12-2014

A Transformation-Based Foundation for Semantics-Directed Code Generation

Arthur Nunes-Harwitt

A Transformation-Based Foundation for Semantics-Directed Code Generation

by

Arthur Nunes-Harwitt

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the B. Thomas Golisano College of
Computing and Information Sciences
Ph.D. Program in Computing and Information Sciences
Rochester Institute of Technology

Author signature _____

Approved by _____

Director of Ph.D. Program Date

December 2014

Rochester, NY, USA

B. THOMAS GOLISANO COLLEGE OF
COMPUTING AND INFORMATION SCIENCES
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

The Ph.D. Degree Dissertation of Arthur Nunes-Harwitt
has been examined and approved by the
dissertation committee as complete and satisfactory for the
dissertation requirement for Ph.D. degree
in Computing and Information Sciences

Dr. Matthew Fluet — Date

Dr. James Heliotis — Date

Dr. William McKeeman — Date

Dr. S. Manian Ramkumar — Date
Dissertation Examination Chairperson

Dr. Axel Schreiner — Date
Dissertation Co-Supervisor

Dr. Pengcheng Shi — Date
Dissertation Supervisor

December 2014

A Transformation-Based Foundation for Semantics-Directed Code Generation

by

Arthur Nunes-Harwitt

December 2014

Abstract

Interpreters and compilers are two different ways of implementing programming languages. An interpreter directly executes its program input. It is a concise definition of the semantics of a programming language and is easily implemented. A compiler translates its program input into another language. It is more difficult to construct, but the code that it generates runs faster than interpreted code.

In this dissertation, we propose a transformation-based foundation for deriving compilers from semantic specifications in the form of four rules. These rules give apriori advice for staging, and allow explicit compiler derivation that would be less succinct with partial evaluation. When applied, these rules turn an interpreter that directly executes its program input into a compiler that emits the code that the interpreter would have executed.

We formalize the language syntax and semantics to be used for the interpreter and the compiler, and also specify a notion of equality. It is then possible to precisely state the transformation rules and to prove both local and global correctness theorems. And although the transformation rules were developed so as to apply to an interpreter written in a denotational style, we consider how to modify non-denotational interpreters so that the rules apply. Finally, we illustrate these ideas by considering a larger example: a PROLOG implementation.

Acknowledgements

I am forever grateful to my advisor Professor Axel Schreiner. He not only saw my potential, but he also played an active role in actualizing it when I thought all hope was lost. Further, he tirelessly read draft upon draft of my papers, always providing helpful suggestions. I would like to acknowledge Professor Pengcheng Shi for his support and advice. He made sure to slow me down when I was too eager to rush ahead, and he made sure to push me forward when I needed it. I am thankful to Professor Bill McKeeman for nudging my research in its current direction. I am thankful to Professor Matthew Fluet for carefully reading my proofs and reading drafts of my papers. I am thankful to Professor Jim Heliotis for his helpful discussions. I would also like to acknowledge Melissa Nunes-Harwitt for providing helpful feedback on all of my papers.

Professor Linwei Wang generously offered LaTeX assistance.

Dr. Arthur C. Nunes Jr., Professor Axel Schreiner, Melissa Nunes-Harwitt, Professor Stanislaw Radziszowski, Menachem Kiria, and Professor Hossein Shahmohamad helped me eliminate errors in my epigraphs.

I am grateful for the encouragement of Brent Devere, and my gratitude also goes to all my friends, living far away or around me.

To my father, and to my mother, for setting an example. I humbly follow in your path.

To my wife Melissa, for your unconditional love and support. All this would have been impossible without all that you have done for me.

We choose to [...] do these [...] things, not because they are easy, but because they are hard, because that goal will serve to organize and measure the best of our energies and skills....

- John Fitzgerald Kennedy

Contents

List of Figures

Chapter 1

Introduction

המתרגם פסוק כצורתו, הרי
זה בדאי ˙ והמוסיף עליו, הרי
זה מחרף ומגדף.

If one translates a verse literally, he is a fabulator – but if he adds to it, he is a blasphemer and a libeller.

B. Talmud Kiddushin 49a

A *compiler* is a computer program that when executed translates[1] *source* code, or input code, to *target* code, or output code; it is understood that the source code is in some fixed high-level programming language and that the target code has the same meaning as the source code.

What approaches are there that ensure that the target code preserves the semantics of the source code? One approach is to guess what the target code should look like and then prove that it preserves the semantics after the fact. Another approach is to derive

[1] Often the word "compiler" is used when translating to a low-level language, and the word "translator" is used when translating to another high-level language. We will *not* make this distinction.

the target code from a semantic specification such as an interpreter. We argue that certain transformations are an effective means of doing exactly that.

1.1 A Semantics-Directed Approach to Code Generation

Commonly, compilers are implemented via a syntax-directed approach which involves writing code to generate instructions for each piece of abstract syntax. The code to be generated is *not* derived from any specification; rather it is chosen by the compiler writer in an ad-hoc fashion. In contrast, a semantics-directed code generator is a code generator that has been derived from a semantic specification. Semantics-based approaches to code generation have a number of benefits: correctness, ease of implementation, maintainability, and rational justification. Two techniques for semantics-based code generation are *partial evaluation* and *staging*.

Like Burstall and Darlington [17], the concern here is on fundamental rules for deriving programs. Their approach involves ideas at the core of equational reasoning: folding and unfolding. Instead, here the focus is on an extension of the λ-calculus [8, 19, 38] formulation of the S-m-n theorem [59, 70].

This thesis makes the following contributions. It identifies a transformation technique in the form of four essential rules, and proves properties about those rules. An analysis shows how this transformation technique can be applied to a class of interpreters even broader than the class of denotational-style interpreters. Case studies suggest the utility of these transformation rules and the transformation approach in general for deriving a compiler from an interpreter.

This approach to creating compilers has the same benefits as other semantics-based approaches. These benefits are particularly valuable for domain-specific languages (DSLs) [106] and experimental languages. In contrast to partial-evaluation based approaches, ad-

vantages include the ability to derive a compiler, and the possibility of naturally adding additional stages to the transformation sequence so as, for example, to use more sophisticated algorithms. In contrast to other work in staging, the advantage is a more formal foundation for program transformation.

1.2 Overview

The rest of this thesis examines the transformation technique in detail. Chapter 2 is devoted to the various ideas related to the transformation technique. Chapter 3 introduces the transformation technique informally. Chapter 4 formalizes the notions here so as to state the rules precisely and to prove correctness results. Although the inspiration for the transformation technique is denotational semantics, chapter 5 investigates to what extent interpreters written in other styles may be modified so that the transformation technique can be applied. Chapter 6 considers a more significant example: a PROLOG implementation. Chapter 7 outlines some thoughts concerning additional work that might be done. Finally, chapter 8 concludes with a summary.

Chapter 2

Background and Related Work

> Gutta cavat lapidem non bis,
> sed saepe cadendo; sic homo fit
> sapiens non bis, sed saepe
> legendo.
> *A drop hollows out the stone by*
> *falling not twice, but many times;*
> *so too is a person made wise by*
> *reading not two, but many books.*
>
> Giordano Bruno

The ideas discussed in this thesis lie in the intersection of three related areas: compiler generation, partial evaluation, and staged computation. A historical section introduces these areas. Then a section devoted to each area follows.

2.1 Historical Overview

The term *compiler* was first used by Grace Hopper [48, 62] to describe program generation software. These early compilers gathered together, or compiled, subroutines from subroutine libraries to form whole programs. Mike Karr [57] suggests that John Backus referred to his FORTRAN project as a 'compiler' project to emphasize the utility of the project independent of the ambitious translation aspect.

Every programmer has had the experience of finding a bug in even the simplest program. Writing bug-free software is challenging. Frederick Brooks [15] observes that there are layers to the challenge of writing bug-free software. He defines a *programming system* as software that consists of a collection of coordinated, interacting programs, and he defines a *programming product* as software that will be run and maintained by others. Brooks suggests that a programming systems component is three times more difficult[1] than an ordinary program, and that a programming systems product is nine times more difficult. At the very least, a compiler is inherently a component of a system involving the operating system, an editor, and perhaps other programs involved in writing the source and running the target code. As such, a compiler is quite difficult to write. Indeed, a well-known series of compiler texts [2–4] emphasizes the difficulty of writing a compiler by depicting the complexity of compiler design as a dragon.

It is possible, and it is quite typical, to split a compiler into two components: the *front-end*, and the *back-end*. The front-end involves computations associated with the source such as lexical analysis, parsing, and type-checking. The back-end involves computations associated with the target such as optimization and code-generation. The advantage of splitting a compiler into front-end and back-end components is that it is then possible to re-use the front-end for multiple targets and the back-end for multiple source languages.

[1] Brooks characterizes this notion of difficulty as cost to achieve a specified level of quality.

Further, it is common to refine this division one step further by tripartitioning the compiler into the front-end, the middle-end, and the back-end. The front-end then performs tasks associated with input such as lexical analysis and parsing. The middle-end performs tasks that manipulate internal representations such as type-checking, generation of intermediate-code, and optimization. The back-end involves computations associated with output such as target-optimization and generation of target-code. The front-end, middle-end, and back-end all have posed challenges.

Early programming language efforts used ordinary English to specify the syntax; the associated ad-hoc parsers required tremendous effort to develop. When working on Algol 58, John Backus [7] made an important breakthrough[2] when he introduced BNF[3] notation to express a context-free grammar (CFG) that specified the syntax. Because the class of CFGs was so clearly specified, it was not long before general parsing algorithms (e.g., CYK, Earley), parsing algorithms for the regular subset, and linear time compiler-oriented parsing algorithms appeared (e.g., Precedence Parsing, LL(k), LR(k), and LALR(k)). Further, tools (e.g., XA [69], lex [65], and yacc [49]) emerged that effectively translated little more than formal specifications into efficient parsing programs.

Code generation in early programming language efforts also involved ad-hoc techniques. But there have been many advances in code-generation technology as well. In particular, for back-ends tree-tiling [4, 80] and declarative machine descriptions [33] have been used to specify at various levels how assembly code should be generated. Unfortunately, for the generation of intermediate-code in middle-ends, ad-hoc techniques continue to be used; indeed, compiler textbooks take a 'cookbook' approach.

A principled approach to intermediate-code generation goes under the heading *semantics-*

[2]It may be that Backus was influenced by Chomsky [14] or Post [105]. In any case, the ideas seem to have been in the air, and it was he who introduced them to the programming language community.

[3]There is a disagreement about whether BNF stands for 'Backus Normal Form' or 'Backus Naur Form.' Knuth [60] has noted that BNF is *not* a normal form in the usual mathematical sense. Peter Naur asserts that his role in developing the notation was minuscule [105].

directed compiler generation. Diehl [34] observes that a compiler generated from a semantic specification has the following advantages: correctness, ease of implementation — only the specification needs to be written from scratch, maintainability — only the specification needs to be modified, and rational justification for the code generated. For automatic (middle-end) compiler generation, some form of *partial evaluation* [51] (i.e., a program specialization technique) is typically used; although occasionally other approaches are used. The idea of *staging* [56,74,87] (i.e., separating a computation into one that runs earlier and another that runs later) is more general than partial evaluation and may refer to automatic or manual approaches to generation. These techniques have been leveraged to partially or fully systematize code-generation. Nevertheless, the success achieved with front-ends, namely the ability to automatically translate a specification into an efficient program, in the realm of intermediate-code generation remains elusive.

2.2 Compiler Generators

Narrowly speaking, a *compiler generator* is a program that accepts a specification of a source language $\mathcal{S}$ that describes both the syntax (S) and semantics, accepts a specification of a target language $\mathcal{T}$ that describes both the syntax (T) and semantics, and generates a compiler that accepts S programs and generates T programs. Broadly speaking, we use the term even if aspects are missing (e.g., the parser, or the target specification) or an interpreter is generated instead.

The first compiler generator is Peter Mosses' Semantic Implementation System (SIS) [73]. The specifications for syntax and semantics are tightly integrated. The lexical and grammatical structure is expressed in a form called *GRAM*, which resembles BNF. The semantics is specified in the *Denotational Semantics Language* (DSL) which is an extension of the λ-calculus. Mosses' system generates reasonable λ-terms; however, his term reducer

is quite slow.

Another pioneering compiler generator was CERES by H. Christiansen and Neil D. Jones. Mads Tofte [101] subsequently simplified the CERES compiler generator by observing that a compiler generator is a kind of compiler. He appears to deemphasize the use of a general partial evaluator.

Peter Lee [63, 64] criticizes Mosses' work and others [79] and points out that the generation process, the generated compiler and the target code generated by the generated compiler are all too slow and inefficient. Lee puts the blame squarely on denotational semantics [95, 102]. In particular, he identifies the following four problems: (1) *lack of separability* — when certain features are added, the entire semantics must be rewritten[4]; (2) *poor semantic engineering* — the static and dynamic aspects are blurred[5]; (3) *minimalist semantic explication* — some language features are described at a very high level that provide no hint of low-level implementation (e.g., no distinction is made between parameters and other variables); and (4) *lack of modularity* — there are no semantic modules, and interaction with the store relies on specific concrete mathematical details. While Lee preserves the compositional[6] aspect of denotational semantics, he replaces the details with a two level semantics: a macrosemantics and a microsemantics. The macrosemantics is expressed in a *prefix-form operator expression* (POE), which appears to be a standardized abstract syntax. The microsemantics can be expressed in a number of ways ranging from a high-level denotational semantics to a low-level assembly language description. However, the different microsemantics are not required to agree; for example, the assembly language semantics for array indexing does no bounds checking. Lee's system generates

[4]Subsequently, some in the denotational camp addressed this problem using the category theoretical construct of *monads* [36, 71, 91, 93].

[5]This blurring of the the static and dynamic aspects can also be seen as an advantage [42] allowing for a simpler description of the meaning.

[6]A *compositional* [102] semantics is one in which the meaning of an expression is a function of the meaning of its sub-expressions.

compilers that can parse, statically type-check, and generate assembly code; further, it is able to generate faster compilers that generate faster code. However, it is not clear how flexible his compiler generator is, nor is it clear how confident one can be in its correctness.

In contrast, Jesper Jørgensen [54, 55] generates a realistic compiler for a Haskell-like language that is specified using denotational semantics. Jørgensen calls the language he implements BAWL. It resembles Miranda even more closely than Haskell. The specifications for syntax and semantics are not tightly integrated. The lexical and grammatical structure is expressed in a form suitable for a YACC-like parser generator. Programs are assumed to be type correct; there is no attempt at type-checking. Jørgensen does discuss how a meta-interpreter would allow the denotational definition language to be used directly, and he gives some small examples. He further speculates that determining binding times might be easier with this approach. Nevertheless, the denotational semantics is hand-translated[7] into Scheme [58, 92] and then the resulting interpreter is enhanced to perform 'optimizations.' If performed at run-time, the 'optimizations' would slow down execution; however, they are designed so that the partial evaluator will perform them during the static phase. The resulting compiler which targets Scheme, when coupled with an optimizing Scheme compiler, performs better than a non-optimizing Miranda compiler from Research Software Limited. However, an experimental optimizing research compiler from Chalmers University has even better performance.

Charles Consel and Siau Cheng Khoo [24] generate a simple PROLOG [32,104] compiler that is specified using denotational semantics. Issues of syntax specification are ignored. The denotational semantics is hand-translated into side-effect-free Scheme, and a partial evaluator is used to create a compiler that targets Scheme. Consel and Khoo report that compiled code runs six times faster than interpreted code, but they do not compare their compiler to other PROLOG implementations. This compiler suffers from a number of is-

[7] Lazy evaluation was implemented via 'suspensions' rather than graph reduction.

sues. First, because their partial evaluator does not support side-effects, their approach cannot take advantage of more sophisticated unification algorithms. Second, their examples of compiled programs show duplications of the database. Third, the generated compiler cannot handle recursive PROLOG programs; recursion causes their compiler to loop.

In order to demonstrate a new technique for doing binding time analysis, Olivier Danvy and René Vestergaard [29] generate a compiler for a simplified Pascal-like language. Issues of syntax specification and type-checking are ignored. The denotational semantics is hand-translated into Scheme, where all globals have been turned into parameters because their binding time analyzer requires its input to be closed. Their partial evaluator is then used to create a compiler that targets Scheme. Danvy and Vestergaard report that generated code runs four times faster than interpreted code. In addition, they point out that their compiler generates code that resembles three-address code [3,4]; however, it may be that that is a result of the semantics being expressed in a low level style.

Stephan Diehl [34] generates compilers for two simple languages. Issues of syntax are ignored. From the specified big-step operational semantics [84, 102], an abstract machine is generated together with a compiler that targets that abstract machine. Diehl's system does this by first transforming the big-step operational semantics into a small-step operational semantics [84, 102]. This transformation is accomplished by recording the additional information of the big-step derivation tree in an auxiliary context data structure. Then *pass-separation*[8] ensures that instructions either introduce other instructions or modify the context data, but not both. The capability of generating instructions for an abstract machine is impressive, but the kind of abstract machine generated is restricted to what is embodied in the transformation algorithm.

Michael J. A. Smith [90] focuses on the meta-level and provides a complete semantics-

[8]The idea and implementation of pass-separation was first explored by Hannan [46].

directed compiler generator: SEMCOM. Syntax and semantics are specified in two parts: the dynamic semantics and the static semantics. The dynamic semantics take the form of an operational semantics (either big or small step) involving concrete syntactic forms, from which an interpreter is generated. The static type system is powerful enough to express polymorphic type systems and is used to generate the lexer, parser, and type-checker. The comprehensiveness of the SEMCOM specification language is impressive; however, it generates only interpreters, not compilers.

There have been various approaches to constructing compiler generators. As of yet, no approach has been competitive with hand-crafted compilers. Partial evaluation is an approach often taken for constructing a compiler generator. We explore this idea next.

2.3 Partial Evaluation

Partial evaluation [51] is a transformation technique for specializing programs. Program specialization can mean simply replacing some of a function's parameters with values; however, specialization is usually understood to involve using those values to perform some of the computation that does not depend on the remaining parameters. Jones et. al. [51] suggest that Lionello A. Lombardi first used the term 'partial evaluation' essentially as it is used today; however, Lombardi's 1964 paper [66] does not seem to contain the phrase 'partial evaluation,' and instead discusses 'incremental computation.' In the same year as Lombardi's paper, Peter J. Landin [61] used the term 'partial evaluation' when discussing mechanisms for expression evaluation. Rodney M. Burstall and John Darlington [17] provide the conceptual foundation for partial evaluation: *unfolding*, or expanding definitions, and *folding*, or reducing definitions.

Kleene's S-m-n theorem establishes that the minimal form of specialization is computable, so programs can be written that perform this task. A *partial evaluator* is a program

that performs partial evaluation. Lionello A. Lombardi hinted at the use of a program to do partial evaluation. Yoshihiko Futamura [42] was the first to clearly describe a partial evaluator and point out what are now known as the first and second Futamura projections: that a partial evaluator can be used to do compilation and to create compilers.

The Futamura projections concern the following observations. We model a programming language $\mathcal{L}$ as a three-tuple $\mathcal{L} = \langle \mathcal{E}, L, D \rangle$, where D is the set of data that the language processes, $L \subseteq D$ is the set of program representations, and $\mathcal{E} : L \times D^* \rightarrow D$ is the evaluator that maps a program representation and inputs to output. Given a programming language $\mathcal{L}$, a partial evaluator m has the property that for any $\mathcal{L}$ program e, $(\mathcal{E}\ e\ (d_1, d_2)) = (\mathcal{E}\ (\mathcal{E}\ m\ (e, d_1))\ d_2)$. Now first observe that if e is an interpreter for $\mathcal{L}_2$ and p is an $\mathcal{L}_2$ program, then $(\mathcal{E}\ e\ (p, d)) = (\mathcal{E}\ (\mathcal{E}\ m\ (e, p))\ d)$. Thus $(\mathcal{E}\ m\ (e, p))$ can be regarded as the target code, and $\lambda p.(\mathcal{E}\ m\ (e, p))$ can be regarded as a compiler. Second, observe that $(\mathcal{E}\ m\ (e, p)) = (\mathcal{E}\ (\mathcal{E}\ m\ (m, e))\ p)$. Thus we can reify the previous compiler abstraction as $(\mathcal{E}\ m\ (m, e))$; i.e., the partial evaluator is applied to itself.

In order to create a compiler using a partial evaluator m, m must be self-applicable. Futamura [42] and other early researchers [10,35] were unable to construct a partial evaluator with this property. One point of view is that the problem was that the partial evaluator could not tell which which specializations were important and which were not. Jones, Sestoft, and Søndergaard [52,53] identified important specializations with 'obvious' ones — specializations that could be identified via static analysis. They then made a distinction between *online* partial evaluators (i.e., those that in a single phase decide what to specialize and then perform that specialization) and *offline* partial evaluators (i.e., those that have two phases: one which performs a *binding time analysis* that determines what specialization should take place, and one which performs the specializations that the analysis indicated). Their offline partial evaluator 'mix' was the first capable of self-application; it was used to generate compilers for toy languages.

Anders Bondorf and Olivier Danvy designed the offline self-applicable partial evaluator Similix [13]. The language it handled was a subset of Scheme that included mutable global variables. It also improved on previous efforts by better preserving termination properties and by avoiding code duplication.

Charles Consel created the offline self-applicable partial evaluator Schism [20, 21]. Initially, the language was restricted to a first-order side-effect-free subset of Scheme, but it was subsequently expanded to allow higher-order functions as well.

Subsequent partial evaluators such as Tempo [22, 23] were frequently extensions of well-established partial evaluators such as Schism.

The cogen approach [12, 99] is an alternative to traditional partial evaluation. Like the technique presented in this thesis, the emphasis is on generating a code generator. The cogen approach borrows from the ideas involved in off-line partial evaluation. To create a code generator, a binding time analysis is performed and the input program is annotated. Instead of using the annotated program for partial evaluation, the annotations are reified to generate the generator. Then the generator can be used for partial evaluation, if desired. PGG [99, 100] is a partial evaluator that follows the cogen approach. According to Neil D. Jones [50], it is one of the most sophisticated partial evaluation programs available at the time of this writing.

Recently, despite beliefs that only offline partial evaluators could be self-applicable, Robert Glück [43] developed an *online* self-applicable partial evaluator. The language is a flowchart language that allows recursive calls. Glück observes that no new techniques were used but rather the order of the techniques and recursive polyvariant specialization were the key to the implementation.

Early partial evaluators had trouble with assignment and/or higher-order functions [20, 21]. Although contemporary partial evaluators are more powerful, they are not always successful. Coming up with the right binding time improvements to help a par-

tial evaluator can be challenging because partial evaluation algorithms are quite complicated [29]. Some prefer manual staging because it is more transparent. We consider this idea next.

2.4 Staged Computation

A *staged computation* is a computation that is organized so that part of the computation occurs at one stage, or time, and the rest of the computation occurs at another. Partial evaluation is a technique for staging; however, this notion has broader scope and includes (and today usually refers to) manual techniques.

LISP programmers have long enjoyed a kind of staging in the form of the eval operator and macros. Further, LISP [58,68,92,94] dialects have always had a quotation mechanism [9,68] which has facilitated meta-programming. The eval operator allows staging in that s-expressions can be constructed that look like programs. These programs can then be evaluated by the eval operator in the global environment[9]. Macros allow staging in that arbitrarily complex programs can be written that generate code in the form of an abstract syntax tree usually expressed as an s-expression[10] . This generated code is executed later[11] with the non-macro portion of the program.

It appears that Ulrik Jørring and William L. Scherlis [56] were the first to use the term 'staging.' In their provocative paper, they show many examples of turning small interpreters into code generators. However, they are somewhat vague about the details of transforming an interpreter. They comment, "[We] reorder computations a bit to separate the stages..." but they do not explain how to do so.

A year later, Marc Feeley's paper [37] appears concerning a closure-based approach

[9] As of R5RS [58], Scheme allows environments other than the global environment to be specified.
[10] Dialects of Scheme have used explicit abstract syntax representations [92].
[11] In some LISP dialects, when a macro would run was obscure.

to code generation. Although he does not reference Jørring and Scherlis and he does not characterize his approach as one involving a transformation, his presentation of an interpreter and a compiler side by side is suggestive. Feeley is very practical and has performance results to show that his approach yields simple but high performing implementations.

Flemming Nielson and Hanne R. Nielson [75] view staging from the perspective of implementing a traditional compiler for functional programming languages. They are concerned that with the proliferation of higher-order functions there remains a lot of computation that can be done at compile time but discovering this computation requires a careful analysis. To aid such analyses, they introduce 'two-level languages.'

Rowan Davies and Frank Pfenning [31] extend the work of Nielson and Nielson and develop a multi-level language. Again following Nielson and Nielson they develop a type system for binding-time analysis. Davies and Pfenning have in mind that the analysis should be specifically useful for partial evaluation. Their type system is based on modal logic and allows for the eval operator but not for open terms.

Since open terms are occasionally needed in partial evaluation, Rowan Davies [30] developed a type system based on linear-time temporal logic. This type system allows for open terms by abandoning the soundness of the eval operator[12]. This limitation entails that the eval operator is not allowed.

Subsequently, Walid Taha and Tim Sheard [98] argue that multi-stage languages are useful programming languages and should not just be viewed as an intermediate languages in a compiler (or other programming language oriented program). Although Jørring and Scherlis are not referenced, Nielson and Nielson, and Davies and Pfenning are referenced. Nevertheless they develop their own type system and deliver the programming language MetaML. Unfortunately, this design is unsound because free vari-

[12]With open terms, the eval operator could attempt to evaluate an unbound variable.

ables may be encountered during evaluation. Taha, Benaissa, and Sheard [97] correct the soundness issue.

Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard [72, 96] return to the design of the type system for MetaML. Their new approach borrows much from Davies and Pfenning. They succeed in integrating both the linear-time temporal logic and the modal logic providing MetaML with a sound type system that allows for both the eval operator and open terms.

Tim Sheard and Simon Peyton Jones [88] add meta-programming features to Haskell. Their approach differs from MetaML. The emphasis is on compile time meta-programming like LISP macros. Unfortunately, also like LISP macros the type system does very little to ensure correctness.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy [18] explore an alternative meta-programming implementation and extend OCaml to MetaOCaml. Their implementation makes use of abstract syntax trees, gensym, and reflection. They report that this new implementation yields respectable performance.

More recently, Aleksandar Nanevski and Frank Pfenning [74] extend the work of Davies and Pfenning and incorporate into that logic some of the ideas from MetaOCaml involving generated names. ν-abstraction is added to the language and a set of names qualifies the modal operator. Thus their calculus and type system go beyond closed terms and allow both open terms and the sound use of the the eval operator with only a single modal operator.

2.5 Summary

At one end of the spectrum, partial evaluation is a mostly automatic program specialization technique. It can be used to turn interpreters into compilers. At the other end of

the spectrum, staged computation is a mostly manual technique for specializing code. It involves writing code that given a single input, does as much computation as possible, and returns a function or function text that characterizes the rest of the computation that depends on additional inputs. However, issues remain for both of these approaches.

Deriving a compiler using partial evaluation is cumbersome. Such a derivation is more challenging than deriving a string-matching algorithm [28] because compiler generation requires the partial evaluation of a traditional partial evaluator on the interpreter, or a binding time analysis when using a cogen-based partial evaluator.

Further, partial evaluators have shortcomings both because they are automatic and because they are not as automatic as claimed. Because they are automatic, they cannot replace transparent definitional algorithms with more sophisticated algorithms. Fully automatic partial evaluators are often slow and/or generate slow code. Yet fully automatic partial evaluators often do not succeed in eliminating static computation [29]. Thus manual "binding time improvements" are needed to help the partial evaluator. Coming up with the right improvements can be challenging because partial evaluation algorithms are often quite complicated.

And so constructing a compiler completely automatically is unreasonable. Nevertheless, there should be a rational basis for compiler construction. The ideas of staging [56] hint that a manual approach may be feasible, but they provide no notion of how to construct the staged program.

This thesis presents a manual transformation technique that can be used to derive a compiler from an interpreter. Because it is a manual technique, improved algorithms can be introduced at any point. But even though the it is a manual technique, the transformation rules indicate exactly how to go about staging to the code.

Chapter 3

The Transformation Technique by Example

Какой-то математик сказал, что наслаждение не в открытии истины, но в искании ее.
A mathematician once said: pleasure lies not in discovering truth, but seeking it.

Leo Tolstoy

The motivation for the transformation technique comes from several places: Marc Feeley's closure based approach to code generation [37], Kleene's S-m-n theorem, and denotational-semantics [95, 102]. The transformation technique involves applying four rules: currying, lambda lowering, expression lifting, and quoting. To build intuition, we start with a very small example that is not even an interpreter. Then we present a slightly

larger example.

3.1 A Small Example

In this section, the transformation rules are motivated by the desire to both specialize and generate text. For the sake of brevity, mathematical functions are used as examples.

3.1.1 Currying Dynamic Variables

Currying is a mathematical trick to make all functions take one argument; it transforms a function of two or more arguments into a function of one argument that returns a function. For example, the multiplication function $m(x, y) = x \times y$ becomes $m(x) = \lambda y.x \times y$. If we have in mind that x is known statically, but y is known dynamically, then applying the curried form to a statically known value specializes the multiplication function. For example, applying m to 2 results in the following term: $m(2) = \lambda y.2 \times y$. Thus the application of a curried function is a form of code generation.

Many programming languages, especially today, allow for first-class functions. In Scheme [58], the multiplication example looks as follows.

```
(define (m x y) (* x y))
```

When curried, it becomes the following.

```
(define (m x) (lambda (y) (* x y)))
```

However, applying `m` to 2 yields an opaque result rather than the desired term. Something more is needed to see the text of the resulting procedure.

```
> (m 2)
#<procedure>
```

3.1.2 Code Via Quoting

To fix the problem in section 3.1.1, we want to see the text of the function rather than the function itself (which may not be displayable). To return text, rather than a function, we can use Scheme's quotation and un-quotation mechanisms: backquote and comma. The lambda expression is quoted and the lambda expression's local free variables are unquoted. Upon making this change, the term comes out as expected, although now eval is needed to actually apply this function.

```
(define (m x) `(lambda (y) (* ,x y)))
```

```
> (m 2)
(lambda (y) (* 2 y))
```

But consider the following more complicated example of raising b to the nth power using a recursive function and what happens when applying these currying and quoting transformations.

```
(define (p n b) ; original
   (if (= n 0)
       1
       (* b (p (- n 1) b))))

(define (p n) ; curried
   (lambda (b)
      (if (= n 0)
          1
          (* b ((p (- n 1)) b)))))
```

```
(define (p n) ; quoted
   `(lambda (b)
       (if (= ,n 0)
           1
           (* b ((p (- ,n 1)) b)))))

> (p 3)
(lambda (b)
   (if (= 3 0) 1 (* b ((p (- 3 1)) b))))
```

The result this time is inadequate because a substantial amount of static computation remains. In particular, the conditional does not depend on the parameter b and should not be there. The code generated also assumes a run-time environment in which the curried form of p that returns a number is defined. Of course, the goal is to eliminate the need for such a run-time function.

3.1.3 Lambda Lowering

To fix the problem in section 3.1.2, we need to evaluate the test in the conditional. A way to do that is to move the function with the formal parameter b inside the conditional after currying. Upon making this sequence of transformations, applying the code generating function does yield a simpler term.

```
; original

; curried

(define (p n) ; lambda lowered
   (if (= n 0)
       (lambda (b) 1)
       (lambda (b) (* b ((p (- n 1)) b)))))

(define (p n) ; quoted
   (if (= n 0)
       `(lambda (b) 1)
       `(lambda (b) (* b ((p (- ,n 1)) b)))))
```

```
> (p 3)
(lambda (b) (* b ((p (- 3 1)) b)))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function p.

3.1.4 Expression Lifting

To fix the problem in section 3.1.3, we need to evaluate the recursive call. Since it resides in a λ-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term[1].

```
; original

; curried

; lambda lowered

(define (p n) ; expression lifted
   (if (= n 0)
       (lambda (b) 1)
       (let ((f (p (- n 1))))
          (lambda (b) (* b (f b))))))

(define (p n) ; quoted
   (if (= n 0)
       `(lambda (b) 1)
       (let ((f (p (- n 1))))
          `(lambda (b) (* b (,f b))))))
```

[1]This example is intended merely to illustrate the four transformation rules. A more serious algorithm for computing powers would use repeated squaring. Further, this approach all by itself is insufficient for loop unfolding since this code will unfold arbitrarily large powers.

```
> (p 3)
(lambda (b)
   (* b ((lambda (b)
           (* b ((lambda (b)
                   (* b ((lambda (b) 1) b)))
                 b)))
         b)))
```

Although ideally the generated code would be more readable, we can make it more pleasant looking by post-processing[2].

```
(lambda (b) (* b (* b (* b 1))))
```

3.1.5 Rule Ordering

When applying the rules above, they are performed in the following order. First the function is curried. Then the lambda lowering and expression lifting rules are applied repeatedly until those rules can no longer be applied. Finally the quoting rule is applied to all λ-expressions derived from the curried function.

3.2 A Longer Example

To illustrate the technique, consider the application of regular expression matching. A regular expression matching interpreter takes a regular expression and a string, and determines if the string is in the language denoted by the regular expression. Often, the regular expression is fixed, and we would like the code that answers whether a string is in the language denoted by that fixed regular expression.

[2]The post-processing consists of copy-propagation and dead-code elimination. Again, we only make this effort for human readers; the computer executes the unprocessed form.

Definition 1. *A* regular expression *is one of the following, where the predicate testing each option is in parentheses.*

- *The empty string.* (`null?`)
- *A character in the alphabet.* (`char?`)
- *The union of two regular expressions.* (`or?`)
- *The concatenation of two regular expressions.* (`cat?`)
- *The Kleene star of a regular expressions.* (`star?`)

The matching algorithm is expressed in Scheme using continuation passing style; the continuation (`k`) is the property that must be satisfied by the remainder of the string. In the code below, a string is represented as a list of characters (`cl`).

```
(define (match regexp cl k)
  (cond ((null? regexp) (k cl))
        ((char? regexp)
         (if (null? cl)
             #f
             (and (eq? (car cl) regexp) (k (cdr cl)))))
        ((or? regexp)
         (or (match (exp1<-or regexp) cl k)
             (match (exp2<-or regexp) cl k)))
        ((cat? regexp)
         (match (exp1<-cat regexp)
                cl
                (lambda (cl2)
                  (match (exp2<-cat regexp) cl2 k))))
        ((star? regexp)
         (let loop ((cl2 cl))
           (or (k cl2)
               (match (exp<-star regexp)
                 cl2
                 (lambda (cl3)
                   (if (eq? cl2 cl3) #f (loop cl3)))))))
        (else (error 'match "match's input is bad"))))
```

When `regexp` is the empty string, `match` invokes the continuation on the character list. Note that the initial continuation verifies that the character list is empty. When `regexp` is a character, `match` checks that the first character in the character list is that character, and invokes the continuation on the tail of the character list. When `regexp` is a union, `match` first tries the first option, and if that fails it backtracks and tries the second option. When `regexp` is a concatenation, `match` recursively matches the first component and adds a check for the second to the continuation. When `regexp` is a Kleene star, `match` loops checking if either the continuation is satisfied (i.e., Kleene star corresponds to the empty string) or the pattern to be repeated is matched; thus the shortest prefix is matched.

In the following sections, we will now apply the technique to this interpreter and derive a code generator.

3.2.1 Currying

A compiler for regular expressions must be a function that takes a regular expression; hence the dynamic parameters are `cl` and `k`. They are removed from the top-level parameter list and put into the parameter list of the λ-expression. The recursive calls are modified to account for this new protocol.

```
(define (match regexp)
  (lambda (cl k)
    (cond ((null? regexp) (k cl))
          ((char? regexp)
           (if (null? cl)
               #f
               (and (eq? (car cl) regexp) (k (cdr cl)))))
          ((or? regexp)
           (or ((match (exp1<-or regexp)) cl k)
               ((match (exp2<-or regexp)) cl k)))
          ((cat? regexp)
           ((match (exp1<-cat regexp))
                    cl
                    (lambda (cl2)
                      ((match (exp2<-cat regexp)) cl2 k))))
          ((star? regexp)
           (let loop ((cl2 cl))
             (or (k cl2)
                 ((match (exp<-star regexp))
                  cl2
                  (lambda (cl3)
                    (if (eq? cl2 cl3) #f (loop cl3)))))))
          (else (error 'match "match's input is bad")))))
```

3.2.2 Lambda lowering

Since `(cond (`e_1 e_2`) ...)` $\equiv$ `(if` e_1 e_2 `(cond ...))`, it is possible to apply the conditional form of the lambda lowering rule several times. The lambda just below the definition in `match` is lowered into each branch of the cond-expression[3].

[3] An exception to the rule is made in the error case; the lambda is not lowered. The motivation is practical: it is preferable to find out right away that the input is invalid.

```
(define (match regexp)
  (cond ((null? regexp) (lambda (cl k) (k cl)))
        ((char? regexp)
         (lambda (cl k)
           (if (null? cl)
               #f
               (and (eq? (car cl) regexp) (k (cdr cl))))))
        ((or? regexp)
         (lambda (cl k)
           (or ((match (exp1<-or regexp)) cl k)
               ((match (exp2<-or regexp)) cl k))))
        ((cat? regexp)
         (lambda (cl k)
           ((match (exp1<-cat regexp))
            cl
            (lambda (cl2)
              ((match (exp2<-cat regexp)) cl2 k)))))
        ((star? regexp)
           (lambda (cl k)
             (let loop ((cl2 cl))
               (or (k cl2)
                   ((match (exp<-star regexp))
                    cl2
                    (lambda (cl3)
                      (if (eq? cl2 cl3) #f (loop cl3)))))))))
        (else (error 'match "match's input is bad"))))
```

3.2.3 Expression lifting

Since the recursive calls have been curried and do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. In this example, it is clear that the calls will halt since the recursive calls are always on smaller structures.

```
(define (match regexp)
  (cond ((null? regexp) (lambda (cl k) (k cl)))
        ((char? regexp)
         (lambda (cl k)
           (if (null? cl)
               #f
               (and (eq? (car cl) regexp) (k (cdr cl))))))
        ((or? regexp)
         (let ((f1 (match (exp1<-or regexp)))
               (f2 (match (exp2<-or regexp))))
           (lambda (cl k) (or (f1 cl k) (f2 cl k)))))
        ((cat? regexp)
         (let ((f1 (match (exp1<-cat regexp)))
               (f2 (match (exp2<-cat regexp))))
           (lambda (cl k)
             (f1 cl (lambda (cl2) (f2 cl2 k))))))
        ((star? regexp)
           (let ((f (match (exp<-star regexp))))
             (lambda (cl k)
               (let loop ((cl2 cl))
                 (or (k cl2)
                     (f cl2 (lambda (cl3)
                              (if (eq? cl2 cl3)
                                  #f
                                  (loop cl3)))))))))
        (else (error 'match "match's input is bad"))))
```

3.2.4 Quoting

Now each λ-expression is quoted. The Scheme backquote syntax is used to allow some sub-expressions to be evaluated. In particular, non-global free variables are unquoted in the text.

```
(define (match regexp)
  (cond ((null? regexp) `(lambda (cl k) (k cl)))
        ((char? regexp)
         `(lambda (cl k)
            (if (null? cl)
                #f
                (and (eq? (car cl) ,regexp) (k (cdr cl))))))
        ((or? regexp)
         (let ((f1 (match (exp1<-or regexp)))
               (f2 (match (exp2<-or regexp))))
           `(lambda (cl k) (or (,f1 cl k) (,f2 cl k)))))
        ((cat? regexp)
         (let ((f1 (match (exp1<-cat regexp)))
               (f2 (match (exp2<-cat regexp))))
           `(lambda (cl k)
              (,f1 cl (lambda (cl2) (,f2 cl2 k))))))
        ((star? regexp)
           (let ((f (match (exp<-star regexp))))
             `(lambda (cl k)
                (let loop ((cl2 cl))
                  (or (k cl2)
                      (,f cl2 (lambda (cl3)
                                (if (eq? cl2 cl3)
                                    #f
                                    (loop cl3)))))))))
        (else (error 'match "match's input is bad"))))
```

3.2.5 Output

When the regular expression is a*(a ∪ b), the simplified output becomes the following[4].

[4]Although this example is more elaborate, it too is merely intended to be illustrative. The algorithm to match a regular expression is inefficient: recall that union is implemented via backtracking.

```
(lambda (cl k)
  (let ((k
         (lambda (cl2)
           (or (if (null? cl2)
                   #f
                   (and (eq? (car cl2) #\a)
                        (k (cdr cl2))))
               (if (null? cl2)
                   #f
                   (and (eq? (car cl2) #\b)
                        (k (cdr cl2))))))))
    (let loop ((cl2 cl))
      (or (k cl2)
          (let ((cl cl2)
                (k (lambda (cl3)
                     (if (eq? cl2 cl3) #f (loop cl3)))))
            (if (null? cl)
                #f
                (and (eq? (car cl) #\a) (k (cdr cl)))))))))
```

3.3 Summary

The transformation technique is not difficult to perform manually. By using four rules (the transformation technique) we can turn an algorithm that computes a result into an algorithm that generates code to compute the result. Finally, we see that the generated code is respectable, but not breathtaking. The code that is generated can be no more subtle than the algorithm or interpreter it is based on.

Chapter 4

A Formal Model

Μεταβάλλον ἀναπαύεται

Even as it changes, it stands still.

Heraclitus

The examples from chapter 3 illustrate how a procedure can be modified so that it generates a λ-term. There the transformation rules were described informally. Here, we will describe them formally. In order to do so, it is first necessary to articulate a suitable language for writing interpreters. Once the interpreter language and the transformation rules have been articulated, it is possible to prove correctness results.

4.1 The Interpreter Language

The examples in chapter 3 are written in Scheme. More generally, we are inspired by Scheme, Common LISP, ML, and Haskell. Therefore, it is natural to construct a model interpreter language based on the call-by-value λ-calculus; this calculus is extended with constants, conditionals, and quotation (see figure 4.1). Constants and conditionals are commonplace. In addition, a let-form is understood in the usual way to abbreviate the

$$
\begin{array}{lrcl}
\text{Terms} & e & ::= & c \\
 & & ::= & x \\
 & & ::= & ,y \\
 & & ::= & \lambda\bar{x}.e \\
 & & ::= & [\![e]\!] \\
 & & ::= & e_0(\bar{e}) \\
 & & ::= & \mathsf{let}\ ,y = e\ \mathsf{in}\ e' \\
 & & ::= & \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
\text{Values} & v & ::= & c \\
 & & ::= & x \\
 & & ::= & \lambda\bar{x}.e \\
 & & ::= & [\![e]\!]
\end{array}
$$

$\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is syntactic sugar. (See below.)

$\mathsf{eval}(e)$ is syntactic sugar. (See below.)

Figure 4.1: Interpreter language syntax.

application of a lambda expression, or *abstraction*[1].

Definition 2. *The form* $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ *is syntactic sugar for* $(\lambda x.e')(e)$.

Definition 3. *The form* $\mathsf{let}\ x_1 = e_1, \cdots, x_n = e_n\ \mathsf{in}\ e$ *is syntactic sugar for*
$\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ e$.

There have been various approaches for describing program text in the context of the λ-calculus[2] [72, 74, 96]. Quotation and the let-form associated with quotation are inspired by Nanevski and Pfenning [74] and require more discussion.

In LISP, both programs and data are parenthesized expressions. Data is distinguished from programs by putting a quotation mark in front. Thus `(+ 2 3)` performs addition, but `'(+ 2 3)` is a list. In the interpreter language above, $+(2, 3)$ performs addition and

[1] We use the term "abstraction" when discussing both the meta-language (interpreter language) and the language.

[2] Other models of computation such as Turing machines and recursive function theory naturally incorporate the notion of program text.

$[\![+(2,3)]\!]$ is data. The notation here differs somewhat from LISP in that LISP allows an arbitrary form to be quoted; thus `'(1 2 3)` is simply a list of numbers. While the interpreter language syntax allows the term $[\![1(2,3)]\!]$, there is no other interpretation other than application and so the term does not make sense. LISP also makes it possible to substitute values into a quoted form. The comma operator is used to unquote an expression. Thus the LISP expression `(let ((y 2)) '(+ ,y 3))` evaluates to a list whose second component is `2`. In the interpreter language above, the comma is not an operator; rather a second kind of variable is introduced, the comma-variable, which is intended to resemble LISP's application of the comma operator to a variable. The let-form for comma-variables is used for substituting into a quoted expression. In the interpreter language, the comma example is written let $,y = 2$ in $[\![+((,y),3)]\!]$. Further, it is natural to use this let-form to define the operator eval.

Definition 4. *The form* eval(e) *is syntactic sugar for* let $,y = e$ in $,y$.

The meaning of this interpreter language λ-calculus is mostly standard (see figure 4.2). A function δ is assumed that characterizes how constant/primitive operators act on values. Again, the approach to modeling quotation requires some discussion. In LISP, we have that `(let ((y '(+ 3 4))) '(* 2 ,y))` evaluates to the list `(* 2 (+ 3 4))`. This can be understood as removing the quotation and replacing the comma-variable with the unquoted term. In LISP, the body of the let-form cannot be a comma-variable; in LISP the comma operator must appear inside a quasi-quote form. However, in the interpreter language it is possible. Observe that when the body of the let-form is the comma-variable, the quoted term is unquoted thereby implementing the eval operator.

But what should the result be when applying eval to an unquoted value, and more generally how should the let-form for comma-variables behave? Traditionally LISP allows the application of `eval` to unquoted expressions. Thus `(eval 2)` evaluates to `2`. In

$$\frac{\delta(c_{op}, \bar{v}) = v'}{c_{op}(\bar{v}) \to v'} \qquad \frac{}{(\lambda \bar{x}.e)(\bar{v}) \to e[\bar{x} \ := \ \bar{v}]}$$

$$\frac{v \neq [\![e]\!]}{\mathsf{let}\ , y = v\ \mathsf{in}\ e_b \to e_b[, y \ := \ v]} \qquad \frac{}{\mathsf{let}\ , y = [\![e]\!]\ \mathsf{in}\ e_b \to e_b[, y \ := \ e]}$$

$$\frac{}{\mathsf{if\ False\ then}\ e_1\ \mathsf{else}\ e_2 \to e_2} \qquad \frac{v \neq \mathsf{False}}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to e_1}$$

$$\frac{e_i \to e_i' \qquad 0 \leq i \leq n}{v_0(v_1 \cdots v_{i-1}\ e_i\ e_{i+1} \cdots e_n) \to v_0(v_1 \cdots v_{i-1}\ e_i'\ e_{i+1} \cdots e_n)}$$

$$\frac{e \to e'}{\mathsf{let}\ , y = e\ \mathsf{in}\ e_b \to \mathsf{let}\ , y = e'\ \mathsf{in}\ e_b} \qquad \frac{e \to e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}$$

Figure 4.2: Interpreter language semantics.

the discussion of syntax above, there was an example in which a comma-variable was bound to the unquoted value 2. However, the following example exposes a complication; consider `(let ((y (list 1 2 3))) '(car ,y))`. It evaluates to the list `(car (1 2 3))`, which is not what we wanted; we wanted the list `(car '(1 2 3))`. Should something additional happen with unquoted values or not? We argue that the example with the `2`, and *not* the example with the list, gets at the essence and that the anomaly with the list is due to unfortunate syntax. If we could write something like `(let ((y [1 2 3])) '(car ,y))` in LISP, there would not be a problem[3]. And so the rule for the let-form with comma-variables is merely to replace the comma-variable with the already unquoted value.

Reduction can be extended to an equivalence relation by making sure the relation is reflexive, symmetric, and transitive (see figure 4.3). In addition, when making arguments it is useful to be able to say that sub-structural equality implies equality. Hence the equational rules for structural equality are added.

[3]Lurking in the background is the issue of equality. Scheme has the operator `eq?` which is not extensional. We will assume that only extensional equality is used.

$$\frac{}{e = e}\ \text{reflexive} \qquad \frac{e = e'}{e' = e}\ \text{symmetric} \qquad \frac{e = e'\ e' = e''}{e = e''}\ \text{transitive}$$

$$\frac{e \to e'}{e = e'}\ \text{reduction} \qquad \frac{e \equiv_\alpha e'}{e = e'}\ \alpha \qquad \frac{e = e'}{\lambda\bar{x}.e = \lambda\bar{x}.e'}\ \xi$$

$$\frac{e_i = e_i' \qquad 0 \le i \le n}{e_0(e_1 \cdots e_{i-1}\ e_i\ e_{i+1} \cdots e_n) = e_0(e_1 \cdots e_{i-1}\ e_i'\ e_{i+1} \cdots e_n)}$$

$$\frac{e = e'}{\mathsf{let}\ , y = e\ \mathsf{in}\ e_b = \mathsf{let}\ , y = e'\ \mathsf{in}\ e_b} \qquad \frac{e = e'}{\mathsf{let}\ , y = e''\ \mathsf{in}\ e = \mathsf{let}\ , y = e''\ \mathsf{in}\ e'}$$

$$\frac{e = e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \qquad \frac{e = e'}{\mathsf{if}\ e_0\ \mathsf{then}\ e\ \mathsf{else}\ e_2 = \mathsf{if}\ e_0\ \mathsf{then}\ e'\ \mathsf{else}\ e_2}$$

$$\frac{e = e'}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e'} \qquad \frac{e = e'}{[\![e]\!] = [\![e']\!]}$$

Figure 4.3: Interpreter language term equality.

4.2 The Transformation Rules

In chapter 3, we mentioned that the transformation technique was inspired by denotational semantics. That is because a denotational definition can be understood as a compiler: given a term, we are free to evaluate the recursive calls and derive a λ-term. Yet the call-by-value evaluation strategy prevents reducing the applications inside abstractions. The key idea behind the transformations is the following: An expression within an abstraction cannot be evaluated, and so the code is restructured so that the expression is no longer within the abstraction. The formal transformations are in figure 4.4.

Rules (4.1) and (4.2) are about currying. The equivalence of functions and their curried counterparts is well known. Although the rules are expressed as local changes, rule (4.2) must be applied completely using non-local assumptions and information.

Rules (4.3) and (4.4) are about lambda lowering. These rules involve moving an expression that is just inside an abstraction and does not depend on the parameters of an abstraction out of the abstraction. In particular, if the abstraction body is a conditional, but the conditional does not depend on the abstraction's parameters, we may regard the

$$\lambda\bar{s}, \bar{d}.e \hookrightarrow \lambda\bar{s}.\lambda\bar{d}.e \tag{4.1}$$

$$e_c(\bar{e_s}, \bar{e_d}) \hookrightarrow e_c(\bar{e_s})(\bar{e_d}) \tag{4.2}$$

if e_c is an expression that reduces to a curried function.

$$\lambda\bar{x}.\textsf{if } e \textsf{ then } e_1 \textsf{ else } e_2 \hookrightarrow \textsf{if } e \textsf{ then } (\lambda\bar{x}.e_1) \textsf{ else } (\lambda\bar{x}.e_2) \quad \textsf{if } x_i \notin \textsf{FV}(e) \tag{4.3}$$

$$\lambda\bar{x}.\textsf{let } z = e \textsf{ in } e_b \hookrightarrow \textsf{let } z = e \textsf{ in } \lambda\bar{x}.e_b \quad \textsf{if } x_i \notin \textsf{FV}(e) \textsf{ and } z \neq x_i \tag{4.4}$$

$$\lambda\bar{x}.e'[u \ := \ e] \hookrightarrow \textsf{let } z = e \textsf{ in } \lambda\bar{x}.e'[u \ := \ z] \quad \textsf{if } z \textsf{ is fresh and } x_i \notin \textsf{FV}(e) \tag{4.5}$$

$$\begin{aligned}&\textsf{let } z_1 = e_1 \textsf{ in } \cdots \textsf{ let } z_n = e_n \textsf{ in } \lambda\bar{d}.e \hookrightarrow \\ &\textsf{let } , z_1 \ = \ e_1 \textsf{ in } \cdots \\ &\quad \textsf{let } , z_n \ = \ e_n \textsf{ in} \\ &\qquad [\![\lambda\bar{d}.e[z_1 \ := , z_1] \cdots [z_n \ := , z_n]]\!] \\ &\textsf{when } \textsf{FV}(\lambda\bar{d}.e) = \{z_1, \ldots, z_n\}\textsf{, each } , z_i \textsf{ is fresh,} \\ &\textsf{and } \textsf{FV}(e_i) \cap \{z_1, \ldots, z_n\} = \varnothing \textsf{ for each } i\end{aligned} \tag{4.6}$$

The transformation relation is denoted by $\hookrightarrow$.

Figure 4.4: Transformations

conditional as specifying one of two abstractions. Or, if the abstraction body defines an intermediate value that does not depend on the parameters, we may regard the definition as occurring outside the body of the abstraction.

Rule (4.5) is expression lifting. This rule is similar to lambda lowering insofar as both involve moving an expression out of an abstraction. However, with expression lifting, the entire expression is moved completely out of the abstraction if it does not depend on the parameters of the abstraction. Typically, the expression being lifted is an application.

Rule (4.6) is about quotation. It transforms an expression that returns an abstraction into an expression that returns the text that represents that abstraction. Note that the rule states that the variables that become comma-variables are exactly those that are in scope from the surrounding let. In practice, we relax this restriction slightly and allow a function's formal parameters to be unquoted without being in a let. Following the formal requirement of the rule in that case is trivial but wordy.

4.3 Local Correctness

The correctness of rules (4.3), (4.4), and (4.5) relies only on local reasoning. Note that they all assume that the evaluation of e terminates. If that is not the case, looping outside of an abstraction is always observed, but looping inside an abstraction is observed only if the abstraction is called. In practice, it is clear for rules (4.3) and (4.4) whether or not e terminates: typically it is a call to a structure predicate and it does not loop. The termination of e in rule (4.5) is more subtle. If it is a recursive call on sub-structure it will terminate. If it is a recursive call on the same structure it will not terminate. Otherwise, termination is not obvious.

For rule (4.3), concerning lambda lowering for a conditional, we first need a technical lemma. (The proof of the lemma is in appendix A.)

Lemma 1. *If* $e \to^* v$ *then* $(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \to^* (\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)$.

Informally, the argument for the correctness of rule (4.3) is that if e reduces to a value v, then the body of the abstraction depends on v. When false, the body is e_2; otherwise the body is e_1. And that is what the right-hand-side says.

Theorem 1. *If* $e \to^* v$, *and* $x_i \notin \mathsf{FV}(e)$ *then* $(\lambda\bar{x}.\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) = (\mathsf{if}\ e\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2)$.

Proof. By case analysis on v.

- Suppose $v \neq \mathsf{False}$.

$$\begin{aligned}\lambda\bar{x}.\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &= \lambda\bar{x}.\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ &= \lambda\bar{x}.e_1 \\ &= \mathsf{if}\ v\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2 \\ &= \mathsf{if}\ e\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2\end{aligned}$$

- Suppose $v = \mathsf{False}$.

  The argument is similar.

□

For rule (4.4), concerning lambda lowering for a let-binding, we first need a technical lemma. (The proof of the lemma is in appendix A.)

Lemma 2. *If* $e \to^* v$ *then* $(\mathsf{let}\ z = e\ \mathsf{in}\ e_b) \to^* (\mathsf{let}\ z = v\ \mathsf{in}\ e_b)$.

Informally, the argument for the correctness of rule (4.4) is that if e reduces to a value v, then the let on the left-hand-side substitutes v for z in e_b. The let on the right-hand-side

substitutes v for z in the abstraction, but it passes right through and becomes a substitution in e_b since z is distinct from the formal parameters.

Theorem 2. *If* $e \rightarrow^* v$, $z \neq x_i$, *and* $x_i \notin \mathsf{FV}(e)$ *then* $(\mathsf{let}\ z = e\ \mathsf{in}\ \lambda\bar{x}.e_b) = (\lambda\bar{x}.\mathsf{let}\ z = e\ \mathsf{in}\ e_b)$.

Proof.

$$\begin{aligned}
\mathsf{let}\ z = e\ \mathsf{in}\ \lambda\bar{x}.e_b &= \mathsf{let}\ z = v\ \mathsf{in}\ \lambda\bar{x}.e_b \\
&= (\lambda\bar{x}.e_b)[z\ :=\ v] \\
&= \lambda\bar{x}.(e_b[z\ :=\ v]) \\
&= \lambda\bar{x}.\mathsf{let}\ z = v\ \mathsf{in}\ e_b \\
&= \lambda\bar{x}.\mathsf{let}\ z = e\ \mathsf{in}\ e_b
\end{aligned}$$

□

For rule (4.5), concerning expression lifting, we first need a technical lemma. (The proof of the lemma is in appendix A.)

Lemma 3. *If* $e' = v$ *then* $(e[u\ :=\ e']) = (e[u\ :=\ v])$.

Informally, the argument for the correctness of rule (4.5) is that if e reduces to a value v, then the body of the abstraction on the left-hand-side will replace u with v. The let on the right-hand-side also ultimately replaces u with v since the substitution for z passes right through the abstraction.

Theorem 3. *If* $e' \rightarrow^* v$, z *is fresh, and* $x_i \notin \mathsf{FV}(e')$ *then* $(\mathsf{let}\ z = e'\ \mathsf{in}\ \lambda\bar{x}.e[u := z]) = (\lambda\bar{x}.e[u := e'])$.

Proof.

$$\begin{aligned}
\mathsf{let}\ z = e'\ \mathsf{in}\ \lambda\bar{x}.e[u := z] &= \mathsf{let}\ z = v\ \mathsf{in}\ \lambda\bar{x}.e[u := z] \\
&= (\lambda\bar{x}.e[u := z])[z := v] \\
&= \lambda\bar{x}.(e[u := z][z := v]) \\
&= \lambda\bar{x}.e[u := v] \\
&= \lambda\bar{x}.e[u := e']
\end{aligned}$$

□

With rule (4.6), the transformed expression reduces to a different value from the original, and so here the notion of correctness is different.

4.4 Global Correctness of a Sum Language Example

Correctness for rule (4.6) means that applying the eval operator to the text that results from applying the transformed interpreter results in the same value that the original interpreter yields. Rule (4.6) requires non-local information and assumptions; it must be applied to all branches of a conditional. Here we consider a concrete sum language example. For the sake of brevity, we write the evaluator in pseudo-code based on ML and Haskell that can readily be translated into the interpreter language. It is used to implement a simple sum language evaluator for a language involving numbers, variables, and sums. The evaluator is transformed, and the resulting compiler is proved correct.

First it is necessary to define the sum language. We can imagine that N refers to a set of numbers, V refers to a set of variables, and S is a constructor that builds syntactic sums.

Definition 5. *Given sets N and V, $L(N, V)$ is the smallest set satisfying the following.*

- $n \in L(N, V)$ *if* $n \in N$, *and*
- $x \in L(N, V)$ *if* $x \in V$, *and*
- $\mathsf{S}(t_1, t_2) \in L(N, V)$ *if* $t_1, t_2 \in L(N, V)$.

Now we can define the sum language evaluator. We can imagine that a is the operator that applies an environment to a variable, and that p is the plus function.

$\mathcal{E} : L(N, V) \times R \to N$
$\mathcal{E}(n, \rho) = n$
$\mathcal{E}(x, \rho) = a(\rho, x)$
$\mathcal{E}(\mathsf{S}(t_1, t_2), \rho) = p(\mathcal{E}(t_1, \rho), \mathcal{E}(t_2, \rho))$

Note that the interpreter language in figure 4.1 does not include a form in which a function is defined by a set of equations. Rather, this pseudo-code is shorthand for the following term, where $\mathcal{E}$ is a variable, $\perp$ is a constant, Y is the call-by-value fixed-point/recursion operator, $t \in N$ refers to a number predicate, $t \in V$ refers to a variable predicate, $\mathsf{s?}$ is the sum predicate, and s_1 and s_2 are the sum selectors.

let $\mathcal{E} = \mathsf{Y}(\lambda f.\ \lambda(t, \rho).$
  if $t \in N$ then t
  else if $t \in V$ then $a(\rho, t)$
  else if $\mathsf{s?}(t)$ then $p(f(\mathsf{s}_1(t), \rho), f(\mathsf{s}_2(t), \rho))$
  else $\perp)$
in $\mathcal{E}$

4.4.1 Applying the Transformations

In the following sub-sections, we will now apply the technique to this evaluator and derive a code generator.

Currying and Lambda Lowering

A compiler for sum expressions must be a function that takes a sum expression. The dynamic parameter is the environment variable ρ because values for variables are not known until run-time. This parameter is removed from the top-level parameter list and put into the parameter list of the λ-expression. The recursive calls are modified to account for this new protocol. Since, in this shorthand, currying involves introducing a λ and moving a parameter from the left of the equal-sign to the right, lambda lowering occurs as well.

$\mathcal{E}_1(n) = \lambda\rho.n$
$\mathcal{E}_1(x) = \lambda\rho.a(\rho, x)$
$\mathcal{E}_1(\mathsf{S}(t_1, t_2)) = \lambda\rho.p(\mathcal{E}_1(t_1)(\rho), \mathcal{E}_1(t_2)(\rho))$

Expression Lifting

Since the recursive calls have been curried and do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. It is clear that the calls will halt since the recursive calls are always on smaller structures.

$\mathcal{E}_2(n) = \lambda\rho.n$
$\mathcal{E}_2(x) = \lambda\rho.a(\rho, x)$
$\mathcal{E}_2(\mathsf{S}(t_1, t_2)) = \mathsf{let}\ f_1 = \mathcal{E}_2(t_1), f_2 = \mathcal{E}_2(t_2)\ \mathsf{in}\ \lambda\rho.p(f_1(\rho), f_2(\rho))$

Quoting

Now each λ-expression is quoted, and the let-bound variables are unquoted.

$\mathcal{E}_3(n) = [\![\lambda\rho.n]\!]$
$\mathcal{E}_3(x) = [\![\lambda\rho.a(\rho, x)]\!]$
$\mathcal{E}_3(\mathsf{S}(t_1, t_2)) = \mathsf{let}\ , f_1 = \mathcal{E}_3(t_1),\ , f_2 = \mathcal{E}_3(t_2)\ \mathsf{in}\ [\![\lambda\rho.p((, f_1)(\rho), (, f_2)(\rho))]\!]$

4.4.2 Correctness

Correctness of the compiler $\mathcal{E}_3$ means that the text that $\mathcal{E}_3$ generates, when evaluated and supplied with the dynamic parameters, produces the same result as the evaluator $\mathcal{E}$. This result is established by showing the text $\mathcal{E}_3$ generates is the same as $\mathcal{E}$.

Theorem 4. *For any $t \in L(N, V)$, $\mathcal{E}_3(t) = [\![\lambda\rho.\mathcal{E}(t, \rho)]\!]$.*

Proof. By structural induction on t.

- Suppose $t = n$. $\mathcal{E}_3(n) = [\![\lambda\rho.n]\!] = [\![\lambda\rho.\mathcal{E}(n, \rho)]\!]$
- Suppose $t = x$. $\mathcal{E}_3(x) = [\![\lambda\rho.a(\rho, x)]\!] = [\![\lambda\rho.\mathcal{E}(x, \rho)]\!]$
- Suppose $t = \mathsf{S}(t_1, t_2)$.

$$
\begin{aligned}
\mathcal{E}_3(\mathsf{S}(t_1, t_2)) &= \mathsf{let}\ , f_1 = \mathcal{E}_3(t_1),\ , f_2 = \mathcal{E}_3(t_2)\ \mathsf{in}\ [\![\lambda\rho.p((, f_1)(\rho), (, f_2)(\rho))]\!] \\
&= \mathsf{let}\ , f_1 = [\![\lambda\rho.\mathcal{E}(t_1, \rho)]\!],\ , f_2 = [\![\lambda\rho.\mathcal{E}(t_2, \rho)]\!]\ \mathsf{in}\ [\![\lambda\rho.p((, f_1)(\rho), (, f_2)(\rho))]\!] \\
&= [\![\lambda\rho.p((\lambda\rho.\mathcal{E}(t_1, \rho))(\rho), (\lambda\rho.\mathcal{E}(t_2, \rho))(\rho))]\!] \\
&= [\![\lambda\rho.p(\mathcal{E}(t_1, \rho), \mathcal{E}(t_2, \rho))]\!] \\
&= [\![\lambda\rho.\mathcal{E}(\mathsf{S}(t_1, t_2), \rho)]\!]
\end{aligned}
$$

□

Given theorem 4, the correctness result is merely a matter of applying the eval operator to eliminate the quotation.

Corollary 1. *For any $t \in L(N, V)$, for any $\rho \in R$,* $\mathsf{eval}(\mathcal{E}_3(t))(\rho) = \mathcal{E}(t, \rho)$.

Proof.

$$\begin{aligned}\mathsf{eval}(\mathcal{E}_3(t))(\rho) &= \mathsf{eval}([\![\lambda\rho.\mathcal{E}(t,\rho)]\!])(\rho) \\ &= (\lambda\rho.\mathcal{E}(t,\rho))(\rho) \\ &= \mathcal{E}(t,\rho)\end{aligned}$$

□

4.5 Global Correctness of an Abstract Denotational Example

Although the evaluator in section 4.4 was described fairly concretely, two key functions a and p were never formally defined. Therefore it is possible to view that evaluator as an abstract interpreter. In this section, we take that sort of abstraction to the extreme so that we can claim correctness for all denotational-style interpreters of this form.

Here too it is necessary to define the language the evaluator will operate on. Instead of being dependent on two sets N and V, we generalize and allow for dependence on a collection of sets $\bar{X}$. An element from one of the sets in the collection $\bar{X}$ is a base case; the operators C_j are constructors that involve sub-terms and possibly elements from sets in the collection.

Definition 6. *Given a finite collection of sets $\bar{X}$, $L(\bar{X})$ is the smallest set satisfying the following.*

- $x_i \in L(\bar{X})$ *if* $x_i \in X_i$, *and*
- $\mathsf{C}_j(\bar{x_j}, \bar{t_j}) \in L(\bar{X})$ *if* $x_j^i \in X_i$ *and* $t_j^i \in L(\bar{X})$.

Now we can define the abstract evaluator. In addition to terms from the language $L(\bar{X})$, it also takes some number of static parameters and some number of dynamic parameters and returns an answer. If the first input is a base case, the result is a function (g_i) of a computation of the input with the static parameters (h_i^{s}) and a computation of the input with the dynamic parameters (h_i^{d}). If the first input is a compound structure characterized by a constructor, the result is a function (g_j) of a computation of the non-recursive part of the input with the static parameters (h_j^{cs}), a computation of the non-recursive part of the input with the dynamic parameters (h_j^{cd}), and recursive calls on the recursive parts of the input.

$\mathcal{E} : L(\bar{X}) \times \bar{S} \times \bar{D} \to A$
$\mathcal{E}(x_i, \bar{s}, \bar{d}) = g_i(h_i^{\mathsf{s}}(\bar{s}, x_i), h_i^{\mathsf{d}}(\bar{d}, x_i))$
$\mathcal{E}(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}, \bar{d}) = g_j(h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), \mathcal{E}(t_j^1, \bar{s}, \bar{d}), \cdots, \mathcal{E}(t_j^{|\bar{t_j}|}, \bar{s}, \bar{d}))$

Again, shorthand notation is used to express the following term, where it is assumed that there is a predicate to test $t \in X_i$, $\mathsf{c}_j?$ are predicates, $\mathsf{c}_j^{\mathsf{x}}$ is a selector for the non-recursive component of the jth compound structure, and c_j^i are the selectors for the recursive components of the jth compound structure.

$$
\begin{array}{l}
\mathsf{let}\ \mathcal{E} = \mathsf{Y}(\lambda f.\ \lambda(t, \bar{s}, \bar{d}). \\
\qquad\qquad \mathsf{if}\ \cdots \\
\qquad\qquad \mathsf{else\ if}\ t \in X_i\ \mathsf{then}\ g_i(h_i^{\mathsf{s}}(\bar{s}, t), h_i^{\mathsf{d}}(\bar{d}, t)) \\
\qquad\qquad \vdots \\
\qquad\qquad \mathsf{else\ if}\ \mathsf{c}_j?(t)\ \mathsf{then}\ g_j(h_j^{\mathsf{cs}}(\bar{s}, \mathsf{c}_j^{\mathsf{x}}(t)), h_j^{\mathsf{cd}}(\bar{d}, \mathsf{c}_j^{\mathsf{x}}(t)), f(\mathsf{c}_j^1(t), \bar{s}, \bar{d}), \cdots) \\
\qquad\qquad \vdots \\
\qquad\qquad \mathsf{else}\ \bot) \\
\mathsf{in}\ \mathcal{E}
\end{array}
$$

4.5.1 Applying the Transformations

In the following sub-sections, we will now apply the technique to this abstract evaluator and derive an abstract code generator.

Currying and Lambda Lowering

Based on the notation used, it is clear which variables to curry. The dynamic parameters are removed from the top-level parameter list and put into the parameter list of the λ-expression. The recursive calls are modified to account for this new protocol. Since, in this shorthand, currying involves introducing a λ and moving a parameter from the left of the equal-sign to the right, lambda lowering occurs as well. While it doesn't occur in this formulation, were there an if or a let on the right-hand side, there would be the need for additional lambda lowering.

$$
\begin{array}{l}
\mathcal{E}_1(x_i, \bar{s}) = \lambda\bar{d}.g_i(h_i^{\mathsf{s}}(\bar{s}, x_i), h_i^{\mathsf{d}}(\bar{d}, x_i)) \\
\mathcal{E}_1(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}) = \lambda\bar{d}.g_j(h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), \mathcal{E}_1(t_j^1, \bar{s})(\bar{d}), \cdots, \mathcal{E}_1(t_j^{|\bar{t_j}|}, \bar{s})(\bar{d}))
\end{array}
$$

Expression Lifting

Since the recursive calls have been curried and do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. It is clear that the calls will halt since the recursive calls are always on smaller structures. There are also non-recursive calls involving the static parameters that can be lifted out. We explicitly assume that those functions are total.

$$
\begin{array}{l}
\mathcal{E}_2(x_i, \bar{s}) = \mathsf{let}\ y = h_i^{\mathsf{s}}(\bar{s}, x_i)\ \mathsf{in}\ \lambda\bar{d}.g_i(y, h_i^{\mathsf{d}}(\bar{d}, x_i)) \\
\mathcal{E}_2(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}) = \\
\quad \mathsf{let}\ \ y = h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), \\
\qquad\quad u_1 = \mathcal{E}_2(t_j^1, \bar{s}), \\
\qquad\quad \vdots \\
\qquad\quad u_{|\bar{t_j}|} = \mathcal{E}_2(t_j^{|\bar{t_j}|}, \bar{s}) \\
\quad \mathsf{in}\ \ \lambda\bar{d}.g_j(y, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), u_1(\bar{d}), \cdots, u_{|\bar{t_j}|}(\bar{d}))
\end{array}
$$

Quoting

Now each λ-expression is quoted, and the let-bound variables are unquoted.

$$
\begin{array}{l}
\mathcal{E}_3(x_i, \bar{s}) = \mathsf{let}\ , y = h_i^{\mathsf{s}}(\bar{s}, x_i)\ \mathsf{in}\ [\![\lambda\bar{d}.g_i(, y, h_i^{\mathsf{d}}(\bar{d}, x_i))]\!] \\
\mathcal{E}_3(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}) = \\
\quad \mathsf{let}\ \ , y = h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), \\
\qquad\quad , u_1 = \mathcal{E}_3(t_j^1, \bar{s}), \\
\qquad\quad \vdots \\
\qquad\quad , u_{|\bar{t_j}|} = \mathcal{E}_3(t_j^{|\bar{t_j}|}, \bar{s}) \\
\quad \mathsf{in}\ \ [\![\lambda\bar{d}.g_j(, y, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), (, u_1)(\bar{d}), \cdots, (, u_{|\bar{t_j}|})(\bar{d}))]\!]
\end{array}
$$

4.5.2 Correctness

Again, correctness of the compiler $\mathcal{E}_3$ means that the text that $\mathcal{E}_3$ generates, when evaluated and supplied with the dynamic parameters, produces the same result as the evaluator $\mathcal{E}$. This result is established by showing the text $\mathcal{E}_3$ generates is the same as $\mathcal{E}$. For this result, a little more must be assumed.

Definition 7. *Given a term h, h is* total *if for any terms $\bar{e}$, $h(\bar{e}) = v$.*

Definition 8. *Given a term h, h is* quote free *if for any terms $\bar{e}$, $h(\bar{e}) = v$ implies $v \neq [\![e']\!]$.*

As mentioned in section 4.5.1, because functions other than recursive calls are lifted out, there must be an explicit assumption that those functions are total. Further, it has been an implicit assumption that the evaluator makes no use of quoted terms. That assumption becomes explicit here.

Theorem 5. *For any $t \in L(\bar{X})$, for any $\bar{s} \in \bar{S}$, if h_i^{s} and h_j^{cs} are total and quote free, then $\mathcal{E}_3(t, \bar{s}) = [\![\lambda \bar{d}.\mathcal{E}(t, \bar{s}, \bar{d})]\!]$.*

Proof. By structural induction on t.

- Suppose $t = x_i$. Since h_i^{s} is total and quote free, $h_i^{\mathsf{s}}(\bar{s}, x_i) = v$ and $v \neq [\![e]\!]$.

$$
\begin{aligned}
\mathcal{E}_3(x_i, \bar{s}) &= \mathsf{let}\ , y = h_i^{\mathsf{s}}(\bar{s}, x_i)\ \mathsf{in}\ [\![\lambda \bar{d}.g_i(, y, h_i^{\mathsf{d}}(\bar{d}, x_i))]\!] \\
&= \mathsf{let}\ , y = v\ \mathsf{in}\ [\![\lambda \bar{d}.g_i(, y, h_i^{\mathsf{d}}(\bar{d}, x_i))]\!] \\
&= [\![\lambda \bar{d}.g_i(v, h_i^{\mathsf{d}}(\bar{d}, x_i))]\!] \\
&= [\![\lambda \bar{d}.g_i(h_i^{\mathsf{s}}(\bar{s}, x_i), h_i^{\mathsf{d}}(\bar{d}, x_i))]\!] \\
&= [\![\lambda \bar{d}.\mathcal{E}(x_i, \bar{s}, \bar{d})]\!]
\end{aligned}
$$

- Suppose $t = \mathsf{C}_j(\bar{x_j}, \bar{t_j})$. Since h_j^{cs} is total and quote free, $h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}) = v$ and $v \neq [\![e]\!]$.

$$
\begin{aligned}
\mathcal{E}_3(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}) = \mathsf{let}\ & , y = h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), \\
& , u_1 = \mathcal{E}_3(t_j^1, \bar{s}), \\
& \vdots \\
& , u_{|\bar{t_j}|} = \mathcal{E}_3(t_j^{|\bar{t_j}|}, \bar{s}) \\
\mathsf{in}\ & [\![\lambda \bar{d}.g_j(, y, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), (, u_1)(\bar{d}), \cdots, (, u_{|\bar{t_j}|})(\bar{d}))]\!] \\
= \mathsf{let}\ & , y = v, \\
& , u_1 = [\![\lambda \bar{d}.\mathcal{E}(t_j^1, \bar{s}, \bar{d})]\!], \\
& \vdots \\
& , u_{|\bar{t_j}|} = [\![\lambda \bar{d}.\mathcal{E}(t_j^{|\bar{t_j}|}, \bar{s}, \bar{d})]\!] \\
\mathsf{in}\ & [\![\lambda \bar{d}.g_j(, y, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), (, u_1)(\bar{d}), \cdots, (, u_{|\bar{t_j}|})(\bar{d}))]\!] \\
= [\![\lambda \bar{d}.g_j(v, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), (\lambda \bar{d}.\mathcal{E}(t_j^1, \bar{s}, \bar{d}))(\bar{d}), \cdots, (\lambda \bar{d}.\mathcal{E}(t_j^{|\bar{t_j}|}, \bar{s}, \bar{d}))(\bar{d}))]\!] \\
= [\![\lambda \bar{d}.g_j(v, h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), \mathcal{E}(t_j^1, \bar{s}, \bar{d}), \cdots, \mathcal{E}(t_j^{|\bar{t_j}|}, \bar{s}, \bar{d}))]\!] \\
= [\![\lambda \bar{d}.g_j(h_j^{\mathsf{cs}}(\bar{s}, \bar{x_j}), h_j^{\mathsf{cd}}(\bar{d}, \bar{x_j}), \mathcal{E}(t_j^1, \bar{s}, \bar{d}), \cdots, \mathcal{E}(t_j^{|\bar{t_j}|}, \bar{s}, \bar{d}))]\!] \\
= [\![\lambda \bar{d}.\mathcal{E}(\mathsf{C}_j(\bar{x_j}, \bar{t_j}), \bar{s}, \bar{d})]\!]
\end{aligned}
$$

□

Again, the correctness result follows immediately from theorem 5, and so the assumptions of the theorem must be duplicated.

Corollary 2. *For any $t \in L(\bar{X})$, for any $\bar{s} \in \bar{S}$, for any $\bar{d} \in \bar{D}$, if h_i^{s} and h_j^{cs} are total and quote free, then* $\mathsf{eval}(\mathcal{E}_3(t, \bar{s}))(\bar{d}) = \mathcal{E}(t, \bar{s}, \bar{d})$.

Thus the transformation technique works for denotational-style interpreters. This time we omit the proof since the argument is essentially the same as the argument for corollary 1.

Chapter 5

Beyond Denotational Interpreters

فحررت هذا الكتاب و جمعت فيه
جميع ما يحتاج اليه الحاسب محترزا
عن اشباع ممل و اختصار مخل

I wrote this book and compiled in it everything that is necessary for the computer, avoiding both boring verbosity and misleading brevity.

Ghiyath al-Din Jamshid al-Kashi

As we have seen, the transformation technique can be applied to an interpreter written in a denotational style. But if one attempts to apply the transformation technique directly to a non-denotational interpreter, it is quite likely that the transformation technique will fail. Nevertheless, often such an interpreter can be modified so that minor changes put it back in the realm of the denotational.

This chapter focuses on specific ways in which interpreters and interpreter related algorithms might not be denotational and gives examples of such algorithms and of an appropriate modification. The first section discusses the issue of the static and the dynamic

being entangled. The second and third sections discuss issues that can arise from an operational style interpreter. The last section summarizes the modifications. Throughout this chapter pseudo-code is used that resembles ML and Haskell.

5.1 Disentangling the Static and the Dynamic

When writing an interpreter it can happen that some parameter that should be static is dependent on a dynamic parameter. In particular, pattern matching and unification algorithms for very high level languages often suffer from this problem. If the static and dynamic are entangled then the transformation technique cannot be applied because the lowering and lifting rules will be blocked. Often such an entanglement is the result of the algorithm having to immediately check the value of a dynamic parameter that is passed back into the function because its type is the Maybe type (or something similar). The modification is then to express the algorithm using continuations instead.

5.1.1 Unification Example

We consider the concrete example of unification. In order to discuss unification algorithms, it is first necessary to define what the domain of the algorithm is. It is also worthwhile to formally define the meaning of unification. With that background, unification algorithms and their interaction with the transformation technique can be discussed.

The set of terms about which we ask the question of unification is defined as follows.

$$\begin{array}{lll} \mathsf{Term} & ::= & n \\ & ::= & s \\ & ::= & X \\ & ::= & [] \\ & ::= & \mathsf{Cons}(t_1, t_2) \end{array}$$

where n is a number, $s \in \mathsf{Sym}$ is a symbol, $X \in \mathsf{Var}$ is a variable, and t_k is a Term.

Definition 9. *A* substitution θ *is a function* $\theta : \mathsf{Var} \to \mathsf{Term}$.

Definition 10. *Given a substitution θ, θ^* is the function $\theta^* : \mathsf{Term} \to \mathsf{Term}$ that extends the domain of θ to all terms.*

$$\begin{array}{lcl} \theta^*(n) & = & n \\ \theta^*(s) & = & s \\ \theta^*(X) & = & \theta(X) \\ \theta^*([]) & = & [] \\ \theta^*(\mathsf{Cons}(t_1, t_2)) & = & \mathsf{Cons}(\theta^*(t_1), \theta^*(t_2)) \end{array}$$

Definition 11. *Given terms $t_1, t_2 \in \mathsf{Term}$, t_1 and t_2* unify *if there exists a substitution θ such that $\theta^*(t_1) = \theta^*(t_2)$. Such a substitution is called a* unifier.

To verify that two terms unify, it is simplest to rely on the definition above. But a somewhat more efficient verifier combines equality checking with the use of the substitution leading to an algorithm that performs structural equality on each kind of term while applying the unifier to variables. The substitution, when a unifier, can be viewed as a certificate that proves the terms unify. A starting point for a unification algorithm is this verification procedure; however, instead of taking a certificate as a parameter, the unification algorithm must take a substitution and return a certificate or an indication of failure. This parameter is initialized to be the identity substitution. A natural choice for the parameter type and the return type is $\mathsf{Maybe}(U)$, where U is the type of the representation of the substitution[1]. These ideas lead to the following unification algorithm.

[1]The substitution could be represented by a function type, but since the infinite portion is the identity function, it could also be represented as a finite data structure such as a set or list.

$\mathcal{U}$: Term $\times$ Term $\times$ Maybe$(U) \rightarrow$ Maybe(U)
$\mathcal{U}(\_, \_, \mathsf{Nothing}) = \mathsf{Nothing}$
$\mathcal{U}(n_1, n_2, \theta) =$ if $n_1 = n_2$ then θ else Nothing
$\mathcal{U}(s_1, s_2, \theta) =$ if $s_1 = s_2$ then θ else Nothing
$\mathcal{U}(X, t_2, \theta) = \mathcal{U}_{\mathsf{Var}}(t_2, X, \theta)$
$\mathcal{U}(t_1, X, \theta) = \mathcal{U}_{\mathsf{Var}}(t_1, X, \theta)$
$\mathcal{U}([], [], \theta) = \theta$
$\mathcal{U}(\mathsf{Cons}(t_1, t_2), \mathsf{Cons}(t'_1, t'_2), \theta) = \mathcal{U}(t_2, t'_2, \mathcal{U}(t_1, t'_1, \theta))$
$\mathcal{U}(\_, \_, \theta) = \mathsf{Nothing}$

Note that the order of the clauses above is significant. The function $\mathcal{U}_{\mathsf{Var}}$ is a specialized unification function; its second argument must be a variable. We omit the details of $\mathcal{U}_{\mathsf{Var}}$.

5.1.2 Applying the Technique: A First Attempt

We now apply the transformation technique to the above unification algorithm. It is necessary to decide what is static and what is dynamic. The unifier must be dynamic, in general. It seems reasonable to allow both terms to be static; although we may reconsider this choice in the future.

When written in the form above, currying and some lambda lowering happen at once. When attempting to apply those transformations, we see that lambda lowering cannot be performed. We are stuck. To emphasize the point, consider the algorithm written in Scheme notation.

```
(define (unify t1 t2 theta)
  (if (nothing? theta)
      *nothing*
      (cond ((and (number? t1) (number? t2))
             (if (= t1 t2) theta *nothing*))
             ...
                )))
```

Currying is not a problem:

```
(define (unify t1 t2)
  (lambda (theta)
     (if (nothing? theta)
         *nothing*
         (cond ((and (number? t1) (number? t2))
                (if (= t1 t2) theta *nothing*))
                ...
                   ))))
```

Notice that lambda lowering is not possible because the free variables of the test expression include the formal parameter `theta`. In this sense the static and the dynamic are entangled.

We suggest that the Maybe type[2] may be harmful for staged computation. Another approach involves using continuations instead. Continuation passing style has been used successfully in partial evaluation: a static term stuck in a dynamic context is replaced with a static calculation coupled with a dynamic continuation parameter. Although the issue of context here is in some ways similar since the inner conditional is in a dynamic context, making the context a parameter is not viable. Further, such a context exists only to check if it is necessary to fail right away. Continuations allow for a more natural[3] and efficient way to express such a notion.

5.1.3 Revising the Algorithm

To solve the problem from the previous section we introduce two new dynamic parameters: a success continuation and a failure continuation. Now the θ parameter will represent only one thing: the unifier. The new unification algorithm now follows.

[2]Both Maybe and continuations are examples of monads. That provided a hint for the switch.
[3]An alternative would be to move the test on θ just before it is needed and into the case for cons.

$\mathcal{U} : \mathsf{Term} \times \mathsf{Term} \times U \times (U \to \mathsf{Maybe}(U)) \times (() \to \mathsf{Maybe}(U)) \to \mathsf{Maybe}(U)$
$\mathcal{U}(n_1, n_2, \theta, \kappa_s, \kappa_f) = \mathsf{if}\ n_1 = n_2\ \mathsf{then}\ \kappa_s(\theta)\ \mathsf{else}\ \kappa_f()$
$\mathcal{U}(s_1, s_2, \theta, \kappa_s, \kappa_f) = \mathsf{if}\ s_1 = s_2\ \mathsf{then}\ \kappa_s(\theta)\ \mathsf{else}\ \kappa_f()$
$\mathcal{U}(X, t_2, \theta, \kappa_s, \kappa_f) = \mathcal{U}_{\mathsf{Var}}(t_2, X, \theta, \kappa_s, \kappa_f)$
$\mathcal{U}(t_1, X, \theta, \kappa_s, \kappa_f) = \mathcal{U}_{\mathsf{Var}}(t_1, X, \theta, \kappa_s, \kappa_f)$
$\mathcal{U}([], [], \theta, \kappa_s, \kappa_f) = \kappa_s(\theta)$
$\mathcal{U}(\mathsf{Cons}(t_1, t_2), \mathsf{Cons}(t_1', t_2'), \theta, \kappa_s, \kappa_f) = \mathcal{U}(t_1, t_1', \theta, (\lambda\theta'.\mathcal{U}(t_2, t_2', \theta', \kappa_s, \kappa_f)), \kappa_f)$
$\mathcal{U}(\_, \_, \theta, \kappa_s, \kappa_f) = \kappa_f()$

We observe that the transformation technique can be applied to this version successfully with the caveat that some dynamic unification may be necessary because the dynamic unifier parameter could contain terms to unify.

5.2 Introducing Explicit Fixed-Points

Interpreters may be based on an operational semantics rather than a denotational semantics. The transformation technique may then fail to be applicable. In particular, iteration constructs are often defined in terms of themselves in operational-style interpreters. The transformation technique will lead to infinite loops on this sort of expansive recursion. Following Gunter [45], we solve this problem in the interpreter by explicitly identifying the fixed-point and eliminating the expansive recursion.

5.2.1 Example: While Loops

For example, consider the following interpreter snippet for a while-loop construct. If the test expression b evaluates to False then the command c is not executed. If the test expression b evaluates to True then the command c is executed at least once. The interpreter ($\mathcal{I}$) is invoked on c and s, where s is the interpreter state, and it returns the new state. Then iteration is achieved by invoking the interpreter on the entire while command.

$\mathcal{I}(\text{While } b\ c, s) =$
  if $\mathcal{E}(b, s) = \text{False}$ then s else $\mathcal{I}(\text{While } b\ c, \mathcal{I}(c, s))$

After currying and lambda lowering that snippet becomes the following.

$\mathcal{I}(\text{While } b\ c) =$
  $\lambda s.$ if $\mathcal{E}(b)(s) = \text{False}$ then s else $\mathcal{I}(\text{While } b\ c)(\mathcal{I}(c)(s))$

But attempting to lift $\mathcal{I}(\text{While } b\ c)$ will lead to non-termination, and so the expression lifting rule should not be used.

$\mathcal{I}(\text{While } b\ c) =$
  let $f_1 = \mathcal{E}(b)$
      $f_2 = \mathcal{I}(\text{While } b\ c) \leftarrow$ *causes non-termination*
      $f_3 = \mathcal{I}(c)$
      in $\lambda s.$ if $f_1(s) = \text{False}$ then s else $f_2(f_3(s))$

However, if we let $g = \mathcal{I}(\text{While } b\ c)$ it becomes apparent that this function can be computed; g is the fixed-point function.

$g(s) = \text{if } \mathcal{E}(b)(s) = \text{False}$ then s else $g(\mathcal{I}(c)(s))$

5.2.2 Example: Regular Expressions

Another example involves revisiting the regular expression code from chapter 3 so that it includes Kleene-star in a different way. Suppose we had not anticipated deriving a compiler and wrote the Kleene-star case in an operational style.

```
...
((star? regexp)
 (or (k cl)
     (match (exp<-star regexp)
            cl
            (lambda (cl3)
               (if (eq? cl cl3) #f (match regexp cl3 k))))))
```

The transformation technique does not succeed on this augmented interpreter. If we apply currying, lambda lowering, and start to apply expression lifting, it becomes apparent that one expression cannot be lifted because it will not terminate outside the λ-expression.

```
...
((star? regexp)
 (let ((f1 (match (exp<-star regexp))))
   (lambda (cl k)
     (or (k cl)
         (f1
          cl
          (lambda (cl3) ; if lifted,  (match regexp)  will loop!
            (if (eq? cl cl3) #f ((match regexp) cl3 k))))))))
```

It is clear that the expression `(match regexp)` will loop if lifted. Again, we solve the problem by introducing the explicit fixed-point function. Let $f_2 = (\texttt{match regexp})$, then $(\texttt{match regexp}) = \lambda(c\ell, k).\cdots(\texttt{match regexp})\cdots$ becomes $f_2 = \lambda(c\ell, k).\cdots f_2\cdots$. We then get the following code.

```
...
((star? regexp)
 (let ((f1 (match (exp<-star regexp))))
   (letrec ((f2 (lambda (cl k)
                  (or (k cl)
                      (f1
                       cl
                       (lambda (cl3)
                         (if (eq? cl cl3) #f (f2 cl3 k))))))))
     f2)))
```

The body of the let is not what the quoting rule needs, and so we eta-expand.

```
...
((star? regexp)
 (let ((f1 (match (exp<-star regexp))))
   (lambda (cl k)
     ((letrec ((f2 (lambda (cl k)
                     (or (k cl)
                         (f1
                          cl
                          (lambda (cl3)
                            (if (eq? cl cl3) #f (f2 cl3 k))))))))
        f2) cl k))))
```

Now the quoting rule can be applied. When performed, we get a code generator for regular expressions that includes Kleene-star forms.

5.3 Replacing Text with Denotation

Another way that operationally based definitions can lead to problems for the transformation technique is when an operational-style interpreter manipulates program text rather than some denoted value. When this happens in an interpreter that supports first-class functions, often the portion of the interpreter concerning abstractions cannot be turned

into a code-generator since there is no recursive call on the body of the abstraction. There may be other kinds of terms for which this issue arises as well. When this happens in an interpreter in which functions are *not* expressed values, a portion of the interpreter when turned into a code-generator will loop on procedure definitions that make use of recursion. The solution in these cases is to modify the interpreter so that it returns functions containing recursive calls to the interpreter on the text rather than the text itself. We will first consider the case of interpreters that manipulate text, and then the case of procedure environments that contain text.

5.3.1 Interpreters Manipulating Terms

For both big-step and small-step operational semantics, it is customary for an abstraction to "evaluate to itself." Thus for a big-step semantics we see rules such as $\lambda x.M \Downarrow \lambda x.M$. And for small step semantics, we see no rule at all for abstractions. Operational-style interpreters are just as unsuitable for the transformation technique; abstractions evaluate to closures that contain program text. The transformation technique cannot be applied to an interpreter written in such a style. The modification is straightforward: introduce a call to the interpreter on the body of the abstraction that is suitably shielded. We illustrate this modification for both big-step and small-step oriented interpreters.

Big-Step Interpreters

For example, consider the following big-step oriented interpreter snippet for an abstraction construct. An abstraction evaluates to a closure: $\mathcal{E}(\mathsf{Fun}\ x\ e, \mathrm{env}) = \mathsf{Closure}(x, e, \mathrm{env})$, where $\mathcal{E}$ is the evaluator, x is a variable, e is an expression, and env is the environment. The transformations do not introduce calls, and so there cannot be a compiler call on the sub-expression e since there is not one in the interpreter. Nevertheless, such an interpreter

can be modified to be denotational. The call to the evaluator can be moved[4] inside the closure hidden inside an abstraction: $\mathcal{E}(\mathsf{Fun}\ x\ e, \mathrm{env}) = \mathsf{Closure}(x, \lambda(\mathrm{env}').\mathcal{E}(e, \mathrm{env}'), \mathrm{env})$.

For another big-step example, we choose Abelson and Sussman's interpreter from their classic text [1]. The code is somewhat lengthy so we only include the essentials concerning abstractions and applications.

Their evaluator has several cases. Abelson and Sussman prefer to reserve the word "closure" for the mathematical notion, and name the function that makes a closure `make-procedure`. In the case of an application, the value of the operator is applied to the values of the operands.

```
(define (sicp-eval exp env)
  (cond ...
    ((lambda? exp)
     (make-procedure (lambda-parameters exp)
                     (lambda-body exp)
                     env))
    ...
    ((application? exp)
     (sicp-apply (sicp-eval (operator exp) env)
                 (list-of-values (operands exp) env)))
  ...))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
```

The application function must distinguish user defined procedures from primitive procedures. A user defined procedure is applied by evaluating its body in the extended environment.

[4]Of course, the meaning of the application must be correspondingly modified as well. Our solution is not unique, but we feel that it is the simplest.

```
(define (sicp-apply procedure arguments)
  (cond ...
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        ...))
```

Observe that an abstraction, or lambda expression, evaluates to a closure that contains the *text* of the abstraction. When applying a closure to a list of values, it is this text that is evaluated in the extended environment. The modification is then to move the recursive evaluation call to the construction of the closure storing a (meta-level) function instead. Applying the closure is then a matter of invoking that (meta-level) function. Note that the new version of `make-procedure` builds a structure containing a function instead of text.

```
(define (make-procedure parameters body env)
  (list 'procedure
        parameters
        (lambda (env2) (eval-sequence body env2))
        env))
```

The new application function calls the function in the closure rather than the evaluator.

```
(define (sicp-apply procedure arguments)
  (cond ...
        ((compound-procedure? procedure)
         ((procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        ...))
```

With this change, the transformation technique can be used to derive a compiler. The code portions following the transformation are below.

```
(define (sicp-eval exp env)
  (cond ...
    ((lambda? exp)
     (let ((f (make-procedure (lambda-parameters exp)
                              (lambda-body exp))))
       `(lambda (env) (,f env))))
    ...
    ((application? exp)
     (let ((f1 (sicp-eval (operator exp)))
           (f2 (list-of-values (operands exp))))
       `(lambda (env) (sicp-apply (,f1 env) (,f2 env)))))
  ...))

(define (make-procedure parameters body)
  (let ((f (eval-sequence body)))
    `(lambda (env)
       (list 'procedure
             ,(reify parameters)
             (lambda (env2) (,f env2))
             env))))
```

A Small-Step Interpreter

For the small-step oriented interpreter, we choose Felleisen's CEK-machine [38,39]. It is a small-step operational semantics for the call-by-value λ-calculus. (See figure 5.1.) It was the first virtual machine derived from a term calculus.

These rules state that evaluating an application first involves evaluating the components of the application. When working on one component, the other must be saved as part of the continuation. When both (all the) components have been reduced to values the closure, or constant operator, is applied to its argument. There is also a rule for looking up variables.

$$E \in \text{Env} ::= [] \mid (x, (V, E)) :: E$$
$$K \in \mathcal{K} ::= \mathsf{k}_\emptyset \mid \mathsf{fun}(V, E, K) \mid \mathsf{arg}(M, E, K)$$
$$\mathsf{lookup}(x, (y, (V, E)) :: E') = \begin{cases} (V, E) & \text{if } x = y \\ \mathsf{lookup}(x, E') & \text{otherwise} \end{cases}$$

$$C \in \mathcal{C} ::= \langle M, E, K \rangle$$

$$\begin{array}{lcll}
\langle (M\ N), E, K \rangle & \longmapsto & \langle M, E, \mathsf{arg}(N, E, K) \rangle & \\
\langle V, E, \mathsf{arg}(N, E', K) \rangle & \longmapsto & \langle N, E', \mathsf{fun}(V, E, K) \rangle & \text{if } V \notin \text{Variables} \\
\langle c, E, \mathsf{fun}(c_{op}, E', K) \rangle & \longmapsto & \langle \delta(c_{op}, c), [], K \rangle & \\
\langle V, E, \mathsf{fun}((\lambda x.M), E', K) \rangle & \longmapsto & \langle M, (x, (V, E)) :: E', K \rangle & \text{if } V \notin \text{Variables} \\
\langle x, E, K \rangle & \longmapsto & \langle \mathsf{lookup}(x, E)_1, \mathsf{lookup}(x, E)_2, K \rangle &
\end{array}$$

Figure 5.1: CEK-machine

This interpreter can be re-expressed to emphasize structural recursion as follows.

$$\mathcal{E}_{CEK}(c, e, \kappa) = \kappa(c, e)$$
$$\mathcal{E}_{CEK}(x, e, \kappa) = \kappa(v, e') \text{ where } (v, e') = \mathsf{lookup}(x, e)$$
$$\mathcal{E}_{CEK}((\lambda x.M), e, \kappa) = \kappa((\lambda x.M), e)$$
$$\mathcal{E}_{CEK}((M\ N), e, \kappa) = \mathcal{E}_{CEK}(M, e, \mathsf{mkArg}(N, e, \kappa))$$

$$\mathsf{mkArg}(M, e, \kappa) = \lambda(v, e').\mathcal{E}_{CEK}(M, e, \mathsf{mkFun}(v, e', \kappa))$$
$$\mathsf{mkFun}(v, e, \kappa) = \begin{cases} \lambda(v', e').\kappa(\delta(c_{op}, c), e) & \text{if } v = c_{op} \text{ and } v' = c \\ \lambda(v', e').\mathcal{E}_{CEK}(M, (x, (v', e')) :: e, \kappa) & \text{if } v = (\lambda x.M) \end{cases}$$

In the second version, the continuations are functions. For values, the continuation is invoked on the value.

Note that in either way of expressing the interpreter, the meaning of an abstraction is characterized by its environment and its *textual* body. Also when evaluating an application, the operand is saved as *text*. As in the previous example, we introduce early calls to the interpreter that are shielded with abstractions. The notation f is used to denote meta-level functions.

$\mathcal{E}_{CEK}((\lambda x.M), e, \kappa) = \kappa(\mathsf{op}(x, \lambda(e', \kappa').\mathcal{E}_{CEK}(M, e', \kappa')), e)$
$\mathcal{E}_{CEK}((M\ N), e, \kappa) = \mathcal{E}_{CEK}(M, e, \mathsf{mkArg}(\lambda(e', \kappa').\mathcal{E}_{CEK}(N, e', \kappa'), e, \kappa))$

$\mathsf{mkArg}(f, e, \kappa) = \lambda(v, e').f(e, \mathsf{mkFun}(v, e', \kappa))$
$$\mathsf{mkFun}(v, e, \kappa) = \begin{cases} \lambda(v', e').\kappa(\delta(c_{op}, c), e) & \text{if } v = c_{op} \text{ and } v' = c \\ \lambda(v', e').f((x, (v', e')) :: e, \kappa) & \text{if } v = \mathsf{op}(x, f) \end{cases}$$

Now the transformation technique can be used to derive a compiler.

$\mathcal{E}_{CEK}(c) = [\![\lambda(e, \kappa).\kappa(c, e)]\!]$
$\mathcal{E}_{CEK}(x) = [\![\lambda(e, \kappa).\kappa(v, e') \text{ where } (v, e') = \mathsf{lookup}(x, e)]\!]$
$\mathcal{E}_{CEK}((\lambda x.M)) = \mathsf{let}\ , f = \mathcal{E}_{CEK}(M)\ \mathsf{in}\ [\![\lambda(e, \kappa).\kappa(\mathsf{op}(x, \lambda(e', \kappa').(, f)(e', \kappa')), e)]\!]$
$\mathcal{E}_{CEK}((M\ N)) = \mathsf{let}\ , f_1 = \mathcal{E}_{CEK}(M),\ , f_2 = \mathcal{E}_{CEK}(N)$
$\qquad\qquad\qquad\qquad \mathsf{in}\ [\![\lambda(e, \kappa).(, f_1)(e, \mathsf{mkArg}(\lambda(e', \kappa').(, f_2)(e', \kappa'), e, \kappa))]\!]$

5.3.2 Environments Containing Terms

Another place that code text can linger is in an environment. With a first-order language, function definitions are not treated like other expressions. Consider the following first-order language.

$$\begin{array}{lcl} \Pi & ::= & \mathsf{Let}\ \Delta\ e \\ \Delta & ::= & \varepsilon \\ & ::= & y(x) = e, \Delta \\ e & ::= & n \\ & ::= & x \\ & ::= & e_1 + e_2 \\ & ::= & e_1 - e_2 \\ & ::= & e_1 \times e_2 \\ & ::= & \mathsf{If0}\ e_1\ e_2\ e_3 \\ & ::= & y(e) \end{array}$$

where n is a number, x is a variable, and y is a function variable.

An instance of Π, a program, is a collection of function definitions, together with a main expression that is evaluated. Expressions are either numbers, variables, arithmetic

expressions, conditionals, or function calls. The variable name y is used to distinguish function names from variables denoting expressed values. The notation $\Delta(y)$ is used to look up a function definition.

Now consider the following interpreter for this language.

$\mathcal{P}(\mathsf{Let}\ \Delta\ e) = \mathcal{E}(e, \Delta, \rho_0)$

$\mathcal{E}(n, \Delta, \rho) = n$
$\mathcal{E}(x, \Delta, \rho) = \rho(x)$
$\mathcal{E}(e_1 + e_2, \Delta, \rho) = \mathcal{E}(e_1, \Delta, \rho) + \mathcal{E}(e_2, \Delta, \rho)$
$\mathcal{E}(e_1 - e_2, \Delta, \rho) = \mathcal{E}(e_1, \Delta, \rho) - \mathcal{E}(e_2, \Delta, \rho)$
$\mathcal{E}(e_1 \times e_2, \Delta, \rho) = \mathcal{E}(e_1, \Delta, \rho) \times \mathcal{E}(e_2, \Delta, \rho)$
$\mathcal{E}(\mathsf{If0}\ e_1\ e_2\ e_3, \Delta, \rho) = \text{if } \mathcal{E}(e_1, \Delta, \rho) = 0 \text{ then } \mathcal{E}(e_2, \Delta, \rho) \text{ else } \mathcal{E}(e_3, \Delta, \rho)$
$\mathcal{E}(y(e), \Delta, \rho) = \mathcal{E}(e', \Delta, \rho_0[x \mapsto \mathcal{E}(e, \Delta, \rho)]) \text{ where } (x, e') = \Delta(y)$

This interpreter evaluates expressions in the context of the function definitions (Δ) and the environment (ρ). It may at first appear to be denotational, but it is not. There is no denotation for the function definitions (Δ). In particular, if we curry $\mathcal{E}$ and let $\Delta = y(x) = \mathsf{If0}\ x\ 1\ (x \times y(x-1)), \varepsilon$ then when we try to determine the meaning of $y(x-1)$ we find we cannot. Either Δ is dynamic and we cannot statically compute e', or Δ is static and there is an infinite loop.

$$\begin{aligned}\mathcal{E}(y(x-1), \Delta) &= \mathcal{E}(\mathsf{If0}\ x\ 1\ (x \times y(x-1)), \Delta) \\ &= \lambda\rho.\text{if } \mathcal{E}(x, \Delta)(\rho) = 0 \text{ then } \mathcal{E}(1, \Delta)(\rho) \text{ else } \mathcal{E}(x \times y(x-1), \Delta)(\rho)\end{aligned}$$

Further, when expanding the last interpreter application we get the following.

$\mathcal{E}(x \times y(x-1), \Delta) = \lambda\rho.\mathcal{E}(x, \Delta)(\rho) \times \mathcal{E}(y(x-1), \Delta)(\rho)$

Thus expanding $\mathcal{E}(y(x-1), \Delta)$ requires the expansion of $\mathcal{E}(y(x-1), \Delta)$.

To modify this interpreter, it is necessary to determine the meaning of the function definitions (Δ) as well. The notation g is used to denote a meta-level function.

$\mathcal{P}(\mathsf{Let}\ \Delta\ e) = \mathcal{E}(e, \mathcal{D}(\Delta), \rho_0)$

$\mathcal{D}(\Delta) = \{(y, (\lambda(v, \delta).\mathcal{E}(e, \delta, \rho_0[x \mapsto v]))) \mid y \text{ is a function variable}, (x, e) = \Delta(y)\}$

$\mathcal{E}(n, \delta, \rho) = n$
$\mathcal{E}(x, \delta, \rho) = \rho(x)$
$\mathcal{E}(e_1 + e_2, \delta, \rho) = \mathcal{E}(e_1, \delta, \rho) + \mathcal{E}(e_2, \delta, \rho)$
$\mathcal{E}(e_1 - e_2, \delta, \rho) = \mathcal{E}(e_1, \delta, \rho) - \mathcal{E}(e_2, \delta, \rho)$
$\mathcal{E}(e_1 \times e_2, \delta, \rho) = \mathcal{E}(e_1, \delta, \rho) \times \mathcal{E}(e_2, \delta, \rho)$
$\mathcal{E}(\mathsf{If0}\ e_1\ e_2\ e_3, \delta, \rho) = \text{if } \mathcal{E}(e_1, \delta, \rho) = 0 \text{ then } \mathcal{E}(e_2, \delta, \rho) \text{ else } \mathcal{E}(e_3, \delta, \rho)$
$\mathcal{E}(y(e), \delta, \rho) = g(\mathcal{E}(e, \delta, \rho), \delta) \text{ where } g = \delta(y)$

Now the transformation technique can be used to derive a compiler.

$\mathcal{P}(\mathsf{Let}\ \Delta\ e) = \mathsf{let}\ f = \mathcal{E}(e), \delta = \mathcal{D}(\Delta)\ \mathsf{in}\ [\![\lambda().(, f)((, \delta), \rho_0)]\!]$

$\mathcal{D}(\Delta) = \{(y, \mathsf{let}\ f = \mathcal{E}(e)\ \mathsf{in}\ [\![\lambda(v, \delta).(, f)(\delta, \rho_0[x \mapsto v])]\!]) \mid y \text{ is a function variable}, (x, e) = \Delta(y)\}$

$\mathcal{E}(n) = [\![\lambda(\delta, \rho).n]\!]$
$\mathcal{E}(x) = [\![\lambda(\delta, \rho).\rho(x)]\!]$
$\mathcal{E}(e_1 + e_2) = \mathsf{let}\ , f_1 = \mathcal{E}(e_1),\ , f_2 = \mathcal{E}(e_2)\ \mathsf{in}\ [\![\lambda(\delta, \rho).(, f_1)(\delta, \rho) + (, f_2)(\delta, \rho)]\!]$
$\mathcal{E}(e_1 - e_2) = \mathsf{let}\ , f_1 = \mathcal{E}(e_1),\ , f_2 = \mathcal{E}(e_2)\ \mathsf{in}\ [\![\lambda(\delta, \rho).(, f_1)(\delta, \rho) - (, f_2)(\delta, \rho)]\!]$
$\mathcal{E}(e_1 \times e_2) = \mathsf{let}\ , f_1 = \mathcal{E}(e_1),\ , f_2 = \mathcal{E}(e_2)\ \mathsf{in}\ [\![\lambda(\delta, \rho).(, f_1)(\delta, \rho) \times (, f_2)(\delta, \rho)]\!]$
$\mathcal{E}(\mathsf{If0}\ e_1\ e_2\ e_3) =$
$\quad \mathsf{let}\ , f_1 = \mathcal{E}(e_1)$
$\qquad\quad , f_2 = \mathcal{E}(e_2)$
$\qquad\quad , f_3 = \mathcal{E}(e_3)$
$\quad \mathsf{in}\ [\![\lambda(\delta, \rho).\text{if } (, f_1)(\delta, \rho) = 0 \text{ then } (, f_2)(\delta, \rho) \text{ else } (, f_3)(\delta, \rho)]\!]$
$\mathcal{E}(y(e)) = \mathsf{let}\ , f = \mathcal{E}(e)\ \mathsf{in}\ [\![\lambda(\delta, \rho).g((, f)(\delta, \rho), \delta) \text{ where } g = \delta(y)]\!]$

Above, we see that the function definitions environment is dynamic, but that the functions are still compiled because of $\mathcal{D}$. Hence recursive calls pose no problems.

5.4 Summary

The arguments in chapter 4 showed that the rules deal successfully with interpreters written in a denotational style. The current chapter has shown that certain other coding styles can be modified so that the rules will still succeed: Section 5.1 showed that if static and dynamic parameters seem to be entangled, programming with continuations might be the appropriate modification, especially if the entanglement results from using successful results wrapped into Maybe to accommodate an algorithm failure. Section 5.2 showed that if an interpreter is written in an operational-style and recursively calls itself on its input term, the interpreter can be modified so that the algorithm is expressed using a fixed-point and in that way the interpreter is made denotational. Section 5.3 showed that if an interpreter is written in an operational-style and manipulates text where a denotational-style interpreter would manipulate a denoted value, the interpreter can be modified so that the text is eliminated by introducing calls to the interpreter in those places. Altogether, the rules provide code generation for a broader class of algorithms and coding styles. However, there is no proof that they will always be applicable, nor is it clear whether or not there are other cases that are missing.

Chapter 6

A Larger Example: PROLOG

> Was beweisbar ist, soll in der Wissenschaft nicht ohne Beweis geglaubt werden.
> *In science, what can be proved should not be believed without a proof.*
>
> Richard Dedekind

In his now classic text *Paradigms of Artificial Intelligence Programming* [76], Norvig discusses two PROLOG [32] implementations. The first is a naïve interpreter and the second is a fairly sophisticated compiler that targets Common LISP [94]. Although Norvig discusses both an interpreter and a compiler, he makes no explicit connection between them. In contrast, we use the ideas from chapters 4 and 5 to derive a compiler. We start with a naïve PROLOG interpreter written in a natural style similar to Norvig's interpreter. This interpreter is transformed into a denotational-style interpreter with efficient unification. Then the rules are applied so as to derive a compiler. Finally, we compare the performance

of our compiler to Norvig's compiler.

6.1 A Naïve Interpreter

Our naïve PROLOG interpreter is not identical to Norvig's. While he represents a success continuation as a single list of terms to prove, we represent both success and failure continuations explicitly as functions. Although the naïve interpreter is written in a functional style, there is no notion of a denotation of the program/database (Δ). In this chapter as well, the notation for the pseudo-code borrows from ML and Haskell.

First we introduce the types used in the interpreter pseudo-code below. The class of predicates Pred and the class of terms Term are at the core of a description of PROLOG syntax. A predicate looks like a Boolean-valued function call, and a term is either a number, a variable, or a predicate. Backus-Naur form is used to formally define these classes:

$$\begin{array}{rcl} t \in \mathsf{Term} & ::= & n \\ & ::= & X \\ & ::= & P \\ P \in \mathsf{Pred} & ::= & i(t_1, \ldots, t_n) \end{array}$$

where n is a number, i is an identifier, Var is the set of all variables, $X \in \mathsf{Var}$ is a variable, and t_k is a Term.

A substitution is used to map variables to terms. Sub denotes the class of substitutions, which is shorthand for $\mathsf{Var} \rightarrow \mathsf{Term}$. A clause is a predicate that is the consequence of some conjunction of predicates. Clause denotes the class of clauses, which is shorthand for $\mathsf{Pred} \times [\mathsf{Pred}]$. Now a PROLOG program/database is understood as a sequence, or list, of clauses.

Pseudo-code for a naïve PROLOG interpreter consists of four functions charged with proving results and support functions for unification and for copying terms. The last four arguments for the proving functions are the unification substitution (θ), the database (Δ),

the success continuation (κ_s), and the failure continuation (κ_f). A success continuation takes a failure continuation so that additional solutions can be found. The types of these variables now follow[1].

θ : Sub

Δ : [Clause]

κ_s : Sub $\times$ (() $\rightarrow$ Answer) $\rightarrow$ Answer

κ_f : () $\rightarrow$ Answer

The interactive entry point to the proving functions is the prove-query function ($\mathcal{P}_Q$) which attempts to prove all the predicates of a query using the initial substitution and the initial continuations. It takes a list of predicates and a database.

Π : [Pred]

$\mathcal{P}_Q(\Pi, \Delta) = \mathcal{P}_A(\Pi, \theta_0, \Delta, \kappa_{s0}, \kappa_{f0})$

Prove-query delegates to a function prove-all ($\mathcal{P}_A$) that attempts to prove all the predicates in the list Π: if the list is empty it succeeds, otherwise prove ($\mathcal{P}$) is called on the first term with the success continuation extended to prove the rest.

$\mathcal{P}_A([], \theta, \Delta, \kappa_s, \kappa_f) = \kappa_s(\theta, \kappa_f)$
$\mathcal{P}_A(P :: \Pi, \theta, \Delta, \kappa_s, \kappa_f) =$
$\mathcal{P}(P, \theta, \Delta, (\lambda(\theta', \kappa'_f).\mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa'_f)), \kappa_f)$

The function prove ($\mathcal{P}$) attempts to prove a predicate by delegating to prove-goal ($\mathcal{P}_G$).

$\mathcal{P}(P, \theta, \Delta, \kappa_s, \kappa_f) = \mathcal{P}_G(\Delta, P, \theta, \Delta, \kappa_s, \kappa_f)$

Finally, the function prove-goal ($\mathcal{P}_G$) attempts to prove a predicate by proving one clause (η) from its database of clauses. If the list of clauses is empty it fails. Otherwise it attempts to unify the predicate with the predicate in the first clause. If that succeeds it

[1]The specific type of Answer is left undefined. It is possible for the initial continuations to convey the results using a variety of types.

then attempts to prove all the predicates in the clause predicate list by invoking prove-all ($\mathcal{P}_A$); if that fails it tries another clause.

$\mathcal{P}_G([], P, \theta, \Delta, \kappa_s, \kappa_f) = \kappa_f()$
$\mathcal{P}_G(\eta :: \Delta', P, \theta, \Delta, \kappa_s, \kappa_f) =$
 let $(P', \Pi) = \mathsf{copy}(\eta)$
  $\kappa'_f = \lambda().\mathcal{P}_G(\Delta', P, \theta, \Delta, \kappa_s, \kappa_f)$
  in $\mathcal{U}(P', P, \theta, (\lambda(\theta').\mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa'_f)), \kappa'_f)$

The function prove-goal ($\mathcal{P}_G$) makes use of two functions: copy and $\mathcal{U}$. The function copy copies terms and renames any variables in the copied clause. It is this mechanism that ensures that local variables do not affect other local variables of the same name. Generating fresh variable names requires state, which could be maintained using an additional parameter; the details are not shown here.

The function $\mathcal{U}$ determines if two terms unify. If so, the unify success continuation, which takes only a substitution parameter, is called. If not, the unify failure continuation is called. The details of the unification implementation are also not shown.

Missing features include primitive predicates, cut, and special cases for tail-recursion. These pose no particular challenge and are omitted for the sake of brevity. A hook for primitive predicates could be added to prove-all ($\mathcal{P}_A$). Supporting cut requires a third continuation parameter initialized by prove ($\mathcal{P}$). Supporting tail-recursion involves adding cases for singleton lists to prove-all ($\mathcal{P}_A$) and prove-goal ($\mathcal{P}_G$).

6.2 Efficiency and Denotation

The function prove-goal ($\mathcal{P}_G$) looks through the entire data-base (Δ) to find a match. Thus $\mathcal{U}$ is invoked on the head of every clause. Unification is somewhat heavyweight. It is possible to filter out clauses that cannot unify using lighter-weight comparisons based on

the clause head's identifier and the clause head's arity. We use the notation $\Delta(i/n)$ to indicate filtering by identifier and arity because this filtering resembles an environment look-up. The pseudo-code is modified as follows.

Instead of passing the predicate to prove, prove-all ($\mathcal{P}_A$) passes the components of the predicate: the identifier, the arity, and the list of terms.

$$\mathcal{P}_A([], \theta, \Delta, \kappa_s, \kappa_f) = \kappa_s(\theta, \kappa_f)$$
$$\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \theta, \Delta, \kappa_s, \kappa_f) =$$
$$\mathcal{P}(i, n, [t_1, \ldots, t_n], \theta, \Delta, (\lambda(\theta', \kappa'_f).\mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa'_f)), \kappa_f)$$

Prove ($\mathcal{P}$) then uses the identifier and the arity to find only the relevant clauses in the database. Since all the clauses passed to prove-goal ($\mathcal{P}_G$) have the same identifier, only the list of terms (τ) needs to be passed.

$$\mathcal{P}(i, n, \tau, \theta, \Delta, \kappa_s, \kappa_f) = \mathcal{P}_G(\Delta(i/n), \tau, \theta, \Delta, \kappa_s, \kappa_f)$$

Since prove-goal ($\mathcal{P}_G$) now takes a term-list parameter rather than a predicate, the unification function must be correspondingly adjusted.

$$\mathcal{P}_G([], \tau, \theta, \Delta, \kappa_s, \kappa_f) = \kappa_f()$$
$$\mathcal{P}_G(\eta :: \Delta', \tau, \theta, \Delta, \kappa_s, \kappa_f) =$$
$$\text{let } (i(t_1, \ldots, t_n), \Pi) = \mathsf{copy}(\eta)$$
$$\kappa'_f = \lambda().\mathcal{P}_G(\Delta', \tau, \theta, \Delta, \kappa_s, \kappa_f)$$
$$\text{in } \mathcal{U}([t_1, \ldots, t_n], \tau, \theta, (\lambda(\theta').\mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa'_f)), \kappa'_f)$$

From this point of view it becomes apparent that $\Delta(i/n)$ maps to a list of clauses (i.e., program text) and not to any denoted value. As discussed in chapter 5, the transformation rules may fail on an interpreter that is not denotational. In fact, if the transformation technique were applied to the interpreter above, it would yield a compiler that looped on recursive PROLOG programs. We make use of the solution discussed in chapter 5, and we

introduce a new function $\mathcal{D}$ that maps a database to its denotation. The call to prove-goal ($\mathcal{P}_G$) is moved from prove ($\mathcal{P}$) to the denotation of a collection of clauses. A database denotation can be understood either as a table or a function that maps an identifier and arity to a single function that succeeds or fails based on whether or not the terms provided satisfy the indicated predicate. We use $\delta = \mathcal{D}(\Delta)$ as notation for a database denotation.

$\mathcal{D}(\Delta) = \{(i/n, (\lambda(\tau, \theta, \delta, \kappa_s, \kappa_f).\mathcal{P}_G(\Delta(i/n), \tau, \theta, \delta, \kappa_s, \kappa_f))) \mid i \text{ is an identifier}, n \in \mathbb{N}\}$

Prove-query, prove-all, prove, and prove-goal must all be modified to take the database denotation.

$\mathcal{P}_Q(\Pi, \delta) = \mathcal{P}_A(\Pi, \theta_0, \delta, \kappa_{s0}, \kappa_{f0})$

$\mathcal{P}_A([], \theta, \delta, \kappa_s, \kappa_f) = \kappa_s(\theta, \kappa_f)$
$\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \theta, \delta, \kappa_s, \kappa_f) =$
$\mathcal{P}(i, n, [t_1, \ldots, t_n], \theta, \delta, (\lambda(\theta', \kappa'_f).\mathcal{P}_A(\Pi, \theta', \delta, \kappa_s, \kappa'_f)), \kappa_f)$

Prove ($\mathcal{P}$) now looks very different. Instead of supplying prove-goal ($\mathcal{P}_G$) with some text from the database, it calls the function associated with the identifier and arity.

$\mathcal{P}(i, n, \tau, \theta, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\tau, \theta, \delta, \kappa_s, \kappa_f)$

$\mathcal{P}_G([], \tau, \theta, \delta, \kappa_s, \kappa_f) = \kappa_f()$
$\mathcal{P}_G(\eta :: \Delta, \tau, \theta, \delta, \kappa_s, \kappa_f) =$
$\quad \mathsf{let}\ (i(t_1, \ldots, t_n), \Pi) = \mathsf{copy}(\eta)$
$\qquad \kappa'_f = \lambda().\mathcal{P}_G(\Delta, \tau, \theta, \delta, \kappa_s, \kappa_f)$
$\qquad \mathsf{in}\ \mathcal{U}([t_1, \ldots, t_n], \tau, \theta, (\lambda(\theta').\mathcal{P}_A(\Pi, \theta', \delta, \kappa_s, \kappa'_f)), \kappa'_f)$

At this point, the interpreter in the form of the four proving functions is expressed in denotational-style, and the transformation technique applies. Before applying the technique we improve the efficiency of unification.

6.3 Improving Unification

Norvig comments that a possible next step is improving unification. He forgoes adding a union-find based unification algorithm to the interpreter in order to jump directly to a compiler. In this section, we add a more sophisticated unification algorithm to the interpreter so as to derive a compiler. The substitution (θ) is removed. We also take the opportunity here to avoid completely copying a term, and instead determine a mapping from variables to logic variables. This mapping is written in factored form ($\pi \circ \phi$) to allow more static computation to occur. The first factor is referred to as the pre-frame and the second as the frame.

A pre-frame is the class of mappings from variables to natural numbers. PreFrame denotes the class of pre-frames, which is shorthand for Var $\to \mathbb{N}$. A frame is the class of mappings from natural numbers to logic variables. Frame denotes the class of frames, which is shorthand for $\mathbb{N} \to$ LogicVar.

Additional types are necessary when the unification algorithm changes. The class of terms is still used to describe what a PROLOG program looks like, but since the union-find approach to unification turns variables into data-structures, a new class of values is needed that describes the run-time data structures. A functor looks like a constructor function call, and a value is either a number, a logic variable, or a functor. Funct denotes the class of functors and Value denotes the class of values. Backus-Naur form is used to formally define these classes.

$$
\begin{array}{rcl}
v \in \mathsf{Value} & ::= & n \\
& ::= & \chi \\
& ::= & F \\
F \in \mathsf{Funct} & ::= & i(v_1, \ldots, v_n)
\end{array}
$$

where n is a number, i is an identifier, $\chi \in$ LogicVar is a logic variable, and v_k is a Value.

It is possible to say more about the structure of logic variables. When displaying logic variables it is useful to have the original variable from the program and a number to

distinguish it from others, but the essence is simply a container. Thus LogicVar is shorthand for Var $\times \mathbb{N} \times$ Location.

Now the arguments to the proving functions are more varied. The last two arguments for all of them are again the success continuation and the failure continuation. Because a substitution is no longer used, the type of the success continuation has changed.

$\kappa_s : (() \to \mathsf{Answer}) \to \mathsf{Answer}$

$\kappa_f : () \to \mathsf{Answer}$

Prove-all ($\mathcal{P}_A$) takes a pre-frame and a frame.

π : PreFrame

ϕ : Frame

Prove ($\mathcal{P}$) and prove-goal ($\mathcal{P}_G$) take a list of values.

ν : [Value]

The previous pseudo-code is transformed to make use of the more sophisticated unification algorithm. Destructive unification and logic variables require state. Here too the state parameter and the implementation details are not shown. This improvement to unification does improve the overall performance of the interpreter.

The functions in the database denotation are now constructed so that the arguments correspond to the arguments that the new version of prove-goal ($\mathcal{P}_G$) needs.

$\mathcal{D}(\Delta) = \{(i/n, (\lambda(\nu, \delta, \kappa_s, \kappa_f).\mathcal{P}_G(\Delta(i/n), \nu, \delta, \kappa_s, \kappa_f))) \mid i \text{ is an identifier}, n \in \mathbb{N}\}$

Instead of merely supplying an initial substitution, prove-query ($\mathcal{P}_Q$) must construct an initial pre-frame and frame based on the variables in the predicate list Π to supply to prove-all ($\mathcal{P}_A$). The function varsFromPred returns a list of all the unique variables in a predicate list. The function newPreFrame turns a list of variables into a map from variables to numbers, and the function newFrame turns that same list of variables into a map from numbers to logic variables.

$\mathcal{P}_Q(\Pi, \delta) =$
 let $\ell = \mathsf{varsFromPred}(\Pi)$
  $\pi_0 = \mathsf{newPreFrame}(\ell)$
  $\phi_0 = \mathsf{newFrame}(\ell)$
  in $\mathcal{P}_A(\Pi, \pi_0, \phi_0, \delta, \kappa_{s0}, \kappa_{f0})$

Prove-all ($\mathcal{P}_A$) now uses the function toValues to convert its term list to a list of values so that prove ($\mathcal{P}$) will not need to keep track of a frame.

$\mathcal{P}_A([], \pi, \phi, \delta, \kappa_s, \kappa_f) = \kappa_s(\kappa_f)$
$\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \pi, \phi, \delta, \kappa_s, \kappa_f) =$
 $\mathcal{P}(i, n, \mathsf{toValues}([t_1, \ldots, t_n], \pi, \phi), \delta, (\lambda(\kappa'_f).\mathcal{P}_A(\Pi, \pi, \phi, \delta, \kappa_s, \kappa'_f)), \kappa_f)$

$\mathcal{P}(i, n, \nu, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\nu, \delta, \kappa_s, \kappa_f)$

Prove-goal ($\mathcal{P}_G$) does not take a substitution any more, nor does it take a frame. However, it creates a frame based on the variables in the first clause from its list of clauses using the function varsFromClause. This frame is supplied to the unification algorithm so that the values associated with the terms can be unified with the list of values (ν) that were supplied as an argument. The unification function is adjusted again so that destructive unification is used and logic variables are set and unset. The unsetting is hidden in the unification failure continuations.

$\mathcal{P}_G([], \nu, \delta, \kappa_s, \kappa_f) = \kappa_f()$
$\mathcal{P}_G(\eta :: \Delta, \nu, \delta, \kappa_s, \kappa_f) =$
 let $(i(t_1, \ldots, t_n), \Pi) = \eta$
  $\ell = \mathsf{varsFromClause}(\eta)$
  $\pi = \mathsf{newPreFrame}(\ell)$
  $\phi = \mathsf{newFrame}(\ell)$
  in $\mathcal{U}([t_1, \ldots, t_n], \pi, \phi, \nu, (\lambda(\kappa'_f).\mathcal{P}_A(\Pi, \pi, \phi, \delta, \kappa_s, \kappa'_f)),$
  $(\lambda().\mathcal{P}_G(\Delta, \nu, \delta, \kappa_s, \kappa_f)))$

At this point, not only is the interpreter expressed in a denotational-style, but it also uses an efficient unification algorithm. The transformation technique can now effectively be applied to this interpreter. The next subsections describe their application.

6.4 Currying and Lambda Lowering

Currying involves little more than moving the dynamic parameters from the left to the right side of the equal sign. Since conditionals are implicit and involve separate equations, this movement also achieves lambda lowering. The lambda lowering that remains is to lower the lambda through the let in prove-goal ($\mathcal{P}_G$). Of course, all the calls to the curried functions must be modified.

The pseudo-code from the previous subsection is transformed using rules (1), (2), and (3) from chapter 4.

When prove-goal ($\mathcal{P}_G$) is curried, it produces a function. Thus the abstraction in the database denotation can be eta-reduced.

$\mathcal{D}(\Delta) = \{(i/n, \mathcal{P}_G(\Delta(i/n))) \mid i \text{ is an identifier}, n \in \mathbb{N}\}$

For prove-query ($\mathcal{P}_Q$), the query itself is static, but the database denotation remains dynamic and is curried. We also see that prove-all ($\mathcal{P}_A$) has been curried and that the call has been changed.

$\mathcal{P}_Q(\Pi) =$
$\lambda(\delta).$
let $\ell = \mathsf{varsFromPred}(\Pi)$
$\pi_0 = \mathsf{newPreFrame}(\ell)$
$\phi_0 = \mathsf{newFrame}(\ell)$
in $\mathcal{P}_A(\Pi, \pi_0)(\phi_0, \delta, \kappa_{s0}, \kappa_{f0})$

The static parameters for prove-all ($\mathcal{P}_A$) are the list of predicates and the pre-frame.

The frame itself, the database denotation, and the continuations remain dynamic and are curried. We see that the recursive call to $\mathcal{P}_A$ and the call to toValues have been adjusted.

$$
\begin{array}{l}
\mathcal{P}_A([], \pi) = \lambda(\phi, \delta, \kappa_s, \kappa_f).\kappa_s(\kappa_f) \\
\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \pi) = \\
\quad \lambda(\phi, \delta, \kappa_s, \kappa_f).\mathcal{P}(i, \\
\qquad\qquad\qquad n, \\
\qquad\qquad\qquad \mathsf{toValues}([t_1, \ldots, t_n], \pi)(\phi), \\
\qquad\qquad\qquad \delta, \\
\qquad\qquad\qquad (\lambda(\kappa'_f).\mathcal{P}_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\qquad \kappa_f)
\end{array}
$$

The function prove ($\mathcal{P}$) is, in effect, a call operator. All of the parameters are dynamic, so no currying occurs here.

$$\mathcal{P}(i, n, \nu, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\nu, \delta, \kappa_s, \kappa_f)$$

For prove-goal ($\mathcal{P}_G$), only the list of clauses is static. All the other parameters are curried. Not only are the calls to prove-goal and prove-all adjusted, but also the call to $\mathcal{U}$ is adjusted as well since it is also curried.

$$
\begin{array}{l}
\mathcal{P}_G([]) = \lambda(\nu, \delta, \kappa_s, \kappa_f).\kappa_f() \\
\mathcal{P}_G(\eta :: \Delta) = \\
\quad \lambda(\nu, \delta, \kappa_s, \kappa_f). \\
\quad\quad \mathsf{let}\ (i(t_1, \ldots, t_n), \Pi) = \eta \\
\qquad\quad \ell = \mathsf{varsFromClause}(\eta) \\
\qquad\quad \pi = \mathsf{newPreFrame}(\ell) \\
\qquad\quad \phi = \mathsf{newFrame}(\ell) \\
\qquad\quad \mathsf{in}\ \mathcal{U}([t_1, \ldots, t_n], \pi) \\
\qquad\qquad\quad (\phi, \\
\qquad\qquad\quad \nu, \\
\qquad\qquad\quad (\lambda(\kappa'_f).\mathcal{P}_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\quad (\lambda().\mathcal{P}_G(\Delta)(\nu, \delta, \kappa_s, \kappa_f)))
\end{array}
$$

6.5 More Lambda Lowering

In addition to lowering the lambdas into the conditional expressions, we lower the lambdas into the let as well. This move allows the variables to be determined statically. Then only the frame is allocated dynamically.

The previous pseudo-code is now transformed using rule (4) from chapter 4.

The initial pre-frame (π_0) depends only on the static list of predicates (Π), so the lambda can be lowered into the first let. However, the initial frame (ϕ_0) implicitly depends on the dynamic store since allocating a frame involves storage allocation. Therefore, the lambda cannot be lowered into that let.

$\mathcal{P}_Q(\Pi) =$
 let $\ell = \mathsf{varsFromPred}(\Pi)$
  $\pi_0 = \mathsf{newPreFrame}(\ell)$
  in $\lambda(\delta)$. let $\phi_0 = \mathsf{newFrame}(\ell)$
  in $\mathcal{P}_A(\Pi, \pi_0)(\phi_0, \delta, \kappa_{s0}, \kappa_{f0})$

Similarly, in prove-goal ($\mathcal{P}_G$) the lambda can be lowered past the pre-frame but no lower.

$\mathcal{P}_G([]) = \lambda(\nu, \delta, \kappa_s, \kappa_f).\kappa_f()$
$\mathcal{P}_G(\eta :: \Delta) =$
 let $(i(t_1, \ldots, t_n), \Pi) = \eta$
  $\ell = \mathsf{varsFromClause}(\eta)$
  $\pi = \mathsf{newPreFrame}(\ell)$
  in $\lambda(\nu, \delta, \kappa_s, \kappa_f)$.
   let $\phi = \mathsf{newFrame}(\ell)$
    in $\mathcal{U}([t_1, \ldots, t_n], \pi)$
     $(\phi,$
     $\nu,$
     $(\lambda(\kappa'_f).\mathcal{P}_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa'_f)),$
     $(\lambda().\mathcal{P}_G(\Delta)(\nu, \delta, \kappa_s, \kappa_f)))$

6.6 Expression Lifting

Since the calls to the prove functions do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. We do this now so that the functions can be generated statically rather than dynamically.

Now the pseudo-code from the previous section is transformed using rule (5) from chapter 4.

In prove-query ($\mathcal{P}_Q$), the call to prove-all ($\mathcal{P}_A$) only depends on the list of predicates and the pre-frame, so it can be lifted.

$$
\begin{array}{l}
\mathcal{P}_Q(\Pi) = \\
\quad \mathsf{let}\ \ell = \mathsf{varsFromPred}(\Pi) \\
\qquad \pi_0 = \mathsf{newPreFrame}(\ell) \\
\qquad f = \mathcal{P}_A(\Pi, \pi_0) \\
\qquad \mathsf{in}\ \lambda(\delta).\ \mathsf{let}\ \phi_0 = \mathsf{newFrame}(\ell) \\
\qquad\qquad\qquad \mathsf{in}\ f(\phi_0, \delta, \kappa_{s0}, \kappa_{f0})
\end{array}
$$

Prove-all ($\mathcal{P}_A$) has a recursive call. This call is lifted as well as a call to toValues which depends only on the predicate and the pre-frame.

$$
\begin{array}{l}
\mathcal{P}_A([], \pi) = \lambda(\phi, \delta, \kappa_s, \kappa_f).\kappa_s(\kappa_f) \\
\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \pi) = \\
\quad \mathsf{let}\ f_1 = \mathsf{toValues}([t_1, \ldots, t_n], \pi) \\
\qquad f_2 = \mathcal{P}_A(\Pi, \pi) \\
\qquad \mathsf{in}\ \lambda(\phi, \delta, \kappa_s, \kappa_f).\mathcal{P}(i, \\
\qquad\qquad\qquad\qquad n, \\
\qquad\qquad\qquad\qquad f_1(\phi), \\
\qquad\qquad\qquad\qquad \delta, \\
\qquad\qquad\qquad\qquad (\lambda(\kappa'_f).f_2(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\qquad\qquad \kappa_f)
\end{array}
$$

In prove-goal ($\mathcal{P}_G$), it is also possible to lift the call to prove-all ($\mathcal{P}_A$). In addition, the recursive call is lifted as is the static portion of unification.

$$
\begin{array}{l}
\mathcal{P}_G([]) = \lambda(\nu, \delta, \kappa_s, \kappa_f).\kappa_f() \\
\mathcal{P}_G(\eta :: \Delta) = \\
\quad \mathsf{let}\ (i(t_1, \ldots, t_n), \Pi) = \eta \\
\qquad \ell = \mathsf{varsFromClause}(\eta) \\
\qquad \pi = \mathsf{newPreFrame}(\ell) \\
\qquad f_1 = \mathcal{U}([t_1, \ldots, t_n], \pi) \\
\qquad f_2 = \mathcal{P}_A(\Pi, \pi) \\
\qquad f_3 = \mathcal{P}_G(\Delta) \\
\qquad \mathsf{in}\ \lambda(\nu, \delta, \kappa_s, \kappa_f). \\
\qquad\quad \mathsf{let}\ \phi = \mathsf{newFrame}(\ell) \\
\qquad\qquad \mathsf{in}\ f_1(\phi, \\
\qquad\qquad\qquad \nu, \\
\qquad\qquad\qquad (\lambda(\kappa'_f).f_2(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\qquad (\lambda().f_3(\nu, \delta, \kappa_s, \kappa_f)))
\end{array}
$$

6.7 Code Generation

Since we are interested in generating text, we quote the relevant portions: the functions with dynamic variables. Rule (6) is applied to the previous pseudo-code.

In prove-query ($\mathcal{P}_Q$), the abstraction inside the let is quoted using the bracket notation, and the variables w and f are unquoted using the comma notation.

$$
\begin{array}{l}
\mathcal{P}_Q(\Pi) = \\
\ \mathsf{let}\ \ell = \mathsf{varsFromPred}(\Pi) \\
\qquad \pi_0 = \mathsf{newPreFrame}(\ell) \\
\qquad , w = \ell \\
\qquad , f = \mathcal{P}_A(\Pi, \pi_0) \\
\qquad \mathsf{in}\ [\![\lambda(\delta).\ \mathsf{let}\ \phi_0 = \mathsf{newFrame}(, w) \\
\qquad\qquad\qquad\quad \mathsf{in}\ (, f)(\phi_0, \delta, \kappa_{s0}, \kappa_{f0})]\!]
\end{array}
$$

The prove-query function now generates text that interfaces with the text of the database denotation.

The abstraction in the let is quoted in prove-all ($\mathcal{P}_A$). The variables f_1 and f_2 are unquoted as are i and n.

$$
\begin{array}{l}
\mathcal{P}_A([], \pi) = [\![\lambda(\phi, \delta, \kappa_s, \kappa_f).\kappa_s(\kappa_f)]\!] \\
\mathcal{P}_A((i(t_1, \ldots, t_n)) :: \Pi, \pi) = \\
\quad \mathsf{let}\ , i = i \\
\qquad , n = n \\
\qquad , f_1 = \mathsf{toValues}([t_1, \ldots, t_n], \pi) \\
\qquad , f_2 = \mathcal{P}_A(\Pi, \pi) \\
\qquad \mathsf{in}\ [\![\lambda(\phi, \delta, \kappa_s, \kappa_f).\mathcal{P}((, i), \\
\qquad\qquad\qquad\qquad (, n), \\
\qquad\qquad\qquad\qquad (, f_1)(\phi), \\
\qquad\qquad\qquad\qquad \delta, \\
\qquad\qquad\qquad\qquad (\lambda(\kappa'_f).(, f_2)(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\qquad\qquad \kappa_f)]\!]
\end{array}
$$

For prove-goal ($\mathcal{P}_G$), the abstraction in the let is again quoted. The variables f_1, f_2, f_3, and w are unquoted.

$$
\begin{array}{l}
\mathcal{P}_G([]) = [\![\lambda(\nu, \delta, \kappa_s, \kappa_f).\kappa_f()]\!] \\
\mathcal{P}_G(\eta :: \Delta) = \\
\quad \mathsf{let}\ (i(t_1, \ldots, t_n), \Pi) = \eta \\
\qquad \ell = \mathsf{varsFromClause}(\eta) \\
\qquad \pi = \mathsf{newPreFrame}(\ell) \\
\qquad , w = \ell \\
\qquad , f_1 = \mathcal{U}([t_1, \ldots, t_n], \pi) \\
\qquad , f_2 = \mathcal{P}_A(\Pi, \pi) \\
\qquad , f_3 = \mathcal{P}_G(\Delta) \\
\qquad \mathsf{in}\ [\![\lambda(\nu, \delta, \kappa_s, \kappa_f). \\
\qquad\quad \mathsf{let}\ \phi = \mathsf{newFrame}(, w) \\
\qquad\qquad \mathsf{in}\ (, f_1)(\phi, \\
\qquad\qquad\qquad\qquad \nu, \\
\qquad\qquad\qquad\qquad (\lambda(\kappa'_f).(, f_2)(\phi, \delta, \kappa_s, \kappa'_f)), \\
\qquad\qquad\qquad\qquad (\lambda().(, f_3)(\nu, \delta, \kappa_s, \kappa_f)))]\!]
\end{array}
$$

The database denotation $\mathcal{D}(\Delta)$ is now text that constructs a table associating identifiers augmented with the arity with program text generated by prove-goal ($\mathcal{P}_G$), and δ is initialized to the table generated by that text when executed. Hence $\mathcal{D}$ is now a compiler. To use this compiler interactively, the database denotation must be loaded, and any query text generated must be subsequently executed.

6.8 Performance

To gauge the performance of the compiler derived in this chapter we make use of a simple benchmark: the naïve Fibonacci function using Peano arithmetic. We found Norvig's compiler implementation to be quite respectable with timing results that appeared to be the same as SWI PROLOG (version 6.6.1). We directly compare our implementation only to Norvig's. We used SBCL Common LISP version 1.1.6.0-3c5581a using the default configuration on a MacBook Pro running Mac OS 10.7.5 on a 2GHz Intel Core i7 with 6GB of memory.

fib(15)	Norvig	compiler	interpreter
avg user time (ms)	2.7	4.3	20.0
avg GC time (ms)	0	0	0
avg MB consed	0.5	2.8	6.5

First we compare the denotational interpreter with efficient unification to the derived compiler and observe the performance boost of the transformation technique. The compiler is about 4.7 times faster than the interpreter. The interpreter allocates about 2.3 times as much memory.

We see that Norvig's implementation is only about 1.6 times faster than ours. Our implementation allocates about 5.5 times as much memory. However, when n is only 15, the benchmark does not stress the memory. We consider this next and raise n to 20.

fib(20)	Norvig	compiler	interpreter
avg user time (ms)	17.2	52.0	238.6
avg GC time (ms)	0	160.0	145.3
avg MB consed	5.6	37.7	88.3

The differences are more pronounced here. Nevertheless, without including the garbage collection time, Norvig's implementation is still only about three times faster than ours. In this case, our implementation allocates about 6.7 times as much memory.

We have also looked at other benchmarks, including the n-queens problem. We find that the performance results are very similar. The derived compiler is several times faster than the interpreter and it is in the ballpark of Norvig's implementation. Garbage collection continues to have a significant negative impact on performance, but the variation in the factors has not been completely characterized.

One reason that Norvig's implementation has higher performance is that it has an explicit optimization phase akin to copy-propagation. Further, his implementation effectively allocates frames on the Common LISP run-time stack, whereas our implementation allocates frames in the heap. Even though our implementation currently lacks these enhancements, we consider its performance quite respectable.

6.9 Summary

Inspired by Norvig's PROLOG implementations, we showed how to derive a serious compiler from an interpreter. Having started with a naïve non-denotational PROLOG interpreter, we used the ideas from chapter 5 to derive a denotational one. Then we upgraded the unification algorithm to a fast union-find based implementation. The interpreter was then ready to be transformed by the transformation technique. The transformations were successfully applied. The resulting PROLOG compiler generates code with respectable performance.

Chapter 7

Future Work

J'ai réinventé le passé pour voir
la beauté de l'avenir.
I reinvented the past to see the
beauty of the future.

Louis Aragon

Here we outline additional interesting ideas, additional examples, and theory that we would like to pursue.

7.1 Relationship to Partial Evaluation

Partial evaluation relies on equational reasoning. So too, the correctness proofs of the rules rely on equational reasoning. Hence, the transformation rules are a focused form of equational reasoning. We believe the rules get at the essence of the important equational reasoning that occurs in partial evaluation. We anticipate arguing explicitly that that is the case via both examples and formal proof.

7.2 Additional Examples

Additional examples of the transformation technique can highlight both its conceptual and practical value. Although there have been many early successes creating self-applicable off-line partial evaluators, creating self-applicable on-line partial evaluators has been much more challenging. The only one constructed so far has been for a flowchart language [43]. We believe that we can use the concepts in this thesis to construct a self-applicable on-line partial evaluator for Scheme.

Pattern matching compilers similar to the example in chapter 1 are quite practical. Unfortunately, that particular example is merely a toy for illustration purposes. The usual approach to generating fast pattern matchers from regular expressions involves compiling them into finite automata. Such an approach does not appear to fit well with the transformation technique. However, an alternative approach involving "derivatives" of regular expressions is receiving increased attention [16, 25, 78, 82]. That approach appears to be a good fit and would make a nice practical example. Other kinds of pattern matching, such as on trees, may also make nice practical examples.

7.3 Additional Rules

The transformation technique consists of four rules. We have argued that these rules work, but we might wonder whether and to what extent additional rules would be beneficial.

Are those four rules even enough? For the interpreter language from chapter 4, we argue informally that no additional rules are necessary. The currying rules introduce an abstraction from which sub-expressions can be extracted, the quoting rule turns that abstraction into text, and the remaining rules extract sub-expressions from that abstraction.

Assuming the body of the abstraction has no sub-expressions involving quotation, there are only five cases. For variables and constants, there is nothing to extract. If the body is also an abstraction, we assume the rules are enough for that abstraction, and use them again. If the body is an application, use the expression lifting rule (4.5) on either the sub-expressions or the entire expression; that is all that can be done. If the body is a conditional, use the expression lifting rule (4.5) on sub-expressions or use the lambda lowering rule (4.3) if only the first sub-expression is independent of the parameters; that is all that can be done. We anticipate formalizing this argument.

We briefly observe that the argument above did not use the lambda lowering for a let-binding rule (4.4). It is possible to use only the expression lifting rule (4.5) to achieve a transformation very similar to rule (4.4). Nevertheless, it is convenient and natural during manual transformation to use rule (4.4). Further rule (4.4) avoids the introduction of an extra variable; an additional rule would be needed to eliminate the extra variable.

What about languages that are larger than the interpreter language? In Scheme, we used the lambda lowering rule on `cond` even though the rule is defined only for `if`. In principle, if additional forms are defined in terms of the forms available in the interpreter language, then it is possible to macro-expand these new forms and apply the rules directly. Informally, we feel free to un-expand the forms for readability. Nevertheless, it may be worthwhile to have special derived rules associated with derived forms. If there are additional forms that cannot be defined in terms of the forms available in the interpreter language, then additional rules might be necessary.

In addition to the transformation technique, we have discussed how to extend its applicability by adjusting non-denotational interpreters to make them denotational. These adjustment heuristics are reminiscent of rules. We anticipate investigating whether it is possible to formulate rules that transform an operational semantics into a denotational one, but we are not optimistic.

Recall that rule (4.6) from chapter 4 quoted an abstraction for which all free variables were accounted for. Thus the terms that are generated are always closed. But it is possible to imagine that other rules might yield simpler and possibly more efficient terms by manipulating open terms. We consider as an example the CPS-transform. We observe that given a CPS-style interpreter, the transformation technique yields a Plotkin style CPS-transform [83] which generates code with "administrative" reductions. The interpreter follows.

$$\begin{array}{lcl}
\overline{\overline{c}}\rho\kappa & = & \kappa(c) \\
\overline{\overline{x}}\rho\kappa & = & \kappa(\rho(x)) \\
\overline{\overline{\lambda x.M}}\rho\kappa & = & \kappa(\lambda v\kappa'.\overline{\overline{M}}(\rho[x \mapsto v])\kappa') \\
\overline{\overline{(M\ N)}}\rho\kappa & = & \overline{\overline{M}}\rho(\lambda m.\overline{\overline{N}}\rho(\lambda n.m(n,\kappa)))
\end{array}$$

Applying the transformation technique yields the Plotkin style CPS-transform.

$$\begin{array}{lcl}
\overline{c} & = & [\![\lambda\rho\kappa.\kappa(c)]\!] \\
\overline{x} & = & [\![\lambda\rho\kappa.\kappa(\rho(x))]\!] \\
\overline{\lambda x.M} & = & \mathsf{let}\ , f = \overline{M}\ \mathsf{in}\ [\![\lambda\rho\kappa.\kappa(\lambda v\kappa'.(,f)(\rho[x \mapsto v])\kappa')]\!] \\
\overline{(M\ N)} & = & \mathsf{let}\ , f_1 = \overline{M}\ \mathsf{in}\ \mathsf{let}\ , f_2 = \overline{N}\ \mathsf{in}\ [\![\lambda\rho\kappa.(,f_1)\rho(\lambda m.(,f_2)\rho(\lambda n.m(n,\kappa)))]\!]
\end{array}$$

On $((\lambda x.x)\ 5)$ the transform above yields the following lengthy term.

$$(\lambda\rho\kappa.((\lambda\rho\kappa.\kappa((\lambda v\kappa'.((\lambda\rho\kappa.\kappa(\rho(x)))(\rho[x \mapsto v])\kappa'))))\rho(\lambda m.((\lambda\rho\kappa.\kappa(5))\rho(\lambda n.m(n,\kappa))))))$$

In each case in the interpreter the quoted text begins with $\lambda\rho\kappa$. It seems plausible to push the quotation inward. The function must then take a textual variable, and the generated text must be embedded in an abstraction. The variables f, f_1, and f_2 are no longer text but functions that return text. (For the sake of clarity, we use LISP style unquoting, including the unquoting of full expressions.) This code generator is then similar to the Danvy and Filinski [26,27] optimized CPS transform.

$$\begin{array}{lcl}
\overline{M} & = & [\![\lambda\rho\kappa.,(\overline{\overline{M}}\ [\![\kappa]\!]\ [\![\rho]\!])]\!] \\
\overline{\overline{c}} & = & \lambda\rho\kappa.[\![(,\kappa)(c)]\!] \\
\overline{\overline{x}} & = & \lambda\rho\kappa.[\![(,\kappa)((,\rho)(x))]\!] \\
\overline{\overline{\lambda x.M}} & = & \mathsf{let}\ , f = \overline{\overline{M}}\ \mathsf{in}\ \lambda\rho\kappa.[\![(,\kappa)(\lambda v\kappa'.,(f\ [\![(,\rho)[x \mapsto v]]\!]\ [\![\kappa']\!]))]\!] \\
\overline{\overline{(M\ N)}} & = & \mathsf{let}\ , f_1 = \overline{\overline{M}}\ \mathsf{in}\ \mathsf{let}\ , f_2 = \overline{\overline{N}}\ \mathsf{in}\ \lambda\rho\kappa.(f_1\ \rho\ [\![(\lambda m.,(f_2\ \rho\ [\![(\lambda n.m(n,(,\kappa)))]\!]))]\!])
\end{array}$$

On $((\lambda x.x)\ 5)$ the transform above yields the following shorter term.

$(\lambda\rho\kappa.((\lambda m.((\lambda n.m(n,\kappa))5))(\lambda v\kappa'.(\kappa'(\rho[x \mapsto v])(x)))))$

Suitable rules embodying the transformation above involving open terms could extend the work on the CPS-transform to all compilers.

7.4 Additional Languages

All the examples in this thesis have been implemented in either Scheme or Common LISP. Can the transformation technique be used in other languages? What are the key characteristics needed so that the transformation technique can be applied? Are there languages other than Scheme and Common LISP that posses these characteristics?

The needed characteristics are implicit in the definition of the interpreter language from chapter 4. In particular, we need a high-level language with first-class functions and some means of quotation. We also suggest garbage collection as a key feature so that programming will not be too painful.

There are experimental languages that are specifically designed for meta-programming: Meta-ML [98] and Template Haskell [88]. These languages were designed to have the needed characteristics.

Most other LISP dialects also have these characteristics. Thus popular alternative LISP implementations such as Racket [40, 41] and Closure [85] can be used.

There have also been some languages inspired by LISP that are becoming popular that retain the relevant features. For example, Scala [77, 86] and Julia [11] both have first class functions and quotation.

What about more mainstream languages such as C# and Java? For some time, C# [47] has had first class functions. First class functions have become a part of Java with Java 8 [44]. But neither one has the kind of quotation discussed in this thesis. It is still possible

to fake quotation and generate text with the more primitive notions of strings and string concatenation. It should be possible to use the transformation technique in this more limited fashion for these languages.

We anticipate exploring examples of the transformation technique using these other languages.

7.5 Automation

We believe that attempting to fully automate the conversion of an interpreter into a compiler is not worthwhile. For example, in chapter 6 we saw that to construct a realistic compiler, it was necessary to replace the naïve unification algorithm with the more sophisticated union-find based implementation. In general, no automatic system will be capable of such insight. Nevertheless, it may be useful for larger programs to have tools to help apply the transformation rules.

Implementing currying appears straightforward. If the name of the function to be curried and the static parameters are specified, it is trivial to modify the code so that a function involving the static parameters is returned. Correcting the calling protocol requires more effort[1]. Assuming all possible call locations are circumscribed, these call locations would need to be identified. Further, it would be necessary to verify that no other functions flow to those locations. For such a flow analysis, 0CFA [89] should be adequate.

Implementing rules (4.3), (4.4), and (4.5), the rules that stage the computation, should also be mostly straightforward. As hinted at in section 7.3, when automating, rule 4.4 would be replaced with a rule to contract applications applied to variables. One difficulty when implementing rules (4.3) and (4.5) involves verifying that particular terms termi-

[1]Flow analysis seems like a natural choice for identifying call sites; however, for a statically typed language, typing checking could also be used.

nate. The solution could be to have a simple function attempt to prove that the term in question does terminate. If it succeeds, then the transformation can proceed. If it fails, it could either log that it does not actually know the term terminates, or it could query a human oracle. Another difficulty is if some parameters, such as the store, are implicit. A careless implementation might automatically lift terms incorrectly that make use of these implicit parameters. Such implicit parameters are not arbitrary; they are typically restricted to the store parameter and perhaps the I/O parameter. Thus it is likely that an analysis could identify which functions perform side-effects and thereby make explicit the implicit parameters.

Finally, there are two reasons why quoting may be the trickiest to implement. First, one would like to relax the restriction that the free variables are all locally let-bound. This restriction makes it easy to see that not only are all the free variables under consideration, but also that they are all suitably modified so that they become comma-variables. In practice, we have not followed this rule to the letter (for example in chapter 6); we allowed parameters that are not let-bound to be included among those that were unquoted. It is likely that the way to proceed is to have a rule preprocessing phase that identifies the non-global free variables in an abstraction and adds the needed let forms so that the original rule applies.

Second, the semantics of Scheme unquoting is more complicated than the model of unquoting in this thesis. In particular, we allow a comma-variable to be replaced with unquoted values. For Scheme, this semantics entails that numbers and symbols are treated the same way. But that is a problem in Scheme. If a symbol is inserted into a list representing code, the symbol becomes a variable.

For example, `(let ((s 'x)) `(memq ,s '(x)))` will generate code with an unbound variable. Instead, the code `(let ((s "x")) `(memq ,s '(x)))` will generate the desired code.

For a more accurate model, we can preserve comma-variables, but require that the let form only substitute quoted values. This change leads to several complications. Since we would like to be able to achieve the same effect as replacing a comma-variable with an unquoted value, we need a new reify operator that converts a value into appropriate text. Formalizing the semantics of such an operator might look something like the following.

$$\frac{}{\ulcorner v \urcorner \rightarrow [\![v]\!]}$$

$$\frac{e \rightarrow e'}{\ulcorner e \urcorner \rightarrow \ulcorner e' \urcorner}$$

When we change the let form and add the reify operator, we find that some expressions among the let-bound variables need to be reified and some do not. How can we tell which is which? We conjecture that the answer involves a type analysis using a type system similar to that found in [74].

Chapter 8

Conclusion

All's Well That Ends Well

William Shakespeare

In this chapter we summarize what has been accomplished. Specifically, we review the transformation technique, mention how it can often be extended to non-denotational interpreters, and comment on the benefits.

8.1 The Transformation Technique

This thesis has presented a new transformation technique for deriving a compiler from an interpreter. This technique consists of the application of four rules: Currying, Lambda Lowering, Expression Lifting, and Quoting. Currying splits the static and dynamic parameters and turns a function with static and dynamic parameters into one that returns a function with dynamic parameters. Lambda lowering moves the testing in a conditional out of an abstraction. Expression lifting moves an expression out of an abstraction. Quoting turns a function that returns a function into a function that returns text. We have proved that lambda lowering and expression lifting preserve the meaning of terms. And

we have proved that the transformation technique yields a correct compiler given an abstract denotational interpreter.

8.2 Denotational Interpreters and Beyond

One of the ideas motivating the transformation technique comes from denotational semantics: a denotational semantics is a compiler. Mathematically, we are free to use a reduction strategy in which applications inside abstractions are reduced. To get a compiler in the context of a reduction strategy that does not go inside abstractions, we must move the expressions that would be reduced out of the abstraction. Thus the transformation technique is designed to work on interpreters written in a denotational style.

Yet interpreters are not always written in a denotational style. A common alternative is basing an interpreter on an operational semantics. Certain key operational idioms cannot be trivially interpreted denotationally. For example, neither recursive interpreter calls on forms that are not substructures nor the interpreter leaving program text in the environment or returned values can be viewed denotationally. In those cases, we showed how to turn an operational style interpreter into a denotational one by, for example, introducing a fixed-point function or inserting calls to the interpreter. At that point, the transformation technique applies.

8.3 Comparison to Partial Evaluation and Staging

In chapter 2, we discussed partial evaluation as a technique similar in spirit to the transformation technique. There we argued that although one can derive target code for particular examples using partial evaluation, deriving a code-generator is laborious because the equational reasoning must be performed on both the interpreter and the partial evalu-

ator. We also argued that using cogen-based partial evaluation to derive a code-generator is similarly laborious because of the need for the separate step of the binding time analysis. In chapter 6, we implicitly argued that the transformation technique has practical and conceptual advantages over partial evaluation. The practical advantage is that, in contrast to Consel and Khoo [24], we were able to use the transformation technique to construct a PROLOG compiler capable of compiling recursive clauses. The conceptual advantage of the transformation technique is that while partial evaluation obscured the issue of recursion, the rules proposed here dealt with the issue successfully.

Chapter 2 also discussed the manual technique of staging. From the beginning, Jørring and Scherlis [56] hinted that interpreters could be mechanically transformed into compilers, but they did not say how to do so. However, subsequent work [18, 74, 98] in staging relies on the programmer guessing a staged form of an algorithm; a type-checking algorithm provides a post-facto verification. In contrast, we believe that the transformation technique presented in this thesis fulfills the vision of Jørring and Scherlis by providing apriori advice to help the programmer perform the staging.

8.4 Practical Benefits

The advantage of an interpreter is that it is easier to write. The advantage of a compiler is that it generates code that runs faster than the interpreter. The transformation technique turns an interpreter into a compiler. One benefit is that the derived compiler is guaranteed to preserve the semantics of the interpreter. Another benefit is that deriving the compiler takes a marginal effort beyond the effort of writing the interpreter. Thus one gets a compiler for about the effort of an interpreter. Finally, a benefit is that the generated code does indeed run faster. In our experiments, compiled code runs almost five times faster than the interpreted code.

Appendix A

Lemmata

Lemma 1. *If* $e \to^* v$ *then* (if e then e_1 else e_2) $\to^*$ (if v then e_1 else e_2).

Proof. By mathematical induction on the number of reduction steps.

- Suppose $e \to^0 v$. Then e is identical to v and the result follows immediately.
- Assume that $e' \to^k v$ implies (if e' then e_1 else e_2) $\to^*$ (if v then e_1 else e_2).
  - Suppose $e \to^{k+1} v$. Then $e \to e' \to^k v$.

    The reduction $e \to e'$ implies (if e then e_1 else e_2) $\to$ (if e' then e_1 else e_2). Since $e' \to^k v$, it follows from the assumption that (if e' then e_1 else e_2) $\to^*$ (if v then e_1 else e_2). Hence the result follows.

□

Lemma 2. *If* $e \to^* v$ *then* $(\lambda z.e_b)(e) \to^* (\lambda z.e_b)(v)$.

Proof. By mathematical induction on the number of reduction steps.

- Suppose $e \to^0 v$. Then e is identical to v and the result follows immediately.
- Assume that $e' \to^k v$ implies $(\lambda z.e_b)(e') \to^* (\lambda z.e_b)(v)$.
  - Suppose $e \to^{k+1} v$. Then $e \to e' \to^k v$.
    The reduction $e \to e'$ implies $(\lambda z.e_b)(e) \to (\lambda z.e_b)(e')$. Since $e' \to^k v$, it follows from the assumption that $((\lambda z.e_b)(e') \to^* (\lambda z.e_b)(v)$. Hence the result follows.

□

Lemma 3. *If* $e' = v$ *then* $(e[u := e']) = (e[u := v])$.

Proof. By structural induction on e.

- Suppose e is a constant c. Then the two expressions are identical and equality follows from the fact that equality is reflexive.
- Suppose e is a variable x.
  - Suppose $x = u$. Then equality follows from the assumption that $e' = v$.
  - Suppose $x \neq u$. Then the two expressions are identical and equality follows from the fact that equality is reflexive.
- Suppose e is a comma variable $, y$. Then the two expressions are identical and equality follows from the fact that equality is reflexive.
- Assume $e' = v$ implies $(e''[u := e']) = (e''[u := v])$ if e'' is a substructure of e.

– Suppose e is an abstraction $(\lambda\bar{x}.e'')$. Without loss of generality, assume $u \neq x_i$. Then $(\lambda\bar{x}.e'')[u := e']$ is $(\lambda\bar{x}.(e''[u := e']))$, and $(\lambda\bar{x}.e'')[u := v]$ is $(\lambda\bar{x}.(e''[u := v]))$. It follows from the assumption that $(e''[u := e']) = (e''[u := v])$. Hence by the abstraction equality rule (ξ), the abstractions are equal.

– Suppose e is a quoted form $[\![e'']\!]$. Note that $[\![e'']\!][u := e']$ is $[\![(e''[u := e'])]\!]$, and $[\![e'']\!][u := v]$ is $[\![(e''[u := v])]\!]$. It follows from the assumption that $(e''[u := e']) = (e''[u := v])$. Hence by the quoted form equality rule, the quoted forms are equal.

– Suppose e is an application $e_0(e_1, \cdots, e_n)$. Note that $e_0(e_1, \cdots, e_n)[u := e']$ is $(e_0[u := e'])(e_1[u := e'], \cdots, e_n[u := e'])$, and $e_0(e_1, \cdots, e_n)[u := v]$ is $(e_0[u := v])(e_1[u := v], \cdots, e_n[u := v])$. It follows from the assumption that for every component term e_i, $(e_i[u := e']) = (e_i[u := v])$. Hence by repeated application of the application equality rule, the applications are equal.

– Suppose e is a let-form $\mathsf{let}\ , y = e_1\ \mathsf{in}\ e_2$. Note that $(\mathsf{let}\ , y = e_1\ \mathsf{in}\ e_2)[u := e']$ is $\mathsf{let}\ , y = (e_1[u := e'])\ \mathsf{in}\ (e_2[u := e'])$, and $(\mathsf{let}\ , y = e_1\ \mathsf{in}\ e_2)[u := v]$ is $\mathsf{let}\ , y = (e_1[u := v])\ \mathsf{in}\ (e_2[u := v])$. It follows from the assumption that for each component term e_i, $(e_i[u := e']) = (e_i[u := v])$. Hence by application of the two let-form equality rules, the let-forms are equal.

– Suppose e is a conditional $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Note that $(\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)[u := e']$ is $\mathsf{if}\ (e_0[u := e'])\ \mathsf{then}\ (e_1[u := e'])\ \mathsf{else}\ (e_2[u := e'])$, and $(\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)[u := v]$ is $\mathsf{if}\ (e_0[u := v])\ \mathsf{then}\ (e_1[u := v])\ \mathsf{else}\ (e_2[u := v])$. It follows from the assumption that for each component term e_i, $(e_i[u := e']) = (e_i[u := v])$. Hence by application of the three conditional form equality rules, the conditionals are equal.

□

Bibliography

[1] H. Abelson, G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press; 2nd edition, 1996.

[2] A. V. Aho, J. D. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977.

[3] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley; 2nd edition, 2007.

[5] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[6] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991.

[7] J. W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of Zürich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing*, UNESCO, 125–132, 1959.

[8] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1985.

[9] A. Bawden. Quasiquotation in Lisp. *Partial Evaluation and Semantic-Based Program Manipulation*, 4–12, 1999.

[10] L. Beckman, A. Haraldson, Ö. Osarksson, E. Sandewall. A Partial Evaluator, and its Use as a Programming Tool. *Artificial Intelligence*, Vol. 7, No. 4, 319 – 357, 1976.

[11] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. arXiv:1209.5145 [cs.PL].

[12] L. Birkedal, M. Welinder. Hand-Writing Program Generator Generators. *Programming Language Implementation and Logic Programming*, Springer, 1994.

[13] A. Bondorf, O. Danvy. Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. *Science of Computer Programming*, Vol. 16, 151–195, 1991.

[14] G. Booch. Oral History of John Backus. `http://archive.computerhistory.org/resources/text/Oral_History/Backus_John/`.

[15] F. P. Brooks, Jr. *The Mythical Man Month*. Addison Wesley; 2nd edition, 1995.

[16] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, Vol. 11, No. 4, 481–494, 1964.

[17] R. M. Bustall, J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, Vol. 24, No. 1, 44–67, 1977.

[18] C. Calcagno, W. Taha, L. Huang, X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. *Generative Programming and Component Engineering*, 57–76, 2003.

[19] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[20] C. Consel. A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 145–154, 1993.

[21] C. Consel. New Insights into Partial Evaluation: The Schism Experiment. *Lecture Notes in Computer Science*, Vol. 300, 236–246, Springer-Verlag, 1988.

[22] C. Consel. *Tempo Specializer - History and Contributions*. `http://phoenix.labri.fr/software/tempo/doc/tempo-doc-contrib.html#history`.

[23] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, J. Noyé. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, Vol. 30, No. 3es, 1998.

[24] C. Consel, S. C. Khoo. Semantics-Directed Generation of a Prolog Compiler. *Science of Computer Programming*, Vol. 21, No. 3, 263–291, 1993.

[25] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[26] O. Danvy, A. Filinski. Abstracting Control. *Proceedings of the 1990 ACM Conference on LISP and Functional Progamming*, 151–160, 1990.

[27] O. Danvy, A. Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, 361 – 391, 1992.

[28] O. Danvy, H. K. Rohde. On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation. *Information Processing Letters*, Vol. 99, No. 4, 158–162, 2006.

[29] O. Danvy and R. Vestergaard. Semantics Based Compiling: A Case Study in Type Directed Partial Evaluation. *Eighth International Symposium on Programming Language Implementation and Logic Programming*, 182–497, 1996.

[30] R. Davies. A Temporal-Logic Approach to Binding-Time Analysis. *Proceedings of the Symposium on Logic in Computer Science*, 184–195, 1996.

[31] R. Davies, F. Pfenning. A Modal Analysis of Staged Computation. *Journal of the ACM*, Vol. 48, No. 3, 555–604, 2001.

[32] P. Deransart, A. Ed-Dbali, L. Cervoni. *Prolog: The Standard: Reference Manual*. Springer, 1996.

[33] J. Dias. *Automatically Generating the Back End of a Compiler Using Declarative Machine Descriptions*. PhD thesis, Harvard University, Cambridge, MA, USA, 2008.

[34] S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, University of the Saarland, Saarbrücken, Germany, 1996.

[35] A. P. Ershov. On the Essence of Compilation. *Formal Description of Programming Concepts*, 391–420, North-Holland, 1978.

[36] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, New York, NY, USA, 1995.

[37] M. Feeley, G. LaPalme. Using Closures for Code Generation. *Computer Language*, Vol. 12, No. 1, 47–66, 1987.

[38] M. Felleisen, R. B. Findler, M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[39] M. Felleisen, D. P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. *Formal Description of Programming Concepts III* edited by M. Wirsing, 193–217. Elsevier, 1986.

[40] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, M. Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, Vol. 12, No. 2, 159–182, 2002.

[41] M. Flatt et al. *The Racket Reference*. `http://docs.racket-lang.org/reference/`, 2014.

[42] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, Vol. 2, No. 5, 45–50, 1971.

[43] R. Glück. A Self-Applicable Online Partial Evaluator for Recursive Flowchart Languages. *Software: Practice and Experience*, Vol. 42, No. 6, 649–673, 2012.

[44] J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha, A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[45] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[46] J. Hannan. Operational Semantics-Directed Compilers and Machine Architectures. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, 1215–1247, 1994.

[47] A. Hejlsberg, S. Wiltamuth, P. Golde. *C# language specification*. Addison-Wesley, 2003.

[48] G. Hopper. The Education of a Computer. *Proceedings of the Association for Computing Machinery Conference*, 243–249, May 1952.

[49] S. C. Johnson. *Yacc: Yet Another Compiler Compiler*. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ 07974, 1975.

[50] N. D. Jones. Personal Communication, 2013.

[51] N. D. Jones, C. K. Gomard, P. Sestoft, L. O. Andersen, T. Mogensen. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[52] N. D. Jones, P. Sestoft, H. Søndergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. *Lecture Notes in Computer Science*, Vol. 202, 124–140, Springer-Verlag, 1985.

[53] N. D. Jones, P. Sestoft, H. Søndergaard. MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *LISP and Symbolic Computation*, Vol. 2, No. 1, 9–50, 1989.

[54] J. Jørgensen. Compiler Generation by Partial Evaluation. Master's Thesis, DIKU, University of Copenhagen, Denmark, 1992.

[55] J. Jørgensen. Generating a Compiler for a Lazy Language by Partial Evaluation. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 258–268, 1992.

[56] U. Jørring, W. L. Scherlis. Compilers and Staging Transformations. *Symposium on Principles of Programming Languages*, 86–96, 1986.

[57] M. Karr. Personal Communication.

[58] R. Kelsey, W. Clinger, J. Rees editors. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, 26–76, 1998.

[59] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1964.

[60] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, Vol. 7, No. 12, 735–736, 1964.

[61] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, Vol. 6, 308–320, 1964.

[62] H. Lawson. `http://www.computerhistory.org/events/lectures/cobol_06121997/index.shtml`.

[63] P. Lee. *Realistic Compiler Generation*. MIT Press, 1990.

[64] P. Lee, U. Pleban. A Realistic Compiler Generator Based on High-Level Semantics: Another Progress Report. *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 284–295,1987.

[65] M. E. Lesk. *Lex – A Lexical Analyzer Generator*. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, NJ 07974, 1975.

[66] L. A. Lombardi. Lisp as the Language for an Incremental Computer. *The Programming Language LISP: Its Operation and Applications*, MIT Press, 204–219, 1964.

[67] D. Maier, D. S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin Cummings, 1988.

[68] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[69] W. M. McKeeman, J. J. Horning, D. B. Wortman. *A Compiler Generator*. Prentice-Hall, 1970.

[70] E. Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall/CRC, 2009.

[71] E. Moggi. Notions of Computation and Monads. *Information and Computation*, Vol. 93, 55-92, 1991.

[72] E. Moggi, W. Taha, Z. Benaissa, T. Sheard. An Idealized MetaML: Simpler, and More Expressive. *Proeedings of the European Symposium on Programming*, 193–207, 1999.

[73] P. Mosses. *SIS, Semantics Implementation System: Reference Manual and User Guide*. DAIMI Technical Report MD-30, University of Aarhus, Denmark, 1979.

[74] A. Nanevski, F. Pfenning. Staged Computation with Names and Necessity. *Journal of Functional Programming*, Vol. 15, No. 6, 893–939, 2005.

[75] F. Nielson, H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[76] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1991.

[77] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger. *An overview of the Scala programming language*. No. LAMP-REPORT-2004-006. 2004.

[78] S. Owens, J. Reppy, A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, Vol. 19, No. 2, 173-190, 2009.

[79] L. Paulson. A Semantics-Directed Compiler Generator. *Symposium on Principles of Programming Languages*, 224–232, 1982.

[80] E. Pelegrí-Llopart, S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. *Symposium on Principles of Programming Languages*, 294–308, 1988.

[81] F. C. N. Pereira, D. H. D. Warren. Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, Vol. 13, No. 3, 231–278, 1980.

[82] N. Pippenger. *Theories of Computability*. Cambridge University Press, 1997.

[83] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, Vol. 1, 125–159, 1975.

[84] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[85] A. Rathore. *Clojure in Action*. Manning Publications, 2011.

[86] T. Rompf, M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, Vol. 55, no. 6, 121 – 130, 2012.

[87] T. Sheard, Z. Benaissa. From Interpreter to Compiler Using Staging and Monads. *Proceeeding of the 1998 International Conference on Functional Programming*, 1998.

[88] T. Sheard, S. P. Jones. Template meta-programming for Haskell. *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 1 – 16, 2002.

[89] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA , USA, 1991.

[90] M. J. A. Smith. *Semantics-Directed Compiler Generation*. Part II Dissertation, University of Cambridge, Computer Laboratory, `http://lanther.co.uk/compsci/semcom/semcom.pdf`, Cambridge, UK, 2005.

[91] J. Sobel, E. Hilsdale, R. K. Dybvig, D. P. Friedman. Abstraction and Performance from Explicit Monadic Reflection. Unpublished manuscript, 1999.

[92] M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten editors. Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`.

[93] G. L. Steele Jr. Building Interpreters by Composing Monads. *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 472 – 492, 1994.

[94] G. L. Steele Jr. *Common LISP: The Language*. Digital Press; 2nd edition, 1984.

[95] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981.

[96] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Hillsboro, Oregon, USA, 1999.

[97] W. Taha, Z. Benaissa, T. Sheard. Multi-Stage Programming: Axiomatization and Type Safety. *Automata, Languages and Programming*, 918–929, 1998.

[98] W. Taha, T. Sheard. Multi-Stage Programming with Explicit Annotations. *Workshop on Partial Evaluation and Semantics Based Program Manipulation*, 203–217, 1997.

[99] P. J. Thiemann. Cogen in Six Lines. *ACM SIGPLAN Notices*, Vol. 31, No. 6. ACM, 1996.

[100] P. J. Thiemann. *The PGG system–user manual*. 2000.

[101] M. Tofte. *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do*. Springer, 1990.

[102] F. Turbak, D. Gifford, M. A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.

[103] D. H. D. Warren. An Abstract Prolog Instruction Set. Report 309, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1983.

[104] D. H. D. Warren, L. M. Pereira. PROLOG — The Language and It's Implementation Compared with LISP. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. SIGPLAN Notices Vol. 12, No. 8, 1977.

[105] R. L. Wexelblat. *History of Programming Languages*. Academic Press, 1981.

[106] J. Zeng. *Partial Evaluation for Code Generation from Domain-Specific Languages*. PhD thesis, Columbia University, New York, NY, USA, 2007.