  Closure* continuation = action->getInlineArg(1)->as<Closure>();
  FILE* file = fopen(filename, "r");
  int dimX, dimY;
  fscanf(file, "%d%d", &x, &y);
  float* matrix = new float[dimX*dimY];
  for (int y = 0; y<dimY; ++y) {
    for (int x = 0; x<dimX; ++x) {
      fscanf(file, "%f", &matrix[y*dimX+x]);
    }
  }
  fclose(file);
  interpreter->bindArgument(continuation, 0, new Pointer(matrix));
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
```

Listing 6.8: *A single ManyDSL-aware function loading a data matrix stored in a file. The ManyDSL API is used to read values from core nodes and pass the results at the end through the* `interpreter`*. The actual computation, performed by the extending function can be provided in plain C++, with or without the ManyDSL API.*

While ManyDSL-aware functions are complicated to write, they provide a flexible way of defining extensions to the core DeepCPS. Both standard C functions and ManyDSL-aware functions appear as ordinary function value in DeepCPS. They can be invoked or passed as arguments, indistinguishably to functions defined directly in DeepCPS.

6.1.3 Memory Operations

In most previous examples we use DeepCPS in a purely functional way. However DeepCPS can be used in imperative programming as well and supports memory operations. In the previous example Listing 6.7 we represent a matrix as a memory-allocated array.

Naturally, a matrix could be represented as a giant tuple or tuple-of-tuples to simplify the 2-dimensional access. However, when the size is potentially big and accessing it through dynamic indexing, an array type is better suited. DeepCPS does not provide an array type though, since at the low level those behave similarly to a memory region.

DeepCPS supports pointer arithmetic, explicit allocation and deallocation. Currently, it does not provide any smart pointers that would automatically provide deallocation when the pointer is no longer referenced. This aligns with the philosophy of DeepCPS as a low-level language that does nothing automatically.

To further exemplify imperative programming in DeepCPS, consider the example Listing 6.9. The provided code loads a 2-dimensional, monochromatic image I,

```
let matrixGet(matrix, dimX, dimY, x, y, return) {
  matrix+(y*dimX+x) . (addr)
  load . addr (value)
  return . value
}
let matrixSet(matrix, dimX, dimY, x, y, value, return) {
  matrix+(y*dimX+x) . (addr)
  store . addr value ()
  return .
}

loadMatrix . kernel (kDimX, kDimY, kernel)
(kDimX-1)/2 . (kOffsetX)
(kDimY-1)/2 . (kOffsetY)
-kOffsetX . (negkOffsetX)
-kOffsetY . (negkOffsetY)
loadMatrix . image (iDimX, iDimY, image)
allocMatrix . iDimX iDimY (output)

//for each output image pixel
formd . [0,0] [iDimX,iDimY] (outx, outy, break, continue) {

  //for each kernel element
  formd . 0 [negkOffsetX,negkOffsetY] [kOffsetX,kOffsetY]
    (acc, kx, ky, break, continue) {
    outx+kx . (inx)
    outy+ky . (iny)

    //check out of bounds
    inx>=0 and inx<iDimX and iny>=0 and iny<iDimY . (ok)
    ifelse . ok
      (endif) {
        //load input image value
        matrixGet . image iDimX iDimY inx iny (inValue)

        //load kernel value
        kx+kOffsetX . (kx)
        ky+kOffsetY . (ky)
        matrixGet . kernel kDimX kDimY kx ky (kernelValue)

        //add to the accumulator acc
        acc+inValue*kernelValue . (acc)
        endif . acc
      }
      (endif) {//else
        endif . acc
      }
    (acc) //endif
    continue . acc
  } (acc)
  matrixSet . output iDimX iDimY outx outy acc ()
  continue .
} ()
```

Listing 6.9: *2D convolution in DeepCPS.* `matrixSet`*/*`matrixGet` *perform pointer computation and access the memory through* `load` *and* `store`*. These are unpure functions, and while they return nothing, they cannot be removed from the code. Similarly, the outher multidimentional for loop* `formd` *(defined in Listing 6.5) has no data explicitly exchanged between the loops, but its body is unpure as well.*

a kernel K and performs a convolution $I * K$. In this simple version we ignore problems arising at the borders of the image.

We use the multidimensional loops defined in Listing 6.5. The outer loop iterates over all pixels of the input image. Each iteration, for given coordinates $(\texttt{outx}, \texttt{outy})$ computes the value of the convolution $\texttt{image} * \texttt{kernel}$. The result is stored in the image `output`.

The inner loop iterates over 2-dimensional kernel, in the range between corners $\left(-\left\lfloor\frac{\texttt{kDimX}-1}{2}\right\rfloor, -\left\lfloor\frac{\texttt{kDimY}-1}{2}\right\rfloor\right)$ and $\left(\left\lfloor\frac{\texttt{kDimX}-1}{2}\right\rfloor, \left\lfloor\frac{\texttt{kDimY}-1}{2}\right\rfloor\right)$. The inner `formd` function takes an additional accumulator argument that is initially set to 0. This argument is passed to all iterations of the loop. The value `acc` is increased by the product between pixel value and kernel value, and the result is returned through the loop continuation. The final result, also named `acc`, is stored into the corresponding pixel of the output image.

Let us compare the outer and inner loops with respect of their effect on the program. Each iteration of the inner loop takes an additional argument (`acc`), modifies it producing a new value, and then returns it. This is how loops in purely functional programming usually look like.

The outher loop function however takes no additional arguments and produces no result. It would seem that each iteration of the outer loop has no effect on the rest of the program. However, since we modify a memory within the loop by writing to the `output` image, the loop cannot be simply removed.

6.2 Staging

In Section 4.1 we introduce the concept of Dynamic Staging and identify basic patterns in Section 4.1.7 We now show more involved examples, how to use staging and the patterns.

Loop unrolling

One of the simplest tasks for staging is loop unrolling. In this example we use the generic `for` loop defined in Listing 6.3, but any other loop can be used as well. We use the fragment chaining pattern of Listing 4.9. The loop acts as the master function and a single code fragment — the loop body — is called repeatedly. The loop body takes and returns a stage chain control variable.

In the example Listing 6.10 we use a chain `s` to unroll a simple for loop. In each iteration the same code fragment is invoked. Its body is executed, but a block between `@sin:` and `[sout]` is deferred. The `sout` chain control variable is returned to the loop function and used as `sin` in the next iteration. This way, the next instance of the `sin`-`sout` block is chained together with the previous one.

It should be noted that no change is required to the general `for` loop that was given in Listing 6.3. We merely use the general tuple `!args` to pass the stage parameter in and out of the body of the loop.

```
for . s from to (sin, idx, continue) {
        ...
        ... ()[jmp]
  @sin: ...
        ...
        ... ()[sout]
  @jmp: ...
        ...
        continue . sout
} (sfinal)
...
```

Listing 6.10: *Using staging to unroll a loop. The fragment of the loop body staged in the* s *chain remains intact for every iteration of the loop.*

Staging and Memory

The staging mechanism of DeepCPS is explicit and does not rely upon compiler deciding what is and what is not safe. This puts dynamic staging in a unique position where it allows staging over unpure functions and memory operations. It is the programmer who defines what is and what is not safe. The programmer has additional information about the intended behavior of the program, often not expressed in code in any way.

While the user has more freedom compared to similar solutions, it also puts more responsibility upon them. DeepCPS does not try to catch any misuse of staging. Code that is correct in natural staging may lead to a crash when used with staging. Apart from purely functional hazards induced by staging, such as infinite execution, executing the memory operations in a different order may lead to new kinds of problems.

- Accessing a variable before it is allocated, or after it is deleted.
- Accessing a variable allocated on different machine – when one staging chain is interpreted and another is compiled and then executed elsewhere.

Consider an example Listing 6.11. It is a similar convolution function as given in Listing 6.9, but uses staging to specialize itself for a specific kernel.

The function takes a file name containing a description for a kernel. With it, it resolves the parts of the convolution function that depend only on the kernel and not on the input image. This involves computing on the dimensions of the kernel, unrolling the inner loop and loading values from the kernel array.

Technically, this is done with two staging chains: the function-time `ft`, and a preparation `prep`. The inner `formd` uses the loop unrolling pattern explained earlier in this chapter: The loop itself is executed in `prep` stage chain, but the `ft` staging chain control variables are passed into each iteration of the loop. The parts staged in the `ft` chain remain intact, and reappear one after another in the produced code, shown in Listing 6.12.

The result of executing the `prep` chain is given in Listing 6.12. It is assumed that a file is named, which contains a simple Sobel-Feldman Operator kernel, typically used in image processing for edge detection [123]:

```
let convolve(kernelName, return)[prep0] {
  return . (inName, outName, return)[ft0]
  @prep0:
  loadMatrix . kernelName (kDimX, kDimY, kernel)
  (kDimX-1)/2 . (kOffsetX)
  (kDimY-1)/2 . (kOffsetY)
  -kOffsetX . (negkOffsetX)
  -kOffsetY . (negkOffsetY)[prep1]
  @ft0:
  loadMatrix . inName (iDimX, iDimY, image)
  allocMatrix . iDimX iDimY (output)

  //for each output image pixel
  formd . [0,0] [iDimX,iDimY] (outx, outy, break, continue)[ft1] {

    @prep1:
    //for each kernel element
    formd . ft1 0 [negkOffsetX,negkOffsetY] [kOffsetX,kOffsetY]
      (ft2, acc, kx, ky, break, continue)[prep2] {
      @prep2:
      let [prep3];       //Stage Branch pattern, Section 4.1.7
      @ft2:
      outx+kx . (inx)
      outy+ky . (iny)

      //check out of bounds
      inx>=0 and inx<iDimX and iny>=0 and iny<iDimY . (ok)
      ifelse . ok
        (endif) {
          //load input image value
          matrixGet . image iDimX iDimY inx iny (inValue)[ft2]

          @prep2:
          //load kernel value
          kx+kOffsetX . (kx)
          ky+kOffsetY . (ky)
          matrixGet . kernel kDimX kDimY kx ky (kernelValue)

          //add to the accumulator acc
          @ft2:
          acc+inValue*kernelValue . (acc)
          endif . acc
        }
        (endif) {//else
          endif . acc
        }
      (acc)[ft3] //endif
      @prep3:
      continue . ft3 acc
    } (ft4, acc)
    @ft4:
    matrixSet . output iDimX iDimY outx outy acc ()
    continue .
  } ()
  return .
}
```

Listing 6.11: *Staged convolution, specialized for a specific kernel given in a file* `kernelName`*. We use staging despite unpure functions, such as* `matrixGet` *and* `matrixSet` *are present (defined in Listing 6.9). Operations on the kernel matrix are performed in* `prep` *staging chain, while image matrix is used only at* `ft` *staging chain.*

```
// after applying kernel 3x3:
// [ -1 0 +1 ]
// [ -2 0 +2 ]
// [ -1 - +1 ]

(inName, outName, return)
  loadMatrix . inName (iDimX, iDimY, image)
  allocMatrix . iDimX iDimY (output)

  //for each output image pixel
  formd . [0,0] [iDimX,iDimY] (outx, outy, break, continue) {

    outx-1 . (inx)
    outy-1 . (iny)
    inx>=0 and inx<iDimX and iny>=0 and iny<iDimY . (ok)
    ifelse . ok
      (endif) {
        matrixGet . image iDimX iDimY inx iny (inValue)
        0+inValue*(-1) . (acc)
        endif . acc
      }
      (endif) { endif . 0 }
    (acc)

    outx . (inx)
    outy-1 . (iny)
    inx>=0 and inx<iDimX and iny>=0 and iny<iDimY . (ok)
    ifelse . ok
      (endif) {
        matrixGet . image iDimX iDimY inx iny (inValue)
        acc+inValue*0 . (acc)
        endif . acc
      }
      (endif) { endif . 0 }
    (acc)

    outx+1 . (inx)
    outy-1 . (iny)
    inx>=0 and inx<iDimX and iny>=0 and iny<iDimY . (ok)
    ifelse . ok
      (endif) {
        matrixGet . image iDimX iDimY inx iny (inValue)
        acc+inValue*1 . (acc)
        endif . acc
      }
      (endif) { endif . 0 }
    (acc)

    ... 6 more similar blocks ...

    matrixSet . output iDimX iDimY outx outy acc ()
    continue .
  } ()
  return .
}
```

Listing 6.12: *Result of calling* `convolve` *from Listing 6.11 with a name of a file containing the Sobel-Feldman Operator.*

```
3 3
-1 0 +1
-2 0 +2
-1 0 +1
```

With the contents of the `kernel` matrix populated with the actual data, the inner `formd` loop is unrolled, producing 9 copies of the body. The matrix contents appear as constants within expressions incrementing the accumulator data `acc`.

Other languages that perform partial evaluation would have difficulties to perform such optimization, because it is generally unsafe. The compiler requires guarantees regarding pointer aliasing and has no knowledge of what the C functions in `loadMatrix` actually do. Only by moving the responsibility towards the user, as it is done in DeepCPS, such an evaluation is possible.

At this point DeepCPS makes no attempts to further optimize the code. However, when the underlying compiler is invoked, it performs the typical LLVM optimizations, which include constant propagation and dead code elimination. With it, the unrolled version can be further simplified, e.g. by completely eliminating the code that ends up multiplying with a constant 0.

Dependent Staging

Let us now show another unique use of dynamic staging: In some scenarios it is worth for staging to change, depending on certain conditions or as a result of some computation. Consider for example Listing 6.13 — a recursive implementation of a 1D Cooley-Tukey algorithm for the Fast Fourier Transform [63]. We assume that the input size of the image is a power of 2.

It is a divide and conquer algorithm of complexity $\Theta(n \log n)$. In each recursion step `recfft` calls itself twice for the first and second half of the array. The recursion is terminated when the size reaches 1. When each of the halves of the array have their respective FFT computed, the merge step occurs, which linearly iterates over the whole data range to compute the FFT of the combined array.

In practice the above function is slow and can be improved. The problem is, that the recursive `recfft` is called several times for very small values of `size`. In these calls, the same trigonometric functions are recomputed each time. Compared to those, the actual computation on the `image` takes little time.

We could specialize `fft` for any input value `size`, even when the actual contents of `image` remain unknown. This would unroll the recursion and compute all the trigonometric functions beforehand. The problem is, that we obtain $n \log n$ instructions, which for high n values may again slow down the program. What we need is to choose whether we unroll or not depending on the size n.

To make the most informative decision for staging, we could check the properties of the target hardware or even perform profiling. However, for the purpose of our example let us take a simple assumption instead: We unroll the recursion when `size` is at most 8. For other values of n we still specialize `recfft`, but keep the recursive calls intact.

With dynamic staging, such condition can be encoded as in the example List-

```
fix reindex(image, size, return) { ... } in
fix recfft(image, size, return) {
  size=1 . (done)
  if . done () { return . image } ()
  size/2 . (sizeh)
  recfft(image, sizeh) . ()
  recfft(image+sizeh, sizeh) . ()
  sin(pi/sizeh) . (wtemp)
  -2 * sqr(wtemp) . (wpr)
  -sin(2*pi/sizeh) . (wpi)
  for . wtemp 1 0 0 sizeh (wtemp, wr, wi, idx, break, continue) {
    image[i+sizeh]*wr - image[i+sizeh+1]*wi . (tempr)
    image[i+sizeh]*wi + image[i+sizeh+1]*wr . (tempi)
    store(image[i+sh], image[i]-tempr) . ()
    store(image[i+sh+1], image[i+1]-tempi) . ()
    store(image[i], image[i]+tempr) . ()
    store(image[i+1], image[i+1]+tempi) . ()
    wr + wr*wpr - wi*wpi . (nextwr)
    wi + wi*wpr + wr*wpi . (nextwi)
    continue . wr nextwr nextwi
  } (wtemp, wr, wi)
  return . image
} in
let fft(image, size, return) {
  reindex . image size (image)
  recfft . image size (image)
  return .
} ...
```

Listing 6.13: *An implementation of 1D Cooley-Tukey algorithm for FFT in DeepCPS without staging. The* `reindex` *is a straightforward, function which reindexes the input data according to the algorithm. We skip its implementation for the sake of brevity.*

```
fix reindex(image, size, return) { ... } in
fix recfft(image, size, return) {
  @size:
  size=1 . (done)
  if . done () { return . image } ()
  size/2 . (sizeh)
  let recffth(image, return) {                    Specialization of recfft
    @sizeh:
    recfft . image sizeh (image)
    @return:
    return . image
  }
  size<=8 . (dounroll)                            Decision on staging
  ifelse . dounroll
    (endif) { endif . always }
    (endif) { endif . image }
  (unroll)
  @unroll:                                        Dependently staged recursive call
  recffth(image) . (imgleft)
  recffth(image+sizeh) . (imgright)[recdone]
  @imgleft&imgright: let [imgrec];
  @sizeh:
  sin(pi/sizeh) . (wtemp)
  -2 * sqr(wtemp) . (wpr)
  -sin(2*pi/sizeh) . (wpi)[const]
  @const & recdone:
  for . imgrec wtemp 1 0 0 sizeh
    (imgrec, wtemp, wr, wi, idx, break, continue)[s] {
    @imgrec:
    image[i+sizeh]*wr - image[i+sizeh+1]*wi . (tempr)
    image[i+sizeh]*wi + image[i+sizeh+1]*wr . (tempi)
    store(image[i+sh], image[i]-tempr) . ()
    store(image[i+sh+1], image[i+1]-tempi) . ()
    store(image[i], image[i]+tempr) . ()
    store(image[i+1], image[i+1]+tempi) . ()[imgrec]
    @s:
    wr + wr*wpr - wi*wpi . (nextwr)
    wi + wi*wpr + wr*wpi . (nextwi)
    continue . wr nextwr nextwi
  } (imgrec, wtemp, wr, wi)
  return . imgrec
} in
let fftgen(image, size, return) {
  @image:
  reindex . image size (image)
  @size:
  recfft . image size (imgdone)
  @imgdone:
  return .
}
let fft1024(image, return) {
  @fft:
  fft . image 1024 return
}
```

Listing 6.14: *An implementation of 1D Cooley-Tukey algorithm for FFT in DeepCPS with staging. Each call to* `recfft` *causes a recursive specialization* `recffth` *of itself for* `size/2`*. Moreover, if* `size` *is at most 8, the recursive calls are actually invoked and the remaining code spliced into the context of parent* `recfft`*.*

ing 6.14. We define recfft function as before, but with an intent that it may be invoked with unknown image where only its size is known.

First, within the function recfft we define a specialization recffth, that performs the very same computation as the parent function, but for half the image size. Through the recursive specializations, all necessary versions of recffth are created. Note that in each step of the recursion we create exactly one specialization, despite invoking it twice later in the algorithm. We do not duplicate code — for each value of size there is exactly one function. We ultimately create $\Theta(\log n)$ versions of specialized recfft.

Before we invoke the specialized functions, we decide at which stage that invocation should happen. To achieve this we check the value of size and depending on its value we invoke the continuation (unroll) . . ., passing either always or image as a stage parameter.

The recursive calls return a stage parameter imgleft and imgright, indicating that the respective halves of the image have been processed. We need these parameters to ensure that all image operations — which are memory operations — are actually performed in the correct order. The trailing line

```
@imgleft&imgright: let [imgrec];
```

defines a new staging variable imgrec which becomes active when both left and right images are processed.

With the value size known, the trigonometric functions can be evaluated early, incorporating constants into the body of specialized recfft. The loop however, exactly the same as with recursive calls, is unrolled only for size at most 8. For any higher size, the loop executes only after the recursive calls are performed.

Within the loop, when unrolling, values wr and wi – the principal roots of unity W from the Cooley-Tukey's algorithm – are computed early, since those do not depend on the image data. On the other hand, the instructions that alter image are staged to be executed in the correct order, after the previous recursive calls.

Finally, at the end, we return the last imgrec staging variable, which becomes active only when all image operations of the function are completed.

We should note that the staging and specialization part in this example is a little bit more involved, but is still manageable and — for the first time — uses dependent staging. Despite the unusual additions, the core logic of the algorithm remains the same.

Reverse Capture

Consider the following scenario: We want to specialize a generic function fgen, defining a new, specialized version fspec. As in many examples above, including the introductory power72 from Listing 4.5, this is done by invoking fgen from within the body of fspec, before fspec itself is called. The code of fgen that is skipped over due to staging is spliced into the context of fspec.

However, as the generic fgen is executed as part of the fspec body, it may

```
let fgen(..., fn[resT,auxT] return) {
    ...
    return . result auxiliary
}
alloc . auxT (auxT* mem)[memAv]
@memAv: let [memSet];
let fspec(return) {
    @fgen: fgen . ... (result, auxiliary)
    @memAv: store . mem auxiliary ()
    @result: return . result
}
@memSet:
load . mem (auxiliary)
...
```

Listing 6.15: *The reverse capture pattern – retrieving an auxilary information from within a body of a function. Function `fgen` is specialized within `fspec`, producing an unknown `result` and known `auxiliary`. The `auxiliary` value is stored in `mem` and then retrieved outside of the `fspec` body.*

compute some other valuable information, such as the typing information. Unfortunately, such auxiliary information i is spliced together with the remaining code into the body of `fspec`. Consequently, outside of `fspec`, i cannot be accessed easily. This may be a problem if the content of i hints on how `fspec` may be used — for example, when a language implements type deduction for `fspec` based on i.

In other words, we want to obtain a value from within a lambda and use it outside of its scope. We refer to it as *reverse capture*, as the operation is a reverse of an inner lambda capturing a value from its outer context.

In a pure lambda calculus, reverse capture is not possible. Instead, in a pure functional programming, one should keep the actual f and the computation of i separate. This may be inconvenient for the programmer, however, when the computation flow of f and i are the same or similar. Moreover, partial results obtained during computation of f and i may incrementally interact with each other. Thus keeping the computation for f and i in separate functions may be impractical.

In DeepCPS we propose an unpure but more pragmatic solution. As shown in Listing 6.15 we use memory to obtain i, while maintaining `fspec`. Suppose that the generic `fgen` returns two values `result` and `auxiliary`, of type `resT` and `auxT` respectively. The value `result` is passed into the context of `fspec` to be later passed into the continuation `return`. The `auxiliary` value however is stored into a previously defined memory location `mem`, and never passed to `return`. The `mem` can be then accessed outside of the `fspec` body, to retrieve the `auxiliary` value.

Care has to be made to ensure that the value is actually stored into `mem` before it is retrieved. For that we use a stage branch from Section 4.1.7: The stage variable `memAv` ensures that the value is set before stage `memSet` is active. All loads of `mem` are then performed after `memSet`.

Moreover, it is important that `@fgen:` is executed before `memAv`. Otherwise, `store` would fail when trying to store a symbolic value under `mem`.

```
let specialize(fgen, fargs, return) {
                //fgen has type fn[ ... , fn[ retT, auxT ]]
  typeof(fgen).last.last . (auxT)
  alloc . auxT (mem)
  @mem: let [memSet];
  let fspec(!args, return) {
    fargs . !args (!argsComb)
    @fgen: fgen . !argsComb (result, auxiliary)
    @mem: store . mem auxiliary ()
    @result: return . result
  }
  @memSet:
  load . mem (auxiliary)
  return . fspec auxiliary
}
```

Listing 6.16: *The abstraction over the reverse capture pattern. The generic* `fgen` *takes an arbitrary set of arguments* `!argsComb`*, and a returning continuation that accepts the* `result` *and the* `auxiliary` *result. The function* `fargs` *combines the* `fspec` *arguments with arbitrary constants over which* `fgen` *is specialized. The type of the auxiliary (*`auxT`*) is extracted from the type of* `fgen`*, so that proper memory is allocated.*

In theory, with a small extension to `alloc`, it is possible to store symbolic values in memory as well. In practice it can lead to hard-to-track bugs and in general should be avoided. Recall from Section 5.1.2 that when a function is invoked, a deep cloning of all nested code is performed. The node representing the symbolic `auxiliary` value is cloned whenever any lambda containing it is invoked (continuations excluded, as explained in Section 5.1.3). In particular, any lambda encapsulating the whole section of code we are currently discussing could potentially issue cloning. However, such cloning procedure would not alter the values stored in `mem`. It would always point to the original, and not the cloned symbolic value.

For that reason, using memory for symbolic values is safe only if it can be guaraneed that the referred value is not cloned between storing and loading.

The above pattern for specializing a function and extracting an auxiliary information can be abstracted out, for example the way it is done in Listing 6.16. This specialization function takes the generic version `fgen` and produces its specialization `fspec`. As the function `fgen` is partially evaluated, it produces the auxiliary information of type `auxT`.

6.3 Compilation and Performance

The ManyDSL underlying compilation module uses Thorin [84] and LLVM [80] as described in Section 5.5. Let us show how the compilation can be performed. We also want to verify that ManyDSL can actually produce efficient machine code.

Consider a function `P` representing a program that needs to be compiled. In order to obtain an LLVM module containing the function `P` it suffices to write:

```
#include<compiler.cps>
let P(...) { ... }
```

```
...
compiler_create . (c)
compiler_add . c P "P" ()
compiler_finalize . c ()
compiler_optimize . c ()
compiler_emitLLVM . c ("P.bc")
compiler_destroy . c ()
```

All `compiler_*` values are imported C functions, provided by the header file `compiler.cps`. In the six steps we perform:

- Create a new compiler context
- Add a new closure `P` into the context. It also automatically adds all dependencies that are captured by it. The string value is an arbitrary name we assign to the function that is used for linking purposes.
- Finalize the compilation, indicating that no other functions are to be added. At this step we translate `P` into Thorin.
- Invoke transformations within Thorin, so that code can be generated.
- Emit LLVM bytecode into a specified file
- Close and remove the compiler context.

In the current version of ManyDSL it is not possible to invoke the produced code from DeepCPS. However, doing so should not impose any theoretical or technological obstacles.

In our case, we produced a complete, standalone program. In order to evaluate the performance of it we performed a series of tests. We have implemented 4 problems discussed in this work, namely:

- The power function of Listing 4.6a computing a value x^{72}.
- A convolution of an unknown 1D data set of size 256 with a kernel [-1, -2, 0, +2, +1], using an implementation similar to one discussed in Listing 6.11
- The same convolution with an additional code to handle borders. The staged version moves the border cases outside the main loop.
- The Cooley-Tukey algorithm for FFT from Listing 6.14 applied to a 1D signal of length 2^{24}.

Each of these problems was implemented in four different variants:

- Standard C++ dynamic code without any explicit specialization
- In C++ using metaprogramming to force partial evaluation at compile-time
- In DeepCPS but without staging
- In DeepCPS using staging

Table 6.1: *Absolute execution times of different implementations of the power function and 1D convolution. Two variants of the convolution are considered, one using mirroring as boundary handling (BH) and one without boundary handling.*

	power	convolution BH: none	convolution BH: mirror	FFT
Number of iterations	10^8	10^6	10^6	1
Standard C++	1402 ms	1079 ms	1171 ms	2510 ms
Templated C++	192 ms	419 ms	421 ms	551 ms
CPS without staging	1685 ms	1012 ms	1047 ms	2584 ms
CPS with staging	194 ms	413 ms	420 ms	573 ms

The C++ sources have been compiled into LLVM bytecode using clang 3.3. The CPS implementations are compiled into LLVM bytecode using Thorin [84]. Regardless of the source, C++ or CPS, all versions of the bytecode are then compiled and linked into native code using LLVM with optimizations enabled (`opt -O3`).

All produced functions have been called from the same main testing loop, repeating the call several times. The parameters were flagged as volatile to prevent any further optimizations between the loop iterations. Without the volatile flag, the compiler was often able to detect that we compute the same thing all the time and completely remove the testing loop, defeating the purpose of the test.

The produced executables have been run on a computer equipped with an Intel i7-2600K 3.4 GHz CPU and 8 GB of DDR3 (1333MHz) memory, running 64-bit Ubuntu 12.04.2 LTS. Each executable was run 5 times and the average timing was used.

From Table 6.1 we can see, unsurprisingly, that partially evaluated code can perform significantly better than unspecialized code. What is important, however, is that the code produced by ManyDSL is comparable in performance to the existing and well-adopted C++ compiler.

We do not beat C++ in terms of performance, but that was never our goal. Instead, we beat it in terms of staging mechanisms: We use highly flexible dynamic staging, while C++ requires the inconvenient template metaprogramming approach – that we discussed in Section 2.3.1. We show that despite the additional abstraction layers and use of CPS, we remain competitive.

6.4 Language Creation Challenges

In Section 4.2 we explained what tools ManyDSL provides to facilitate language creation. We did not explain in detail how these can be actually used. In this chapter we show typical challenges encountered when creating a language and how ManyDSL can be used to resolve them. We particularly seek solutions that can be used across many languages. Such generic solutions make the language definitions easier to maintain, but also simplify code that interacts with multiple

languages at once.

6.4.1 Multiple Passes

DeepCPS is a single-pass language. Consequently, all names must be defined before they are used. In a higher-level language this may be inconvenient. If in a given scope multiple objects or functions are mutually recursive, writing all declarations before their definitions may be cumbersome.

This raises a question, how to incorporate a multi-pass language in the Syntax Directed Execution format? We choose to use builders, defined in Section 4.3.1, to find a solution. Observe, that the order of builder invocations and the order in which fragment functions are connected are independent. Builders are typically called in the order of parsing. However, each action can contain multiple builders, and the generated fragments are no longer constrained by the parse order.

Multiple pass language can be achieved by creating two or more series of fragment functions. As the source is parsed, fragment functions are glued to these two series, for example `Decl` and `Def`. Upon completion, `Decl` containing all the declarations is put before `Def` having all the appropriate definitions.

As an example, consider a simple DSL for specifying directed graphs. Each entry consists of a head vertex, followed by an edge list naming all adjacent vertices:

```
VertexName -> [VertexName [ , VertexName [ , ... ]]] ;
```

The DSL should read such a graph description and form an indexed tuple describing this graph. An entry at an index i should be an adjacency list of the vertex i. For example, given an input

```
graph {
  Start -> X, Y;
  X -> Y;
  Y -> X;
}
```

our DSL should produce a graph-describing tuple:

```
[[2,3], [3], [2]]
```

In the Listing 6.17 we present a complete implementation of such language. In the main rule `Graph` we begin building not one, but two functions: `Decl` and `Def`. These are passed to a list of entries.

Incremental fragments of `Decl` have 2 recurring parameters: `names` containing all recorded vertex names, and `exit` continuation function to be invoked at the very end. Each graph entry names a new vertex in a graph, followed by its adjacency list. The `Decl` is modified only by the vertex name. A new name is added at the end of the `names` tuple.

Fragments of `Def` have 3 recurring parameters: `names` and `exit` are the same as in `Def`. The `Def` fragments assume that `names` already contains the names of all vertices in the graph. A new parameter `graph` is a tuple representation of the graph that we build. The `Def` function is modified when reading the adjacency list.

```
Graph->(P) ::= "graph" "{"
  ()->(Decl,Def) {    //create two fragment chains: Decl and Def
    build . (exit, cont) { cont . [] exit } (Decl)
    build . (names, exit, cont) { cont . [] names exit } (Def)
    return . Decl Def
  }
  (Decl,Def)->EntryList->(Decl,Def)
  (Decl, Def)->(P) { //here, as the whole graph is parsed, connect Decl to Def
    build . (graph, names, exit) {
      exit . graph
    } (Fend)
    glue . Decl Def (FMain)
    glue . FMain FEnd (FComplete)
    glue . FComplete (P)
    return . P
  } "}";

(Decl,Def)->EntryList->(Decl,Def) ::= epsilon;
(Decl,Def)->EntryList->(Decl,Def) ::= Entry EntryList;

(Decl,Def)->Entry->(Decl,Def) ::= Name->(name)
  (name,Decl)->(Decl) {
      //new vertex name is encountered, add it to the tuple of names
    build . (names, exit, cont) {
      concatenate(names,[name]) . (names)
      cont . names exit
    } (F)
    glue . Decl F return
  }
  "->"
  (Def)->(Def) {
    build . (graph, names, exit, cont) {
      cont . [] graph names exit
    } (F)
    glue . Def F return
  }
  (Def)->Adj->(Def)
  (Def)->(Def) {
    build . (adj, graph, names, exit, cont) {
      concatenate(graph, [adj]) . (graph)
      cont . graph names exit
    } (F)
    glue . Def F return
  } ";" ;

(Def)->Adj->(Def) ::= epsilon;
(Def)->Adj->(Def) ::= AdjElement AdjCont;
(Def)->AdjCont->(Def) ::= epsilon;
(Def)->AdjCont->(Def) ::= "," AdjElement AdjCont;

(Def)->AdjElement->(Def) ::= Name->(name)
  (name,Def)->(Def) {
      //new adjacent name - look it up in names array, filled in Decl
    build . (adj, graph, names, exit, cont) {
      find(names,name) . (idx)      //such that names[idx]=name
      concatenate(adj,idx) . (adj)
      cont . adj graph names exit
    } (F)
    glue . Decl F return
  }
```

Listing 6.17: *DSL for a graph description. Multi-pass is achieved by creating two fragment functions:* `Decl` *and* `Def` *and gluing them together only at the end of the description.*

Within the Adj rule, an additional recurring parameter adj is temporarily added to Def and it holds the numeric values of the adjacent vertices. For each parsed name in the adjacent vertex list, we add new code to the Def function: The names tuple is searched to find the specified name and obtain the corresponding index. The result is appended at the end of adj. When, at the end, the adjacency list of the current vertex is completed, it is appended to the graph tuple.

Finally, at the very end — that is, the last action of the Graph rule, Decl and Def functions are glued together. This puts all the vertex naming operations in front of all adjacency list building. This way, all insertions to names happen before lookups. The assumption we made that names is complete within Def fragments is in fact true. As long as the graph description is correct, all searches in the names tuple succeed even if in the source code vertex declaration name appears after its use.

This way, a language that originally would be considered to need two passes, is actually built in a single pass. Multiple functions are being built while parsing, and at the very end they are combined to form the program.

6.4.2 Building Recursive Functions

In Section 4.3.5 we have shown how fix nodes can be incorporated within builders. Unfortunately, within native DeepCPS the fix cannot be built in a generic way, the node cannot be split into smaller components or built incrementally. In this section we show how this limitation can be bypassed by manipulating ManyDSL nodes directly. We also show, how to manipulate ManyDSL nodes to form a recursion without using fix nodes at all.

Building Fix Nodes

We use ManyDSL-aware C++ functions designed to directly manipulate fix nodes. These functions are provided by us as a library, and their precise definition is provided in the Appendix B.

- `fixDeclare (n, return)` creates a new, incomplete, fix node. The function taking 2 arguments: a number n and a returning continuation. It creates a new ManyDSL fix node, with n functions. The functions however have no definitions yet. The fix node is also missing the continuation code, which normally appears after the **`in`**.

- `fixDefine (fixnode, idx, definition, return)` modifies the fix node by adding a single function definition at a given integer position `idx`. It returns through the parameter-less continuation.

- `fixFinish (fixnode, in_caluse, return)` finalizes the fix node once all definitions are provided. The `fixFinish` takes the remaining fix node continuation code (`inclause`), given as an parameter-less lambda. With it, fix node is then complete. It is put into another lambda, and passed into `return` continuation as a proper ManyDSL value.

```
Graph->(P) ::= "graph" "{"
  ()->(Decl,Def) {
    build . (exit, cont) { cont . [] exit } (Decl)
    build . (names, exit, cont) {
      arity(names) . (size)
      fixDeclare . size (graph, fcts)
      cont . graph names fcts exit
    } (Def)
    return . Decl Def
  }
  (Decl,Def)->EntryList->(Decl,Def)
  (Decl, Def)->(P) {
    build . (graph, names, fcts, exit) {
      fixFinish . graph () {
        exit . fcts
      } (fixlambda)
      fixlambda .
    } (Fend)
    glue . Decl Def (FMain)
    glue . FMain FEnd (FComplete)
    glue . FComplete (P)
    return . P
  } "}";
```

Listing 6.18: *The definition of the top grammar rules of a DSL for graph description. In this version, each vertex corresponds to a function returning its adjacency list. All these mutually-recursive functions are put manually into a single* **`fix`** *node of TR.*

In order to show how to use the above functions, let us again consider the graph-building language from before. This time however, each vertex is a function, which returns (through continuation) the adjacency list. Thus, for the same input:

```
graph {
  Start -> X, Y;
  X -> Y;
  Y -> X;
}
```

we produce something equivalent to:

```
(graph) {
  fix
    f1 = (return) { return . [f2,f3] }
    f2 = (return) { return . [f3] }
    f3 = (return) { return . [f2] }
  in
    graph . [f1,f2,f3]
}
```

The basic grammar of the language remains the same as in the Listing 6.17. The language changes only in terms of the actions.

Within the `Graph` production, in Listing 6.18, we modify the actions to initialize and finalize fix node creation. At the beginning of `Def`, we create the fix node. At that time the `names` tuple is already populated and we know its size. The `fixDeclare` creates a fix node which is then used as the graph representation. In this version, the `Def` also gets another recurring parameter: `fcts` which is a tuple containing all declared functions within the fix node.

At the end of the `Def` we assume that the graph representation is complete

```
(Def)->AdjElement->(Def) ::= Name->(name)
  (name,Def)->(Def) {
    build . (adj, graph, names, exit, cont) {
      find(names,name) . (idx)       //such that names[idx]=name
      fcts[idx] . (fct)
      concatenate(adj,fct) . (adj)
      cont . adj graph names exit
    } (F)
    glue . Decl F return
  }
```

Listing 6.19: *The construction of the adjacency list in a graph-describing DSL using functions as vertices. The difference from Listing 6.17 is minimal: We use the additional array `fcts` to retrieve the actual function to be put into the adjacency list. In this version, each vertex corresponds to a function returning its adjacency list. All these mutually-recursive functions are put manually into a single **`fix`** node of TR.*

```
(Decl,Def)->Entry->(Decl,Def) ::= Name->(name)
  ...
  (Def)->Adj->(Def)
  (Def,name)->(Def) {
    build . (adj, graph, names, fcts, exit, cont) {
      "find(names,name)" . (idx)
      fixDefine . graph idx
        (return) { return . adj }
      ()
      cont . graph names fcts exit
    } (F)
    glue . Def F return
  } ";" ;
```

Listing 6.20: *Adding the completed adjacency list into the graph described as the **`fix`** node.*

within the fix node. We specify the **`in`** clause which returns the tuple containing all the nodes of the graph — that is, all the functions of the fix node. The produced, complete fix node is put into a function `fixlambda` by the `fixFinish` which we immediately invoke.

Building the adjacency list does not differ much. We start with an empty `adj` tuple as before. When a new name is encountered, we no longer add an index into the adjacency. Instead, we reference a function of the fix node under that index, and store that function in the adjacency tuple, as in Listing 6.19.

Finally, when the adjacency list is ready, we add the entry to the graph. Instead of concatenating it, we use the `fixDefine` function to specify the entry in the fix node (Listing 6.20).

Ultimately, as it can be seen the above example, with the fix node manipulation functions, the changes are only superficial and the underlying algorithm for the DSL remains unchained. The produced program contains fix node manipulation functions, which is equivalent to what we wanted to obtain.

If one wishes to obtain the intended output exactly, with a single fix node, staging can be used. The fix manipulation functions should be executed similarly as it was done in Section 4.3.2. In that setting, the actual fix manipulation would be performed in the build-time chain, while the lambda containing the fix

```
(graph) {
  allocNode . (m1)                                   Region 1: node allocation
  allocNode . (m2)
  allocNode . (m3)
  let [decl];
  loadNode . m1 (px1)                    Region 2: load references in late stage
  loadNode . m2 (px2)
  loadNode . m3 (px3)
  let f1(return) { return . [px2,px3] };            Region 3: use references
  let f2(return) { return . [px3] };
  let f3(return) { return . [px2] };
  let [done];
  @decl:
  storeNode . m1 f1 ()                  Region 4: set symbolic references early
  storeNode . m2 f2 ()
  storeNode . m3 f3 ()
  @done:
  graph . [f1, f2, f3]          Region 5: return mutually-recursive functions
}
```

Listing 6.21: *Creating recursive functions without using* **`fix`** *node. Instead, we use memory to hold references to TR nodes. With the help of staging regions 1 and 4 (blue) are executed first – we store symbolic* `f*` *values in the memory nodes* `m*`*. Only then, regions 2, 3 and finally 5 (green) are executed in that order. Within them, we retrieve the recursive references stored in the memory nodes* `m*`*.*

(`fixlambda`) would be executed in the function-time chain.

Looping through Staging

Another way to build a series of mutually-recursive functions incrementally is to refer to these functions without the fix node at all. While DeepCPS syntax does not allow that, this limitation can be bypassed through staging and memory operations. As we explained in Section 5.1.1, Target Representation does not actually treat fix nodes in any special way and interpretation would work even if such nodes were not present at all.

The key idea of the approach is to use staging and memory to pass references up to early parts of the program. As before, we use three ManyDSL-aware C++ functions. This time however, they are much simpler than before:

- `allocNode (return)` allocates memory to store any ManyDSL value node and returns through the continuation with the pointer to the node as a result.
- `loadNode (nodeptr, return)` loads the ManyDSL value node from a node pointer, and returns the actual value through the continuation.
- `storeNode (nodeptr, value, return)` takes the value and stores a reference to it under the ManyDSL node pointer `nodeptr`.

Consider the graph-creating DSL as in the previous examples. This time, instead of creating a **`fix`** node we generate code as shown in Listing 6.21. It consists of 5 characteristic regions. In order of their appearance in the code, we have:

1. For each graph vertex we declare a memory location storing a single ManyDSL value (`m1`, `m2`, `m3`).
2. We load the function references that are set in region 4.
3. We use the function references to provide full function definitions.
4. We store the defined functions into the memory locations
5. We return the full graph, represented as a tuple of functions.

Because of staging, the order of execution does not match the order of appearance. Right after the region 1 we use the stage branch pattern Section 4.1.7 and set region 4 to execute before regions 2 and 3. While region 4 is being executed, the values `f1`, `f2`, and `f3` are not yet set and remain merely as symbolic values. Still, these values can be stored in our special memory locations and are loaded as proxy values `px1`, `px2`, `px3` in region 2.

Since formally the above code has no recursion, building it is straightforward. For each vertex, a line of code is added to regions 1, 2 and 4. The adjacency list is used to build a function, referencing only proxy values `px1`, `px2`, and `px3`. At the very end of the building, all regions are glued together and appropriate staging is added.

Similarly to previous example, the whole node manipulation can be staged early, in the build-time chain, ensuring that memory nodes and proxy values are elided.

6.4.3 Environment

One of important aspects of almost any language is name binding. What scopes does a DSL provide and how can names be mapped to values? How can multiple languages be combined if they provide different scoping rules?

The Syntax-Directed Execution scheme (Section 4.2.2) does not provide, nor put any constrain on solutions. As with everything else, name lookups are defined by arbitrary language semantics. Typically, the code handling variable naming is staged to be executed early in order to avoid overhead later on. However, both static and dynamic name resolution can be handled by almost the same code.

In a purely functional approach, binding can be handled by a map M embedded as a closure in functions `insert` and `lookup`. We refer to such a pair `[insert,lookup]` as an *environment*. In the example Listing 6.22, such a map is implemented merely as a tuple containing pair of `[name, value]`. The `insert` function adds a new element at the end of the tuple, and recursively creates a new `[insert, lookup]` closure with the updated M tuple. The `lookup` finds the given name in the tuple and returns the associated value.

The actual underlying implementation could be made more efficient, by employing search trees, dictionaries or hash tables. The precise structure of M is however completely orthogonal to the problem and we abstract away from it.

The above solution is just an example of how name binding can be realized. Depending on the user requirements, the environment may behave differently.

```
fix makeEnv = (M, return) {
  let insert(str, value, return) {
    concatenate(M,[[str,value]]) . (newM)
    makeEnv . newM return
  }
  let lookup(str, return) {
    foreach . M (entry, idx, break, continue) {
      entry[0] . (entryStr)
      entryStr==str . (match)
      if . match
        () {
          entry[1] . return
        }
        continue
    } ()
    return . []
  }
  return . insert lookup
} in
let newEnv(return) { makeEnv . [] return }
```

Listing 6.22: *Simplest name binding in DeepCPS. A name-value pairs are represented in a tuple M, within the closures* `insert` *and* `lookup`*. Upon insertion, a new* `insert,lookup` *pair is created that includes an updated tuple.*

For example it may include name mangling, overloading, or automatic insertion of new variables for names not previously encountered. In the following sections we present how some of those features can be implemented.

It should be noted, that when the lookup fails to find an element, an empty tuple is returned as a result. Failure to find a name in an environment does not throw an error. This is good practice, because in some languages and contexts it is indeed not an error and a well-defined behavior is assigned to such a case. We make it more apparent in the next examples.

Environment in C++

Environment operations are frequent during parsing. If their implementation is inefficient, they may noticeably increase the parsing time. In the view of this, it may be practical to implement an environment in C++ and extend DeepCPS with it. In Section 6.1.2 we have shown how extensions in general can be added. Let us look how environments can be implemented as an extension.

We use an object-oriented approach: The environment is represented as a single mutable object `env` with accessor functions `insert` and `lookup`. These functions take the object as an argument, rather than capturing it as a closure. The functions themselves are constant — they are not replaced with a newer version whenever the map changes.

In the Listing 6.23 we define three functions for environment manipulation: Its creation, insertion, and lookup. The environment is represented as a standard C++ map, which in DeepCPS is encapsulated as a `Pointer` value. Obviously, the map uses a string as its key, but the type of its value is problematic. Not only the value type may be different from one entry to another, we also want the environment to store values that are yet unknown, represented symbolically

```
typedef std::map<std::string, sp<Value>>() EnvMap;
extern "C" void envNew(Interpreter* interpreter, Action* action) {
  Closure* continuation = action->getInlineArg(0);
  EnvMap* env = new EnvMap();
  Pointer* penv = new Pointer(env);
  interpreter->bindArgument(continuation, 0, penv);
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
extern "C" void envInsert(Interpreter* interpreter, Action* action)
    {
  EnvMap* env = digPtr<EnvMap>(action->getArg(0));
  std::string name = digString(action->getArg(1));
  Value* val = action->getArg(2);
  Closure* continuation = action->getInlineArg(3);
  env->insert(name, val);
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
extern "C" void envLookup(Interpreter* interpreter, Action* action)
    {
  EnvMap* env = digPtr<EnvMap>(action->getArg(0));
  std::string name = digString(action->getArg(1));
  Closure* continuation = action->getInlineArg(2);
  EnvMap::iterator it = env->find(name);
  if (it == env->end())
    interpreter->bindArgument(continuation, 0, new Tuple());
  else
    interpreter->bindArgument(continuation, 0, it->second());
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
```

Listing 6.23: *Environment implementation in C++ as an extension of DeepCPS. `envNew` creates a new object and packs it into TR pointer. `envInsert` puts a new `Value` object as-is into the environment map. `envLookup` retrieves the value. Upon failure, it returns an empty tuple.*

in the code. For that reason, we do not store actual values in the memory, but pointers to the underlying nodes of the ManyDSL representation. Such pointers may refer not only to concrete values but also to unbound parameters. In the map such pointers are stored as `sp<Value>`, which are strong pointer to any `Value` node. The pointer is declared as strong to ensure that the node is not removed by the ManyDSL garbage collector, as explained in Section 5.1.1.

Storing an explicit pointer to ManyDSL node may cause an implementation-specific pitfall that a programmer must be aware of: As we explained in Section 5.1.2, our implementation of ManyDSL interpreter performs a full copy of a function whenever it is invoked. However, an environment object is not aware of this process. If an environment references a node that is cloned, it remains pointing to the original. Depending on the situation this may or may not be a desired effect.

To ensure correctness, one should store the reference after cloning and not before. In the context of program building, as described in Section 4.3.1, environment maps should be filled only *after* the code has been assembled from all the semantic actions. Only then, possibly in a separate staging chain, all environment insertions and lookups should be performed. Intermixing function building and environment insertions may cause lookups to refer to nodes which are part of the semantic action itself and not its instanced clone that builds up the program.

Environment Polymorphism

Before discussing more involved examples of environments, let us ask an important question: How to use multiple kinds of environments within a single program? We want to abstract away how `insert` and `lookup` functions work for a given environment object.

For example, consider two versions of the environment:

- The first one is a `SimpleEnvironment`, which works exactly how it was shown in the previous example – Listing 6.23.
- The second one, `MangledEnvironment`, mangles the input names in some way, e.g. to contain only alphanumeric characters.

Both versions create their own kinds of insert and lookup functions, for example `simpleEnvNew`, `mangledEnvLookup`, etc. However, when we are given an environment object `env` we do not want to manually discover, which version of these functions should be used with it. In most cases, there is actually a single correct one, and other may lead to some unexpected behavior.

This kind of problem can be solved through dynamic polymorphism – a key concept in object oriented languages. In a functional language such as DeepCPS, dynamic polymorphism can be achieved by storing the proper functions as a part of the object – for example as it is done in functions `newSimple` and `newEmngled` in Listing 6.24. We then define a single global dispatch function `insert` and `lookup` which take the version stored with the environment object and call

```
let newSimple(return) {
  simpleEnvNew . envData
  return . [envData, simpleEnvInsert, simpleEnvLookup]
}
let newMangled(return) {
  mangledEnvNew . envData
  return . [envData, mangledEnvInsert, mangledEnvLookup]
}
let insert(env, str, value, return) {
  env[0] . (envData)
  env[1] . (envInsert)
  envInsert . envData str value return
}
let lookup(env, str, return) {
  env[0] . (envData)
  env[2] . (envLookup)
  envLookup . envData str return
}
```

Listing 6.24: *Polymorphism achieved by storing accessor functions together with data in a single tuple. Each environment tuple consists of 3 elements: the data pointer and the appropriate* `insert` *and* `lookup` *functions.*
Do not confuse `newSimple` *with* `simpleEnvNew`*: The former creates the appropriate tuple; the latter initializes only the data, as it was done in the previous examples.*

it. This way the use of `env` remains unchanged: the user invokes the dispatch function and obtain the result. However, it is the object itself that defines which actual function is being used.

Nested Environments

Many programming languages organize their data in a form of scopes. Certain named variables and entities are accessible only within a certain scope, while others may be accessible globally. Typically, scopes are nested. When a variable name is not found in one scope, it is searched again in its parent scope.

This relation between scopes can be described as a type of environment. For each scope in the code we create a new environment object. Each environment performs its local lookups, but when it fails, it delegates the job to the parent one. However, for this to work, environments need to be somehow connected.

In the example Listing 6.25 we create a special environment which connects two existing environments named `parent` and `child`. Each lookup request is first searched in `child` and if it fails, it is invoked in the `parent`. Both child and parent may be another stacked object, forming an arbitrarily nested chain of scopes.

Environment objects can also be given names and be inserted as entries into another environment. This way, compound naming, such as in structs and classes, can be implemented.

Environment and Recursion

In Section 6.4.2 we have shown how mutually recursive functions can be built incrementally. Creating recursive references through staging is particularly

```
let stackedEnv(parent, child, return) {
  return . [child, parent]
}
let stackedInsert(env, str, value, return) {
  env[0] . (child)
  insert . child str value return
}
let stackedLookup(env, str, return) {
  env[0] . (child)
  lookup . child env str (value)
  value!=[] . (found)
  if . found
    () { return . value }
  ()
  env[1] . (parent)
  lookup . parent env str return
}
```

Listing 6.25: *A stacked environment which connects two arbitrary environments in the child-parent relation. When looking for a name, it is first searched in the child. If it fails to find it, it repeats the process in the parent.*

convenient when using an object-oriented approach for the environment, with a C++ implementation. The `allocNode`, `loadNode`, and `storeNode` functions can be completely replaced by the environment operations.

To repeat the example: We now build a graph-describing DSL. Each entry consists of a vertex name, followed by an adjacency list. We want to convert each vertex to a function, which returns (through a continuation) a tuple referencing all adjacent vertices — that is, the adjacent functions. For example, for the input:

```
graph {
  Start -> X, Y;
  X -> Y;
  Y -> X;
}
```

we want to produce something that is equivalent to:

```
(graph) {
  fix
    f1 = (return) { return . [f2,f3] }
    f2 = (return) { return . [f3] }
    f3 = (return) { return . [f2] }
  in
    graph . [f1,f2,f3]
}
```

We build the structure incrementally, with the use of staging and memory as in Listing 6.26. The role of the memory is replaced by the environment. Through staging, the execution flow jumps into the block containing insertions. Symbolic values `f1`, `f2`, and `f3` are put into the environment before they are bound to concrete function values. Later on, these values are looked up at an earlier location within the program, which allows us to form recursion.

Notice, that this approach can be used for forward declarations, but also for languages which allow functions to be used before any declaration even appeared.

It is key, however, that environment lookups are performed when the code is

```
newEnv . (env)
(graph) {
  let [decl];
  lookup . env "f1" (px1)
  lookup . env "f2" (px2)
  lookup . env "f3" (px3)
  let f1(return) { return . [px2,px3] };
  let f2(return) { return . [px3] };
  let f3(return) { return . [px2] };
  let [done];
  @env:
  insert . env "f1" f1 ()
  insert . env "f2" f2 ()
  insert . env "f3" f3 ()
  @done:
  graph . [f1, f2, f3]
}
```

Listing 6.26

already being built, preferably as a separate staging chain. If the lookups are performed during actual parsing, the necessary recursive definitions may not even be read from the source.

Auxiliary Information

An environment may store more than a mere value under a given name. It may hold additional information, such as the type of the value. We discuss how type system can be implemented in detail in Section 6.4.4. For now, let us assume a simple system with atomic types only. For every value stored in an environment, we also want to store its type. For example, in a C-like language containing a statement:

```
int value = 5;
```

we want to remember that `"value"` has a value 5, and is of type **`int`**. This can be done in three ways.

- By storing the value and type together in a tuple.
- By having two environment objects, separate for value and a type.
- By storing value and type as two entries within the same environment.
- By storing a polymorphic object which can be queried for each information component.

The first approach changes the signature of the `insert` and `lookup` functions as shown in Listing 6.27. The `insertTyped` now requires a type to be specified in addition to the value. Within the function we pack them together in a pair and use the previously defined `insert`. Similarly, `lookupTyped` is changed so that it packs the tuple and provides all the information to its continuation `return`.

While the solution is simple and intuitive, it makes it difficult for inter-language communication. Any language using such environment must adhere to this standard. If a language needs even more auxiliary information components, the

```
let insertTyped(env, name, value, type, return) {
  insert . env name [value, type] return
}
let lookupTyped(env, name, return) {
  lookup . env name (entry)
  entry[0] . (value)
  entry[1] . (type)
  return . value type
}
```

Listing 6.27: *Storing an auxiliary information (type) together with a value within a single environment, by putting all the information within a single pair.*

```
let insertTyped(env1, env2, name, value, type, return) {
  insert . env1 name value ()
  insert . env2 name type return
}
let lookupTyped(env1, env2, name, return) {
  lookup . env1 name (value)
  lookup . env2 name (type)
  return . value type
}
```

Listing 6.28: *Storing an auxiliary information (type) separately from a value, in an additional, independent environment object.*

order of their appearance in a tuple must be globally set. A language that does not use the extra information must still be aware of the new signatures of `insert` and `lookup`.

A second option is to keep multiple environment objects, one for every component of the information (Listing 6.28). This partially solves the problem of inter-language communication. A language that does not use types may simply ignore `env2` and merely use `env1` for its lookups.

Unfortunately, the usage of such environment setup becomes cumbersome and holds a high degree of redundancy. Each environment map has implicitly the same structure and differs only in terms of its stored values. In addition, multiple environments have to be created for each scope and passed around between the fragment functions.

Third solution is to store components separately but within a single environment, as in Listing 6.29. The value is stored under the name provided by the user,

```
let insertTyped(env, name, value, type, return) {
  name+".type" . (typename)
  insert . env name value ()
  insert . env typename type return
}
let lookupTyped(env, name, return) {
  name+".type" . (typename)
  lookup . env name (value)
  lookup . env typename (type)
  return . value type
}
```

Listing 6.29: *Storing an auxiliary information (type) separately from a value, in an additional, independent environment object.*

while type (and other components) gain a name suffix, e.g. ".type". Assuming that a dot cannot be part of a name provided by the user, there is no chance for a name clash caused by the suffixes.

This solution is simple to use. Languages that do not support types or other auxiliary information component simply never query for them and still get meaningful results. A single environment object is used per scope. This is the method we use internally for most of our typed languages.

However, the solution is not without hazards. Consider the following scenario:

- A typed language defines a variable named "x"
- An untyped language in a nested scope defines a variable under the same name
- Then, again a typed language tries to retrieve a variable under the name "x"

in this scenario, the variable lookup returns the second variable "x", but the query for "x.type" may succeed as well, returning the type of the first variable.

The most robust solution is for the environment to return a polymorphic object, with all information about the value and any associated information is retrieved only through the accessor functions. It is up to the language that creates this entry to decide what and how the information is stored.

The additional level of abstraction when accessing an environment entry makes use of these values that much more complex. We have chosen not to continue on this path, because at the current state of development we believe this would be over-engineering. In the future however, it may be a necessity for programming in a multi-DSL system.

Name Overloading

So far we considered environments that use the name as the only criteria for the lookup. However, since the user has freedom of defining the environment of the language, a more advanced system is possible that takes auxiliary information into the account, such as the type information. In the previous section we have shown how auxiliary information can be stored together with values in an environment. Now, we use that information to alter the lookup procedure.

Before providing examples on how this can be implemented, let us ask an important question: Who is responsible for resolving an overload? That means, if we have two entries with the same name, who decides which one is to be used in a given context?

Within a single language the above question has no real meaning. It is entirely up to the language designer to decide where the auxiliary information is being compared. However, in a multi-DSL environment with possibly different auxiliary information being used, it is vital to make such decision clear. There are two options:

```
extern "C" void envInsertMulti ...
extern "C" void envLookupMulti ...
let insertMulti(env, name, value, type, return) {
  envInsertMulti . env name [value,type] return
}
let lookupMulti(env, name, typematch, return) {
  envLookupMulti . env name (results)
  foreach . [[],0] results (found, result, idx, break, continue) {
    result[0] . (value)
    result[1] . (type)
    overloadQuality(type,typematch) . (quality)
    quality>found[1] . (better)
    if . better () {
      continue . [value, quality]
    } () {
      continue . found
    }
  } (found)
  found[0] . return
}
```

Listing 6.30: *An example of an environment, supporting overloading with respect to types. The overloading is resolved within the lookup function. C++ functions implement an environment where multiple entries under the same name can be stored, without any additional logic to it. DeepCPS implementation of* `lookupMulti` *however, iterates over all found entries, checking how well to they match to the provided type signature. An entry with the best match, as evaluated by some* `overloadQuality` *cost function, is picked. Corner cases, such as ambiguous overloading is not handled.*

- Overloading is resolved by the language that defines the entries.
- Overloading is resolved by the caller, i.e. the language where the name lookup is initialized.

In the first approach, in order to resolve the overload, the necessary information must be provided by the caller. The insert and lookup function signature must be changed to include the auxiliary information. Within the lookup we include the overloading logic. We do not stop searching at the first found entry, but continue doing so until the best alternative is found (Listing 6.30).

While this works within a single language, it may become problematic when multiple languages are connected. The auxiliary information of one language may be incompatible with another, or worse — the calling language may provide no such information at all.

For that reason we advocate the second solution: Deferring the overload to the caller. In this setting, `lookup` does not take any additional parameters, besides the name, as a search criterion. Instead, it returns a tuple of all possible candidates. Then, the caller makes a decision which version to choose from. This approach is similar to how lookup failures are handled as well, which we discussed when we first introduced environments.

This approach is not without flaws either: It does not magically fix incompatibilities between languages, but gives a better setup for the user to provide such a fix. A language X, where terms are being defined, cannot predict all languages that these terms are going to be used in. However, a language Y which uses

terms defined in previous language X has knowledge of such dependency and may provide necessary code for Y to interact with X gracefully.

6.4.4 Type System

The core DeepCPS uses a very basic type system that provides only the basic runtime type checking. However, with the power of staging, program constraints — such as type correctness — can be checked before the actual program is run. As we discussed at the end of Section 2.2.2, type checking can be expressed entirely as an auxiliary computation performed by the interpreter [52].

Representing type checking as actual computable code in a multi-staged program has many consequences:

- It bridges the gap between statically and dynamically typed languages. The same type-checking code can be chosen to run at any stage of the program, or not run at all.
- In general, auxiliary code is Touring complete, just as any other code. There is no general guarantee that auxiliary program will halt and produce a correct result. It is up to the DSL designer to ensure correctness of the type checking program.
- Complex type interference, such as type polymorphic, recursive, or dependent types is possible.

In the following, we give a few examples of how early type checking can be implemented to enrich custom DSLs. The list is not exhaustive, as there are many type systems known in the literature.

Simple Types

Simple types can be represented as auxiliary information stored together with values, for example the way we discussed in Section 6.4.3. A potential type checker is then represented as an auxiliary computation on this additional data. The role of a typical type checker is to check if types match and make necessary derivations.

For example, let us assume that a given language supports a binary `*` operator on two integers, producing another integer as a result. We expect a type checker to check the arguments and infer the result. Formally, it is defined as:

$$\frac{\Gamma \vdash t_1 : \texttt{int}, t_2 : \texttt{int}}{\Gamma \vdash t_1 * t_2 : \texttt{int}}$$

Such derivation can be represented as a computation. To keep the actual program, and type computation separate one can use builders to generate two separate functions, such as `Def` and `TC`. To achieve that we can use builders as in the example Listing 6.31.

If the executed code depends on types, e.g in a case of overloading, it is better not to separate type checking and code completely. Instead, a separate staging

```
build (ft, env, cont) {
  lookup . env "t1" (t1val)
  lookup . env "t2" (t2val)[bt]
  @ft: t1*t2 . (result)[ft]
  @bt: cont . ft env result
} (Def)
build (env, cont) {
  lookup . env "t1.type" (t1type)
  lookup . env "t2.type" (t2type)
   "t1.type"==int and "t2.type"==int  . (typesOK)
  if . typesOK () {
    cont . env int
  } () {
    print . "Type error" ()
    $exit .
  }
} (TC)
```

Listing 6.31: *Code being built for expression $t_1 * t_2$, with an assumption that terms* `t1` *and* `t2` *and their types are given in the environment. We build actual executable code in* `Def`*, separate from type checking code which is in* `TC`*. If both arguments to $*$ have the correct type, we return* **`int`** *as the type of the whole expression.*

```
build (ft, tc, env, cont) {
  lookup . env "t1" (t1val)
  lookup . env "t2" (t2val)
  lookup . env "t1.type" (t1type)
  lookup . env "t2.type" (t2type)[bt]
  @tc:
   "t1.type"=="int" and "t2.type"=="int"  . (ints)
  ifelse . ints (endif)[tc] {
    @tc&ft: t1*t2 . (result)[ft]
    @tc: endif . ft result "int"
  } (endif) {
     "t1.type"=="complex" and "t2.type"=="complex"  . (complexs)
    if . complexs ()[tc] {
      @tc&ft: [t1[0]*t2[0]-t1[1]*t2[1],
               t1[1]*t2[0]+t1[0]*t2[1]]  . (result)[ft]
      @tc: endif . ft result "complex"
    } () {
      print . Type error ()
      $exit .
    }
  } (ft, result, resultType)[tc]
  @bt:
  cont . ft tc env result resultType
} (Def)
```

Listing 6.32: *Code being built for expression $t_1 * t_2$, with an assumption that terms* `t1` *and* `t2` *and their types are given in the environment. We support the operator for* **`int`** *and* `complex` *types. Depending on types, different code is being produced. To achieve that, both type checking and actual code are produced within the same function, but we use separate stage chains to resolve type checking early.*

chain should be used. This way, type checking does not incur any overhead in the final code, while it may still influence it.

As an example, consider the same language with the $*$ operator, which may work with integers as well as complex numbers. In the example Listing 6.32, depending on the type of the arguments, different code is produced.

```
build (ft, tc, env, cont)[bt] {
  @never: let myType = [];
  @tc: insert . env "myType" myType ()
         insert . env "myType.structure" [int,int] ()[tc]
  @bt: cont . ft tc env
}
```

Listing 6.33: *The parameter* `myType` *is never bound to any concrete value. As a result a ManyDSL parameter node is placed in the environment, guaranteeing that it is never captured or mistaken with a different type with the same name. The actual type hidden behind the* `myType` *name is a pair of two integers. However, the structure is not checked when comparing types — just the identifier.*

In the above examples we used a binary operator. However, the same approach can be applied to any function with any number of parameters. Similar mechanisms can be used to step beyond atomic types, and support compound types, such as tuple and function types.

Named Types

Custom DSL type system does not rely on the underlying ManyDSL types. Instead it can use any data and computation to represent it. Consequently, types can use any kind of annotation to represent a type. This freedom, however, enables pitfalls for the DSL developer.

Consider, for example, a problem of supporting named types. Suppose that a DSL allows the user to give custom names to simple or compound types. Two types with a different name are considered different even if their content is the same.

In the previous examples, we used strings literals `"int"` and `"complex"` for the basic types. These values were stored as an auxiliary information within the environment. A naive implementation of custom type names would extend that approach: custom name literals would be stored as the auxiliary information of values.

That can be dangerous, however, if the same name is used to identify different types in two different scopes. This can lead to type name clashes. Two unrelated types may actually be identified as the same type by the type checker. It is a problem similar to name capturing and a quick solution is to manually provide necessary substitutions to avoid that.

Notice however, that DeepCPS already solves that problem for parameters. As we explained in Section 5.1.1, in TR the parameters are not referred to by name, but by a direct reference to the parameter object. This guarantees that all parameters are unique, defined just by the memory address of the object. Actual textual names are used only during parsing.

The same mechanism can be used for naming custom types, as in Listing 6.33. Instead of using textual name to represent a type, a ManyDSL node itself can be used, e.g. an unbound parameter. A built-in equality operator can still be used with symbolic values, yielding true if exactly the same parameter is referenced. With the help of staging, a parameter can remain unbound.

```
(env, head)->LambdaBody->() ::=
  "{"
  ()->(lang) {
    $executorGetLanguage . $executor (lang)
    $executorSetLanguage . $executor $deepcps ()
    return . lang
  }
  (env, head)->!StagedAction
  (lang)->() {
    $executorSetLanguage . $executor lang return
  }
  "}"
  ;
```

Listing 6.34: *Language switching in LangDSL. The language is temporarily switched from the current one to the built-in `$deepcps`. In this context, the term `StagedAction` refers to a nonterminal from DeepCPS.*

In this example, even if a variable of type `myType` leaves the scope of the textual name `"myType"`, it remains uniquely identified by the unbound `myType` parameter. The unbound parameter is cloned, effectively yielding a different type, only when the function containing the type declaration is copied.

6.4.5 Language Switching

Throughout this work we highlight that in ManyDSL languages can be combined. Let us give a concrete example how this can be achieved. Consider the syntax of the semantic action in LangDSL:
`(in_args)->(ut_args) { ... DeepCPS action... }`

Such a construct is pared by the LangDSL parser up to the opening curly brace. Then, however, a language is changed, and the body of the semantic action is read by DeepCPS parser instead. When the matching closing brace is encountered, the parser switches back to LangDSL.

The semantic action syntax is defined in `LangDSL` as follows:

- The input and output argument list are parsed.
- Based on those, a `Closure` object `head` is created using a ManyDSL-aware function. The closure object has no body.
- The input names are used to give names to `Parameter`-s building up the `head`.
- Finally, the `LambdaBody` production is invoked where the language switching occurs.

The definition of `LambdaBody` is shown in Listing 6.34. After reading an opening curly brace, an action is triggered which access the global control `$executor` object (Section 5.3). Within the action, we obtain the current language object, and replace it with the built-in `$deepcps`. With that change, we set the parser to use the DeepCPS language.

We then invoke a `StagedAction` nonterminal from within the DeepCPS language. The parser reads a portion of DeepCPS source code that builds the body of the `head` `Closure` node. Afterwards, `head` becomes a valid `Closure`, that can be executed by the interpreter at any moment.

The final step is to invoke the `$executor` again and switch parsing to the previous language, which we held in the parameter `lang`. The closing curly brace is read as part of LangDSL.

6.5 Language Example: The Array Processing DSL

In the previous section we have shown some of the problems that one can encounter when defining a DSL and how these can be addressed in ManyDSL. Let us now give a small but complete DSL example.

In Section 3.2, G3 we argue that an array-processing language can be created without an explicit loop fusion transformation. Instead, when an arithemtic expression is performed over arrays, a single loop body can be filled with instructions incrementally. Now we have the necessary tools to explicitly define such a language.

To reiterate what precisely we want to achieve: We want a DSL such that the following operations are possible:

- Define a new array with its contents: `A = [1, 1, 2, 3, 5, 8, 13];`
- Perform arithemtic operations, component-wise, on arrays: `A=B*C+D;`
- Detect errors when we try to operate on arrays of different size
- Perform operations on scalars in addition to array operations
- Obtain a scalar value from an array `A[3]`
- Define relative indexing for array operations: `A=B[-1]*C+D[+1]` (which would be equivalent to `A[i]=B[i-1]*C[i]+D[i+1]`).
- Support indirect addressing, e.g. `A=B[C]`.

Array Definition

The code written in our language shall be a sequence of instructions. One of such instruction should be a definition of a new array, in the format `<name> = <definition> ;`.

First, in Listing 6.35, we define the outher shell of the program. The entry rule of the DSL is `ArrayProgram`. When parsing is succesful the rule produces a new function `P` that is ready to be invoked. This is a familiar pattern that we have used in other DSLs, e.g. in Section 4.3.2: The first action creates the first

```
ArrayProgram->(P) ::=
  ()->(F) {
    build . 1 (ft, end, cont) {
      envNew . (env)'[bt]'
      cont . ft env end
    } return
  }
  (F)->StatementList->(F)
  (F)->(program) {
    build . 0 (ft, env, end) {
      @ft: end .
    } (Fend)
    glue . F Fend (Fcomplete)
    finalize . Fcomplete return
  };
```

Listing 6.35: *The top level of the array DSL. The produced program* `P` *consists of a series of statements.*

```
(F)->StatementList->(F) ::=
  lassoc<Statement, ";",
    (left, right)->(val) { glue . left right return }
  >;
  (F,val)->(F) { glue . F val return }
```

Listing 6.36: *The statement list is defined as an expression using the binary left-associative operator* `";"` *over the* `Statement` *terms. This approach allows us to reuse the previously defined* `lassoc` *grammar abstraction, instead of explicit grammar rules.*

fragment function that all pieces of the program are glued to. In the last action the `finalize` the function (Listing 4.29) seals the accumulated fragments and converts them to a single program function. In between lies the `StatementList` that incrementally appends new fragments to the program body.

The statement list, that defines the bulk of the produced program, is defined in Listing 6.36 using the `lassoc` grammar abstraction. Recall from Section 4.2.5 that `lassoc` defines a binary left-associative operator, which in our case is the semicolon. The action of the operator is to simply glue the new statement (`right` argument) with the program already produced (`left` argument). When defined like this, the last statement in the list does not need the trailing semicolon.

A single statement in Listing 6.37 is an assignment of the form `Name = <Expr>`. The statement produces a new fragment `Fstmt` that is returned back to `StatementList` and ultimately glued to the `F`. Each statement fragment takes and returns three arguments: `ft`, `env`, `end` in addition to the continuation `cont`:

- The `ft` is the function-time staging chain variable. Operations staged upon `ft` are executed when the generated program is executed.
- The `env` is the environment object variable.
- The `end` is the continuation to be invoked by the genreated program in order to end it.

When parsing an actual expression we use two additional parameters:

```
token Name [[:alpha:]][[:alnum:]]*
Statement->(Fstmt) ::= Name->(name) "=" Expr->(Fexpr)
  (name,Fexpr)->(Fstmt) {
    build . 1 (ft, expr, arity, env, end, cont) {
      '@ft:'
        let '[ft]' value = expr;
      '@bt:'
      env.insert(name,value) . (env)
      env.insert(name+".arity",arity) . (env)
      cont . ft env end
    } (FExprEnd)
    glue . Fexpr FExprEnd return
  };
```

Listing 6.37: *The single statement of the form* `Name = <Expr>`*. The value of the expression is given in the fragment function* `FExpr` *which provides 2 additional arguments:* `expr` *and* `arity`*. The action of the statement finishes the expression, by putting its result into the environment* `env`*. In the end, we continue to* `cont` *with only 3 values that are glued to fragments from the* `ArrayProgram` *and other* `Statement`*-s.*

- The `expr` is the array value produced at function-time.
- The `arity` stores the arity of the produced array at build-time. It is used both for checking and controlling how the number of iterations that component-wise operators need to perform.

Once the `Expr` rule finishes, the calling `Statement` puts the results `expr` and `arity` into the environment `env`. The value `expr` is stored under the given name. The auxiliary information `arity` adds a suffix `".arity"` to the name and is put into the same environment.

The expression `Expr` at the moment can be only a constant array value given explicitly. In Listing 6.38 we are again using the `lassoc` with the comma operator in order to incrementally build the list. The `Singleton` rule creates an array of size 1, while the `lassoc` action connects two existing arrays into a single, bigger one. Effectively, we glue the fragments in a tree-like fashion, where each comma operator has two children: `left`, `right`, and a single `parent`.

Component-wise operations

At this point it is possible to define new arrays in our DSL. Now we would like to define component-wise operations on these arrays. To do that, we need to add more productions for the `Expr` term.

First, let us define in Listing 6.39 the barebone arithmetic operations, similarly to how it is done in MinusDiv grammar from Section 4.2.4. We define what operations are possible and what is their precedence, but we do not define their semantics just yet. It is provided by the rules `Connect` which connects two expression subtrees, and `Reference` which provides a reference to an array name.

In Listing 6.40 we provide the actual semantics. It does not differ much from the scalar setting: the only difference is that all fragment functions take an additional `index` parameter which is used when dereferencing an array. At this

```
token Number [+|-][[:digit:]]+;
Singleton->(F) ::= Number->(V)
  (V)->(F) {
    build . 0 (ft, env, end, parent) {
      parent . ft [V] 1 env end
    } return
  };
 Expr->(F) ::= "["
  lassoc<Singleton, ",",
    (left, right)->(val) {
      build . 2 (ft, env, end, parent, left, right) {
        left . ft env end (ft, Lexpr, Larity, env, end)
        right . ft env end (ft, Rexpr, Rarity, env, end)
        Larity+Rarity . (arity)
        concatenate(Lexpr,Rexpr) . (expr)
        parent . ft expr arity env end
      } (Op)
      glue . Op left (Op)
      glue . Op right (Op)
      return . Op
    }>
  "]"
  (val)->(F) {
    build . 1 (ft, env, end, cont) {
      val . ft env end cont
    } return
  }
  ;
```

Listing 6.38: *Parser for a constant array expression [<number>, <number>, ... , <number>]. Each number creates a fragment of arity 0, producing a singleton array that is returned to the continuation* `parent`*. Each comma operator combines its children into a bigger array. At the very end, we obtain a fragment* `val` *of arity 0 that constitutes the whole array. The last action produces a fragment of arity 1, supplying the unknown* `cont` *as the parent to* `val`*.*

```
Priority1->(op) ::= "*" ()->(op) { return . $add };
Priority1->(op) ::= "/" ()->(op) { return . $sub };
Priority2->(op) ::= "+" ()->(op) { return . $mul };
Priority2->(op) ::= "-" ()->(op) { return . $div };

ExprAtIndex1->(F) ::= lassoc<ExprAtIndex2, Priority2, Connect>;
ExprAtIndex2->(F) ::= lassoc<ValueAtIndex, Priority1, Connect>;
ValueAtIndex->(F) ::= ( ExprAtIndex1 );
ValueAtIndex->(F) ::= Reference;
```

Listing 6.39: *The barebone grammar of math expressions in array DSL. This defines the structure and the operator precedence, but the actual semantic operation is hidden in* `Connect`*.*

```
(left,right,op)->Connect->(F) ::= (left,right,op)->(F) {
  build . 2 (ft, env, index, parent, left, right) {
    left . ft env index (ft, env, Lvalue, Larity)
    right . ft env index (ft, env, Rvalue, Rarity)[bt]
    @ft: op . Lvalue Rvalue (value)[ft]
    @bt: parent . ft env value Larity
  } (F)
  glue . F left (F)
  glue . F right (F)
  return . F
};

Reference->(F) ::= Name->(name) (name)->(F) {
  build . 0 (ft, env, end, index, parent) {
    env.lookup(name) . (A)
    env.lookup(name+".arity") . (arity)[bt]
    @ft:
    A[index] . (value)[ft]
    @bt:
    parent . ft env end value arity
  } return
};
```

Listing 6.40: *The semantics of the math expressions in array DSL. All fragment function take an additional* index *parameter which is used to dereference each component in the array objects.*

point index is not defined anywhere, but the intention is that it is provided by the for loop we are building around our expression.

At this point any basic mathematical expression can be parsed by the ExprAtIndex1, producing a piece of code containing the computational code. The rule cannot be used directly as an Expr however. An Expr represents an expression on the whole array, while ExprAtIndex1 creates fragments for a scalar computation under a given array index. In order to connect one to another, one must produce the for loop, with the fragment of ExprAtIndex1 used as a body.

The function produced by ExprAtIndex1 not only performs the computation on the components, however. Its secondary job is to check the arity of the accessed arrays. This auxiliary information is necessary to define the limits of the for loop. The fragment function should be invoked twice: Before the loop to define its limits and in the loop to perform the computation.

This problem in stageless functional programming would require ExprAtIndex1 to return not one but *two* functions – one for arity computation, and the other for the actual array operation. Alternatively, one would create an AST for the underlying expression and inspect it twice, extracting a different information each time. However, in DeepCPS, representing this two-pass computation in a single function is possible thanks to staging.

The fragment returned by ExprAtIndex1 – the Fbody expects no further fragment continuations, since its code is a complete evaluation of an expression. We use the specialize function performing the reverse capture from Listing 6.16 to finalize and execute the fragment, specializing it with respect to bt and env. As the specialized version of the body bodySpec is created, the auxiliary information about the arity is also computed.

```
Expr->(F) ::= ExprAtIndex1->(Fbody) (Fbody)->(F) {
  build . 1 (ft, expr, arity , env, end, cont) {
    specialize .
      (ft, end, index, return) {
        Fbody . [] (body)
        body . ft env end index (ft, env, value, arity)
        return . [ft, value] arity
      }
      (ft, index, end, return) { return . ft env end end index }
      (bodySpec, arity)[bt]
    @ft:
    for . [] 0 arity (array, index, continue)[ft] {
      @bt: bodySspec . ft index (result)
           split(result) . (ft, value)[bt]
      @ft: concatenate(array,[value]) . (array)
           continue . array
    } (arr)[ft]
    @bt:
    cont . ft arr arity env end
  } (Ffor)
  return . Ffor
}
```

Listing 6.41: *The single* `for` *loop created for the whole array expression. The* `Fbody` *fragment – complete at this stage – is specialized, leaving only the* `ft` *code intact. During specialization the* `arity` *value is computed, that is reverse-captured to specify the size of the* `for` *loop.*

With `bodySpec` and `arity` we can now build the loop. It loops from 0 to `arity` and in each step, the next component of a new `array` is computed. Once ready, it is returned back through `cont`, presumabely, to assign a name in the `Statement` rule.

Detecting Arity Errors

It is an error to try to perform the component-wise operations on arrays of different sizes. Typically such an error is resolved by annotating the objects with a type information. For an array-processing language the type system must store the array size within the data types and define the necessary rules to use this information. Then, in a separate compiler phase the types are checked to find any misuse of the annotated data objects.

In our approach, the necessary information is already stored in the environment `env` at build-time. The only change needed is to perform the actual check at build-time when the arithmetic operation is performed. The needed changes are highlighted in Listing 6.42.

The arity computation is abstracted in a separate function. Yet, the call to it is explicit from within the production action.

Scalar Operations

The simplest way of handling scalar values is to immediately promote them to an indexed value as in example Listing 6.43. The produced code fragment can be immediately used in the `ExprAtIndex` rules, forming bigger scalar expressions. The scalars are automatically replicated for each `index` and can be used with

```
let combineArity(Larity, Rarity, return) {
  "Larity == Rarity" . (ok)
  if . ok
    () { return . Larity }
    () {
      "print error: operation on arrays of different size" . ()
      $exit .
    }
}

(left,right,op)->Connect->(F) ::= (left,right,op)->(F) {
  build . 2 (ft, env, index, parent, left, right) {
    left . ft env index (ft, env, Lvalue, Larity)
    right . ft env index (ft, env, Rvalue, Rarity)
    combineArity . Larity Rarity (arity)
    @ft: op . Lvalue Rvalue (value)[ft]
    @bt: parent . ft env value arity
  } (F)
  glue . F left (F)
  glue . F right (F)
  return . F
};
```

Listing 6.42: *The change to semantics of the math expressions from Listing 6.40. The language now checks at build-time if the left and right array arguments are of the same size.*

```
let combineArity(Larity, Rarity, return) {
  "(Larity == Rarity) or (Larity == 0) or (Rarity == 0)" . (ok)
  if . ok
    () { max(Larity, Rarity) . return }
    () {
      "print error: operation on arrays of different size" . ()
      $exit .
    }
}

ValueAtIndex->(F) ::= Number->(v) (v)->(F) {
  build . 0 (ft, env, end, index, parent) {
    parent . ft env end v 0
  } return
};
```

Listing 6.43: *A scalar value that can be used in an array expression. The constructed function does not depend on* `index` *and we return an arity 0 to the parent.*

arrays as well. The only attention has to be made in the arity checking to handle a value of 0 in a special way.

Altering the Index

So far, all array operands of an expression take the same single index. We would like to extend the language so that the index itself may be an expression. To do that we allow an optional `[]` construct for each array reference. In Listing 6.44 we change the previously defined rule `Reference`, so that both forms `A` and `A[...]` are allowed.

In both cases, we assume that there is a child fragment function `IdxF`, which for given loop iteration, `index`, computes the actual index that a value is read from,

```
Reference->(F) ::= Name->(name) OptIdx->(IdxF) (name,IdxF)->(F) {
  build . 1 (ft, env, end, index, parent, indexFunc) {
    indexFunc . ft env index (ft, env, indexComputed, indexArity)
    env.lookup(name) . (A)
    env.lookup(name+".arity") . (arity)
    combineArity . arity indexArity . (arity)[bt]
    @ft:
    A[indexComputed] . (value)[ft]
    @bt:
    parent . ft env end value arity
  } (F)
  glue . F idxF return
};

OptIdx->(F) ::= "[" IdxExpr "]";
IdxExpr->(F) ::= ExprAtIndex1;

OptIdx->(F) ::= IdentityArray;
IdentityArray->(F) ::= ()->(F) {
  build . 0 (ft, env, end, index, parent) {
    parent . ft env end index 0
  } return
};
```

Listing 6.44: *The grammar rules for an optional square bracket index specification, which may appear after the array name. The index is another arbitrary array expression, allowing for nested indexing, e.g.* `A[B[idx]]`

i.e. `A[indexComputed]`.

Only within the rule `OptIdx` we detect if the `[]` syntax is actually used. If it is, any indexed expression is allowed. If the square braces are omitted, an identity array of unspecified arity is used.

The use of the rule `ExprAtIndex1` makes the square bracket syntax recursive. It allows us to use all DSL-specific syntax for the index expression as well. For example, complex indexing patterns, including indirect indexing such as `A[B+C[D+1]]` are valid expressions. Moreover, all code produced by such construct is also fused under a single `for` loop, guided by the common arity of all involved arrays.

To make the example complete, we also include an explicit use of the identity array, using a keyword:

```
ValueAtIndex->(F) ::= "id" IdentityArray;
```

In most typical use case, we are interested in accessing a field relative to the current index, i.e. `A[id+1]` or `A[id*2]`. To simplify this, in Listing 6.45 we introduce a special syntax for a relative indexing, allowing `id` to be skipped if it appears at the first position. The `IdxExpr` and `IdxExprCont` have a structure very similar to a left-recursive binary operation, except that the very first value is given by an epsilon-rule `IdentityArray`.

Now `A[+1]` or `A[*2]` correctly refer to a relative address. Still, more complex expressions such as `A[+B+C[+1]]` are possible.

```
IdxExpr->(F) ::= IdentityArray->(L) (L)->IdxExprCont->(F);
(F)->IdxExprCont->(F) ::= Priority1->(op) ValueAtIndex->(R)
    (L,R,op)->Connect->(F) IdxExprCont;
(F)->IdxExprCont->(F) ::= Priority2->(op) ExprAtIndex1->(R)
    (L,R,op)->Connect->(F);
```

Listing 6.45: *Grammar rule allowing relative indexing. A relative index expression skips the first argument, which defaults to the current base index. For example,* `[+2]` *is equivalent to* `[id+2]`*, where* `id` *is the current base index.*

Further Extensions to the DSL

At this point we achieved all the goals that we have set up at the beginning of the chapter. Apart from the `Expr` rule that requires the call to `specialize`, all other rules and actions are straightforward and intuitive once DeepCPS is mastered.

The array DSL could be extended further, for example:

- Allow writing to an array under a computed index, instead of sequentially.
- Allow defining arrays of infinite/arbitrary size, evaluated lazily. Currently only the `id` acts as such infinite array.

Chapter 7

Conclusion

In this work we have presented a novel and comprehensive solution to language creation. Our work is based on the following three foundations:

- *Dynamic Staging* — a novel approach to staging (Section 4.1)
- *Syntax Directed Execution Scheme* (SDE) — a functional approach for grammar production definitions (Section 4.2), as well as functional semantics specification (Section 4.3).
- *Interleaved Parsing and Execution*, allowing the user code to alter the parsing process (Section 4.2.1).

All these are explained on a theoretical level, and are all realized in the project *ManyDSL*.

Dynamic Staging

In Section 4.1 we have redefined staging, changing how the concept is seen. Staging is no longer viewed as a pass over the whole program, but as a relation between lambda headers and their bodies. This relation defines an order of execution, possibly overriding the natural order that would follow the lambda nesting structure in CPS.

The theoretical foundations of dynamic staging (Section 4.1.3) are more fundamental than the theory of MetaML [136] or LMS [116]. Built on top of CPS, we introduce the concept of staging parameter and staging expression. With the help of *Waiting* and *Active* relations, the operational semantics is reduced to a handful of rules. We do not incorporate quotation nor do we need any complex typing rules. Early/late code blocks that quotation-based staging introduces is only one of many staging patterns possible with dynamic staging (Section 4.1.7).

Despite formal simplicity, staging is a dynamic, first-class citizen of our language. This means, staging values can be a result of arbitrary computation and can

be passed as parameters between functions. Depending on parametrisation, different parts of code may execute early or late. A single generic function can be partially evaluated producing many different forms. Each of these forms can be compiled, producing many highly-optimized versions of the same generic algorithm. To our knowledge, this is the first time such level of flexibility is achieved.

It can be argued that dynamic staging lacks guarantees present in other approaches. In fact, checking correctness of dynamic staging statically can be reduced to solving the halting problem, which is undecidable. This is because of the dynamic nature of our approach.

We believe however that the dynamic staging as presented here can be used as a basis to define more complex systems. We hope that the theory given here may one day become the language we speak in when discussing staging. In the future, a well crafted set of constraints on the staging may allow the program to be checkable against faulty staging annotations, while remaining flexible and useful in practice.

Syntax Directed Execution Scheme

The basic Syntax Directed Execution Scheme (SDE), we presented in Section 4.2.2, from the theoretical standpoint does not differ much from the old Syntax Directed Translation scheme [88, 104]. The main difference is the way of thinking: We no longer translate the input into some structure such as AST, but instead we execute code based on the input. The parser itself acts as an interpreter.

Building new languages using grammars with SDE still requires a good theoretical background from the programmer. Our approach faces the same gramatical restrictions as any other LL1 parser. The difference lies in how the action code – the pieces that get executed upon succesful parsing – is represented.

In our approach language semantics are represented as regular code, encapsulated in lambda functions which we named *fragment functions*. These functions are similar to AST nodes, but only on the surface: They are being built and are connected (glued) together as such. However, unlike AST nodes their contents can be arbitrary complex, are self-defining, and can be written in an entirely functional manner. We forgo AST visitors, pattern matching, tree grammars, or any other ways that can potentially alter the semantics of the original structure. Instead, we rely solely on specialization and abstraction as the only tools for code transformation.

Typically, the fragment functions contain not only the intended semantic of the final program, but all other support behavior is spelled out explicitly as well. This includes, for example, variable lookup code or language-specific checks. The node may also include type-checking, given as an explicit computation.

The generated code is executed simply by evaluating the produced function. We use dynamic staging to define execution phases, ensure that certain operations are performed before others, and remove any unwanted overhead in the final

code. It is up to DSL designer to specify the staging. Depending on the staging context, the same piece of code can selected to run at compile-time or run-time. Consequently, the hard distinction between static and dynamic components of the language is removed, as it only depends on the values of staging parameters.

Such approach is more open-ended and natural for a programmer to express oneself, compared to previous solution that rely on AST construction. Any programming techniques and more complex algorithms can be used within the action code. The programmer is not limited by a short list of node types and their transformations.

Interleaved Parsing and Execution

The ability of ManyDSL to interleave parsing and execution is a unique property of our solution. A program can be partially parsed and then executed, before the rest is loaded. This allows the program to alter the parser on the fly. When the parsing resumes, it may use a different language, use different input or work in some other altered way. Effectively, this allows all operations that are normally controlled in a preprocessing step, to be represented uniformly in the same language (or languages) as the rest of the program.

The ability to execute a program that is only partially parsed has a profound impact on the design of the parser and interpreter — much deeper than we originally anticipated. Consequently, what was originally merely a design problem, had a significant impact on our choice of underlying algorithms (Section 5.7). In the future search of better interpreter implementation this functionality must be taken into account from the very beginning.

The ManyDSL Implementation

Last but not least, our work is accompanied with the actual implementation of the theory laid out. We have tested ManyDSL not only with the small examples shown here, but also with medium-sized multi-language projects, such as LangDSL itself. While far from commercial product quality, ManyDSL needs only a few seconds to process and interpret thousands of lines of code, spanning among a series of different languages.

7.1 Goals Evaluation

With ManyDSL fully explained, let us evaluate our goals and desired properties that we listed in the overview in Section 3 and check if they have been fulfilled. First, let us iterate over the small items.

G1: Simplicity of DSL Definitions

Is ManyDSL simple enough for an average programmer to define new DSLs?

Indeed, with the SDE scheme (Section 4.2.2) and functional building of code (Section 4.3.1) parsing resembles functional programming. Parser actions are

pieces of code with their own self-descriptive meaning.

There is no need to investigate IR structure. No explicit AST nodes are created, and optimization does not rely on their transformations.

However, writing a complete language from scratch is still not trivial. ManyDSL in its core requires its user to comprehend CPS programming and dynamic staging. The programmer must understand the limitations of LL1 parsing. He must also understand how code pieces produced by builders must be glued together to form the program.

We advocate higher-level DSL creation: Using preexisting language fragments, encapsulating typical constructs and incorporating higher-level languages rather than DeepCPS to define action bodies. All these are possible with ManyDSL, but such libraries are very limited at the moment. Only a few such constructs are provided and LangDSL currently supports only DeepCPS as its action language.

Therefore, we conclude that the goal G1 is achieved only partially. Further work is needed to bring language definitions to higher-level programming. Such work, however, no longer requires changes in ManyDSL itself, instead a proper set of support libraries is needed. With the proper set of libraries, the user would be able to define new languages as a combination of pre-existing building blocks, and use a higher-level language to define semantic actions. Ideally, they would never need to explicitly define a grammar rule, or use the underlying DeepCPS language.

G2: Freedom of Syntax

The SDE scheme from Section 4.2.2 allows the DSL creator to specify any set of tokens, coupled with any LL1 grammar for their language. Through language switching, the DSL creator has full control over which features are actually present in the language: There are no constructs imposed by ManyDSL that must be present in any DSL. If the DSL creator chooses to forbid certain programming practices, all they need is to skip grammar rules that lead to such practice from their DSL definition.

This is in contrast to more common approaches of extendible languages (Section 2.6.4), where new grammar rules may be added, but the existing ones cannot be removed.

We continue to search for ways to extend the class of grammars, possibly incorporating Parser Expression Grammars (PEG) [41] in ManyDSL in the future. PEGs are used in other metamorphic languages (Section 2.6). Nevertheless, being able to specify a LL1 grammar, independent from the host language grammar, already provides a great degree of syntactic freedom for embedded DSLs.

G3: Core Flexibility

The core language — DeepCPS — is designed with flexibility in mind. It uses Continuation Passing Style allowing the user to define arbitrary control flow

structures, as shown in Section 6.1.1. It introduces the Dynamic Staging, allowing for flexible definitions of partial execution plans. While DeepCPS embraces functional programming, it allows the user to break away from it, using memory and external C functions — all without limiting the staging capabilities.

In the context of DSL definition, in Section 4.3.3 we have shown how `builder` functions can be used to incrementally build programs with arbitrary control. In Section 6.5 we have shown how one can incrementally create the body of a `for` loop, needed for array-processing DSL example (Listing 3.1) we gave when introducing G3.

For these reason we claim, that the core of ManyDSL as well as the builder functions provide a flexible functional environment that can be used to define complex, higher-level languages.

G4: Language Modularity and Composability

ManyDSL manages languages in the form of grammars and actions. In Section 4.2.4 we have given an example of a DSL designed specifically to define languages. LangDSL produces a function which, when invoked, creates such a grammar.

However, such code — as any other — can be parametrized and pieces can be abstracted out (Section 4.2.5, Section 4.2.6). It is possible to define fragments of grammar to be used in multiple languages. This way the language can be composed from several modules, rather than being written from scratch.

We believe that these tools provide a sufficient environment for allowing different users to define reusable pieces of languages. Unfortunately, the library of such pieces is relatively small at the moment and further research in this area is needed to fully show that our goal is achieved.

G5: DSL Interoperability

ManyDSL can support any number of DSLs. Every language can incorporate an action within its grammar that orders the parser to switch to a completely different language. This way languages can be easily mixed together.

The ability to write in many DSLs is not enough to confirm DSL interoperability. We must assert that it is possible to exchange data between two program pieces written in different DSLs.

The fact that all DSLs translate to the same core language naturally enables communication between the languages. However, for the communication to be meaningful, each DSL must assume certain knowledge of the structure of the other languages it is combined with. For example, if one language looks up a variable name and finds one provided by another language — it must make assumptions on how such value is represented: Is it merely a plain value? Is it a tuple, containing a data type or some additional auxiliary information?

Different styles for naming environments were given in Section 6.4.3. The DSL authors must agree, at least partially, how to store data for other languages to

understand it.

In that respect, ManyDSL provides the means for the communication between DSLs, but does not specify a single, final protocol. Instead, the style of data representation is entirely in the hands of the DSL creators. With the help of grammar abstractions, it is possible to define inter-DSL protocols separately, encouraging DSL developers to use them. We hope that in the future, standard and more convenient ways of storing and exchanging data will be found, further facilitating the DSL interoperability.

G6: Language Sharing

With the interleaved interpretation and parsing (Section 5.7) languages can be shared as header libraries. There is no additional installation required. While it slows down the interpretation process as the language definition needs to be loaded every time the program runs, it guarantees that nothing is ever missed. In the future we hope to introduce a mechanism to precompile language definitions, speeding up the loading process. This however is not essential with respect of the goal G6, that we consider achieved.

G7: Domain-Specific Optimization

With the functional approach to grammar actions, the language designer has full control how the code is being built. Actions can contain arbitrary pieces of program, containing not only the main semantics, but any amount of auxiliary computation as well. With the help of dynamic staging, even most complex algorithms can be scheduled for execution during parsing or compilation, before the actual program is run.

The DSL creator can perform domain-specific analysis of the data and choose the best run-time code to be emitted. In Section 6.2 we have given more involved examples of using staging to benefit image analysis and signal processing. These were programs given in plain DeepCPS, but nothing stands against using the very same constructs in code that is produced by other DSLs.

G8: Generation of Efficient Code

In Section 4.3.2 we have shown how additional code coming from different language layers can be removed through dynamic staging. DSLs are able to produce code that is indistinguishable from one produced by native DeepCPS source. Furthermore, in Section 6.3 we have demonstrated that the underlying compiler can produce highly efficient machine code. The use of dynamic staging and partial evaluation is elided in the produced code.

We also introduce hardware reflection in Section 5.1.6. While the solution is basic, it provides all the necessary means for a DSL creator to define DSL optimizaitons for a specific hardware.

7.1.1 The Main Goal

In Section 7.1 we have evaluated all the minor goals we have set for ManyDSL. All of them are either fully or at least partially achieved. Does it mean that ManyDSL achieves its main goal that we have given in Section 3.1?

We are interested if ManyDSL:

- Allows for separation of concerns between compiler and DSL programmers.
- Bridges the gap between hardware, programming languages, scientific domains, and application creation.

The key element for separating DSL programmers from compiler experts is the success of G1. As we stated however, that goal is achieved only partially, and further work is needed.

The SDE scheme (Section 4.2.2) and LangDSL (Section 4.2.4) provide means for DSL creators to express themselves in a way that is natural to functional programmers. The success or failure of a programming language or a tool, however, is not determined only by its own expressiveness. The supporting libraries pay a major role as well. They allow the language to grow over time and adapt to ever-changing requirements. At this time however ManyDSL does not include many such libraries.

Another key aspect that is helpful for DSL creators is the ability to specify domain-specific optimizations as programs (G7). This too, has a functional flavor, aided by dynamic staging. Programming with staging is a new paradigm – especially in the dynamic context – that an average developer may not be familiar with. This mechanism, however, has strong theoretical foundations (Section 4.1.3) and is applicable to much wider area of cases than just language construction.

No aspect of a DSL – its syntax, nor semantics, nor domain-specific optimizaton, nor checking has to rely on inspection and pattern-matching in IR. Still, efficient code can be created, using only the functional DSL specification (G7,G8).

Creating a DSL in ManyDSL would be useless if it was later hard to incorporate in new and existing projects.

For these reasons we firmly believe that the basic approach provided by ManyDSL is the answer to the problem of language creation by non-experts. However, only through the ongoing effort of bringing and maintaining a solid set of support libraries and languages, ManyDSL will actually succeed.

In our discussion we repetedely name language creators as *DSL developers*, but this group may not necessairly be uniform. One could identify different roles within such a group. For example:

- *Domain expert* understands best what kind of problems the DSL is going to be used to solve. With the help of ManyDSL they are able to define the domain abstractions and incorporate them into a new language.

- *Language expert* understands the relation between different languages. They are most likely to focus on the integration of a given DSL with other languages already used in the project, e.g. through cross-language environments, discussed in Section 6.4.3 and G5. They may also handle the syntactic details or type system of the DSL. Note however, that while good understanding of how ManyDSL operates is needed, this still does not require knowledge of core compiler construction.
- *Hardware expert* tunes an existing DSL for a very specific hardware. By using hardware reflection (Section 5.1.6), one can check for hardware specification in an ordinary 'if' statement, and then remove this checking overhead through staging. If a given device has a functionality not supported by ManyDSL, an extension can be introduced in a form of a ManyDSL-aware function (Section 6.1.2). Because these functions are indistinguishable from existing built-in functions, in most cases ManyDSL itself does not need to be changed to support the new hardware.
- *Software engineer* tunes existing DSLs to best fit the requirements of the given project. By choosing which features of the language are present and which are not, the engineer encourages the programmer to abide to principles they have set up. This way the desired rules are not just written, but actively enforced by the used languages.

Different roles require different familiarity with ManyDSL. With the right language libraries, domain experts will not need to know the internal details of ManyDSL, fulfilling the main goal of the project. At the same time, authors of those libraries may need a deeper understanding of CPS programming, dynamic staging, and DeepCPS extensions.

7.2 Future Work

The journey into the land of language pluralism is not finished. ManyDSL marks only the beginning — a tool to make such pluralism possible. Further research is needed to explore how ManyDSL can be used in the most efficient way.

In particular, we are asking which grammar abstractions will best facilitate creation of new languages. In Section 4.2.5 we gave the foundation for the grammar composition. We gave a simple example of an function for a left- and right-associative operator. This is however a small solution, abstracting over a few productions and hiding a limitation of LL1 grammars.

A much more interesting and practical functions would cover a bigger portion of the language. We could cover, for example:

- All arithmetic expressions using standard operators over unspecified term `T`. Perhaps a more generic version would allow the designer to choose which operators are available and what is their behavior — all specified as arguments, rather than grammar rules.

- C-like scoped declarations (classes, structs, functions) followed by some body `B`.
- List declarations over term `T` and separator `S`, to be used whenever a sequence of `T` is needed. This could be used for example to define parameter lists for function declaration, argument lists for function calls, or sequences of instructions in a block.

We believe that with the right set of grammar functions, given as a library to the user, the actual, explicit grammar rules would become seldom used. Whole, complex languages could be created as a combination of functions, with explicit productions used as a top-level glue or to define the most unique constructs.

Moreover, such abstractions could define a much needed standardization.This way two DSLs would be similar enough to read them all together, but different enough to conveniently capture the unique intrinsics of their own domains. Standardization of how languages manage their entities such as variables or types would further ease the complexity of exchanging such information between different DSLs.

Ultimately, we hope that with the help of ManyDSL one will adjust languages to solve problems, rather than adjust problems and their solutions to express them in rigid languages.

Bibliography

[1] N. I. Adams IV et al. "Revised(5) Report on the Algorithmic Language Scheme". In: *ACM SIGPLAN Notices* 33.9 (Sept. 1998), pp. 26–76.

[2] A. V. Aho and S. C. Johnson. "LR Parsing". In: *ACM Computing Surveys* 6.2 (June 1974), pp. 99–124.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] Eric Allen et al. "Growing a Syntax". In: *Proceedings of Workshop on Foundations of Object-Oriented Languages (FOOL), 2009.* FOOL '09. 2009.

[5] Thorsten Altenkirch, Conor McBride, and James McKinna. *Why Dependent Types Matter*. 2005. URL: www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf. retrieved on 22.04.2016.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[7] Andrea Asperti et al. "The Matita Interactive Theorem Prover". In: *23rd International Conference on Automated Deduction*. 2011, pp. 64–69.

[8] J. W. Backus et al. *Fortran Automated Coding System For the IBM 704*. New York: International Buisness Machines Corporation, 1956.

[9] J. W. Backus et al. "Revised Report on the Algorithm Language ALGOL 60". In: *Communications of the ACM* 6.1 (Jan. 1963). Ed. by P. Naur, pp. 1–17.

[10] J. W. Backus et al. "The FORTRAN Automatic Coding System". In: *Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). Los Angeles, California: ACM, 1957, pp. 188–198.

[11] Alan Bawden. *Quasiquotation in lisp*. Tech. rep. University of Aarhus, 1999.

[12] Yaakov Belch, Sergey Ignatchenko, and Dmytro Ivanchykhin. "Project-specific Language Dialects". In: *Overload* 94 (Dec. 2009), pp. 17–25.

[13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art : The Calculus of Inductive Constructions*. Texts in theoretical computer science. Springer, 2004.

[14] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. "Lost in Translation: Formalizing Proposed Extensions to C#". In: *SIGPLAN Notices* 42.10 (Oct. 2007), pp. 479–498.

[15] Per Bjesse et al. "Lava: Hardware Design in Haskell". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. ACM, 1998, pp. 174–184.

[16] Richard J. Boulton et al. "Experience with Embedding Hardware Description Languages in HOL". In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. North-Holland Publishing Co., 1992, pp. 129–156.

[17] Alex Bourd, ed. *The OpenCL Specification, Version 2.2.* Mar. 2016. URL: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf. retrieved on 27.11.2016.

[18] Kenneth A. Bowen. “Prolog”. In: *Proceedings of the 1979 Annual Conference.* ACM ’79. ACM, 1979, pp. 14–23.

[19] Claus Brabrand and Michael I. Schwartzbach. “Growing Languages with Metamorphic Syntax Macros”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation.* PEPM ’02. ACM, 2002, pp. 31–40.

[20] Felice Cardone and Mario Coppo. “Type Inference with Recursive Types: Syntax and Semantics”. In: *Information and Computation* 92.1 (1991), pp. 48 –80.

[21] Hassan Chafi et al. “A Domain-specific Approach to Heterogeneous Parallelism”. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming.* ACM, 2011, pp. 35–46.

[22] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. In: *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control.* SIGFIDET ’74. ACM, 1974, pp. 249–264.

[23] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68.

[24] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363.

[25] Alain Colmerauer and Philippe Roussel. “History of Programming languages—II”. In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. ACM, 1996. Chap. The Birth of Prolog, pp. 331–367.

[26] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition.* The MIT Press, 2009.

[27] *CUDA C Programming Guide - v8.0.* Sept. 2016. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide. retrieved on 15.10.2016.

[28] Krzysztof Czarnecki et al. “DSL Implementation in MetaOCaml, Template Haskell, and C++”. In: *Domain-Specific Program Generation.* Ed. by Christian Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Springer, 2003, pp. 51–72.

[29] Ole-Johan Dahl. *SIMULA 67 Common Base Language.* Norwegian Computing Center, 1968.

[30] Richard Dallaway and Jonathan Ferguson. *Essential Slick.* Underscore, 2015.

[31] Luis Damas and Robin Milner. “Principal Type-schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 1982, pp. 207–212.

[32] Piotr Danilewski et al. “Specialization Through Dynamic Staging”. In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences.* GPCE 2014. 2014, pp. 103–112.

[33] Arie van Deursen and Tobias Kuipers. “Building Documentation Generators”. In: *1999 International Conference on Software Maintenance*. 1999, pp. 40–49.

[34] Edsger W. Dijkstra. “How do we tell truths that might hurt?” In: *Selected Writings on Computing: A Personal Perspective*. EWD 498. Springer-Verlag, 1982, pp. 129–131.

[35] Edsger W. Dijkstra. “The Humble Programmer”. In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866.

[36] Dawson R Engler, Wilson C Hsieh, and M Frans Kaashoek. “‘C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM. 1996, pp. 131–144.

[37] Sebastian Erdweg et al. “SugarJ: Library-based Syntactic Language Extensibility”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. ACM, 2011, pp. 391–406.

[38] Robert W. Floyd. “The Paradigms of Programming”. In: *Communications of the ACM* 22.8 (Aug. 1979), pp. 455–460.

[39] J. Fokker. “Functional Parsers”. In: *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*. Ed. by J. Jeuring and E. Meijer. Springer, 1995, pp. 1–23.

[40] Bryan Ford. “Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. 2002, pp. 36–47.

[41] Bryan Ford. “Parsing Expression Grammars: A Recognition-based Syntactic Foundation”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2004, pp. 111–122.

[42] Daniel P. Friedman, Christopher T. Haynes, and Eugene" Kohlbecker. “Program Transformation and Programming Environments”. In: ed. by Peter Pepper. Springer, 1984. Chap. Programming with Continuations, pp. 263–274.

[43] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[44] Jeremy Gibbons and Nicolas Wu. “Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl)”. In: *SIGPLAN Notices* 49.9 (Aug. 2014), pp. 339–347.

[45] Jean-Yves Girard. “Une extension de l’interpretation de Godel a l’analyse, et son application a l’elimination des coupures dans l’analyse et la theorie des types”. In: 63 (1971), pp. 63–92.

[46] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[47] Herman Heine Goldstine and Adele Goldstine. “The Electronic Numerical Integrator and Computer (ENIAC)”. English. In: *Mathematical Tables and Other Aids to Computation* 2.15 (1946), pp. 97–110.

[48] Robert Grimm. “Better Extensibility Through Modular Syntax”. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. ACM, 2006, pp. 38–51.

[49] Tobias Grosser, Armin Größlinger, and Christian Lengauer. “Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* 22.4 (2012).

[50] Timothy Hart and Mike Levin. *The New Compiler*. AI Memo 39. MIT, 1962.

[51] J. Heering et al. “The Syntax Definition Formalism SDF — Reference Manual”. In: *SIGPLAN Notices* 24.11 (Nov. 1989), pp. 43–75.

[52] Stephan Andreas Herhut. “Auxiliary Computations: A Framework for a Step-wise, Non-disruptive Introduction of Static Guarantees to Untyped Programs Using Partial Evaluation Techniques”. PhD thesis. University of Hertfordshire, 2010.

[53] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. 2nd ed. New York, NY, USA: Cambridge University Press, 2008.

[54] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60.

[55] Christian Hofer and Klaus Ostermann. “Modular Domain-specific Language Components in Scala”. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. GPCE ’10. ACM, 2010, pp. 83–92.

[56] Paul Hudak. “Modular Domain Specific Languages and Tools”. In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR ’98. IEEE Computer Society, 1998.

[57] Paul Hudak and Mark Jones. *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*. Research Report YALEU/DCS/RR-1049. New Haven, CT: Department of Computer Science, Yale University, 1994.

[58] Harry D. Huskey, M. H. Halstead, and R. McArthur. “NELIAC — Dialect of ALGOL”. In: *Communications of the ACM* 3.8 (Aug. 1960), pp. 463–468.

[59] Graham Hutton and Erik Meijer. “Monadic Parsing in Haskell”. In: *Journal of Functional Programming* 8.4 (July 1998), pp. 437–444.

[60] *Information technology – Programming languages – C++*. ISO/IEC 14882:2011.

[61] Dan Ingalls et al. “Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself”. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’97. ACM, 1997, pp. 318–326.

[62] *Initial Specifications for a Common Business Oriented Language (COBOL) for Programming Electronic Digital Computers*. Tech. rep. Department of Defense of US, Apr. 1960.

[63] John W. Tukey James W. Cooley. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301.

[64] John v. Neumann. “First draft of a report on the EDVAC”. Moore School of Electrical Engineering, University of Pennsylvania. 1945. Reprinted in *IEEE Annals of the History of Computing*, 15(4):27–75, April 1991.

[65] Stephen C. Johnson. *YACC – Yet Another Compiler-Compiler*. Tech. rep. Bell Laboratories, 1975.

[66] Simon P. Jones, E. Meijer, and D. Leijen. “Scripting COM Components in Haskell”. In: *Proceedings of the 5th International Conference on Software Reuse*. IEEE Computer Society, 1998.

[67] Manohar Jonnalagedda et al. “Staged Parser Combinators for Efficient Data Processing”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. ACM, 2014, pp. 637–653.

[68] Vojin Jovanovic et al. “Yin-yang: Concealing the Deep Embedding of DSLs”. In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. ACM, 2014, pp. 73–82.

[69] Guy Lewis Steele Jr. and Garald Jay Sussman. *Lambda: The Ultimate Imperative*. AI Memo 353. MIT, 1976.

[70] Morry Katz and Daniel Weise. “Towards a New Perspective on Partial Evaluation”. In: *PEPM’92, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 1992, pp. 29–37.

[71] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988.

[72] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*. Tech. rep. Bell Laboratories, 1977.

[73] John Kessenich and Google, eds. *The OpenGL Shading Language, Version 4.50*. 2016. URL: {https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf}. retrieved on 07.06.2016.

[74] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. ACM, 1973, pp. 194–206.

[75] Oleg Kiselyov. “Macros That Compose: Systematic Macro Programming”. In: *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. GPCE ’02. Springer-Verlag, 2002, pp. 202–217.

[76] S. C. Kleene. “On notation for ordinal numbers”. In: *Journal of Symbolic Logic* 3 (04 Dec. 1938), pp. 150–155.

[77] Donald E. Knuth. “On the Translation of Languages from Left to Right”. In: *Information and Control* 8.6 (1965), pp. 607 –639.

[78] Donald E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1984.

[79] Eugene Kohlbecker et al. “Hygienic Macro Expansion”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP ’86. ACM, 1986, pp. 151–161.

[80] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. IEEE. Mar. 2004, pp. 75–86.

[81] Daan Leijen and Erik Meijer. “Domain Specific Embedded Compilers”. In: *Proceedings of the 2nd Conference on Domain-Specific Languages*. DSL ’99. ACM, 1999, pp. 109–122.

[82] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Tech. rep. University of Utrecht, 2002.

[83] Roland Leißa, Immanuel Haffner, and Sebastian Hack. “Sierra: A SIMD extension for C++”. In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. 2014, pp. 17–24.

[84] Roland Leißa, Marcel Köster, and Sebastian Hack. “A Graph-Based Higher-Order Intermediate Representation”. In: *International Symposium on Code Generation and Optimization*. 2015, pp. 202–212.

[85] Roland Leißa et al. “Shallow Embedding of DSLs via Online Partial Evaluation”. In: *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM. Pittsburgh, PA, USA, pp. 11–20.

[86] M. E. Lesk and E. Schmidt. *Lex – A Lexical Analyzer Generator*. Tech. rep. Bell Laboratories, 1975.

[87] John R. Levine, Tony Mason, and Doug Brown. *Lex &Amp; Yacc (2Nd Ed.)* O’Reilly & Associates, Inc., 1992.

[88] P. M. Lewis II and R. E. Stearns. “Syntax-Directed Transduction”. In: *Journal of the ACM* 15.3 (July 1968), pp. 465–488.

[89] P.M. Lewis, D.J. Rosenkrantz, and R.E. Stearns. “Attributed translations”. In: *Journal of Computer and System Sciences* 9.3 (1974), pp. 279 –307.

[90] William Maddox. *Semantically-Sensitive Macroprocessing*. Tech. rep. University of California at Berkeley, 1989.

[91] *Mars Climate Orbiter Mishap Investigation Board Phase I Report*. 1999.

[92] John McCarthy. “History of LISP”. In: *SIGPLAN Notices* 13.8 (Aug. 1978), pp. 217–223.

[93] John McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 184–195.

[94] Drew V. McDermott and Gerald Jay Sussman. *The Conniver Reference Manual*. AI Memo 259a. MIT, 1972.

[95] Erik Meijer. “Server Side Web Scripting in Haskell”. In: *Journal of Functional Programming* 10.1 (Jan. 2000), pp. 1–18.

[96] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[98] Eugenio Moggi et al. “An Idealized MetaML: Simpler, and More Expressive.” In: Lecture Notes in Computer Science. Springer, 1999, pp. 193–207.

[99] David A. Moon. “Architecture of the Symbolics 3600”. In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, 1985, pp. 76–83.

[100] Kristen Nygaard and Ole-Johan Dahl. “History of Programming Languages I”. In: ed. by Richard L. Wexelblat. ACM, 1981. Chap. The Development of the SIMULA Languages, pp. 439–480.

[101] Georg Ofenbeck et al. “Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries”. In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. GPCE ’13. ACM, 2013, pp. 125–134.

[102] Gerard O’Regan. “Early Computers”. In: *A Brief History of Computing*. Springer, 2008. Chap. 3.

[103] Gerard O’Regan. “History of Programming Languages”. In: *A Brief History of Computing*. Springer, 2008. Chap. 9.

[104] Jukka Paakki. “Attribute Grammar Paradigms — a High-level Methodology in Language Implementation”. In: *ACM Computing Surveys* 27.2 (June 1995), pp. 196–255.

[105] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013.

[106] Terence Parr and Kathleen Fisher. “LL(*): The Foundation of the ANTLR Parser Generator”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 425–436.

[107] David A. Patterson and David R. Ditzel. “The Case for the Reduced Instruction Set Computer”. In: *SIGARCH Computer Architecture News* 8.6 (Oct. 1980), pp. 25–33.

[108] A. J. Perlis and K. Samelson. “Preliminary Report: International Algebraic Language”. In: *Communications of the ACM* 1.12 (Dec. 1958), pp. 8–22.

[109] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[110] Matt Pharr and William R. Mark. “ispc: A SPMD Compiler for High-Performance CPU Programming”. In: *Innovative Parallel Computing Conference*. 2012, pp. 184 –196.

[111] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002.

[112] Tadeusz Pietraszek and Chris Vanden Berghe. “Defending Against Injection Attacks Through Context-sensitive String Evaluation”. In: *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*. RAID'05. Springer-Verlag, 2006, pp. 124–145.

[113] *Programming languages — C++*. ISO/IEC 14882:1998.

[114] *Programming languages — C*. ISO/IEC 9899:2011.

[115] John C. Reynolds. “Towards a Theory of Type Structure”. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, 1974, pp. 408–423.

[116] Tiark Rompf and Martin Odersky. “Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs”. In: *Communications of the ACM* 55.6 (June 2012), pp. 121–130.

[117] Tiark Rompf et al. “Building-Blocks for Performance Oriented DSLs”. In: *Proceedings IFIP Working Conference on Domain-Specific Languages*. 2011, pp. 93–117.

[118] D. J. Rosenkrantz and R. E. Stearns. “Properties of Deterministic Top Down Grammars”. In: *Proceedings of the First Annual ACM Symposium on Theory of Computing*. ACM, 1969, pp. 165–180.

[119] Philippe Roussel. *Prolog : Manuel de Référence et d'Utilisation*. 1975.

[120] Edward Sapir. “The Status of Linguistics as a Science”. In: *Language*. Vol. 5. 4. Linguistic Society of America, 1929, pp. 207–214.

[121] D. V. Schorre. “META II a Syntax-oriented Compiler Writing Language”. In: *Proceedings of the 1964 19th ACM National Conference*. ACM, 1964.

[122] Chris Seaton. “A Programming Language Where the Syntax and Semantics Are Mutable at Runtime”. MA thesis. University of Bristol, 2007.

[123] Irwin Sobel. *History and Definition of the so-called “Sobel Operator”, more appropriately named the Sobel-Feldman Operator*. 2014. URL: http://www.researchgate.net/publication/239398674_An_Isotropic_3_3_Image_Gradient_Operator. retrieved on 17.03.2016.

[124] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. 2015. URL: https://gcc.gnu.org/onlinedocs/gccint.pdf. retrieved on 29.11.2016.

[125] Andreas Stefik and Susanna Siebert. “An Empirical Investigation into Programming Language Syntax”. In: *ACM Transactions on Computing Education* 13.4 (Nov. 2013).

[126] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Tech. rep. HPL-95-11(R.1). Hewlett-Packard Laboratories, Nov. 1995.

[127] Stephan Diehl. “A Formal Introduction to the Compilation of Java”. In: *Software — Practice and Experience* 28.3 (1998), pp. 297–327.

[128] Bjarne Stroustrup. “Evolving a Language in and for the Real World: C++ 1991-2006”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. ACM, 2007, pp. 4–1–4–59.

[129] Bjarne Stroustrup. "History of Programming languages—II". In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. ACM, 1996. Chap. A History of C++: 1979–1991, pp. 699–769.

[130] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[131] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013.

[132] Arvind K. Sujeeth. *OptiML Language Specification 0.2*. Tech. rep. Stanford University.

[133] S. Doaitse Swierstra and Luc Duponcheel. "Deterministic, Error-Correcting Combinator Parsers". In: *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, 1996, pp. 184–207.

[134] Walid Taha. "Multi-Stage Programming: Its Theory and Applications". PhD thesis. Kuwait University, 1999.

[135] Walid Taha and Tim Sheard. "MetaML and Multi-Stage Programming with Explicit Annotations". In: *Theoretical Computer Science*. ACM Press, 1999, pp. 203–217.

[136] Walid Taha and Tim Sheard. "Multi-Stage Programming with Explicit Annotations". In: *ACM SIGPLAN Notices* 32.12 (1997), pp. 203–217.

[137] Walid Taha, Zine, and Tim Sheard. "Multi-Stage Programming: Axiomatization and Type Safety". In: *Lecture Notes in Computer Science* 1443 (1998), pp. 918–929.

[138] *The jOOQ User Manual*. 2016. URL: http://www.jooq.org/doc/3.8/manual-pdf/jOOQ-manual-3.8.pdf. retrieved on 01.06.2016.

[139] *The RenderMan Interface*. Nov. 2005. URL: https://renderman.pixar.com/products/rispec/rispec_pdf/RISpec3_2.pdf. retrieved on 28.11.2016.

[140] Peter Thiemann. "An Embedded Domain-specific Language for Type-safe Server-side Web Scripting". In: *ACM Transactions on Internet Technology* 5.1 (Feb. 2005), pp. 1–46.

[141] Sam Tobin-Hochstadt et al. "Languages as Libraries". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. ACM, 2011, pp. 132–141.

[142] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985.

[143] Todd Veldhuizen. "C++ Gems". In: ed. by Stanley B. Lippman. SIGS Publications, Inc., 1996. Chap. Expression Templates, pp. 475–487.

[144] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. Indiana University Computer Science, 2003.

[145] Todd L. Veldhuizen. "C++ Templates as Partial Evaluation". In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. BRICS, 1999, pp. 13–18.

[146] Eelco Visser. "Program Transformation with Stratego/XT". In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 216–238.

[147] Philip Wadler. "How to Replace Failure by a List of Successes". In: *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985, pp. 113–128.

[148] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.2 (Apr. 1991), pp. 181–210.

[149] Daniel Weise and Roger Crew. "Programmable Syntax Macros". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. ACM, 1993, pp. 156–165.

[150] Benjamin Lee Whorf. "Science and Linguistics". In: *Technology Review*. MIT, 1940.

[151] Martin Wildmoser and Tobias Nipkow. "Certifying Machine Code Safety: Shallow versus Deep Embedding". In: *Theorem Proving in Higher Order Logics*. Ed. by K. Slind, A. Bunker, and G. Gopalakrishnan. 2004, pp. 305–320.

[152] Konrad Zuse. "Der Plankalkül". 1945.

Appendix A

Example Execution of Staged Builders

In Section 4.3.2 we have shown how staging can be used to eliminate builder overhead. We use simple `build` and `glue` functions:

```
let build
  (code, return) { return .
    (next, return) { return .
      (bt, ft, !args)
        @bt: code . ft !args (ft, !args2)[bt]
        @next: next . bt ft !args2
    }
  }

let glue
  (Fprev, Fnext, return)[g] { return .
    (next, return) {
      @g: Fnext . next (body)
          Fprev . body (body2)
      @return: return . body2
    }
  }
```

As an example on how to use these, we use a language `MinusDiv` to parse an binary expression using minus and divide operators. The language runs semantic actions given in Section 4.3.2, parsing an example input `1-4/2`. This means, that the actions are executed in the order: `S 1 4 2 / - E F`, where `S` is the Starting action, and `E`, `F` are ending and finalize actions.

A.1 Gluing

Let us focus on a subsequence of actions `2 /`, resulting in the execution of the code:

```
build .
  (ft, numbers, end, cont) {                                    code1
    cont . ft [2, numbers] end
  }
  (Fnext1)
build .
  (ft, numbers, end, cont)[bt] {                                code2
    numbers[0] . (R)
    numbers[1] . (second)
    second[0] . (L)
    second[1] . (rest)[bt]
    @ft: L/R . (result)[ft]
    @bt: cont . ft [result,rest] end result
  }
  (Fnext2)
glue . Fnext1 Fnext2 (F5)
```

In this chapter, we show step-by-step how the above code is reduced to a piece of user program without overhead.

First, the `build` function is called twice. Since only natural staging is used for the builder itself, standard substitution rules are applied. The code within the body function is replaced by the brown sections of the user code (`code1` and `code2`). The complete program reduces down to the last application, calling the `glue` function:

```
(Fprev, Fnext, return)[g] { return .                  glue
  (next, return) {
    @g: Fnext . next (body)
        Fprev . body (body2)
   @return: return . body2
  }
} .
  (next, return) { return .                                   Fnext1
    (bt, ft, !args)                                   Fnext1body
      @bt: code1 . ft !args (ft, !args2)[bt]
      @next: next . bt ft !args2
  }
  (next, return) { return .                                   Fnext2
    (bt, ft, !args)                                   Fnext2body
      @bt: code2 . ft !args (ft, !args2)[bt]
      @next: next . bt ft !args2
  }
  (F5)...
```

The italic values *code1* and *code2* are actual lambda functions. We hid it under a name for a sake of brevity. By performing the application we obtain:

```
(F5) { ... } .
  (next, return) {
    @⊤: Fnext2 . next (body)
        Fnext1 . body (body2)
   @return: return . body2
  }
```

The glue function uses staging variable g, which has been activated and is equivalent to ⊤. As a result, the application that was staged upon g — namely the *Fnext2*`. next (body)...` — is performed before the final call to `(F5) ...`.

The continuation `(body){Fnext1. body (body2)... }` is mapped to `return`. This `return` is invoked with the body of fragment function `Fnext2`:

```
(F5) { ... } .
  (next, return) {
    @⊤: (body) {
      Fnext1 . body (body2)
      @return: return . body2
    } .
      (bt, ft, !args)                                   Fnext2body
        @bt: code2 . ft !args (ft, !args2)[bt]
        @next: next . bt ft !args2
  }
```

Notice the `next` within the original *Fnext2body* is substituted by an symbolic value with the same name `next` coming from the lambda function in our program.

Invoking the continuation leads to simple substitution

$$\texttt{body} \mapsto \textit{Fnext2body}$$

```
(F5) { ... } .
  (next, return) {
    @⊤: Fnext1 . Fnext2body (body2)
    @return: return . body2
  }
```

Before continuing, let us expand the contents of *Fnext1*:

```
(F5) { ... } .
  (next, return) {
    @⊤:
    (next, return) { return .                          Fnext1
      (bt, ft, !args)
        @bt: code1 . ft !args (ft, !args2)[bt]
        @next: next . bt ft !args2
    } .
      Fnext2body (body2)
    @return: return . body2
  }
```

In the next step, we invoked *Fnext1* with arguments:

$$\texttt{next} \mapsto \textit{Fnext2body}$$
$$\texttt{return} \mapsto \texttt{(body2)...}$$

This time both arguments are concrete lambda values. Consequently, applications staged upon `next` within *Fnext1* become active.

```
(F5) { ... } .
  (next, return) {
    @⊤: (body2) {
      @return: return . body2
    } .
      (bt, ft, !args)
        @bt: code1 . ft !args (ft, !args2)[bt]
        @⊤: Fnext2body . bt ft !args2
    }
  }
```

The invocation substitutes parameters of *Fnext2body* with arguments of similar names coming from *Fnext1body*. All arguments are symbolic.

```
(F5) { ... } .
  (next, return) {
    @T: (body2) {
      @return: return . body2
    } .
      (bt, ft, !args)
        @bt: code1 . ft !args (ft, !args2)[bt]
        @bt: code2 . ft !args2 (ft, !args2)[bt]
        @next: next . bt ft !args2
    }
  }
```

The last invocation didn't activate any new applications. Consequently, the execution proceeds with the invocation of `(body2) ...` lambda function, yielding:

```
(F5) { ... } .
  (next, return) {
      @return: return .
      (bt, ft, !args) {
        @bt: code1 . ft !args (ft, !args2)[bt]
        @bt: code2 . ft !args2 (ft, !args2)[bt]
        @next: next . bt ft !args2
      }
    }
  }
```

Finally, the obtained function is passed into the `(F5) ...` continuation. Notice, that the result has the same pattern as if it was build by a single `build` function:

```
F5 =
  (next, return) { @return: return .                  fragment
    (bt, ft, !args) {                                body
      @bt: code1 . ft !args (ft, !args2)[bt]
      @bt: code2 . ft !args2 (ft, !args2)[bt]
      @next: next . bt ft !args2
    }
  }
```

A.2 Build-time Staging Chain

Two fragments have been connected into a single one named `F5`. This is however not the end of the building process. When a fragment representing an actual complete program function P, the `finalize` step is performed. Recall, from Listing 4.25 the finalize function:

```
let finalize
  (F, return) {
    F . (P)[bt]
    return .
      (!args)[ft] {
        @bt: P . bt ft !args
      }
  }
```

It triggers the build-time staging chain. Suppose that a fragment function FP contains the contents of F5:

```
FP =
  (return) { @return: return .
    (bt, ft, !args) {
      @bt: ...
      ... preceding code invocations ...
                              ... (ft, !args2)[bt]
      @bt: code1 . ft !args2 (ft, !args2)[bt]   F5
      @bt: code2 . ft !args2 (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
    }
  }
```

We invoke `finalize` function with the FP argument and some `return` continuation:

```
FP . (P)[bt]
return .
  (!args)[ft] {
    @bt: P . bt ft !args
  }
```

Calling FP returns the lambda `(bt, ft, !args)...` as P, and triggers the bt staging variable:

```
return .
  (!args)[ft] {
    @⊤: (bt, ft, !args) {
      @bt: ...
      ... preceding code invocations ...
                              ... (ft, !args2)[bt]
      @bt: code1 . ft !args2 (ft, !args2)[bt]
      @bt: code2 . ft !args2 (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
    } .
      ⊤ ft !args
  }
```

The lambda is invoked with an argument bt which is already equivalent to ⊤. Consequently, the first instruction staged upon bt becomes activated. Note that other bt within the lambda are not ⊤, because those refer the implicit staging parameters of nested continuation lambdas `(ft, !args2)[bt]...`, shadowing the original name.

```
return .
  (!args)[ft] {
    @⊤: ...
    ... preceding code invocations ...
                            ... (ft, !args2)[bt]
    @bt: code1 . ft !args2 (ft, !args2)[bt]
    @bt: code2 . ft !args2 (ft, !args2)[bt]
    ...
    ... following code fragments ...
    ...
  }
```

As a result of fragment chaining, all preceding code functions are invoked in

sequence. We are parsing a sequence `S 1 4 2 / - E F` and *code1*, *code2* correspond to `2 /` part of it. Upon finishing the preceding part we have executed the subsequence `S 1 4`. At this point the arguments are:

- The `bt` upon which the *code1* invocation is staged just became ⊤.
- `ft` corresponds to continuation of the generated code
- `!args2` is set to `[[4, [1, []]], end]` — the `numbers` and `end` recurring parameters.

We have:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: code1 . ft [[4, [1, []]], end] (ft, !args2)[bt]
    @bt: code2 . ft !args2 (ft, !args2)[bt]
    ...
    ... following code fragments ...
    ...
  }
```

The italic *code1* is actually a lambda function and the above code reads:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: (ft, numbers, end, cont) {
      cont . ft [2, numbers] end
    } .
      ft ![[4, [1, []]], end] (ft, !args2)[bt]
      @bt: code2 . ft !args2 (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
  }
```

By invoking it we substitute:

$$
\begin{aligned}
\texttt{ft} &\mapsto \texttt{ft} \\
\texttt{numbers} &\mapsto \texttt{[4, [1, []]]} && \text{(first element of } \texttt{![[4, [1, []]], end]}\text{)} \\
\texttt{end} &\mapsto \texttt{end} && \text{(second element of } \texttt{![[4, [1, []]], end]}\text{)} \\
\texttt{cont} &\mapsto \texttt{(ft, !args2)[bt]@bt:}\textit{code2}\texttt{. ...}
\end{aligned}
$$

and obtain a continuation call:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: (ft, !args2)[bt] {
      @bt: code2 . ft !args2 (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
    } .
      ft [2, [4, [1, []]]] end
  }
```

Calling the continuation we arrive to a version which does not contain `code1` at all. However, the `numbers` tuple is updated to include a new element:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: code2 . ft ![[2, [4, [1, []]]], end] (ft, !args2)[bt]
    ...
    ... following code fragments ...
    ...
  }
```

Recall that *code2* that we are about to invoke is also a concrete lambda function:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: (ft, numbers, end, cont)[bt] {
      numbers[0] . (R)
      numbers[1] . (second)
      second[0] . (L)
      second[1] . (rest)[bt]
      @ft: L/R . (result)[ft]
      @bt: cont . ft [result,rest] end result
    } .
      ft ![[2, [4, [1, []]]], end] (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
  }
```

We invoke the lambda, splitting the `![[2, [4, [1, []]]], end]` between `numbers` and `end` as before:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @⊤: ([2, [4, [1, []]]])[0] . (R)
    ([2, [4, [1, []]]])[1] . (second)
    second[0] . (L)
    second[1] . (rest)[bt]
    @ft: L/R . (result)[ft]
    @bt: (ft, !args2)[bt]
      ...
      ... following code fragments ...
      ...
    } . ft [result,rest] end result
  }
```

The 4 instructions selecting an element from a tuple are executed in a sequence, leading to substitutions:

$$\texttt{R} \mapsto 2$$

$$\texttt{second} \mapsto \texttt{[4, [1, []]}$$

$$\texttt{L} \mapsto 4$$

$$\texttt{rest} \mapsto \texttt{[1, []]}$$

It is important to note that these substitutions are performed at build-time, and are no longer required at function-time.

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @ft: 4/2 . (result)[ft]
    @⊤: (ft, !args2)[bt]
```

```
      ...
      ... following code fragments ...
      ...
    } . ft [result,[1, []]] end result
  }
```

Calling the continuation triggers the remaining part of the build-time chain, reducing the remaining part of the program to include only the function-time code. It should be noted that the `numbers` tuple now contains a symbolic value. This fact however does not harm the reduction process — all operations that depend on `numbers` values are purely symbolic at build-time. Only the `numbers` list itself must be concrete.

Finally, we obtain a reduced program:

```
return .
  (!args)[ft] {
    @ft: ... preceding generated code ...
    @ft: 4/2 . (result)[ft]
    @ft: ... following generated code ...
  }
```

With the help of dynamic staging, the selected actions of subsequence `2 /` reduced to a single instruction in the generated code.

Appendix B

Manual Building of Fix Nodes

In Section 6.4.2 we have argued that generic **fix** nodes cannot be built incrementally in native DeepCPS. This is because the **fix** node arity is a hidden property of the `Action` node that cannot be manipulated programatically from DeepCPS level. The solution is to do it at a lower level, by directly manipulating the TR. Note that throughout the dissertation we actively try to avoid such manipulation, but in this particular case we must make an exception.

We use ManyDSL-aware functions, which we introduce in Section 5.1.4, to implement the necessary functions. First, in Listing B.1 we define the `fixDeclare` ManyDSL-aware function. It creates a new empty **fix** node of arity n. Each entry is represented by an unnamed parameter.

```
extern "C"
void fixDeclare(Interpreter* interpreter, Action* action) {
  int count = dig<int>(action->getArg(0));
  Closure* continuation = action->getInlineArg(1)->as<Closure>();
  Tuple* references = new Tuple();
  Fix* fix = new Fix();
  for (size_t i=0; i<count; ++i) {
    Parameter* p = new Parameter();
    fix->addParam(p);
    references->add(p);
  }
  fix->setArgCount(count);

  SPointer* fixptr = new SPointer(fix);
  interpreter->bindArgument(continuation, 0, fixptr);
  interpreter->bindArgument(continuation, 1, references);
  interpereter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
```

Listing B.1: *The* `fixDeclare` *ManyDSL-aware function creating a new* **fix** *node. The function takes two arguments: integer cont, and a returning continuation. The continuation receives the new* **fix** *node packed into a strong-pointer object, and a tuple of unbound parameters that name the entries within the* **fix**.

```
extern "C" void fixDefine(Interpreter* interpreter, Action* action)
    {
  SPointer* ptr = action->getDerefArgConstant(0)->as<SPointer>();
  Fix* fix = ptr->get()->as<Fix>();
  int idx = dig<int>(action->getArg(1));
  Value* value = action->getDerefArg(2)->as<Value>();
  fix->setArg(idx, value);

  Closure* continuation = action->getInlineArg(3)->as<Closure>();
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
```

Listing B.2: *The `fixDefine` ManyDSL-aware function, defining one of the mutually-recursive functions of **`fix`**. The function takes four arguments: the pointer to the **`fix`** node we build, the index of the parameter we set, the arbitrary value as assign to it and the parameterless continuation.*

```
extern "C" void fixFinish(Interpreter* interpreter, Action* action)
    {
  SPointer* ptr = action->getDerefArgConstant(0)->as<SPointer>();
  Fix* fix = ptr->get()->as<Fix>();
  Value* fixEnd = action->getDerefArg(1)->as<Value>();
  Closure* continuation = action->getInlineArg(2)->as<Closure>();

  fix->addArg(fixEnd);

  Closure* fct = new Closure();
  fct->setBody(fix);
  interpreter->bindArgument(continuation, 0, fct);
  interpreter->invoke(continuation);
  action->substituteWithFollowup(continuation->body());
}
```

Listing B.3: *The `fixFinish` ManyDSL-aware function completes the creation of the **`fix`** node. It takes 3 arguments: the pointer to the **`fix`** node, the function representing the **`in`** clause, and a continuation. After assembling the **`fix`**, it is put into its own lambda and returned as a regular value.*

Note that we create a `Fix` object, but in fact it is a special case of an `Action`. We also create a tuple of all the parameters, so that they may be referred to, forming a recursion.

Then, in Listing B.2 we set a concrete value under the specified index. The operation **`fix`**`->setArg` takes the `Parameter` object at the specified index and binds the value to it. It does not trigger staging associated with that parameter.

Finally, when all parameters of the **`fix`** are set, the **`fix`** node is sealed with `fixFinish` in Listing B.3. It takes one last value that is set at the end of the argument list, representing the **`in`** clause. Finally, the produced **`fix`** node is packed into its own `Closure` object and returned as a value to the continuation.

Appendix C

Y-combinator in CPS

The Y-combinator in Lambda Calculus

A fix point combinator is a construct that models recursion. In languages that do not provide explicit recursive references, it can be defined as a higher-order function. In lambda calculus, one of the most well-known combinators is the Y-combinator:

$$\begin{aligned} Y =& \lambda f.(\lambda x.xx)R_f, \\ \text{where } R_f =& \lambda x.f(xx) \end{aligned}$$

Sometimes, Y is defined as $\lambda f.R_f R_f$. We refer to R_f as a *replication term* with a free variable f, as it causes the recursive call to unroll by replicating the callee.

Suppose that with the help of Y we want to define a recursive function $G = \lambda a....$ Within the body of λa we would like to be able to call G again, but the name is not available in that context. For that reason, we define a term G_r with a free variable r indicating the function G_r itself:

$$G_r = \lambda a.\ ....\text{body with } r....$$

We then put G_r in a lambda specifying r, with its value to be supplied by the Y combinator: We obtain:

$$F = \lambda r.G_r = \lambda r.\lambda a.\ ....\text{body with } r....$$

We say that F is "Y-ready", meaning that the recursion of G_r is achieved as soon as F is applied to Y. Let us show, step-by-step how the Y combinator works:

1. $(YF)v$
1. $(\lambda f.(\lambda x.xx)R_f F)v$

We advance the step number when an actual β-reduction is performed. Representations where a name of already bound value is replaced by its contents (or vice-versa) are considered the same step.

$$
\begin{aligned}
&2. \quad ((\lambda x.xx)R_F)v \\
&3. \quad (R_F R_F)v \\
&3. \quad ((\lambda x.F(xx))R_F)v \\
&4. \quad (F(R_F R_F))v
\end{aligned}
$$

At this point we could reduce the subexpression $R_F R_F$ again, but that would lead to the same steps 3-4 performed as a part of the whole expression. It would be equivalent to unrolling the recursion multiple times, before executing the recursive function. We do not want that. For that reason, we choose to lazily postpone the computation of $R_F R_F$ and applying it as such to F, which is the λr given earlier.

$$
\begin{aligned}
&4. \quad ((\lambda r.\lambda a.....\text{body with } r....)(F_R F_R))v \\
&5. \quad (\lambda a.....\text{body with } (F_R F_R)....)v \\
&6. \quad [a \mapsto v]\text{body with } (F_R F_R)
\end{aligned}
$$

From this point, the body is executed as normal.

$$7. \quad \text{...executing body...}$$

At some point the recursive call is reached, possibly with a new argument v'. The recursion is no longer represented as a name r, but as the replication terms $F_R F_R$:

$$8. \quad (F_R F_R)v'$$

This however is the same as point 3, except for the new argument v'. The recursion is achieved.

The CPS Version

Let us now focus on representing the Y combinator in CPS.

First, let us specify how the recursive function looks like: Let us assume that is takes one argument a plus a continuation c_a:

$$G_r = \lambda a c_a. \text{ ...body with } r\text{...}$$

The free variable r is provided by the Y combinator, passed to a Y-ready function F. The F function has also two arguments — the next step of recursion r, and the continuation c_r where the newly combined recursive function is passed for future use.

$$F = \lambda r c_r.c_r G_r = \lambda r c_r.c_r(\lambda a c_a. \text{ ...body with } r\text{...})$$

We ignore other cases, e.g. where G_r could take more arguments.

We can now take the standard Y-combinator from the lambda calculus and transform it to CPS. All single-argument lambdas now take two arguments, with the second one being the continuation where the result is being returned. The argument f of the combinator is assumed to be in the form of F. With such straightforward approach we obtain:

$$\begin{aligned}\widetilde{Y} =& \lambda f c_f.(\lambda x c_x.x x c_x)\widetilde{R}_f c_f,\\ \text{with } \widetilde{R}_f =& \lambda x c_x.x x(\lambda y.f y c_x)\end{aligned}$$

In DeepCPS notation these functions look as:

```
let Gr(a, ca) { ...body with r... }
let F(r, cr) cr . (a, ca) { ...body with r... }
let Rf(x, cx) {
    x . x (y)
    f . y cx
}
let Y(f, cf) {
    (x, cx) { x . x cx } . Rf cf
}
```

However, the solution we obtain contains a subtle error. Consider the following execution:

1.
```
Y . F (g)
g . v (result) ...
```

1.
```
(f, cf) {
    (x, cx) { x . x cx } . Rf cf
} . F (g)
g . v (result) ...
```

2.
```
(x, cx) { x . x cx } . RF (g)
g . v (result) ...
```

3.
```
RF . RF (g)
g . v (result) ...
```

3.
```
(x, cx) {
    x . x (y)
    F . y cx
} . RF (g)
g . v (result) ...
```

4.
```
RF . RF (y)
F . y (g)
g . v (result) ...
```

It becomes apparent that we are trying to unroll the recursion by endlessly calling `RF` on itself, before we are ever able to invoke `F`. In standard lambda calculus we could arbitrarily choose to lazily pass $R_F R_F$ as a parameter to avoid this problem. With CPS however, the order of evaluation is explicit, and in the definition of $\widetilde{Y}$ we have chosen not to do anything lazily.

Any expression can be made lazy by encapsulating it in its own lambda function and passing it as a parameter. In our case we need to make the replication term act lazily.

$$\lambda x c_x . xx(\lambda y . f y c_x)$$

The replication xx needs to be encapsulated in a new lambda. The lambda shall be invoked every time the recursion is triggered, accepting the same arguments as G_r. Therefore, our corrected R_f looks as:

$$R_f = \lambda x c_x . f(\lambda a c_a . xx(\lambda y . y a c_a)) c_x$$

We can now spell out the *correct* version of Y-combinator in CPS:

$$\begin{aligned} Y =& \lambda f c_f . (\lambda x c_x . x x c_x) R_f c_f, \\ \text{where } R_f =& \lambda x c_x . f(\lambda a c_a . xx(\lambda y . y a c_a)) c_x \end{aligned}$$

And in DeepCPS notation:

```
let Gr(a, ca) { ...body with r... }
let F(r, cr) { cr . (a, ca) ...body with r... }
let Rf(x, cx) {
    f . (a, ca) {
        x . x (y)
        y . a ca
    } cx
}
let Y(f, cf) {
    (x, cx) { x . x cx } . Rf cf
}
```

Let us confirm that our newly defined Y actually behaves as we intend:

1.
```
Y . F (g)
g . v (result) ...
```

1.
```
(f, cf) {
    (x, cx) { x . x cx } . Rf cf
} . F (g)
g . v (result) ...
```

2.
```
(x, cx) { x . x cx } . RF (g)
g . v (result) ...
```

3.

```
RF . RF (g)
g . v (result) ...
```

3.

```
(x, cx) {
    F . (a, ca) {
        x . x (y)
        y . a ca
    } cx
} . RF (g)
g . v (result) ...
```

4.

```
F . (a, ca) {
    RF . RF (y)
    y . a ca
} (g)
g . v (result) ...
```

4.

```
(r, cr) {
    cr . (a, ca) ...body with r as recursion...
} . (a, ca) {
    RF . RF (y)
    y . a ca
} (g)
g . v (result) ...
```

5.

```
let g (a, ca) {
   ...  body with  (a', ca') {
                        RF . RF (y)
                        y . a' ca'
                   } as recursion  ...
}
g . v (result) ...
```

Now a single iteration of the recursive function g is allowed to execute normally. The recursive step however is no longer represented as `r`, but as a lambda `(a, ca)RF . RF (y)y . a ca`.

6. $\left[\begin{array}{c} \texttt{a} \mapsto \texttt{v} \\ \texttt{ca} \mapsto \texttt{(result)}\ldots \end{array}\right]$

```
...  body with  (a', ca') {
                     RF . RF (y)
                     y . a' ca'
                } as recursion  ...
```

When the recursion is reached, with a new argument `v'` and continuation `(result')`... we have:

7.

```
(a, ca) {
    RF . RF (y)
    y . a ca
} . v' (result') ...
```

8.

```
RF . RF (y)
y . v' (result') ...
```

Which is equivalent to what we have in point 3, except for the new arguments `v'` and `(result')...`. The recursion has been achieved.