THE CALCULI OF LAMBDA-v-CS CONVERSION:

A SYNTACTIC THEORY OF CONTROL AND STATE

IN IMPERATIVE HIGHER-ORDER PROGRAMMING LANGUAGES

Matthias Felleisen

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
of the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

August 1987

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree Doctor of Philosophy.

Daniel P. Friedman

Daniel P. Friedman, Ph.D.

Doctoral Committee:

J. Michael Dunn

J. Michael Dunn

Paul W. Purdom

Paul W. Purdom

Mitchell Wand

Mitchell Wand

11 July 1987

To my parents and my grandmother

Abstract

Imperative extensions of higher-order functional languages are highly expressive media for reasoning about programming problems and solutions. They directly support a broad variety of programming paradigms and their associated facilities, *e.g.*, lexical scoping with blocks and modules, object-oriented programming with message-passing entities, backtracking with relations and coroutines. Purely functional languages can also emulate these facilities, but, in general, this requires a (re-) formulation of the entire program in a special style.

The advantage of functional programming languages is that they automatically come with a powerful, symbolic reasoning system. They are syntactically and semantically variants of the λ-calculus, and this connection provides an abstract understanding of programs, independent of any implementation machinery. Thus, for verifications, transformations, and comparisons, programmers can manipulate programs in an algebra-like manner. However, the λ-calculus is too weak to support imperative extensions of functional programming languages. Consequently, it has been impossible to reason algebraically about imperative programs. We have solved this problem with the construction of the λ_v-CS-calculus.

The λ_v-CS-calculus is a conservative extension of the λ-calculus. Its principal syntactic facilities are variables and assignable variables, functional abstractions,

function applications, and two new expression types for the manipulation of evaluation control and program state: ubiquitous building blocks of most practical languages. The extended set of axioms is derived from an abstract machine semantics. They satisfy a variant of the Church-Rosser Theorem, the Curry-Feys Standardization Theorem, and Plotkin's Correspondence Theorems. With the calculus, it is easy to establish and prove equational properties of such facilities as loop exits, cell objects, and generators. Since the calculus-language can express high-level constructs and their low-level, assembly-like implementations, we can also formalize and establish the correctness of classical compiler techniques with simple program transformations. Examples are the elimination of tail-recursion and the implementation of recursion with self-references.

The construction of the λ_v-CS-calculus is a contribution to the study of fundamental concepts in programming languages. It is an attempt to bridge the gap between mathematically elegant and realistic programming languages. A further exploration of the calculus will certainly produce deeper insights into imperative higher-order programming and its paradigms.

Acknowledgements

I owe thanks to my advisor and *Doktorvater* Daniel P. Friedman for introducing me to the fascinating world of closures, continuations, and assignments; for the stimulating discussions on languages and calculi; for being available day and night to give advice; and, most importantly, for his friendship throughout these years of apprenticeship.

Another important support element has been my doctoral committee consisting of Michael J. Dunn, Paul W. Purdom, and Mitchel Wand. Mike Dunn provided many insights into the logic of things; Paul Purdom safely steered me around administrative and Tex-y obstacles; Mitch Wand streamlined many of my windy motivations.

Beyond this official doctoral environment, Bruce Duba and Eugene Kohlbecker have played a crucial role in my endeavor. Without them, I may have never understood Dan's original ideas on rules for our favorite Scheme construct call/cc. In particular the numerous talks on Bruce's front-porch swing have been a constant source of inspiration. I'll miss them.

Furthermore, I thank Carolyn Talcott for her critical reading of most of my papers and a draft of this dissertation. Like nobody else, she went through my proofs with greatest care, pointed out many improvements, and posed a great deal of interesting questions.

Last but not least, I mention my loving and caring wife Helga. Especially in the last few months, she had to put up with a lot of hectic activities and panic between me and our son Christopher. For all that and a lot more: Thank You. We both are indebted to our families for their support throughout these school years and to our American family-extension George Paulik for his wise words on so many issues.

Financial Support. My dissertation has been financed by a teaching assistantship from the Indiana University Computer Science Department, an NSF research assistantship (NSF grants DCR 85-03279 and DCR 85-01277), an IBM Graduate Research Fellowship, a summer internship with the Software Technology Program at the MCC/Austin, and an Indiana University Summer Research Fellowship. The Program in Classical Archaeology provided a comfortable workspace for both me and my wife. I gratefully acknowledge all this help.

Preface

This preface provides a high-level survey of our technical contributions, an explanation about their relationship to mathematics, and some basic definitions of notation and terminology.

Overview. The λ_v-CS-calculus is an equational theory about the ubiquitous programming language concepts of function, control operation, and assignment. The theory is a conservative extension of a variant of Church's λ-calculus. For the treatment of control and assignment operations, the calculus includes two new sets of term relations: reduction and computation rules. The former are freely applicable term relations like Church's β-rule; the latter are program relations that can only be applied to a redex at the root of a program. The division is necessary for the coordination of imperative effects, but it does not constitute an infringement on the calculus's properties or capabilities. The λ_v-CS-calculus satisfies a modified version of the Church-Rosser Consistency Theorem and the Curry-Feys Standardization Theorem.

We derive the calculus from and for a programming language called Idealized Scheme. The language generalizes the principal computational primitives of most sequential imperative programming languages. Most facilities of other languages can be expressed as syntactic abbreviations of Idealized Scheme expressions and can be treated in the calculus with simple equivalences. The syntax of Idealized Scheme

is an extension of the λ-calculus-term language. Beyond constants, variables, λ-abstractions, and applications, the term set contains two new expression categories: $\mathcal{F}$-applications for the manipulation of evaluation control and σ-capabilities for the manipulation of state. The former are related to label values in Algol derivatives and to continuation values in Scheme-like languages; the latter are expression-oriented variants of assignment statements. The semantics of the entire language is defined via a denotational-style abstract machine.

From the abstract machine semantics we systematically derive the above-mentioned set of term relations for the λ_v-CS-calculus. The main result with respect to the relationship between Idealized Scheme and the calculus is a generalization of Plotkin's Correspondence Theorem. The theorem verifies that, under certain conditions, the equality of two programs in the calculus implies operationally indistinguishable behavior and is thus a major tool for reasoning about imperative programs. With such a symbolic reasoning system, a programmer can manipulate imperative programs like algebraic expressions—on a *symbolic-syntactic* level. The programmer can algebraically determine the result of a program, prove the equivalence of two programs, or transform an obviously correct program into a more efficient version. For example, with a few, simple equations we prove the correctness of the elimination of tail-recursive function applications in favor of unconditional jumps and the implementation of recursive functions with self-referential structures, two traditional compiler techniques that are usually justified on informal grounds.

In some sense, our research attempts to reconcile the premisses of two diverging currents in the programming language research community: the one striving for mathematically-based languages, the other for expressive ones. The advantage of a mathematical language is that it automatically comes with an equational reasoning system, *e.g.*, all functional languages directly correspond to some variant of the

λ-calculus. Imperative extensions of functional languages, on the other hand, are much better equipped for expressing an exceptional flow of control and the occurrence of a state-changing event. Unfortunately, because of the additional imperative operations, the usual mathematical theories cannot support equational reasoning with these languages. The λ_v-CS-calculus resolves this dichotomy by providing an algebra-like system for the manipulation of imperative programs.

Prerequisites and Mathematics. The goal of our research is a deeper understanding of imperative computations in an extended functional programming language. The major *tool* for this endeavor is the mathematics of sets, terms, term relations, and induction. Accordingly, we concentrate on the interpretation and application of mathematical results. Informative or constructive proofs are developed together with the theorems; long proofs that do not immediately further the understanding of a topic are moved to subsections. The proofs are kept informal so long as it is easy to fill in the gaps.

We assume that the reader is acquainted with the contents of an undergraduate and a graduate course on the principles of programming languages as they are taught at a university like Indiana University; that the reader has been exposed to the notion of a state transition system; and at various places we assume some superficial familiarity with the organization of a denotational programming language semantics—for example, that the separation of environment and store is necessary for an imperative higher-order language—and its realization as an abstract machine. Further knowledge of denotational semantics is also helpful for the analysis of the dichotomy between functional languages and imperative extensions in Chapter 1.

The necessary mathematical background is developed in Chapters 2 and 3. The second chapter is an introduction to the concepts of abstract machines and calculi for functional languages; the third introduces an abstract machine for imperative

extensions of functional languages. We assume that the reader is knowledgeable of induction principles.

Notation and Terminology. Three basic mathematical notions deserve some general explanation. First, we use the word *domain* when talking about sets for meanings of programs, but also when referring to the range of values that the independent variable of a function can assume. Domains in the first sense are generally defined by a system of mutually recursive equations. In order to improve readability, the meta-language for these equations contains some elements from abstract syntax. For example, an equation of the form

$$A = B \times \mathbf{token} \times C$$

defines the set A as the cartesian product of B, $\{\mathbf{token}\}$, and C. A typical element from this set may be written as

$$(b\ \mathbf{token}\ c).$$

Second, many domains are sets of partial or finite functions from a set A to a set B. We denote these domains with

$$A \multimap B.$$

If f is a finite function on A, then the domain of f, $Dom(f)$, is the subset of A on which f is defined. In some cases it is more convenient to perceive a finite function as a set of pairs. We shall switch to whatever variant is more convenient. The *update* of a finite function f for x with value y is written

$$f[x := y]$$

and denotes a new function such that

$$f[x := y](z) = \begin{cases} y & \text{if } x = z \\ f(z) & \text{otherwise.} \end{cases}$$

Warning. We overload this notation twice: for term substitution and labeled-value substitution. Both are syntactic counterparts to function updates. The textual context should distinguish the different uses. **End**

Finally, besides the relations proper, we frequently need transitive and reflexive closures. If

$$A \longrightarrow B$$

stands for a single step relation from A to B, then

$$A \longrightarrow^{+} B \text{ and } A \longrightarrow^{*} B$$

denote the transitive and transitive-reflexive closures. Exceptions to this rule are clearly indicated.

We use ordinary programming language terminology which generalizes mathematical usage. Thus, we call a programming language object a *function* if it is applicable to different values and yields values, even though it may not correspond to a mathematical function. If it does, we call the function *pure*. We refer to apparent or bound *variables* for the syntactic objects that define an equivalence relation on term positions. With *application* we mean two different things, namely, the syntactic juxtaposition of two objects and the computational process of calling a function. To emphasize that we mean the latter, we sometimes use *function invocation*.

For talking about terms and programs, we make use of abstract syntax terminology and freely mix it with tree terminology. Hence, we shall call the outermost syntactic construction the *root* and parts of a term the *sub-expressions*. Other words should be self-explanatory.

Contents

Definitions

1. Expressiveness versus Mathematics

The programming language research community knows two diverging currents: one striving for mathematically elegant formalisms, the other for highly expressive thought media. In general, the basis of a mathematically-oriented language consists of a single concept from mathematics. The set of syntactic categories is small, each category and its interaction with others is well-understood. The underlying mathematical theory provides a foundation for symbolic reasoning about programs in the language. Two prominent examples are pure[1] Lisp and Prolog. The former is associated with λ-calculus, the latter with Horn-clause logic.

With a symbolic reasoning system, a programmer can manipulate programs like algebraic expressions—on a *symbolic-syntactic* level. This is useful in determining the result of a program, proving the equivalence of two programs, or transforming programs into more readable or efficient versions. In short, a symbolic reasoning system provides an abstract understanding of programs, independent of any implementations or annotations.

The mathematical elegance of a programming language/reasoning-system pair is obviously appealing. The exploration of this direction of language design has led to the discovery of many programming principles, to an improved understanding

[1] Full-funarg, lexically-scoped Lisp.

of structural elements in language designs, and to novel implementation strategies. Yet, when it comes to realistic programming, mathematical languages are usually offered as *impure* realizations with imperative operations. Their addition is justified on grounds of efficiency; their use, however, is discouraged because the underlying mathematical theory cannot account for the imperative effects.

An advocate of expressive languages rejects restrictions on the use of linguistic facilities. If an imperative facility can succinctly express the idea of a particular program, then it is appropriate to use. After all, a programming language is a vehicle for formalizing and organizing thoughts about problems and their solutions, and some problems and solutions are better expressed in an algorithmic, event-based manner. From this perspective, the design of a programming language means anticipating the needs of a language user and building in useful facilities.

At first sight, this argument leads to big and baroque languages, but this is not necessary. A series of experimental language designs from the mid 60's to the late 70's has demonstrated that a practical language can be simultaneously small and expressive. The important insight is to distinguish between a language core and syntactic abstractions. A language core is the set of facilities that are truly necessary essentials of a language; the set of syntactic abstractions comprises all those constructs that are abbreviations of typical usage patterns of core facilities.

From this point of view an analysis of Algol-style languages and sub-languages produces an interesting result: higher-order functions, access to an abstraction of the current thread of program control, and lexical assignment suffice to treat practically all traditional language facilities as syntactic abstractions. The range of expressible facilities includes looping constructs, blocks, modules, structured data objects, function exits, co-routines, and quasi-parallel processes.

The meaning of this finding is clear. Not only are control facilities and assign-

ments ubiquitous building blocks of programming languages, but they also express fundamental concepts that are difficult to formulate within a functional framework. However, as mentioned above, adding these imperative facilities to a functional language invalidates the correspondence between the language and the reasoning system. And thus,

> it appears [that] we have a choice: we can either define weak systems, such as purely functional languages, about which we can prove theorems, or we can define strong systems which we can prove little about.[2]

We are more hopeful and accept both premisses: a language should simultaneously be expressive and associated with a symbolic reasoning system. Consequently, our goal must be the construction of a symbolic reasoning system for the core of expressive languages. In the rest of this chapter, we further explore the notion of expressiveness, first in an historical, then in an analytical setting. The presentation is informal and assumes a superficial knowledge of programming language concepts and algebra-like calculi. In the third section, we investigate and formulate the problem statement. The last section contains an outline for the main body of the report.

1.1. A Short History of Expressiveness

The history of the systematic analysis and design of expressive programming languages contains three major milestones. The first to explore the idea of mathematical programming and its relationship to imperative programming was Landin. His major idea is to map out the space for programming languages in a systematic manner. Reynolds's contribution is the realization that this design space has a small basis with regard to traditional languages, and that language research is

[2] This quote is attributed to M. Minsky by T. Knight [32:105].

to investigate this basis. The credit for making these ideas practical goes to Steele and Sussman.

Landin's starting point was a simple observation: a computation should be construed as the mechanical evaluation of a mathematical expression [40]. Furthermore, to conceal the intricasies of computers, programming languages should formalize mathematical notation in a most general sense. Landin chose the λ-calculus as his notational framework.

The λ-calculus [10] is a formal system for studying "the concept of a function as it appears in various branches of mathematics"[10:1]. It consists of a simple term language, and its semantics is defined via a small set of equational axioms. The term language consists of three classes of expressions: variables, functional or λ-abstractions, and function applications. Whereas variables and function applications are traditional mathematical notation, functional abstractions are somewhat unusual. A λ-abstraction defines a function by combining the independent variable and the result expression in a λ-tagged pair, *e.g.*, $\lambda x.x^2$. Furthermore, a λ-abstraction can occur wherever an ordinary expression can occur. It is thus possible that a function's result is a function; in other words, the calculus is a higher-order system.

Semantically, the calculus is equivalent to other computational formalisms like Turing-machines [5:ch6], but syntactically it is rather austere. Nevertheless, it offers a natural framework into which most mathematical notation can be translated. Landin refers to the calculus language as applicative expressions (AE) and to other notations as "syntactic sugaring." Put differently: all systems of functional programming notation are syntactic variants of the λ-calculus, and the study of functional languages and their reasoning systems is based on the study of the λ-calculus.

The kind of syntactic transformation that Landin had in mind for the expansion

of "syntactic sugar" into AE is best illustrated with an example. In mathematical contexts, statements are often qualified with a where-clause, *e.g.*,

$$x^2 + ax + b = 0 \text{ where } a = 2.$$

Such a qualified statement can be restructured into λ-notation as

$$(\lambda a.x^2 + ax + b = 0)2.$$

More generally,

$$M \text{ where } x = L$$

translates into the application of a function (of argument x) to the value L:

$$(\lambda x.M)L.$$

The translation has three characteristics: (1) it is independent of the context in which the expression occurs, (2) it cannot be formulated in terms of functional abstraction because it manipulates the scope of variables, and (3) sub-expressions of the abbreviation become sub-expressions of the expansion. Due to the first and second characteristics, syntactic variants are also called syntactic abstractions.

Landin's next step was to design an abstract machine for the implementation of the mechanical evaluation rules: the SECD-machine. This abstract machine is a rather simple state-transition system, based on machine-related constructions like stacks, dumps, and control memories. It simultaneously provides a formal semantics and an implementation strategy for AE and for its syntactic variants.

This success in explaining a large body of mathematical symbolism in a single framework led Landin to attempt a bigger project: the formalization of Algol's semantics via a syntactic correspondence to a simple AE-like language [35, 37].

In order to capture the meaning of imperative Algol-statements with syntactic abstractions, Landin [39] introduced the language of *imperative* applicative expressions (IAE). IAE extends AE with two new facilities: the J-operator and the assignment operator. The former is used to model the meaning of labels. In simplistic terms, the J-operator provides the program with instantaneous access to the current control state of the machine [7:ch 2]. The result is a label-like value with the same status as a function. When invoked on a value, it resumes the evaluation at the labeled point and discards the current state. The assignment operator resembles the usual side-effect statements in Algol-like languages, but in conjunction with higher-order functions it can achieve new effects. The language semantics is based on an extended SECD-machine, called sharing machine.

In his final, seminal paper on language design, Landin [38] extrapolated his experience into a single framework: ISWIM. ISWIM is a family of programming languages. The family members share a common abstract part and differ in their choice of primitive mathematical entities and functions,[3] *e.g.*, numbers, strings, vectors, *etc.*, an idea originally due to McCarthy [45]. A particular choice of mathematical entities determines the application area of a member language.

The abstract part of the ISWIM-language is constructed from a language core, the prototypical core being IAE. Other language constructs are expressed via definitional equations as patterns of core expressions. The semantics of the core language is defined by an abstract machine. Introducing different syntactic abstractions or a different core produces a different family of languages. Within a given family, the language design process has become simple. Arbitrary choices are eliminated. Landin explains that "... a new language is a point chosen from a well-mapped

[3] The members also differ in their written representation—concrete syntax, an aspect of language design that we disregard.

space, rather than a laboriously devised construction" [38:164]. He does not further explore the idea of varying sets of syntactic abstractions.

The idea of splitting the abstract part of a language into a core and a set of syntactic abstractions naturally provokes the question of how large the core must be. Reynolds [51] investigated this problem by constructing a language GEDANKEN (for Gedanken experiment). GEDANKEN is a minimalistic, Algol-like language based on the principle of completeness: all values, including functions, labels, and cells can be used in any appropriate context. This implies that functions can return labels, that functions can be stored in structures, and that cells can be passed as arguments. The importance of "GEDANKEN lies primarily in the language features which have been excluded"[51:308]. The language proper only contains five major facilities: variables, functions, applications, assignments, and jumps, but still, all traditional constructs of Algol-like languages and extensions of those can be programmed. Reynolds's examples include functional data structures, lists, records, arrays, co-routines, and quasi-nondeterministic computations. Although the term is never mentioned, the programming style for all but the last example recalls Landin's "syntactic sugar." Reynolds's conclusion is that simplicity does not impair expressiveness, except that the implementation of GEDANKEN is extremely inefficient.

In the mid-70's, Sussman and Steele [68] started another experiment of modeling programming languages with the λ-calculus. To this end, they implemented a programming language called Scheme. Scheme is an extension of the λ-calculus-term set that structurally resembles IAE. The major semantic differences between the two languages are the parameter-passing technique and the control operator. Whereas IAE passes arguments by reference, Scheme passes arguments by value. The manipulation of the flow of control in (original) Scheme is based on **catch-**

expressions. An expression of the form

catch *L Body*

constructs a continuation value—an abstraction of the rest of the computation—and binds it to *L* for the evaluation of *Body*. A continuation value has the same status as a function, it can be the result of an expression, passed to functions, and so on, but upon invocation it behaves like a label value and passes its argument back to the labeled point in the evaluation. Syntactically the facility is equivalent to the J-operator, semantically it is simpler, avoiding interference with λ-axioms [14]. A superficial difference between Scheme and IAE is the concrete syntax. Since early implementations were based on Lisp, Scheme's syntax is Lisp-oriented and it is occasionally called a Lisp dialect. Scheme implementations also inherited Lisp's macro facility. Because of this, experimenting with syntactic abstractions is rather convenient, and unlike in ISWIM, the set of syntactic variants is not fixed.

Sussman and Steele's initial goal was to implement Hewitt's actor model of computation, but they soon discovered that Scheme could also express numerous other linguistic facilities [64]. Some of the results were rediscoveries of Landin's and Reynolds's work, others were new, *e.g.*, models for dynamic variable binding and calling mechanisms such as call-by-name and call-by-need. They realized that practically all facilities could be rephrased in terms of functional notation, but that some, including control and assignment operators, were better left in as primitive computation vehicles. Steele [61, 62, 63] then set out to look for efficient implementation techniques. The outcome was a compiler and a set of compilation techniques, which implemented Scheme and its macro-based syntactic extensions quite efficiently.

In the meantime, Scheme has left the laboratory environment and is used as a teaching, research, and industrial programming vehicle. A textbook [2] has ap-

peared that successfully uses Scheme for teaching the current programming paradigms in a single language framework. Research about Scheme concentrates on improved compiler techniques [34], the use of Scheme as a language development framework [11], and practical additions of syntactic abbreviation as a programming tool [33]. Research and programming with Scheme happens in all areas that in the past have been dominated by Lisp and Algol-derivatives.

The Scheme community has recently begun to standardize the language [49]. The development of Scheme is the proof of practicality of Landin's programming language design philosophy. Imperative higher-order languages not only provide a theoretical test-bed for language design ideas, they are also practical programming tools.

1.2. The Essence of Expressiveness

The preceding history of language design teaches that an expressive language need not be complex. The important distinction between core facilities and syntactic abstractions is the foundation for a structural, hierarchical design of a language. For an analysis of languages, it is crucial to understand the core facilities. Syntactic abstractions can be explained via their definitional equations.[4] Thus, language analysis and design reduces to the question whether a given construct is a core facility or not. The answer hinges on what is acceptable as a syntactic abbreviation technique. This is, reduced to a single point, the problem of expressiveness.

Throughout the evolution of expressive languages, the notion of expressiveness has been kept vague. There is no formalization of the meaning of expressiveness. A few hints come from Landin [38] and Sussman and Steele [64]. The former says that a transliteration of a syntactic variant into its expansion must be indepen-

[4] See also Subsection 7.3.2 on this topic.

dent of whatever its context is; the latter notice that ordinary transformations for syntactic variants are "syntactically local" whereas eliminations of control and assignment operations in favor of functional notations involve complex reformulations. In other words, access to the current continuation and lexical assignment are just as fundamental as functions and function applications.

Landin's syntactic abstractions exhibit another important attribute. As mentioned above, his transliterations are usually of denotational character, that is, the result of a syntactic expansion for a complex term depends only on the subexpressions, not on their structure. This further simplifies the expansion algorithm and preserves the possibility of proving and inferring structural properties by induction.

Based on these characteristics, we define syntactic abstractions as syntactically local (possibly denotational) transformations. For the comparison of languages we concentrate on fundamental concepts and ignore all syntactic extensions because they are trivially equivalent to core expressions. It immediately follows that a functional language with a control operator is more expressive than a (mathematically) functional language. Take as an example the **abort**-operation. When evaluated, the expression **abort**(v) is to terminate a program and must return the value v as the result of the entire program. Now suppose that some functional expression A_v is syntactically equivalent to **abort**(v). Since **abort**(v) is defined so is A_v, but we also know that A_v is equivalent to its value in any context. If A_v is equal to v, then a program of the form

$$F(\mathbf{abort}(1)) \text{ where } F(x) = 0 * x$$

cannot yield 1; similarly, if A_v is not equal to v, the expression

$$\mathbf{abort}(1)$$

is not equal to 1. Consequently, there is no syntactically local translation of **abort**-operations into functional expressions.

In an absolute but trivial sense this statement is counter-intuitive to the development of denotational semantics. The latter has shown that all sequential programming languages can be assigned mathematical meaning, and hence, can be expressed with a functional notation. Indeed, there are well-known techniques for reformulating imperative programs into functional ones, but the emphasis here is on *program.* These reformulations cannot be performed as syntactically local translations of one construct into an equivalent expression: *the entire program must be restructured.* The crucial idea for modeling exceptional flow of control and state variables with a functional program is to make all control and state information explicit. As a result, such a program contains many recurring programming patterns. In addition, the program is generally less modular than its imperative counterpart, since a local change in an imperative program—like the insertion of an **abort**-statement—may require a *global* restructuring of the program.

We illustrate the pattern-oriented programming style of functional versus imperative programs with an example for the (relatively local) emulation of a **catch**-expression. For the example we work in a programming language with a built-in data type *binary number tree.* Such a tree is either empty, which can be tested with the predicate empty?, or it is non-empty, in which case it consists of two sub-trees and a number. The parts of a non-empty tree can be accessed with lson, rson, and info. Provided recursive functions are available, it is straightforward to write a recursive program Σ^* that sums up the numbers in a tree. In an Algol-style language, this could be expressed as

integer procedure $\Sigma^*(T)$; **tree** T;
    if empty?(T) **then** 0
    else info(T) + Σ^*(lson(T)) + Σ^*(rson(T)).

Next we consider a slightly modified version of Σ^*, namely, a function Σ_0^* that walks through the tree and adds up the numbers, but immediately returns 0 upon encountering 0. With a Scheme-like **catch**-facility, this is only a minor extension of the above program:

```
integer procedure Σ*₀(T); tree T;
  catch Exit
    begin
      integer procedure S(T); tree T;
        if empty?(T) then 0 else
        if info(T) = 0 then Exit(0)
        else info(T) + S(lson(T)) + S(rson(T))
      S(T)
    end {Σ*₀}.
```

Informally, when S encounters 0, it invokes the continuation value *Exit* with 0 and thus jumps back to the caller of Σ_0^*, returning 0 as required.

A functional version of this program that preserves the one-parse-early-exit property is more complicated. It is based on the idea of a denotational continuation [1, 21, 44, 67].[5] That is, every function takes an additional parameter that represents the computation after the function invocation. If the function wants to continue the computation, this extra parameter must be invoked in such a position that after its return nothing is left to do. Also, the current function must pass along

[5] Indeed, the origin of the concept of a continuation can be traced back to A. van Wijngaarden who pointed out in a discussion at the IFIP Working Conference on Formal Language Description Languages, 1964 [59:24] that "this implementation [of procedures] is only so difficult because you have to take care of the goto statement." He went on to explain his newly designed implementation strategy:

> [N]o procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure return.

Or, put differently, "... it's exactly the same as a goto, only called in other words"[ibid].

a representation of the rest of its computation to invocations of other functions. When a computation is to be terminated, the extra parameter is simply ignored.[6] In a first-order Algol-style language, this program becomes:

```
integer procedure Σ*0(T); tree T;
  begin
    integer procedure Id(x); integer x; x;
    integer procedure S(T, C); tree T; integer procedure C;
      begin
        integer procedure Sl(suml); integer suml;
          begin
            integer procedure Sr(sumr); integer sumr;
              C(suml + sumr + info(T));
            S(rson(T), Sr)
          end {Sl};
        if empty?(T) then C(0) else
        if info(T) = 0 then 0
        else S(lson(T), Sl)
      end {S};
    S(T, Id)
  end {Σ*0}
```

Our complaint with this second version is not that it is longer. This could be improved with a better syntax and higher-order functions. The important drawback of this programming style is that it introduces recurring programming patterns, *e.g.*, every call to the auxiliary function S uniformly takes an extra parameter. A programming language, however, should not introduce patterns into programs, but it should hide them. Constructing highly repetitive programs is an error-prone task

[6] There are various other ways to achieve the correct effect, *e.g.*, with explicit stacks, but these can be derived from this solution [72].

and should be avoided with appropriate abstractions. This is the essence of the desire for abstract programming notation. To conclude this part of the discussion, we recall McCarthy's argument about the advantages of introducing conditional expressions into the theory of recursive functions:

> [B]oth the original Church-Kleene formalism and the formalism using the minimalization operation use integer calculations to control the flow of the calculations. That this can be done is noteworthy, but controlling the flow in this way is less natural than using conditional expressions which control the flow directly [45:64].

The same arguments hold for the elimination of assignments in favor of functional notation. This technique enforces an explicit passing around of the current state variables of a program [8, 30]. If a program models many objects with state, this structure—generally a state-tree—becomes large. Changing one component means finding the place of the current value in the state structure, putting in the new value, and re-constructing the state structure. This can be expensive, yet, more importantly this strategy induces a notational overhead. With an assignment operation, the first and the third part of this state transformation need not be programmed. Since the program text naturally organizes the state structure in the shape of a tree, these parts are automatically a part of the assignment statement. Again, functional programs that emulate state variables and state changes contain recurring program patterns.

In summary, the essence of expressiveness is a set of fundamental computational *abstractions*. These abstractions are chosen in order to avoid a repetitive programming style. Other facilities that abstract patterns of core expressions are explained via syntactic equivalences. Accordingly, they are called syntactic abstractions. Writing down a syntactic abstraction should be considered as a mere editorial task. For traditional languages, five fundamental facilities are sufficient: a set of names, means for functional abstraction and function application, and oper-

ations for the manipulation of control and state. The need for a naming capability is obvious. Functional abstraction binds expressions together via names and is the main tool for structuring programs. Function application is the means of employing functions: it relates names to values. A control operator has the task of manipulating the thread of control for exceptional cases. An assignment statement finally expresses the occurrence of a state-changing event. Equipped with this understanding of expressiveness, we can now proceed to our problem statement.

1.3. A Calculus for Imperative Higher-Order Programming Languages

The historic evolution and the analysis of expressive programming languages shows that they share with mathematical languages the tendency towards simplicity. The point of contention is the inclusion of two imperative operations for expressing events. As McCarthy remarks:

> A programming language should include both recursive function definitions and Algol-like statements. However, a theory of computation certainly must have techniques for proving algorithms equivalent, and so far it has seemed easier to develop proof techniques like recursion induction for recursive functions than for Algol-like programs [45:65].

The apparent reason for the problem is captured in Landin's complaint that

> [f]or both of these [jumps and assignments] the precise specification is in terms of the underlying abstract machine [38:159].

In other words, there is no method for understanding these operations solely on the basis of program text and program equivalences.

At the outset of this chapter, we briefly alluded to the importance of program equivalences. Both McCarthy and Landin clearly foresaw the possible need for and uses of calculi. The former anticipated transformational programming:

> [Equivalence preserving] transformations can be used to take an algorithm from a form in which it is easily seen to give the right

> answers to an equivalent form guaranteed to give the same answers but which has other advantages such as speed, economy of storage, or the incorporation of auxiliary processes [45:34];

the latter perceived a need to put available techniques on solid ground:

> The practicability of *all* kinds of program-processing (optimizing, checking satisfaction of given conditions, constructing a program satisfying given conditions) depends on there being elegant equivalence rules [38:160]. [7]

That is, if computer science wants a well-founded theory and practice of compilation and optimization or a sound understanding of the construction of programs, then an equivalence theory for expressive programming notations is the absolutely necessary starting point.

The central clue to the precise formulation of our problem statement comes from our characterization of an expressive language core: an imperative extension of a functional language. Given that the λ-calculus is the principal reasoning system for functional languages and that it is based on a small set of basic rules, we must accept Talcott's challenge, who asked in the conclusion of her dissertation:

> What rules should be added to the rules for ... lambda-calculus to obtain a *Rum* calculus with reasonable properties? [70:199],

where *Rum* is a functional language with a syntactic variant of **catch**-expressions.

Our own starting point is a version of the programming language Scheme. Scheme is well-explored and practical. Its current fundamental abstractions are almost orthogonal, that is, their (syntactic and semantic) tasks do not overlap; its syntax is expression-oriented, the only exception being the assignment operator. Scheme is thus an acceptable extension of the λ-calculus-language and a well-suited candidate for our project.

---

[7] Emphasis ours.

The real question then becomes: when do we know that we have the correct calculus? Or, more precisely, what does it mean for a calculus to correspond to a programming language? The first to investigate this question systematically was Plotkin [47]. He noticed that Landin's abstract machine for AE/ISWIM was not in accord with the λ-calculus. There are two problems with the relationship. First, the SECD-machine reduces programs to *values* as opposed to normal froms. Second, the SECD-machine may stop and yield a value when the calculus fails to produce a normal form and *vice versa*. One possible solution is to change the programming language so that it fits to the calculus, but since we are interested in studying programming languages, this is the wrong solution. Instead—and this is Plotkin's insight—we must look for the correct version of the calculus.

For Landin's AE/ISWIM-language the correct calculus is the λ-value-calculus. It differs from the original λ-calculus in the basic axiom and incorporates the notion of a value. Plotkin showed that the relationship between the programming language and the new calculus satisfied a set of correctness conditions. The same criteria are also valid for the correspondence between the original λ-calculus and a modified AE/ISWIM-language. The difference between the two programming languages is the argument evaluation strategy. Whereas Landin's original language evaluates the arguments to a function before the application, the modified version does not. The two strategies are known as call-by-value and call-by-name.

In Algol 60, both argument evaluation strategies are available. Most practical languages, however, realize (a form of) call-by-value over call-by-name.[8] Scheme is among those. There are two reasons for this choice. Since a function generally uses an argument more than once, it is more efficient to evaluate the argument expression

[8] While agreeing on *when* to evaluate an argument, they generally disagree on *what* to pass as an argument, *i.e.*, the value of the argument or a (machine) pointer to the value. We reject the idea of meshing machine concepts with abstract semantics and therefore only accept the first alternative.

only once for every function call. Furthermore, as Church already remarked in conjunction with his own λ-calculus, call-by-name is unnatural in some sense:

> Indeed if we regard these and only these formulas as meaningful which have a normal form, it becomes clearly unreasonable that FN should have a normal form and N have no normal form [10:59].

That is, with call-by-name

$$F(\frac{1}{0}) = 5 \text{ where } F(x) = 5$$

is valid, even though the argument to F is meaningless. The call-by-value strategy is in this sense preferable: it evaluates the argument once and this happens before the function application takes place.[9]

We have now explored all essential aspects of our problem statement. Put briefly, the task is:

— to analyze the correspondence between call-by-value functional languages and the λ_v-calculus,

— to extend the call-by-value based functional language with Scheme-like operations for the manipulation of program control and state, and

— to construct a conservative extension of the λ_v-calculus that reflects the extended programming language and that satisfies the correspondence criteria for languages and calculi.

1.4. Outline

The main body of the thesis has three parts. Chapters 2 and 3 form an introduction

[9] Church's way out of this dilemma was the λ-I-calculus. This calculus has a different term language in which functions must use their argument. This solves the problem by avoiding vacuously abstracted variables. Church's original paper also contains a proposal that is reminiscent of the λ_v-calculus. It is based on a restriction of the ground axiom such that "if M is a meaningful formula containing no free variables, the substitution of $(\lambda x.M)N$ for M ought not to be possible unless N is meaningful" [10:59]. Put differently, the λ-I-calculus solves the problem of vacuous abstraction with a syntactic, *i.e.* static, restriction, while the λ_v-calculus uses a semantic, *i.e.* dynamic, restriction.

to the general area of higher-order languages, calculi, and imperative extensions. Chapters 4 through 6 contain the main results. Chapter 7 summarizes the research and provides a perspective.

Following the above problem statement, we must first develop a formal setting for the analysis of functional programming languages and their relationship to the λ-calculus. This is the contents of Chapter 2. We formalize the concept of a family of programming languages and their semantics, the construction of a calculus, and their mutual correspondence. The descriptions draw on work by Barendregt, Reynolds, and Plotkin, but are presented in a single, unified framework. With a few examples, we illustrate (meta-) programming with functional languages.

In the third chapter, we extend the functional language of Chapter 2 with two imperative operations: $\mathcal{F}$-applications for the manipulation of control and σ-capabilities for the manipulation of state. The former is a semantic (and syntactic) improvement of Scheme's **catch**-expressions, the latter is an expression-oriented syntactic variant of assignment statements. Accordingly, we refer to the extended language as Idealized Scheme. The semantics is defined via an abstract machine, based on a denotational semantics. The chapter ends with a section on programming with Idealized Scheme.

Chapter 4 leads up to and states our first result. In a series of four steps, we transform the abstract machine into a program rewriting system. The new semantics is built on a set of six rules that rewrite a program to another program until a value is reached. The rules only apply to entire programs and are therefore not suited for a calculus. Nevertheless, they provide a symbolic evaluation strategy, which is a major improvement from the perspective of the programmer. At the same time, they are a good starting point for the design of a programming language calculus.

The fifth chapter contains the main result. First, we derive a set of term relations from the rewriting rules such that the context sensitivity of the latter is concentrated on a small subset of the former. From this set we construct the λ_v-CS-calculus. In the remainder of the chapter, we show that the calculus has the usual properties, *i.e.*, it is consistent and has standard sequences, and that it corresponds to Idealized Scheme. The correspondence relationship is more complex than the one for functional languages, but it still induces a method for the construction of program equivalence proofs.

In the sixth chapter we exploit the correspondence theorem and apply it to some programming examples. We prove the correctness of the **catch**-based version of Σ_0^*; an implementation of cells based on higher-order functions and assignments; a fast implementation of recursion in terms of self-referential assignments; the removal of tail-recursion in favor of jumps; and the characteristic idempotent behavior of generators on finite objects. This set of examples covers the two important fields that McCarthy and Landin predicted: program processing and program correctness. For example, the implementations of recursion as self-referential code and the elimination of tail-recursion by jumps are traditional compiler techniques. As far as we know, this is the first time that they are justified with simple, algebraic calculations. The Σ_0^*-example, on the other hand, is a typical case where an obviously correct, but slow program is equivalent to a faster, but less perspicuous program. All the proofs are simple and easy to construct. They demonstrate that reasoning with imperative programs has the same algebraic flavor as reasoning with functional programs. Although we explore a wide range of examples, this field needs more consideration.

The last chapter is devoted to a summary, a comparison to other work, and proposals for future research. Although we are not the first to construct an equational

theory for imperative languages, the design of a symbolic rewriting semantics and a calculus for the full core of traditional imperative languages are unique results. The major limitations of our approach is the concentration on type-free, traditional imperative languages. This is addressed in the section on future work where we suggest an investigation of a type structure and additional fundamental abstractions. There are also numerous other theoretical directions that promise fruitful extensions, *e.g.*, a direct axiomatization of syntactic abstractions, a systematic exploration of proof principles, and the incorporation of intensional calculi. On the practical side, we perceive two main application areas, namely, a visual, animated implementation of our symbolic rewriting semantics as a debugging and learning aid and new implementation strategies for imperative languages.

2. Programming Languages, Calculi, and Correspondence

The three basic concepts of our development are programming languages, their calculi, and the notion of correspondence between the two. The definition of a programming language consists of two parts: one for the syntax and one for the semantics. An associated calculus is approximately an equivalence relation over the same syntax, *i.e.*, it equates programs and program pieces with respect to some behavioral understanding.

Put differently, a programming language semantics and a calculus interpret a common syntax: the semantics is a map from programs to results, the calculus a set of equivalence classes. On the other hand, the semantics also defines a calculus and vice versa. The semantics-based calculus is called *operational equivalence*. Two program pieces are operationally equivalent if one can be replaced by the other in any arbitrary context without changing the result. A calculus-based semantics can be built upon the standardization procedure for derivations in a calculus. For every equation in a calculus, the standardization procedure determines a standard way of deriving this equation. It follows that the standardization procedure associates a *unique* value with a program if the calculus equates the program with any value at all. Thus, the standardization procedure defines a function from programs to values: *the standard reduction function.*

According to the preceding argument, the correspondence between a language semantics and a calculus has two aspects. First, the calculus perceived as a language semantics must equal the original semantics. Second, equivalence in the calculus must imply semantic equivalence. This second criteria accounts for our understanding that we accept the programming language as a given specification of a calculus, and that we actually investigate the correctness of a calculus with respect to this language semantics. We cannot expect the two equivalence relations to be equal because the semantic equivalence collectively assigns all diverging computations to the same equivalence class.

In the following sections we develop the necessary mathematical machinery for a formalization of these basic concepts. Because our work is an extension of the relationship between pure AE/ISWIM and the λ_v-calculus, we explore this correspondence as a prototypical example. The first section contains a specification of pure AE-syntax, *i.e.*, the λ-calculus-term language. In the second one we define an operational semantics in the form of an abstract machine. The third section is a primer on the construction of a calculus, in particular the λ_v-calculus. Section 4 pulls the first three sections together by examining the correspondence question. In the last section we illustrate programming and reasoning with the λ_v-calculus.

2.1. The Programming Language Λ

The term language of the λ-calculus, Λ, is defined inductively over the terminal symbols (,), . (dot), and λ; over an infinite set of variables, *Vars*; and over a set of constants, *Const*. The syntax is formalized in Definition 2.1. The language contains four classes of terms:

— constants: a, which are interpreted symbols;[1]

[1] We do not make a distinction between a *constant symbol* and the *constant* it denotes.

— variables: x, which are placeholders for values;

— λ-abstractions: $\lambda x.M$, which play the role of lexically-scoped, call-by-value functions; accordingly, M is called the function *body*, x is the function *parameter*;

— applications: MN, which serve as *the* computational vehicle, applying the *function part* M to the *argument part* N.

Constants, variables, and abstractions are collectively referred to as *values*.

Definition 2.1: The programming language Λ

Syntactic Domains:

a	$\in$ *Const*	(constants)
x	$\in$ *Vars*	(variables)
M, N	$\in \Lambda$	(Λ-terms)

Abstract Syntax:

$$M ::= a \mid x \mid (\lambda x.M) \mid (MN).$$

The union of constants, variables, and λ-abstractions is the set of (*syntactic*) *values*; we refer to it by *Values*.

Convention. $a, b, \ldots, f, g, \ldots$ are meta-variables for *Const*, but are also used as if they were members of *Const*, and similarly for $x, y, \ldots$, which range over *Vars*. $M, N, \ldots$ denote terms, $U, V, \ldots$ stand for values. **End**

The set of constants represents the primitive or built-in data types of a programming language. It is intentionally left unspecified in order to separate the logical design issues from the application-oriented ones, but we assume that the set is the disjoint union of basic constants, *BConst*, and functional constants, *FConst*. Thus, the language definition for Λ actually specifies a family of programming languages. A particular instantiation of the constant set yields a special-purpose language. In principle, every algebra with its carrier set and its operations is a suitable basis

for a set of constants. For example, if we take the integers with the successor and predecessor function constants, 1^+ and 1^-, we get a primitive language for numerical applications. Some terms in this language are $(\lambda x.1)$, a function that maps everything to 1, $(\lambda x.(1^+(1^+x)))$, a function that increases every integer by 2, and $((\lambda x.1)0)$, an application that combines the first sample function with 0. More useful constant sets include rational and complex numbers for advanced numerical applications, strings for language processing, *etc.* Constants play an important role in the comparison of language semantics and calculi.

We adopt the notational convention of using $\lambda xy.M$ for $(\lambda x.(\lambda y.M))$, LMN for $((LM)N)$, and other shorthands with an obvious meaning; similarly, we write about two-place functions or of the application of a function to two arguments. When we write $\lambda x.\lambda y.M$ we wish to emphasize that this expression is a *higher-order* function, *i.e.*, a function that maps every argument to a function. The usual terminology of (abstract) syntax applies to Λ. For example, the term $(\lambda x.x)z$ contains the *subterms* x and z, or z and $\lambda x.x$ *occur nested* in this term. We also refer to the *root* of a term, meaning the outermost syntactic construction, generally an application.

Two important syntactic notions are the set of free and the set of bound variables, $FV(M)$ and $BV(M)$, of a term M. An occurrence of a variable x is *free* if it is not a part of a $\lambda x.M$-abstraction; otherwise, the occurrence is *bound*. For example, in $(\lambda x.x)xx$ there are two free and two bound occurrences of the variable x. We define the two sets by induction on the structure of a term:

$$\begin{aligned} FV(a) &= \emptyset, & BV(a) &= \emptyset, \\ FV(x) &= \{x\}, & BV(x) &= \emptyset, \\ FV(\lambda x.M) &= FV(M) \setminus \{x\}, & BV(\lambda x.M) &= BV(M) \cup \{x\}, \\ FV(MN) &= FV(M) \cup FV(N); & BV(MN) &= BV(M) \cup BV(N). \end{aligned}$$

Terms with no free variables are *closed terms* and play the role of *programs* in our

language. *Open* terms are sometimes referred to as program pieces.

Bound variables often cause confusion in dealing with substitution. To avoid this, we adopt two of Barendregt's conventions [5:26]:

- Terms that are equal except for some unique renaming of bound variables are identified, *e.g.*, $\lambda x.x \equiv \lambda y.y$, but $\lambda y.zy \equiv \lambda x.zx \not\equiv \lambda z.zz$ [*α-congruence convention*];
- In all mathematical definitions, theorems, *etc.* the sets of bound and free variables of all Λ-terms are assumed to be mutually disjoint [*hygiene convention*].

The first convention reflects the fact that the actual name of a parameter is irrelevant for the functionality of an abstraction. Together, the two conventions permit a naïve treatment of terms and term-substitutions. The definition of the *substitution operation* $M[x := N]$ becomes straightforward. Informally, $M[x := N]$ denotes the term that is like M but with all free occurrences of x replaced by N; formally, we define:

$$a[x := N] \equiv a,$$

$$x[x := N] \equiv N, \quad y[x := N] \equiv y \qquad (x \not\equiv y),$$

$$(\lambda y.M)[x := N] \equiv (\lambda y.M[x := N]),$$

$$(LM)[x := N] \equiv (L[x := N]M[x := N]).$$

For an illustration of the above conventions, suppose we wish to substitute x in $M \equiv \lambda y.xy$ by $N \equiv \lambda z.y$. First, this is illegal because it violates the hygiene convention: M's bound variable is not distinct from N's free one. If we want to apply the substitution algorithm, we must rename the bound variable of M, *e.g.*, $M \equiv \lambda u.xu$. Assuming this, the result of $M[x := N]$ is $\lambda u.(\lambda z.y)u$.

An important fact about substitution is captured in the

Substitution Lemma [5:27]. *If $x \not\equiv y$ and $x \notin FV(L)$, then*

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Several proofs in the main body of the work use this fact.

The last major syntactic notion that we introduce in this section is the concept of a *one-hole context*, which is a term with a hole. We use [ ] to indicate a hole and $C[\ ], \ldots$ as meta-variables ranging over the set of contexts, indicating that a context is a term which is a function of its hole. The formal definition is given by an abstract syntax:

$$C[\ ] ::= [\ ] \mid (\lambda x.C[\ ]) \mid (C[\ ]M) \mid (MC[\ ]).$$

We use the notation $C[M]$ to denote a term that is like the context $C[\ ]$, but with M put into the hole. Unlike terms, contexts do not need to satisfy the above conventions about terms. The filling of a context $C[\ ]$ with a term binds free variables in M. That is, we do not assume that bound variables in $C[\ ]$ are distinct from the free variables in M, *e.g.*, take $M \equiv x$ and $C[\ ] \equiv \lambda x.[\ ]$ which yields $C[M] \equiv \lambda x.x$. Later, contexts will also appear in a more specialized version, namely, as evaluation contexts.

2.2. An Abstract Machine Semantics

The first formal semantics of AE/ISWIM as a programming language was Landin's SECD-machine [7, 39, 36, 40, 47]. The machine is a state transition system and thus provides an abstract operational semantics.[2] The operational character facilitates the comparison with a calculus since a calculus also operationally reduces terms to values. A major disadvantage is that the SECD-machine is hard to compare with a corresponding denotational semantics, the predominant method for formal semantic specifications [54, 66]. To combine the best of both worlds, we follow Reynolds and others [31, 50, 74] and specify the language semantics with an operational version

[2] Considering the results in the following sections, the equational theories also define an operational programming language semantics, but of course a kind that is only remotely related to state-of-the-art machines.

that is derived from a denotational semantics. This should help to clarify the definitions and later it ties in neatly with the design and proof of correctness of our calculus.

The abstract machine for Λ is a classical state transition system. Some states are designated as initial, others as terminal. For the evaluation of a program, the machine's current state is set to some appropriate initial state. Given a current state, a state transition function determines the next current state. When the current state becomes a terminal state, the evaluation stops. Unloading this final state yields the program result.

Since our language definition actually specifies a family of languages, parameterized over constant sets, the semantics depends on an interpretation of constant applications. We assume that the specific sets of constants come equipped with an interpretation function

$$\delta\colon FConst \times BConst \multimap ClosedValues,$$

where *ClosedValues* is the set of closed values. Consider the set of natural numbers with the function constants $+$, 1^+, ... A well-suited definition of δ for this set is

$$\delta(+, n) = n^+ \text{ and } \delta(n^+, m) = n + m.$$

The function $+$ acts like a one-place function and depends on the existence of the functions n^+. Although this treatment is cumbersome, it is sufficient for our investigations.

The formalization of the abstract machine requires the specification of a state space and a state transition function. A *machine state* is a triple of a control string, an environment, and a continuation code. Accordingly, the machine is called CEK-machine.

Definition 2.2: The CEK-machine, part I: the computational domains

Computational Domains:

$$
\begin{array}{lll}
s \in \mathit{States} & = \mathit{Controls} \times \mathit{Envs} \times \mathit{Conts} & \text{(machine states)} \\
c \in \mathit{Controls} & = \Lambda + \ddagger & \text{(control strings)} \\
\rho \in \mathit{Envs} & = \mathit{Vars} \multimap \mathit{Closures} & \text{(environments)} \\
V \in \mathit{Closures} & = \mathit{Values} \times \mathit{Envs} & \text{(closures)} \\
\kappa \in \mathit{Conts} & = \mathbf{ret}\text{-}\mathit{Conts} + \mathbf{p}\text{-}\mathit{Conts} & \text{(continuation codes)}
\end{array}
$$

where **p**-continuations and **ret**-continuations are

$$\mathbf{p}\text{-}\mathit{Conts} = \mathbf{stop} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{fun} \times \mathit{Closures} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{arg} \times \Lambda \times \mathit{Envs}$$
$$\mathbf{ret}\text{-}\mathit{Conts} = \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{ret} \times \mathit{Closures}$$

A *control string* is either a Λ-term or the unique symbol $\ddagger$. If the current control string is a proper term, it determines the next transition step; otherwise, the continuation code is the deciding criteria. In denotational semantics, this latter situation corresponds to the application of a continuation function to a value. We call a transition sequence starting in $\langle M, \bullet, \bullet \rangle$ an evaluation of M. An *environment* is a finite map from variables to closures. The machine uses environments to store the meaning of free variables in the control string. A CEK-machine *closure* is an ordered pair of a constant and the empty environment—a constant-closure—or of an abstraction and an environment whose domain covers the free variables in the abstraction—a λ-closure. Closures are semantic equivalents of values. All these concepts are formalized in Definition 2.2, part I.

The definition of a continuation code is more complex. A *continuation code* remembers the remainder of the computation after the current control string is evaluated. The domain of continuations consists of two subdomains: **p**- and **ret**-

Definition 2.2: The CEK-machine, part II: the transition function

The CEK-transition function maps states to states:

$$States \overset{CEK}{\longmapsto} States,$$

according to the following cases:

$$\langle a, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\ \mathbf{ret}\ \langle a, \emptyset\rangle)\rangle \tag{0}$$

$$\langle x, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\ \mathbf{ret}\ \rho(x))\rangle \tag{1}$$

$$\langle \lambda x.M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\ \mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle \tag{2}$$

$$\langle MN, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle M, \rho, (\kappa\ \mathbf{arg}\ N\ \rho)\rangle \tag{3}$$

$$\langle \ddagger, \emptyset, ((\kappa\ \mathbf{arg}\ N\ \rho)\ \mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle N, \rho, (\kappa\ \mathbf{fun}\ V)\rangle \tag{4}$$

$$\langle \ddagger, \emptyset, ((\kappa\ \mathbf{fun}\ \langle \lambda x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle M, \rho[x := V], \kappa\rangle \tag{5}$$

$$\langle \ddagger, \emptyset, ((\kappa\ \mathbf{fun}\ \langle f, \emptyset\rangle)\ \mathbf{ret}\ \langle a, \emptyset\rangle)\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\ \mathbf{ret}\ \langle \delta(f, a), \emptyset\rangle)\rangle \tag{6}$$

continuations. A **ret**-continuation consists of a **p**-continuation code κ and a semantic value V. It results from an evaluation that started in $\langle M, \rho, \kappa\rangle$, and we therefore say M evaluates to V. The value is supplied to the **p**-continuation so that it can finish whatever is left to do. **p**-Continuations are defined inductively—see Definition 2.2—and have the following intuitive function with respect to an evaluation:

— **(stop)** stands for the initial continuation, specifying that nothing is left to do;

— $(\kappa\ \mathbf{arg}\ N\rho)$ indicates that N is the argument part of an application, that ρ is the environment of the application, and that κ is its continuation;

— $(\kappa\ \mathbf{fun}\ V)$ represents the case where the evaluation of a function part yielded V as a value, and κ is the continuation of the application.

A CEK-machine state is either a triple of the form $\langle \ddagger, \emptyset, \kappa\rangle$ where κ is a **ret**-continuation, or a triple of the form $\langle M, \rho, \kappa\rangle$ where M is a Λ-term, ρ is an environ-

ment that provides a meaning for all free variables in M, and κ is a **p**-continuation. A machine state of the form $\langle M, \emptyset, (\mathbf{stop})\rangle$ is *initial*; $\langle \ddagger, \emptyset, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$ is *terminal*. We sometimes refer to the state components as registers, thinking of every transition as an assignment to the three registers.

The state transition function is displayed in Definition 2.2, part II. The first three transition rules evaluate syntactic values to semantic values in a single step. Indeed, it is because of these single-step evaluation rules that syntactic values are called values: once a value is encountered, it is immediately clear what the result of this term is and no further evaluation of subterms is necessary. The (semantic) value of variables is looked up in the environment; constants and abstractions are combined with the empty and current environment, respectively, to yield closures. Since the occurrence of a λ-abstraction corresponds to the definition of a function, the λ-closure is the result of a definition. The inclusion of the definition-time environment in the closure is necessary for the lexical scoping of its free variables. Otherwise, occurrences of free variables in the abstraction body could not refer to the value they had at definition-time. Rules (CEK3) and (CEK4) cause the machine to evaluate the two parts of an application to values; the function part is hereby considered first. The last two rules perform the actual application of a function value to an argument value. If the first is a λ-closure, the machine continues with an evaluation of the abstraction body in an extended closure-environment that maps the parameter to the argument value. Installing the closure-environment ensures the required lexical scoping of function definitions. If both parts are constant-closures, the first a function, the second a basic constant, the machine uses the δ-function to determine the result. The machine is *stuck* if none of the two application rules matches or if δ is undefined on the given constants.

From the description of the machine it follows that the machine may stop in

the terminal state, it may stop in a *stuck*-state, or it may run forever. When it terminates in $\langle \ddagger, \emptyset, ((\textbf{stop})\ \textbf{ret}\ V)\rangle$, we say that the program M yields the closure V. For a programmer, however, a semantic value makes little sense. The result must be translated back into the comprehensible world of syntax. In case V is a constant-closure, this is simple: the first part directly corresponds to a syntactic value. Otherwise, V is a λ-closure and naturally we would then like to see the abstraction part of such a closure since it determines the functionality. But in order to understand the abstraction completely, we must also say what its free variables mean. This, in turn, can be done by extracting the corresponding values from the environment part. Of course, values in the environment are closures and therefore, this translation is recursive. We call this procedure Unload because it constitutes the interface between the machine and the user and define it as a map from closures to syntactic values:

$$\mathsf{Unload}(\langle V, \rho\rangle) \equiv V[x_1 := \mathsf{Unload}(\rho(x_1))] \ldots [x_n := \mathsf{Unload}(\rho(x_n))]$$
$$\text{where } FV(V) = \{x_1, \ldots, x_n\}.$$

This definition is well-founded due to the finiteness of terms and environments.

With the Unload-function we can formalize an evaluation function that hides the machine details and maps programs to values:

$$eval_{CEK}(M) = \mathsf{Unload}(V) \text{ iff } \langle M, \emptyset, (\textbf{stop})\rangle \overset{CEK^+}{\longmapsto} \langle \ddagger, \emptyset, ((\textbf{stop})\ \textbf{ret}\ V)\rangle.$$

Another view of this definition is that the $eval_{CEK}$-function is the *extensional* semantics of Λ whereas the CEK-transition function is the *intensional* semantics. In other words, $eval_{CEK}$ defines *which* value is the result of a program, and the transition function says *how* this result is computed.

The distinction between extensional and intensional semantics is important for practical purposes. From the extensional point of view, which is that of a programmer, the programming language Λ is an entirely sequential language. Events in an

evaluation are ordered and thus, for example, no function can compute the mathematical (symmetric) or-function.[3] However, the sequentiality of our intensional semantics is not inherent. In Section 2.4 we shall discuss an alternative intensional framework that is extensionally equivalent, but gives rise to parallel evaluations.

A different issue is the nature of the extensional results. The $eval_{CEK}$-function is partial; if it diverges, we say the program under consideration is undefined. If the function returns a value, the value is either a basic constant or a function. In general, basic constants are the final answers which a programmer expects from a computation. They can be effectively compared and the correctness of the computation can be decided. On the other hand, if a function is the result, we must assume that this is an intermediate computation step, and that the final answer is eventually found by further applications of this function. Since there is no effective procedure for deciding on the equality of functions, the display of a λ-abstraction or a primitive function as a result is merely an attempt to provide some information on the progress of the computation.

At this point some examples are appropriate for clarifying the formal definitions. Assume that Λ is defined over the set of natural numbers with the function 1^+ and consider the following three programs:

$$(\lambda x.1^+(1^+x))0, \quad 01^+, \text{ and } \quad (\lambda x.1)((\lambda x.xx)(\lambda x.xx)).$$

The first program should yield 2. This can be verified by tracing the machine steps:[4]

$$\langle(\lambda x.1^+(1^+x))0, \emptyset, \mathbf{(stop)}\rangle$$

[3] A proof of this statement goes beyond the scope of this introduction to the λ-calculus. The reader is referred to either Berry's original work [6] or Barendregt's reformulation [5:375–382].

[4] In evaluation traces we let constants stand for constant-closures.

$$
\begin{aligned}
&\overset{CEK}{\longmapsto} \langle (\lambda x.1^+(1^+x)), \emptyset, ((\textbf{stop})\ \textbf{arg}\ 0\ \emptyset)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{arg}\ 0\ \emptyset)\ \textbf{ret}\ \langle (\lambda x.1^+(1^+x)), \emptyset\rangle)\rangle \\
&\overset{CEK}{\longmapsto} \langle 0, \emptyset, ((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1^+(1^+x)), \emptyset\rangle)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1^+(1^+x)), \emptyset\rangle)\ \textbf{ret}\ 0)\rangle \\
&\overset{CEK}{\longmapsto} \langle 1^+(1^+x), \{(x,0)\}, (\textbf{stop})\rangle \\
&\overset{CEK}{\longmapsto} \langle 1^+, \{(x,0)\}, ((\textbf{stop})\ \textbf{arg}\ (1^+x)\ \{(x,0)\})\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{arg}\ (1^+x)\ \{(x,0)\})\ \textbf{ret}\ 1^+)\rangle \\
&\overset{CEK}{\longmapsto} \langle (1^+x), \{(x,0)\}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle \\
&\overset{CEK}{\longmapsto} \langle 1^+, \{(x,0)\}, (((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{arg}\ x\ \{(x,0)\})\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, ((((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{arg}\ x\ \{(x,0)\})\ \textbf{ret}\ 1^+)\rangle \\
&\overset{CEK}{\longmapsto} \langle x, \{(x,0)\}, (((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{fun}\ 1^+)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, ((((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{fun}\ 1^+)\ \textbf{ret}\ 0)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{ret}\ 1)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, ((\textbf{stop})\ \textbf{ret}\ 2)\rangle.
\end{aligned}
$$

The second program does not return a value because the machine gets stuck:

$$
\begin{aligned}
\langle 01^+, \emptyset, (\textbf{stop})\rangle &\overset{CEK}{\longmapsto} \langle 0, \emptyset, ((\textbf{stop})\ \textbf{arg}\ 1^+\ \emptyset)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{arg}\ 1^+\ \emptyset)\ \textbf{ret}\ 0)\rangle \\
&\overset{CEK}{\longmapsto} \langle 1^+, \emptyset, ((\textbf{stop})\ \textbf{fun}\ 0)\rangle \\
&\overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (((\textbf{stop})\ \textbf{fun}\ 0)\ \textbf{ret}\ 1^+)\rangle.
\end{aligned}
$$

This last clause is not matched by any of the cases in the definition of the CEK-transition function. The third program causes the machine to run forever:

$$
\begin{aligned}
&\langle (\lambda x.1)((\lambda x.xx)(\lambda x.xx)), \emptyset, (\textbf{stop})\rangle \\
&\overset{CEK^+}{\longmapsto} \langle ((\lambda x.xx)(\lambda x.xx)), \emptyset, ((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1), \emptyset\rangle)\rangle \\
&\overset{CEK^+}{\longmapsto} \langle \ddagger, \emptyset, ((((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1), \emptyset\rangle)\ \textbf{fun}\ \langle (\lambda x.xx), \emptyset\rangle)\ \textbf{ret}\ \langle (\lambda x.xx), \emptyset\rangle)\rangle \\
&\overset{CEK^+}{\longmapsto} \langle xx, \{(x, \langle (\lambda x.xx), \emptyset\rangle)\}, ((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1), \emptyset\rangle)\rangle \\
&\overset{CEK^+}{\longmapsto} \langle \ddagger, \emptyset, ((((\textbf{stop})\ \textbf{fun}\ \langle (\lambda x.1), \emptyset\rangle)\ \textbf{fun}\ \langle (\lambda x.xx), \emptyset\rangle)\ \textbf{ret}\ \langle (\lambda x.xx), \emptyset\rangle)\rangle \\
&\overset{CEK^+}{\longmapsto} \ldots
\end{aligned}
$$

This transition sequence proves that the machine returns to the same state after a few steps and hence goes into an infinite loop. Beyond this, the trace exemplifies that a functional simulation of a loop can be implemented as efficiently as a hand-compiled construct.[5] The CEK-machine acts like a true register machine, and, in particular, it implements tail-calls in a goto-like fashion.

To the same degree as the abstract semantics is suited for realistic implementations, it is unfit for human consumption. The above examples illustrate how hard it is to reason about programs with the CEK-machine. The need for a program-based, human-oriented reasoning system is obvious.

2.3. The λ-value-Calculus

A calculus is an equational theory over a term language. There are two equivalent ways to construct a calculus: as a logic-like system with axioms and inference rules or as a term relation built from a set of term reductions. Since reducing a program to a value is closer to the computational understanding of a program evaluation than axiomatically proving its equivalence to a value, we develop the mathematical background for the λ-value-calculus in Barendregt's [5:ch 3] framework of reduction and congruence relations.

A *notion of reduction* is a relation between terms. For example, the β-value relation is

$$\beta_v = \{(((\lambda x.M)N), M[x := N]) \mid M, N \in \Lambda, N \text{ is a value}\}.$$

A more conventional notation is

$$\beta_v : ((\lambda x.M)N) \longrightarrow M[x := N] \text{ provided that } N \text{ is a value.}$$

[5] This is not true for implementations based on the SECD-machine. The very same program causes a constant growth of the machine state, and on a finite computer this sooner or later exhausts the available machine space and (abnormally) terminates this infinite loop.

The δ-function on Λ-constants provides another notion of reduction:

$$\delta : fa \longrightarrow V \text{ provided that } \delta(f, a) = V.$$

If M is related to N via a notion of reduction $\mathbf{R}$, then M is an $\mathbf{R}$-*redex*, N is a *contractum*, and reducing M to N is called $\mathbf{R}$-*contraction* or $\mathbf{R}$-*step*.

Notions of reduction roughly correspond to basic computation steps. For the λ_v-calculus, no notions of reduction are needed for values, *i.e.*, constants, variables, and abstractions: they are already results. On the other hand, the β_v-relation explains how to understand the application of a λ-function to a value, δ does the same for built-in primitives. Since this covers the entire set of syntactic constructors, these two relations are in some sense sufficient for defining computations in Λ. However, these notions of reduction only apply to the actual applications, they do not relate terms in which such applications occur as subterms, *e.g.*, $(\lambda x.(\lambda xy.xy)1^+0)$ or $1^+(1^+0)$. In order to provide computations for these cases, we introduce the concept of compatible term relations.

A relation is compatible with syntactic constructions if a relationship between two terms implies that the relation also holds between all terms that contain the original pair. With the notion of a context this can be expressed more succinctly: a relation $\mathbf{R}$ is *compatible* if $(M, N) \in \mathbf{R}$ implies $(C[M], C[N]) \in \mathbf{R}$ for all M, N, and contexts $C[\ ]$. Since Λ is defined inductively, every notion of reduction has a *compatible closure*, that is, there is a smallest relation that contains the notion of reduction and is compatible. Given $\mathbf{R}$, $\longrightarrow_R$ is its compatible closure and it is defined by

$$(M, N) \in \mathbf{R} \Rightarrow M \longrightarrow_R N$$

$$M \longrightarrow_R N \Rightarrow \lambda x.M \longrightarrow_R \lambda x.N$$

$$M \longrightarrow_R N \Rightarrow LM \longrightarrow_R LN \text{ and } ML \longrightarrow_R NL.$$

The compatible closure of $\mathbf{R}$ is also called *one-step* $\mathbf{R}$-*reduction*. The one-step

Definition 2.3: The λ_v-calculus

The basic notion of reduction is $\mathbf{v} = \beta_v \cup \delta$, where

$$\beta_v : ((\lambda x.M)N) \longrightarrow M[x := N] \text{ provided that } N \text{ is a value, and}$$
$$\delta : fa \longrightarrow V \text{ provided that } \delta(f, a) = V.$$

The *one-step* $\mathbf{v}$*-reduction* $\longrightarrow_v$ is the compatible closure of $\mathbf{v}$:

$$\begin{aligned} (M, N) \in \mathbf{v} &\Rightarrow M \longrightarrow_v N; \\ M \longrightarrow_v N &\Rightarrow \lambda x.M \longrightarrow_v \lambda x.N; \\ M \longrightarrow_v N &\Rightarrow LM \longrightarrow_v LN, ML \longrightarrow_v NL. \end{aligned}$$

The $\mathbf{v}$*-reduction* is denoted by $\longrightarrow\!\!\!\!\rightarrow_v$ and is the reflexive, transitive closure of $\longrightarrow_v$. We denote the smallest equivalence relation generated by $\longrightarrow_v$ with $=_v$ and call it $\mathbf{v}$*-equality*:

$$\begin{aligned} M &=_v M \\ M \longrightarrow_v N &\Rightarrow M =_v N \\ M =_v N &\Rightarrow N =_v M \\ L =_v M, M =_v N &\Rightarrow L =_v N. \end{aligned}$$

Formally, the λ_v-calculus is the congruence relation $=_v$; informally, we also refer to the entire system of relations as λ_v-calculus.

β_v-reduction, for example, relates $(\lambda x.(\lambda xy.xy)1^+0)$ to $(\lambda x.(\lambda y.1^+y)0)$ and furthermore, $(\lambda x.(\lambda y.1^+y)0)$ to $(\lambda x.(1^+0)$. To allow for a direct connection of terms that are related via several single steps, we define $\mathbf{R}$*-reductions*, $\longrightarrow\!\!\!\!\rightarrow_R$, as the reflexive, transitive closures of one-step reductions. Finally, an $\mathbf{R}$*-equality* or $\mathbf{R}$*-congruence*, $=_R$, is the equivalence closure over the one-step reduction. It is customary to write $\lambda\mathbf{R} \vdash M = N$ for $M =_R N$ when the calculus is perceived as an axiom system.

The general development of a calculus is instantiated for the λ_v-calculus in Definition 2.3. The basic notions of reduction are the β_v- and the δ-relation. The

λ_v-calculus is the congruence relation $=_v$.

Two central notions in the study of calculi are normal forms and values. A term is in *normal form* if it contains no redexes. We say a term M *has a normal form* N if M equals N and N is in normal form; the process of going from M to N is called normalization. Similarly, the terminology M *has a value* N means that M equals N and N is a value; going from M to N is an evaluation [in the calculus].

Given the claim that M has a normal form or a value N, the question arises of how to prove or disprove the equivalence between M and N. This directly leads to the more general question whether the calculus does not prove too much, *i.e.*, $M = N$ for all M and N, or, in technical terms, whether the calculus is inconsistent.

Answering the consistency question for a calculus is equivalent to showing that there are distinct normal forms. This, in turn, is true if the *diamond property* holds for the system. A term relation $\longrightarrow$ satisfies the diamond property if for all L, M, and N such that

$$L \longrightarrow M \text{ and } L \longrightarrow N$$

there exists a K such that

$$M \longrightarrow K \text{ and } N \longrightarrow K,$$

i.e., two reductions that start from the same term are confluent. A notion of reduction is *Church-Rosser* if the corresponding reduction, *i.e.*, the reflexive, transitive, and compatible closure, satisfies the diamond property. Given that a notion of reduction is Church-Rosser, we can prove [5:54][6]

Theorem 2.4. *Let* **R** *be a notion of reduction that is Church-Rosser. Then*

(i) $M =_R N$ *implies that there exists an* L *such that* $M \longrightarrow\!\!\!\!\!\rightarrow_R L$ *and* $N \longrightarrow\!\!\!\!\!\rightarrow_R L$;

[6] Unless indicated otherwise, the proofs of the theorems in this chapter can be found in the associated references.

(ii) If M has an **R***-normal form N, then $M \longrightarrow\!\!\!\!\!\rightarrow_R N$.*

The proof of the Church-Rosser property for the original λ-calculus exists in many different variations [52]. The shortest one was developed by Tait and Martin-Löf [5]. The same proof technique also yields a Church-Rosser Theorem for the λ_v-calculus [47]:

Theorem 2.5 (Church-Rosser). *The reduction* $\mathbf{v} = \beta_v \cup \delta$ *is Church-Rosser.*

The theorem implies an appropriate version of Theorem 2.4 and [47]

Corollary 2.6. *If an application M has a value V, then $M \longrightarrow\!\!\!\!\!\rightarrow_v V$.*

With the Church-Rosser Theorem we can now illustrate how the λ_v-calculus facilitates reasoning about Λ-programs. Let us return to the three examples of the preceding section. The first program was $(\lambda x.1^+(1^+x))0$. It reduces to a value and a normal form in three steps:

$$(\lambda x.1^+(1^+x))0 \longrightarrow_v 1^+(1^+0) \longrightarrow_v 1^+1 \longrightarrow_v 2.$$

Our second example was 01^+. This application is already in normal-form, hence, it cannot be further reduced, and therefore, it does not have a value. The third program finally was $(\lambda x.1)\underline{((\lambda x.xx)(\lambda x.xx))}$. Its only redex is the underlined part and a contraction of this redex leaves the term unchanged:

$$(\lambda x.1)((\lambda x.xx)(\lambda x.xx)) \longrightarrow_v (\lambda x.1)((\lambda x.xx)(\lambda x.xx)) \longrightarrow_v \ldots$$

Since the term is not in normal form, by Theorem 2.4 it does not have a normal form; since the application contains a redex in the argument part that never disappears, it does not have a value either.

For the above sample programs an evaluation in the calculus yields the same result as an evaluation on the machine. That this need not be the case is demon-

strated by $(\lambda xy.(\lambda z.z)x)1$, which reduces to two distinct, but equivalent, values:

$$(\lambda xy.(\lambda z.z)x)1 \longrightarrow_v \lambda y.(\lambda z.z)1$$

and

$$(\lambda xy.(\lambda z.z)x)1 \longrightarrow_v (\lambda xy.x)1 \longrightarrow_v \lambda y.1.$$

The CEK-machine evaluation yields the first value. Hence, we must ask whether there is a way to evaluate a program in the calculus such that we find the "right" value, that is, the machine result. This problem, together with the connection between the machine and the calculus in general, is the topic of the next section.

2.4. The Correspondence of Programming Languages and Calculi

In the two preceding sections we have developed an operational semantics and a calculus for Λ. Thus far, we have kept the two perspectives separate. To investigate the mutual relationship, we construct a programming language semantics from the calculus and compare it with the original CEK-machine semantics. Similarly, we define a congruence relation for program pieces based on the CEK-machine and study its connection to the λ_v-calculus.

An evaluation in the λ_v-calculus is a reduction of a program to a value. As demonstrated at the end of the preceding section, the reduction of a program can yield many different values. With respect to the CEK-machine, only one of these values is correct. Thus we must find an algorithm which always reduces a program to the correct value. Put into a broader context, the problem generalizes to the following: given that M reduces to N, is it possible to construct a standardized sequence of reduction steps from M to N? For the traditional λ-calculus this standardization question was raised and solved by Curry and Feys [13, 5]. Plotkin [47] showed that the same theorem holds for the λ_v-calculus and that standardized reduction sequences reduce a program to the machine value.

Definition 2.7: Standard reduction function and sequences

The *standard reduction function*, denoted by $\longmapsto_{sv}$, is defined by:

$$(M, N) \in \mathbf{v} \Rightarrow M \longmapsto_{sv} N;$$
$$M \longmapsto_{sv} M' \Rightarrow MN \longmapsto_{sv} M'N;$$
$$M \text{ is a value, } N \longmapsto_{sv} N' \Rightarrow MN \longmapsto_{sv} MN'.$$

Standard reduction sequences, abbreviated SRS, are defined by:

1. all constants and non-assignable variables are SRS-s;
2. if $M_1, \ldots, M_m$ and $N_1, \ldots, N_n$ are SRS-s, then

$$\lambda x.M_1, \ldots, \lambda x.M_m \text{ and } M_1N_1, \ldots, M_mN_1, \ldots, M_mN_n$$

   are SRS-s;
3. if $M \longmapsto_{sv} M_1$ and $M_1, \ldots, M_m$, then $M, M_1, \ldots, M_m$ is an SRS.

Historically, a standard reduction sequence is defined in terms of positions and residuals of redexes. Plotkin's proof is more elegant. Following his proof strategy, we first define a *standard reduction function*, which maps a term M to a term N by reducing the leftmost-outermost redex not inside a λ-abstraction. The standard reduction function is undefined on values. A *standard reduction sequence* combines a series of terms. It is constructed by applying the standard reduction function to some subterm of a given term and by appending standard reduction sequences with a common beginning and end. The reduced subterm need not contain the leftmost-outermost redex, but once a leftmost-outermost redex is not reduced, it must remain unreduced for the rest of the standard reduction sequence. In short, a standard reduction sequence is approximately a series of terms that are related via *almost*-leftmost-outermost reductions. The two concepts are formalized in Definition 2.7.

An alternative characterization of the standard reduction function is based on

the notion of an *evaluation context.* Such an evaluation context is a context with exactly one hole for which the path from the root to the hole leads through applications only and the terms to the immediate left of the path are values. Letting $C[\ ]$ range over evaluation contexts, we define the set of evaluation contexts by

$$C[\ ] ::= [\ ] \mid C[\ ]M \mid VC[\ ].$$

Since these contexts never contain a hole inside of abstractions, putting an expression into the hole cannot bind free variables. We can now state and prove

Proposition 2.8. *$M \longmapsto_{sv} N$ iff there exists an evaluation context $C[\ ]$ such that $M \equiv C[P]$, $N \equiv C[Q]$, and $(P, Q) \in \mathbf{v}$.*

Proof. Straightforward induction on the standard reduction step. □

This characterization of standard reduction functions will be helpful for the design of the λ_v-CS-calculus.

The importance of standard reduction sequences is captured in the following two theorems. The first says that if there is a reduction from M to N, then there must be a standard reduction sequence [47]:

Theorem 2.9 (Standardization). *$M \longrightarrow\!\!\!\!\!\rightarrow_v N$ iff there exists an* SRS *$L_1, \ldots, L_n$ such that $M \equiv L_1$ and $L_n \equiv N$.*

The theorem determines a semi-decision procedure for finding normal-forms and values. Indeed, the first value in a standard reduction sequence of a program is the correct value with respect to a machine evaluation, *i.e.*, the standard reduction function simulates $eval_{CEK}$ [47]:

Theorem 2.10 (Simulation). *$eval_{CEK}(M) = V$ iff $M \longmapsto_{sv}^* V$.*

This theorem justifies the use of our terminology "evaluation in the calculus" for the reduction of a program to a value, and it motivates the restriction of this idiom

to standard reductions from programs to values. It is essential for the symbolic evaluation of programs by programmers.

The calculus viewpoint is also potentially beneficial for a machine implementation of the language semantics. Whereas the intensional formulation of $eval_{CEK}$ on the basis of the CEK-transition function is sequential, a reformulation according to the standard reduction function reveals opportunities for parallel evaluations:

$$eval_{CEK}(V) = V$$

$$eval_{CEK}(MN) = \begin{cases} \delta(f,a) & \text{if } eval_{CEK}(M) = f,\ eval_{CEK}(N) = a \\ eval_{CEK}(P[x := Q]) & \text{if } eval_{CEK}(M) \equiv \lambda x.P \\ & \text{and } eval_{CEK}(N) \equiv Q. \end{cases}$$

In other words, evaluating a value is immediate; evaluating an application depends on evaluating the two parts and the transition step. Therefore, the evaluation of the two components of applications can proceed in parallel.

A different way of comparing a machine semantics with a calculus-based language semantics was developed by J.H. Morris [46] and adapted for the λ_v-calculus by Plotkin [47]. Mathematically, a *program* maps inputs to outputs and is extensionally equivalent to a function; the intention behind a program is neglected. The desire to compare these functionalities for different frameworks requires a restriction to effectively comparable values.[7] For the traditional λ-calculus, these are normal forms; for the λ_v-calculus, we pick basic constants. Furthermore, without evaluating the application of a Λ-expression to basic constants, it is impossible to determine how many arguments a particular expression consumes until it yields a basic constant. We therefore consider all possible arities $n \geq 0$. With respect to the CEK-machine, the functionality of an expression is determined by the evaluation function. For a given arity n, the machine assigns to a program M the function

[7] In a slightly different context, these values are called *observable* [26].

$\mathcal{M}_M^n$:

$$\mathcal{M}_M^n = \{\langle a_1, \ldots, a_n, c\rangle \mid eval_{CEK}(M a_1 \ldots a_n) = c\}.$$

The calculus, on the other hand, interprets a program according to the equivalence relation:

$$\mathcal{I}_M^n = \{\langle a_1, \ldots, a_n, c\rangle \mid \lambda_v \vdash M a_1 \ldots a_n = c\}.$$

The consistency of this function definition is based on the Church-Rosser Theorem. The above Simulation Theorem, together with the Church-Rosser Theorem, implies that the two functional interpretations of an expression agree [47]:

Theorem 2.11. *For all* $n \geq 0$, $\mathcal{I}_M^n = \mathcal{M}_M^n$.

Since—as discussed above—basic values are what the programmer is interested in, this theorem liberates us from using the standard reduction function for evaluations. Every strategy that reduces programs to values is appropriate.

The theorem also leads to the second central issue of this section. Provided that the calculus is a system for reasoning about the equality of functions and programs, the question arises what equivalence proofs mean for the behavior of programs on the machine. To this end, we derive a compatible equivalence relation on terms from the evaluation function and compare this relation to the λ_v-calculus.

The CEK-evaluation function defines an equivalence relation on programs in a rather natural manner: two programs M and N are equivalent if they return the same answer. However, in order to extend this equivalence relation to a calculus-like system over all kinds of terms, we need to form a compatible closure. The original definition of compatibility in Section 2.3 points out the correct way to construct the appropriate relation as can be seen from the following proposition. It is a generalized construction, originally due to Morris [46:58]:

Proposition 2.12. *Let* $\sim$ *be an equivalence relation on* Λ*-terms. Define* $\simeq$ *as*

follows: $M \simeq N$ *if for all contexts* $C[\ ]$, $C[M] \sim C[N]$. *Then* $\simeq$ *is a compatible equivalence relation.*

In other words, an equivalence relation induces a congruence relation by requiring equivalence in all possible contexts. The proposition reduces our task to specifying precisely the comparison of program answers. Naturally, we would like to use the identity relation, but this is impossible since there are many programs that only return functions for use on other problems. Hence, once again we take refuge in basic constants and otherwise simply require that programs terminate:

Definition 2.13. $M, N \in \Lambda$ *are* operationally equivalent, $M \simeq_{CEK} N$, *if for any program context* $C[\ ]$ *such that* $C[M]$ *and* $C[N]$ *are closed,* $eval_{CEK}$ *is undefined for both* $C[M]$ *and* $C[N]$, *or it is defined for both and if one of the programs yields a basic constant, then the value of the other is the same constant.*

From an extensional point of view this operational equivalence is also how an end-user perceives, tests, and compares programs. For such a user, a program is a black-box that produces some outputs. A comparison of this output with the expected result decides about the correctness of a program. Thus, operational equivalence is a natural means for specifying the correctness of a program. A specification can require either the equivalence of a program to a value or to another program. The second possibility is preferred when a solution is easily specified with a simple, but inefficient program. A proof of correctness in either case means a verification of the operational specification, and in the second case, it also means an improvement of the solution with respect to efficiency. In light of this, the following, second correctness theorem of the λ_v-calculus is important [47]:

Theorem 2.14 (Soundness and Incompleteness). *If* $M =_v N$, *then* $M \simeq_{CEK} N$. *The converse is false.*

The theorem is a consequence of the Church-Rosser Theorem and the Standardization Theorem. A typical example that proves the claim about the converse direction is based on programs with infinite loops. Whereas all diverging programs are operationally equivalent, the calculus cannot prove the equivalence of such programs in general. Consider the two programs $(\lambda x.x)((\lambda x.xx)(\lambda x.xx))$ and $((\lambda x.xx)(\lambda x.xx))$. Both programs diverge. There is clearly no program context that can differentiate the two. Hence, they are equivalent in $\simeq_{CEK}$. But, at the same time, there is no reduction such that λ_v can eliminate the extraneous $\lambda x.x$ in the first program.[8]

The expressiveness of the machine calculus is captured in a stronger variant of the above theorem. It characterizes $\simeq_{CEK}$ as the largest extension of $=_v$ of its kind:

Theorem 2.15. *$\simeq_{CEK}$ is the largest consistent extension of $=_v$ that respects equality on the set of basic constants and that is also*

— *compatible: $M \simeq_{CEK} N$ implies $C[M] \simeq_{CEK} C[N]$ for all $C[\ ]$, and*

— *evaluating: $M \simeq_{CEK} N$ implies $C[M]$ has a value iff $C[N]$ has a value for all contexts $C[\ ]$ that close M and N.*

Proof. Morris's corresponding proof [46] relies on clever techniques by Böhm [4, 5:254–260] for the separation of normal forms. The proof of this theorem is similar, but simpler, because it can assume separability of observable values. □

The main point of the theorem is a reinforcement of the arguments preceding the Simulation Theorem. The machine-calculus is a true calculus that can be used to specify the behavior of program pieces without reference to the program context. Since it is the largest relation of its kind, everything that can be specified at all, must be specifiable in $\simeq_{CEK}$.

[8] Another counterexample concerns the parameterization of recursive functions. If a recursive function is independent of an argument, this argument can be passed before the recursive function is constructed. Although this yields two operationally equivalent programs, the λ_v-calculus cannot prove this equivalence. [Talcott, private communication]

Before leaving the section, we quickly summarize the theoretical part of the presentation. Λ is the set of terms and programs of the λ-calculus; the relation $\equiv$ is the term identity equivalence. The CEK-machine defines an operational semantics for Λ, which is close to machine implementations. For reasoning about Λ-programs, we use the λ_v-calculus. This is based on the fact that

- an evaluation in the calculus agrees with the machine evaluation (Theorem 2.10);
- equality in the calculus, $=_v$, implies behavioral equality on the machine, $\simeq_{CEK}$ (Theorem 2.14).

Together the two respective theorems are called correspondence theorems. The usefulness of the calculus for programming and reasoning in Λ is demonstrated next.

2.5. Programming and Reasoning with Λ

At first glance, Λ is a rather primitive programming language. Here, *primitive* does not refer to computational power, but it means lack of such traditional programming facilities as blocks, branching expressions, complex data structures, *etc.* Fortunately, the literature provides a wealth of studies [2, 5, 7, 35, 40, 51] on how to perceive these presumably fundamental necessities as syntactic embellishments to pure Λ.

In order to demonstrate this point, we present a series of meta-programming examples in Λ. For most examples, we first introduce a well-known programming language construct by giving an intuitive machine behavior; then, if possible, we formalize this specification or some property with the CEK-calculus; and finally we discuss and prove an implementation in pure Λ.

The examples make use of two definitional techniques. We write *combinator* specifications of the form

$$\mathsf{Name} \stackrel{df}{\equiv} \Lambda\text{-term},$$

where Name is some string of characters, the combinator name. A combinator is simply an abbreviation for the closed term on the right-hand side of the definition. New syntactic forms are defined with equations of the type[9]

$$(\textbf{keyword1}\ \textit{Exp1}\ \textbf{keyword2}\ \textit{Exp2}) \stackrel{df}{\equiv} \Lambda\text{-}\textit{term(Exp1, Exp2)}.$$

The meaning of this kind of equation is that every occurrence of the left-hand pattern in an expression is an abbreviation of the right-hand pattern with the expressions *Exp1* and *Exp2* appropriately instantiated. To allow for pyramided dependencies, the defining terms may contain previously defined combinators and syntactic forms. Every occurrence of such a combinator or syntactic form should be thought of as being replaced by the defining side of its specification. With these preliminaries in place, we are now ready for the actual examples.

Some programs in Λ have traditional names and, for completeness sake, we shall introduce them here. The identity function is

$$\mathsf{I} \stackrel{df}{\equiv} \lambda x.x$$

and it maps every value to itself.

$$\mathsf{K} \stackrel{df}{\equiv} \lambda xy.x$$

is a constant-function producing function, *i.e.*, when applied to a value, it returns a function that maps every defined argument to this value. Functional composition is accomplished by

$$\mathsf{B} \stackrel{df}{\equiv} \lambda fgx.f(gx).$$

One more combinator with an important role is

$$\mathsf{S} \stackrel{df}{\equiv} \lambda fgx.fx(gx).$$

[9] Kohlbecker's [33] extend-syntax-facility provides an implementation of this technique for Scheme.

Together, K and S are sufficient to compute every definable function, but of course the respective programs are practically unreadable because of the lack of high-level syntax.

Blocks appear in programming languages in various forms, yet these forms share a number of traits. Most generally, a block consists of a variable declaration and a block expression. The variable declaration is valid inside the block expression as long as the variable is not declared again. Also, outside of the block the variable is invisible. In some languages, a block initializes a variable to a possibly user-specified value; in others, the variable is left uninitialized. In a functional setting only the first version makes sense and a block is essentially a shorthand for improving readability and performance of programs. A suitable syntax for block-notation is

$$(\textbf{let}\ (x\ \mathit{Init})\ \mathit{Body}),$$

where x is the new variable that is initialized to the value of the expression *Init* and that is accessible from within *Body*. From the discussion it is clear that in Λ such a block is syntactically equivalent to an application of the form

$$(\lambda x.\mathit{Body})\mathit{Init}.$$

In this application, the λ-abstraction binds x in *Body*; when *Init* is evaluated to *Val*, the block becomes equivalent to

$$\mathit{Body}[x := \mathit{Val}]$$

making every reference to x a reference to the value of *Init*. The variable conventions automatically take care of redefinitions of x. Therefore, we may define the syntactic abstraction for a block by

$$(\textbf{let}\ (x\ \mathit{Init})\ \mathit{Body}) \stackrel{df}{\equiv} (\lambda x.\mathit{Body})\mathit{Init}.$$

A generalized version of the **let**-facility permits the simultaneous declaration of two or more variables:

$$(\textbf{let}\ ((x_1\ \mathit{Init}_1)\ldots(x_n\ \mathit{Init}_n))\ \mathit{Body}) \stackrel{df}{\equiv} (\lambda x_1\ldots x_n.\mathit{Body})\mathit{Init}_1\ldots\mathit{Init}_n.$$

Of course, the order of variable declarations should not matter. In the machine calculus, we express this for a block with two variables by requiring

$$(\textbf{let}\ ((x\ \mathit{Initx})(y\ \mathit{Inity}))\ \mathit{Body}) \simeq_{CEK} (\textbf{let}\ ((y\ \mathit{Inity})(x\ \mathit{Initx}))\ \mathit{Body})$$

for all *Initx* and *Inity*. The proof of this proposition is easy. Assume that either *Initx* or *Inity* has no value. Since both sides of the equation force the evaluation of these subexpressions, neither side has a value, and hence, they are operationally equivalent. Otherwise, both initialization expressions have values, and the equivalence follows from a simple calculation in the λ_v-calculus:

$$\begin{aligned}
(\textbf{let}\ ((x\ \mathit{Initx})(y\ \mathit{Inity}))\ \mathit{Body}) &\equiv (\lambda xy.\mathit{Body})\mathit{Initx}\ \mathit{Inity} \text{ by definition}\\
&=_v \mathit{Body}[x := \mathit{Initx}][y := \mathit{Inity}]\\
&\equiv \mathit{Body}[y := \mathit{Inity}][x := \mathit{Initx}]\\
&\qquad \text{by hygiene convention, substitution lemma}\\
&=_v (\lambda xy.\mathit{Body})\mathit{Inity}\ \mathit{Initx}\\
&\equiv (\textbf{let}\ ((y\ \mathit{Inity})(x\ \mathit{Initx}))\ \mathit{Body}).
\end{aligned}$$

Convention. *As in the proof above, we shall henceforth treat an expression that has a value as if it were a value.* **End**

The notion of a branching construct in a programming language presupposes the existence of truth values True and False. Given this, a generic branching construct of the form

$$(\textbf{if}\ \mathit{Test}\ \mathit{Then}\ \mathit{Else})$$

evaluates its *Then*-subexpression if *Test* has the value True, and the *Else*-part if *Test* yields False. An important characteristic is that either the *Then*- or the *Else*-branch affects the final outcome, but never both. With our machine calculus this can be formalized as

$$(\textbf{if } \mathsf{True}\ \mathit{Then}\ \mathit{Else}) \simeq_{CEK} \mathit{Then},$$

$$(\textbf{if } \mathsf{False}\ \mathit{Then}\ \mathit{Else}) \simeq_{CEK} \mathit{Else}.$$

In other words, the final value of a program only depends on the correct branch of an **if**-expression.

The programming language Λ can achieve the effect of an **if**-expression by representing True and False as functions which throw away one of their two arguments:

$$\mathsf{True} \stackrel{df}{\equiv} \lambda xy.x,$$

$$\mathsf{False} \stackrel{df}{\equiv} \lambda xy.y;$$

and by encapsulating the *Then*- and *Else*-part in abstractions which delay undesired evaluations:

$$(\textbf{if } \mathit{Test}\ \mathit{Then}\ \mathit{Else}) \stackrel{df}{\equiv} \mathit{Test}(\lambda d.\mathit{Then})(\lambda d.\mathit{Else})\mathsf{I},$$

where d is a fresh variable, *i.e.*, $d \notin FV(\mathit{Then}) \cup FV(\mathit{Else})$. The correctness of this definition is verified by the following analysis. Assume *Test* has the value True, then

$$\begin{aligned}(\textbf{if } \mathsf{True}\ \mathit{Then}\ \mathit{Else}) &=_v \mathsf{True}(\lambda d.\mathit{Then})(\lambda d.\mathit{Else})\mathsf{I} \\ &=_v (\lambda xy.x)(\lambda d.\mathit{Then})(\lambda d.\mathit{Else})\mathsf{I} \\ &=_v (\lambda d.\mathit{Then})\mathsf{I} \\ &=_v \mathit{Then}.\end{aligned}$$

Similarly, we get

$$(\textbf{if } \mathit{Test}\ \mathit{Then}\ \mathit{Else}) =_v \mathit{Else}$$

if *Test* yields False.

Our representation of the **if**-expression blends in with the parameterization of Λ over constant sets. All we require is that predicates in the constant set respect our representation of truth values. For example, a constant zero? over the set of natural numbers is defined by

$$\delta(\mathsf{zero?}, 0) = \mathsf{True} \text{ and } \delta(\mathsf{zero?}, n+1) = \mathsf{False}.$$

Furthermore, with the definition of **if**, it is also possible to realize more elaborate branching facilities like **cond**- and **case**-expressions by reducing them to **if**-expressions.

Next we turn to the specification of recursive functions. λ-abstractions provide functions *per se*. However, the lack of a name for the entire abstraction in the function body prohibits the usual, self-referential definition of recursive functions, which is the main characteristic of a recursive function specification. Nevertheless, recursion can be achieved in Λ. Consider a typical mathematical definition of a recursive function

$$fx = \ldots x \ldots fe_1 \ldots fe_2 \cdots$$

Using λ-abstraction, we can eliminate the argument name x and we can concentrate on the use of f on the right-hand defining side to a single occurrence:

$$f = (\lambda g x. \ldots . x \ldots ge_1 \ldots ge_2 \ldots) f.$$

From this equation, a recursive function can be considered as the fixpoint of a defining functional. Fortunately, it is well-known [51, 70] how to find such fixpoints in the λ-value-calculus. With the combinator

$$\mathsf{Y}_v \stackrel{df}{\equiv} \lambda f x.(\textbf{let } (g\ (\lambda g.f(\lambda x.ggx)))\ (g\ g))x,$$

one can find the fixpoint of any functional F, *i.e.*, $\mathsf{Y}_v F$ satisfies

$$\mathsf{Y}_v F x =_v F(\mathsf{Y}_v F)x \qquad \text{for all } x.$$

Thus, a recursive function definition is a syntactic abstraction of an application of Y_v to a defining functional.

A more readable syntactic form for recursive function definitions is the **rec**-expression:

$$(\textbf{rec}\ (f\ x) = M_{fx}) \stackrel{df}{\equiv} \mathsf{Y}_v(\lambda fx.M_{fx}).$$

The notation M_{fx} indicates that M's free f and x are bound by $\lambda fx.\ldots$.[10]

Recursive functions make the implementation of many iteration and looping facilities easy. Consider the most basic loop-construct for a functional language, namely, the m-fold iteration of a function F over a value V. In other words, if $F^m V$ stands for $(F \ldots m\text{-}times \ldots (FV)\ldots)$ then we are looking for an **iterate**-expression that satisfies

$$(\textbf{iterate}\ F\ \textbf{over}\ V\ \textbf{times}\ m) \simeq_{CEK} F^m V, \qquad m \geq 0.$$

In Λ this construct can be realized with a recursive function:

$$(\textbf{iterate}\ F\ \textbf{over}\ V\ \textbf{times}\ m) \stackrel{df}{\equiv} (\textbf{rec}\ (l\ m) = (\textbf{if}\ (\mathsf{zero?}\ m)\ V\ F(l(1^{-}m))))m.$$

For the correctness proof of this claim let $L \equiv \lambda lm.(\textbf{if}\ (\mathsf{zero?}\ m)\ V\ F(l(1^{-}m)))$.

Since

$$\mathsf{Y}_v Lm =_v L(\mathsf{Y}_v L)m =_v (\textbf{if}\ (\mathsf{zero?}\ m)\ V\ F((\mathsf{Y}_v L)(1^{-}m)))$$

by the fixpoint property of Y_v, it is easy to verify by induction on m, that

$$\mathsf{Y}_v Lm =_v F(\ldots m\text{-}times \ldots (FV)\ldots) \equiv F^m V.$$

The realization of this **iterate**-construct does not rely on jump-operations; it is solely based on functions. This is also true for other looping facilities like iterating a

10 In terms of contexts: $\lambda fx.M_{fx}$ is the result of filling $\lambda fx.[\ ]$ with M_{fx}.

function while or until a condition holds for the current value. It is probably for this reason that these control expressions and related imperative statements are called *structured*: their effect can be explained by a simple mathematical transliteration and is entirely independent of the program context.

Another important ingredient of programming languages is the domain of complex data structures. Prominent examples are pairs, lists, vectors, trees, graphs, and records. Often a restricted form of these structures is built into the constant set of a particular Λ-language. However, Λ itself can perfectly simulate appropriate data sets by means of functional abstraction. The set of arbitrary value-pairs, for example, is isomorphic to the cartesian product of the set of values. Hence, it may be defined via the specification of an injection function or *constructor* and two projection functions or *selectors*. If we let $[\cdot,\cdot]$ be an infix-notation for the constructor and $(\cdot)_1$ and $(\cdot)_2$ infix-notation for the respective selectors, then their operational specification is

$$([U,V])_1 \simeq_{CEK} U \text{ and } ([U;V])_2 \simeq_{CEK} V,$$

for all values U and V. One possible implementation is

$$[\cdot,\cdot] \equiv \lambda uv.\lambda m.muv,$$
$$(\cdot)_1 \equiv \lambda p.p(\lambda uv.u),$$
$$(\cdot)_2 \equiv \lambda p.p(\lambda uv.v).$$

The functions obviously satisfy the above operational specification. If the data structures are only specified via the operational equivalence relation such that other programmers cannot rely on the properties of the concrete realization, then they can safely be called *abstract*. Programmers must use the combinators since they are

not guaranteed that the data structure is not implemented as

$$[\cdot,\cdot] \equiv \lambda uv.\lambda m.mvu,$$

$$(\cdot)_1 \equiv \lambda p.p(\lambda uv.v),$$

$$(\cdot)_2 \equiv \lambda p.p(\lambda uv.u),$$

or something else.

The implementation strategy for pairs generalizes to vectors of arbitrary length. Let

$$\pi_i^n \stackrel{df}{\equiv} \lambda x_1 \ldots x_n.x_i \qquad \text{for } 1 \leq i \leq n.$$

Then we can define an n-place constructor $[\cdot,\ldots,\cdot]_n$ and respective selectors $(\cdot)_{n,i}$ as

$$[\cdot,\ldots,\cdot]_n \stackrel{df}{\equiv} \lambda x_1 \ldots x_n.\lambda m.mx_1 \ldots x_n,$$

$$(\cdot)_{n,i} \stackrel{df}{\equiv} \lambda v.v\pi_i^n.$$

Their characteristic equations are

$$([V_1,\ldots,V_n])_{n,i} =_v V_i \qquad \text{for } 1 \leq i \leq n.$$

Pairs and vectors are a good basis for the implementation of Lisp-like list structures. Recall that a list is either

— the *empty* list, or

— a *node*, consisting of an arbitrary value and a list.

Since the definition contains two clauses, we need two constructor functions. Following Lisp, we call them NIL and cons. The selectors are car and cdr. The predicate that distinguishes between the two cases is null?. In order to correspond to the informal definition, the functions must satisfy

$$\mathsf{null?}\ \mathsf{NIL} \simeq_{CEK} \mathsf{True} \text{ and } \mathsf{null?}(\mathsf{cons}\ V\ L) \simeq_{CEK} \mathsf{False}$$

and

$$\mathsf{car}(\mathsf{cons}\, V\, L) \simeq_{CEK} V \text{ and } \mathsf{cdr}(\mathsf{cons}\, V\, L) \simeq_{CEK} L$$

where L is a list. Our implementation of these list structures is

$$\begin{aligned}
\mathsf{NIL} &\stackrel{df}{\equiv} [\mathsf{True}, \mathsf{False}] \\
\mathsf{cons} &\stackrel{df}{\equiv} \lambda vl.[\mathsf{False}, [v, l]] \\
\mathsf{car} &\stackrel{df}{\equiv} \lambda l.((l)_2)_1 \\
\mathsf{cdr} &\stackrel{df}{\equiv} \lambda l.((l)_2)_2 \\
\mathsf{null?} &\stackrel{df}{\equiv} \lambda l.(l)_1.
\end{aligned}$$

Again, the verification of the correctness of these definitions is straightforward.

With the preceding exercises we have indicated how to embed pure Lisp (with higher-order functions) in Λ. Before we end this exercise on programming in Λ, we reformulate some well-known Lisp-functions with our newly introduced syntax. Given a list of numbers, the function Σ produces the sum of the numbers:

$$\begin{aligned}
\Sigma \equiv (\mathbf{rec}\ (s\ l) = \\
(\mathbf{if}\ (\mathsf{null?}\ l)\ 0 \\
(+(\mathsf{car}\ l)(s(\mathsf{cdr}\ l))))).
\end{aligned}$$

Another useful program is the map-functional which applies a function f to every element of a list l and constructs a list out of the results:

$$\begin{aligned}
\mathsf{map} \equiv \lambda f.(\mathbf{rec}\ (m\ l) = \\
(\mathbf{if}\ (\mathsf{null?}\ l)\ \mathsf{NIL} \\
(\mathsf{cons}\,(f(\mathsf{car}\ l))(m\ (\mathsf{cdr}\ l))))).
\end{aligned}$$

With Lisp-lists we can define other data structures using familiar programming techniques and thus avoid falling back on pure Λ. For example, a binary tree structure as defined in the introduction is an easy programming exercise. If we let

NIL stand for the empty tree, a node can be constructed with two cons-es

$$\mathsf{mk\text{-}nd} \equiv \lambda lvr.\mathsf{cons}\ l(\mathsf{cons}\ v\ r).$$

This is possible because we never ruled out lists as first arguments to cons. The predicate empty? is of course equivalent to null?; the selectors lson, rson, and info are suitable combinations of car and cdr:

$$\begin{aligned}\mathsf{empty?} &\equiv \mathsf{null?},\\ \mathsf{lson} &\equiv \mathsf{car},\\ \mathsf{info} &\equiv \mathsf{B\ car\ cdr},\\ \mathsf{rson} &\equiv \mathsf{B\ cdr\ cdr}.\end{aligned}$$

Finally, the function Σ^* from the introduction becomes:

$$\begin{aligned}\Sigma^* \equiv (\mathbf{rec}\ (s\ t) = \\ (\mathbf{if}\ (\mathsf{empty?}\ t)\ 0\\ (+(\mathsf{info}\ t)(+(s(\mathsf{lson}\ t))(s(\mathsf{rson}\ t)))))).\end{aligned}$$

We have omitted an expansion of these definitions into pure Λ since the additional syntax truly improves the readability of programs.

The programming examples in this section have illustrated how the language can easily support a broad variety of additional features and that programming in an enriched Λ is a feasible endeavor. However, as discussed in the introduction, Λ like any other functional language fundamentally suffers from a lack of imperative facilities. The addition of these facilities to Λ as a programming language is treated in the next chapter.

3. Idealized Scheme: An imperative extension of Λ

The definition of an imperative extension of Λ as a programming language is the first step in constructing a calculus for imperative higher-order languages. Originally, our goal was a calculus for a Scheme-like core language, but in the course of our research we discovered that some minor modifications simplify the resulting calculus[1] [15, 19]. Since the work is a feasibility demonstration, we take the improved version as the new starting point and call it *Idealized Scheme*.

According to our framework, the definition of a new programming language requires the specification of a syntax and a semantics. This is done in the first two sections. In the third section we continue our meta-programming exercise from Section 2.5 and show how to embed imperative programming facilities from traditional and modern programming languages.

3.1. $\Lambda_{\mathcal{F}\sigma}$

The term set of Idealized Scheme is $\Lambda_{\mathcal{F}\sigma}$, an extension of Λ. The syntax is summarized in Definition 3.1. The language contains two new classes of expressions:

— $\mathcal{F}$-applications: $(\mathcal{F}M)$, which are control expressions; M is called the $\mathcal{F}$-argument;

[1] See section 7.3.5 for a discussion on this feedback relation between programming language design and calculus design.

— σ-capabilities: $(\sigma x.M)$, which roughly correspond to assignments; x is the assignable variable, M is the σ-body.

The set of values in $\Lambda_{\mathcal{F}\sigma}$ includes σ-capabilities. A term in the Λ-subset is called *pure*.

When evaluated, an $\mathcal{F}$-application applies its argument to a functional abstraction of its current continuation, *i.e.*, the rest of the computation. This action gives a program complete control over the continuation. Upon application, a continuation performs the actions of the encoded rest of a computation, and, upon completion, a functional continuation returns the final value to the point of its invocation. A σ-capability encapsulates the right to assign the σ-variable a new value. It does *not* bind the variable. The application of a σ-capability to a value is called a σ-application. Intuitively, a σ-application first assigns the variable the new value and then continues to evaluate the σ-body. The value of the σ-body is the result of the entire application. The meaning of the remaining constructs should be adapted accordingly: variables are assignable placeholders for values, abstractions correspond to call-by-value procedures, and applications invoke the result of the function part, possibly a σ-capability, on the value of the argument part. If the result of the function part is a λ-abstraction, the application is a function application; otherwise, we refer to it as σ-application.

The language satisfies our outlined philosophical criteria. First, it is a minimal extension with respect to Λ; there is exactly one new form for evaluation control and one for state manipulation. Second, the new expression types are orthogonal to each other and to the original constructs in Λ. λ-abstraction is still the only binding construct, applications are the only evaluation vehicle. And finally, the language syntax is entirely expression-oriented, whereas generally the introduction of imperative facilities splits a language into a statement and an expression cate-

Definition 3.1: The programming language $\Lambda_{\mathcal{F}\sigma}$

Syntactic Domains:

a	$\in Const$	(constants)
x	$\in Vars$	(variables)
M, N	$\in \Lambda$	(Λ-terms)

Abstract Syntax:

$$M ::= a \mid x \mid (\lambda x.M) \mid (MN) \mid (\mathcal{F}M) \mid (\sigma x.M).$$

All conventions for Λ are applied *mutatis mutandis*.

gory. The advantage of our treatment is that the syntactic homogeneity permits a straightforward transliteration of the framework from Chapter 2. This can already be seen from the re-definition of the static semantics.

The concepts of free and bound variables carry over unchanged. For example, the additional clauses for $FV(\cdot)$ and $BV(\cdot)$ are

$$FV(\mathcal{F}M) = FV(M), \qquad BV(\mathcal{F}M) = BV(M),$$

$$FV(\sigma x.M) = FV(M); \qquad BV(\sigma x.M) = BV(M).$$

All other syntactic definitions and conventions for Λ are adapted with changes similar to these, *e.g.*, the definition of a *one-hole context* becomes

$$C[\ ] ::= [\ ] \mid (\lambda x.C[\ ]) \mid (C[\ ]M) \mid (MC[\ ]) \mid (\mathcal{F}C[\ ]) \mid (\sigma x.C[\ ]).$$

The definition of an evaluation context does not change.

3.2. The CESK-Machine

The formal semantics of $\Lambda_{\mathcal{F}\sigma}$-programs is defined via an extension of the CEK-machine to a CESK-machine. The additional letter S stands for the new store

component. A store is needed to implement the effect of assignments. Since continuations are already present in the machine, the implementation of $\mathcal{F}$-applications only requires the addition of new transition rules.

Definition 3.2: The CESK-machine, part I: the computational domains

Computational Domains:

$$
\begin{array}{lll}
s \in \mathit{States} & = \mathit{Controls} \times \mathit{Envs} \times \mathit{Stores} \times \mathit{Conts} & \text{(machine states)} \\
c \in \mathit{Controls} & = \Lambda_{\mathcal{F}\sigma} + \ddagger & \text{(control strings)} \\
\rho \in \mathit{Envs} & = \mathit{Vars} \xrightarrow{\circ} \mathit{Locs} & \text{(environments)} \\
\theta \in \mathit{Stores} & = \mathit{Locs} \xrightarrow{\circ} \mathit{Closures} & \text{(stores)} \\
V \in \mathit{Closures} & = \mathit{Values} \times \mathit{Envs} + \mathbf{p} \times \mathbf{p}\text{-}\mathit{Conts} & \text{(closures)} \\
\kappa \in \mathit{Conts} & = \mathbf{ret}\text{-}\mathit{Conts} + \mathbf{p}\text{-}\mathit{Conts} & \text{(continuation codes)}
\end{array}
$$

where *Locs* is an arbitrary infinite set

and p-continuations and **ret**-continuations are defined by:

$$\mathbf{p}\text{-}\mathit{Conts} = \mathbf{stop} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{fun} \times \mathit{Closures} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{arg} \times \Lambda_{\mathcal{F}\sigma} \times \mathit{Envs}$$

$$\mathbf{ret}\text{-}\mathit{Conts} = \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{ret} \times \mathit{Closures}.$$

Auxiliary Function:

$$\cdot \otimes \cdot : \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{p}\text{-}\mathit{Conts} \longrightarrow \mathbf{p}\text{-}\mathit{Conts}$$

$$\kappa \otimes (\mathbf{stop}) = \kappa$$

$$\kappa \otimes (\kappa' \ \mathbf{arg}\ N\ \rho) = (\kappa \otimes \kappa' \ \mathbf{arg}\ N\ \rho)$$

$$\kappa \otimes (\kappa' \ \mathbf{fun}\ V) = (\kappa \otimes \kappa' \ \mathbf{fun}\ V)$$

The necessary changes to the computational domains are summarized in Definition 3.2, part I. Environments are divided into an environment part and a store part. The new *environments* map variables to locations; *stores* assign locations to semantic values, or, in more traditional language, a location in the store contains a

value.[2] The set of locations *Locs* is an arbitrary, infinite set. In order to deal with $\mathcal{F}$-applications and σ-capabilities, we extend the domain of closures with σ-closures and **p**-closures. A σ-closure is simply a σ-capability combined with an environment; a **p**-closure is a **p**-tagged **p**-continuation structure.

A CESK-state is either of the form $\langle M, \rho, \theta, \kappa \rangle$, where ρ covers all free variables in M and θ is defined for all locations that occur in ρ or in the environments that occur in the **p**-continuation κ; or a state is of the form $\langle \ddagger, \emptyset, \theta, \kappa \rangle$, where κ is a **ret**-continuation and θ defines the contents of all locations in κ. An *initial* state is of the form $\langle M, \emptyset, \emptyset, (\mathbf{stop}) \rangle$; $\langle \ddagger, \emptyset, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V) \rangle$ is a prototypical *terminal* state.

The CESK-transition function is displayed in Definition 3.2, part II. The first seven clauses are adaptations of the CEK-transition function rules. Merging the environment and the store component yields the CEK-transition function. We also assume that *the δ-function still returns pure, closed values.* This is a minor restriction, but we have never experienced any need for a relaxation. (CESK7) and (CESK8) describe the evaluation of an $\mathcal{F}$-application and the application of a continuation to a value; (CESK9) and (CESK10) define the effect of an assignment application.

The evaluation of an $\mathcal{F}$-application directly corresponds to the informal description of $\mathcal{F}$ at the outset of the first section. The $\mathcal{F}$-argument is applied to γ, which stands for the current continuation. The current continuation is transferred out of the register into a **p**-closure and this gives the program total control over its use. In particular the decision about when to use the continuation is left to the program. The operation call/cc in Scheme does not offer this option: it always copies the continuation into a program accessible structure. From the program's perspective

[2] The decision to keep every semantic value in the store is arbitrary: see Chapter 5.

Definition 3.2: The CESK-machine, part II: the transition function

The CESK-transition function maps states to states:

$$States \overset{CESK}{\longmapsto} States,$$

according to the following cases:

$$\langle a, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\ \mathbf{ret}\ \langle a, \emptyset\rangle)\rangle \tag{0}$$

$$\langle x, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\ \mathbf{ret}\ \theta(\rho(x)))\rangle \tag{1}$$

$$\langle \lambda x.M, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\ \mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle \tag{2}$$

$$\langle MN, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle M, \rho, \theta, (\kappa\ \mathbf{arg}\ N\ \rho)\rangle \tag{3}$$

$$\langle \ddagger, \emptyset, \theta, ((\kappa\ \mathbf{arg}\ N\ \rho)\ \mathbf{ret}\ V)\rangle \overset{CESK}{\longmapsto} \langle N, \rho, \theta, (\kappa\ \mathbf{fun}\ V)\rangle \tag{4}$$

$$\langle \ddagger, \emptyset, \theta, ((\kappa\ \mathbf{fun}\ \langle \lambda x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle \overset{CESK}{\longmapsto} \langle M, \rho[x := n], \theta[n := V], \kappa\rangle \quad \text{where } n \notin Dom(\theta) \tag{5}$$

$$\langle \ddagger, \emptyset, \theta, ((\kappa\ \mathbf{fun}\ \langle f, \emptyset\rangle)\ \mathbf{ret}\ \langle a, \emptyset\rangle)\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\ \mathbf{ret}\ \langle \delta(f, a), \emptyset\rangle)\rangle \tag{6}$$

$$\langle \mathcal{F}M, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle M\gamma, \rho[\gamma := n], \theta[n := \langle \mathbf{p}, \kappa\rangle], (\mathbf{stop})\rangle \quad \text{where } n \notin Dom(\theta),\ \gamma \notin FV(M) \tag{7}$$

$$\langle \ddagger, \emptyset, \theta, ((\kappa\ \mathbf{fun}\ \langle \mathbf{p}, \kappa_0\rangle)\ \mathbf{ret}\ V)\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa \otimes \kappa_0\ \mathbf{ret}\ V)\rangle \tag{8}$$

$$\langle \sigma x.M, \rho, \theta, \kappa\rangle \overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\ \mathbf{ret}\ \langle \sigma x.M, \rho\rangle)\rangle \tag{9}$$

$$\langle \ddagger, \emptyset, \theta, ((\kappa\ \mathbf{fun}\ \langle \sigma x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle \overset{CESK}{\longmapsto} \langle M, \rho, \theta[\rho(x) := V], \kappa\rangle. \tag{10}$$

this is equivalent to saying that a call/cc-continuation is immediately invoked.

The application of a continuation in a $\Lambda_{\mathcal{F}\sigma}$-program results in the concatenation of the applied continuation to the current one. This causes the machine to evaluate the applied continuation as if it were a function. If the applied continuation does not affect its continuation during the evaluation, the result is returned to the point of application. In this respect again, CESK-continuations differ from Scheme-continuations. When a program invokes a continuation in Scheme, the

interpreter first discards its current continuation and then installs the invoked continuation in the continuation register. Put differently, Scheme-continuation objects have an abortive effect on the continuation of their application point. We present an implementation of call/cc in terms of $\mathcal{F}$ in the next section.

In order to clarify the abstract explanation of $\mathcal{F}$ and its comparison with call/cc, we trace the evaluation of $1^+(\mathcal{F}(\lambda k.k(k0)))$:

$$\begin{aligned}
&\langle 1^+(\mathcal{F}(\lambda k.k(k0))), \emptyset, \emptyset, (\textbf{stop})\rangle \\
\xrightarrow{CESK^+}\ &\langle (\mathcal{F}(\lambda k.k(k0))), \emptyset, \emptyset, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle \\
\xrightarrow{CESK}\ &\langle (\lambda k.k(k0))\gamma, \{\gamma \mapsto 1\}, \{1 \mapsto \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle\}, (\textbf{stop})\rangle.
\end{aligned}$$

At this point the continuation is captured although it is not yet under direct control of the $\mathcal{F}$-argument. If we replaced $\mathcal{F}$ by call/cc, this intermediate state would still contain the old continuation:

$$\langle (\lambda k.k(k0))\gamma, \{\gamma \mapsto 1\}, \{1 \mapsto \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle\}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle.$$

With the next few steps the example program reaches a continuation application:

$$\begin{aligned}
&\langle (\lambda k.k(k0))\gamma, \{\gamma \mapsto 1\}, \{1 \mapsto \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle\}, (\textbf{stop})\rangle \\
\xrightarrow{CESK^+}\ &\langle k(k0), \rho_0, \theta_0, (\textbf{stop})\rangle \\
\xrightarrow{CESK^+}\ &\langle k0, \rho_0, \theta_0, ((\textbf{stop})\ \textbf{fun}\ \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle)\rangle \\
\xrightarrow{CESK^+}\ &\langle \ddagger, \emptyset, \theta_0, \\
&\qquad ((((\textbf{stop})\ \textbf{fun}\ \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle)\ \textbf{fun}\ \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle)\ \textbf{ret}\ 0)\rangle \\
\xrightarrow{CESK}\ &\langle \ddagger, \emptyset, \theta_0, ((((\textbf{stop})\ \textbf{fun}\ \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle)\ \textbf{fun}\ 1^+)\ \textbf{ret}\ 0)\rangle,
\end{aligned}$$

where

$$\begin{aligned}
\rho_0 &= \{\gamma \mapsto 1, k \mapsto 2\} \\
\theta_0 &= \{1 \mapsto \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle, 2 \mapsto \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle\}.
\end{aligned}$$

The actual invocation of the continuation does not discard the current continuation. With call/cc, on the other hand, this intermediate state would have been:

$$\langle \ddagger, \emptyset, \theta_0, (((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{ret}\ 0)\rangle.$$

For functional continuations, as in the CESK-machine, the evaluation appends the two continuations:

$$\begin{aligned}
\ldots &\overset{CESK^+}{\longmapsto} \langle \ddagger, \emptyset, \theta_0, (((\textbf{stop})\ \textbf{fun}\ \langle \mathrm{p}, ((\textbf{stop})\ \textbf{fun}\ 1^+)\rangle)\ \textbf{ret}\ 1)\rangle \\
&\overset{CESK^+}{\longmapsto} \langle \ddagger, \emptyset, \theta_0, (((\textbf{stop})\ \textbf{fun}\ 1^+)\ \textbf{ret}\ 1)\rangle \\
&\overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta_0, ((\textbf{stop})\ \textbf{ret}\ 2)\rangle.
\end{aligned}$$

The result is 2. This can also be deduced informally. The continuation of the $\mathcal{F}$-application in the program is an application of the function 1^+. A twofold application of this continuation to 0 returns 2.

The effect of an assignment application depends on three different factors. First, occurrences of variables are disambiguated via the environment, but the store contains the associated current value. Second, the store component of a CESK-evaluation is always present. The only operations on the store are extensions and updates. Unlike the environment, it never shrinks nor is it removed from its register. The transition rules (CESK5) and (CESK7) are responsible for store extensions. They allocate a new location for every semantic value that becomes accessible to the program. Allocating a new location means picking a location that is not in the (finite) domain of the store. *For the moment, we assume that this choice is fixed for a given set of locations.* Third, the rules (CESK9) and (CESK10) jointly affect the location-value association in the store according to an intuitive understanding of assignment. (CESK9) produces a σ-closure for the definition of a σ-capability, which contains the current environment. Thus the transition rule (CESK10) changes the value of the variable that was lexically visible at definition time. Because of the

constant presence of the store, every subsequent occurrence of this variable refers to the new value.

Let us consider the evaluation of $(\lambda x.(\sigma x.x)(\lambda y.x))0$:

$$\langle(\lambda x.(\sigma x.x)(\lambda y.x))0, \emptyset, \emptyset, (\mathbf{stop})\rangle$$
$$\overset{CESK^+}{\longmapsto} \langle(\sigma x.x)(\lambda y.x), \{x \mapsto 1\}, \{1 \mapsto 0\}, (\mathbf{stop})\rangle.$$

At this point, the assignment application is about to be evaluated. After evaluating the value part, the machine alters the appropriate store location, namely 1, and continues with the σ-body x:

$$\overset{CESK^+}{\longmapsto} \langle x, \{x \mapsto 1\}, \{1 \mapsto \langle \lambda y.x, \{x \mapsto 1\}\rangle\}, (\mathbf{stop})\rangle$$
$$\overset{CESK^+}{\longmapsto} \langle \ddagger, \emptyset, \{1 \mapsto \langle \lambda y.x, \{x \mapsto 1\}\rangle\}, ((\mathbf{stop})\ \mathbf{ret}\ \langle \lambda y.x, \{x \mapsto 1\}\rangle)\rangle.$$

The final value is a closure. The environment of the closure maps x to a location that contains the same closure: in higher-order languages with assignment statements it is possible to construct circular values. Because of this problem, the definition of an evaluation function for the CESK-machine is less intuitive than the one for the CEK-machine.

A program evaluation on the CESK-machine begins in $\langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle$ and terminates in $\langle \ddagger, \emptyset, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$ for a store θ and a closure V. Naturally, we would like to map V to a syntactic value, but the possibility of continuation structures or circular location references in V rules out straightforward solutions. We therefore parameterize the evaluation function for the CESK-machine over an unload function in order to permit a later re-definition:

$$eval_{CESK,\mathsf{Unload}} = \mathsf{Unload}(V, \theta) \text{ if } \langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle \overset{CESK^+}{\longmapsto} \langle \ddagger, \emptyset, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle.$$

Unless stated otherwise, we assume that Unload maps V and θ to the pair $\langle V, \theta\rangle$.

The $eval_{CESK,\mathsf{Unload}}$-function defines the extensional semantics of Idealized Scheme, the transition function the intensional one. Since Idealized Scheme is an

extension of the λ-calculus and since furthermore, the two additional facilities are included in order to express events and their ordering, the extensional semantics is sequential. The sequentiality of the CESK-machine is also obvious. However, we shall demonstrate in subsequent chapters that this is not inherent in the language, but due to the granularity of event representations.

For manual program evaluations and comparisons, the CESK-transition function is just as unsuited as its CEK-counterpart. Indeed, the situation is even worse. Because of the manifestation of side-effects in stores, we must compare the store components of intermediate machine states for a comparison of program pieces. However, these stores may contain *garbage*, *i.e.*, locations that are disconnected from the rest of the computation; and the stores may only be isomorphic to each other. We demonstrate both phenomena with two small examples.

The first sample program is $(\lambda x.M)0$ where $x \notin FV(M)$. Intuitively, this should be equivalent to just M in any store, but this is not true as can be seen from the following calculation:

$$\langle (\lambda x.M)0, \rho, \theta, \kappa \rangle \overset{CESK^+}{\longmapsto} \langle M, \rho[x := n], \theta[n := 0], \kappa \rangle.$$

The location n in the store is a typical garbage element. It cannot be reached from M, nor from the rest of the store. These elements clearly prohibit a straightforward comparison of stores. For the problem of relocated values, consider the expressions $(\lambda xy.M)01$ and $(\lambda yx.M)10$. Our intuition says that the two must be equivalent, but the respective evaluations on the CESK-machine only produce isomorphic states:

$$\langle (\lambda xy.M)01, \rho, \theta, \kappa \rangle \overset{CESK^+}{\longmapsto} \langle M, \rho[x := m][y := n], \theta[m := 0][n := 1], \kappa \rangle$$

and

$$\langle (\lambda yx.M)10, \rho, \theta, \kappa \rangle \overset{CESK^+}{\longmapsto} \langle M, \rho[y := m][x := n], \theta[m := 1][n := 0], \kappa \rangle.$$

Although these problems are stringent, equivalence proofs are possible. The central idea is to eliminate garbage from the stores and to compare the resulting stores with isomorphisms. This, however, is hardly practical because the elimination of garbage is generally complicated. An alternative to this technique is developed in the following chapter. It relies on the observation that all store cells needed by a computation are associated with the program text, and that all useless garbage cells must provably be disconnected from the program text.

3.3. Programming with $\Lambda_{\mathcal{F}\sigma}$

Next we give a brief introduction to programming and meta-programming with the language. Without a reasoning system, the examples are necessarily informal and rely on an operational understanding of the CESK-machine. We will resume most of the examples in Chapter 6 after developing a calculus for Idealized Scheme. Like in Chapter 2, we will then prove the correctness and properties of the programs that we present below.

Before the actual introduction, we briefly compare $\mathcal{F}$ and σ with their most closely related counterparts in existing programming languages. This should help clarifying their semantics. Also, some of the examples below may be easier to understand in terms of these equivalence specifications.

As mentioned in the preceding section, $\mathcal{F}$-applications are closely related to the call/cc-function in Scheme. There are two essential differences. First, when call/cc is applied to a function, it provides this function access to the current continuation, but it also leaves the continuation in its register. This effect can be achieved by an $\mathcal{F}$-application if the $\mathcal{F}$-argument immediately invokes the continuation. Second and more important, call/cc applies its argument to a continuation object, which, when applied in turn, discards the current continuation; an $\mathcal{F}$-application, on the other hand, simply provides its argument with a function that upon invocation performs

the same action as the current continuation. Hence, a simulation of call/cc must pass a function to the call/cc argument which throws away its current continuation, or, in terms of $\mathcal{F}$-applications, the object must grab the continuation without using it. Putting all this together produces the following equivalence:

$$\text{call/cc} \equiv \lambda f.\mathcal{F}(\lambda k.k(f(\lambda v.\mathcal{F}(\lambda d.kv)))).$$

Landin's J-operator [39] and Reynolds's **escape**-construct [50] are also closely related language facilities. Both are syntactic variations on call/cc [14] and hence, we omit a more detailed treatment. None of these facilities can implement an $\mathcal{F}$-application as a syntactic extension because of the abortive effect of their respective continuation objects.

The assignment abstraction is more conventional. In traditional expression-oriented languages, statements usually come together with a block statement like **begin** $\langle stmt \rangle$ **result** $\langle exp \rangle$. This is a block that first performs the statement-part and then evaluates the expression-part to return a result. Together with ordinary assignment, this block can express an assignment abstraction as a function:

$$\sigma x.M \equiv (\lambda v.(\textbf{begin } x := v \textbf{ result } M)) \quad \text{where } v \text{ is a fresh variable.}$$

The inverse relationship is expressed by

$$\textbf{begin } x := N \textbf{ result } M \equiv (\sigma x.M)N.$$

As motivated above, the choice of σ-capabilities over assignment statements is for syntactic and technical reasons only.

With the introduction of imperative program facilities, it makes sense to consider traditional Algol-style constructs like compound statements, **while**-loops, *etc.* For example, a sequencing expression

$$(\textbf{begin } Exp_1 \ \ldots \ Exp_n)$$

that returns the value of the last expression after having evaluated Exp_1 through Exp_{n-1} from left to right is equivalent to

$$\pi_n^n\ Exp_1 \ldots Exp_n.$$

This is a purely functional expression, but it only makes sense in the presence of side-effects because the values of the first $n-1$ expressions are thrown away. A similar argument holds for a **while**-loop. It is implemented by

$$(\textbf{while}\ Cond\ \textbf{do}\ Exp) \stackrel{df}{\equiv} \mathsf{Y}_v(\lambda wb.(\textbf{if}\ b\ (\textbf{begin}\ Exp\ (w\,Cond))\ \mathsf{I})Cond.$$

This transliteration is also functional, yet again, its use relies on the presence of side-effects.

Beyond the class of imperative statements whose transliteration is functional, there are also constructs that have functional appearances but are indeed inherently imperative. This is especially true for control constructs that are simple uses of $\mathcal{F}$-applications. For example, a halt-function is realized by grabbing and throwing away the current continuation:

$$\mathsf{halt} \stackrel{df}{\equiv} \lambda x.\mathcal{F}(\lambda d.x).$$

Since grabbing and throwing away a continuation is a recurring program pattern, we introduce a general syntactic abstraction[3]

$$(\textbf{throw}\ LV).$$

Such a **throw**-expression invokes the function L on the value V after eliminating the current continuation, *i.e.*,

$$(\textbf{throw}\ LV) \stackrel{df}{\equiv} \mathcal{F}(\lambda d.LV).$$

[3] This is only superficially related to Common Lisp's **throw**-facility [60].

It resembles a parameter-passing **goto**-facility. With **throw**, the implementation of halt becomes:

$$\mathsf{halt} \stackrel{df}{\equiv} \lambda x.(\mathbf{throw}\ \mathsf{I}x).$$

In other words, halt acts like a **goto** to the initial continuation or, equivalently, the final label.

A function-**exit**-facility is an equally simple use of $\mathcal{F}$. It occurs in an abstraction

$$(\mathbf{function}\ x\ \mathit{Body}),$$

which is like an ordinary function except that its function body may contain **exit**-expressions of the form

$$(\mathbf{exit}\ \mathit{Exp}).$$

The effect of an **exit**-expression is an immediate return to the function caller with the value of *Exp*. When cast into continuation terminology, this description leads to the obvious implementation of **function**- and **exit**-expressions: an **exit**-expression resumes the continuation of the **function**-application:

$$(\mathbf{function}\ x\ \mathit{Body}) \stackrel{df}{\equiv} \lambda x.\mathcal{F}(\lambda\epsilon.\epsilon\mathit{Body}),$$
$$(\mathbf{exit}\ \mathit{Exp}) \stackrel{df}{\equiv} (\mathbf{throw}\ \epsilon\ \mathit{Exp}).$$

We assume that **exit**'s ϵ is bound by the ϵ in the **function**-expansion.

The Σ_0^*-function from the introductory chapter offers a concrete example for the use of an **exit**-facility:

$$\begin{aligned}\Sigma_0^* \equiv \mathbf{function}\ t\ ((\mathbf{rec}\ (s\ t)\ =\ & \\ & (\mathbf{if}\ (\mathsf{empty?}\ t)\ 0 \\ & \quad (\mathbf{if}\ (\mathsf{zero?}(\mathsf{info}\ t))\ (\mathbf{exit}\ 0) \\ & \quad\quad (+(\mathsf{info}\ t)(+(s(\mathsf{lson}\ t))(s(\mathsf{rson}\ t))))))) \\ t)\ .&\end{aligned}$$

The inner application of the recursive function s to the tree t is an example of a generalized loop. Hence, from this definition it should be clear how to build any kind of loop constructs with **exit**-facilities. Although all of these programming constructs look functional, the reader should be aware that the presence of $\mathcal{F}$ makes them imperative.

Assignment capabilities are also imperative in nature, but they give rise to a different class of programs. In conjunction with higher-order functions it is easy to program reference cells with σ-capabilities. The four operations on a cell are: mk-cell, which creates a new cell with a given content; deref, which looks up the current contents of a cell; set-cell!, which changes the contents of a cell to a new value and returns this value upon completion; and, eq?, which compares the identity of two cell-objects.

An implementation of the first three functions is:

$$\begin{aligned} \textsf{mk-cell} &\stackrel{df}{\equiv} \lambda x.\lambda m.mx(\sigma x.x), \\ \textsf{deref} &\stackrel{df}{\equiv} \lambda c.c(\lambda xs.x), \\ \textsf{set-cell!} &\stackrel{df}{\equiv} \lambda c.c(\lambda xs.s). \end{aligned}$$

Upon application to a value, the function mk-cell returns a functional encoding of a pair. The first component of the pair is the current cell-value, the second one a σ-capability for the first component. Accordingly, the functions deref and set-cell! take a cell and apply it to the appropriate selector-function. In particular, the definition of set-cell! clarifies why we call a $\sigma x.$-term "a capability for assigning x a new value." When set-cell! is applied to a cell, it returns the σ-capability for the variable x, which upon some later application changes the contents of the cell.

The function eq? is generally a "built-in"-predicate and relies on the address space of the underlying machine. But this need not be the case: side-effects offer

an alternative solution [42:66].[4] Two cells are identical objects if and only if an alteration of the contents of one affects the other:

$$\begin{aligned}
\mathsf{eq?} \stackrel{df}{\equiv} \lambda c_1 c_2.(&\textbf{let } ((x_1\ (\mathsf{deref}\, c_1))(x_2\ (\mathsf{deref}\, c_2))) \\
&(\textbf{begin} \\
&\quad (\mathsf{set\text{-}cell!}\ c_1\ 1)\ \ (\mathsf{set\text{-}cell!}\ c_2\ 2) \\
&\quad (\textbf{let } (e\ (\stackrel{?}{=}\ (\mathsf{deref}\, c_1)\ 2)) \\
&\qquad (\textbf{begin} \\
&\qquad\quad (\mathsf{set\text{-}cell!}\ c_1\ x_1)\ \ (\mathsf{set\text{-}cell!}\ c_2\ x_2) \\
&\qquad\quad e)))),
\end{aligned}$$

where $\stackrel{?}{=}$ is the usual equality predicate on natural numbers.

The inverse simulation of assignments with cells is more difficult. If assignable variables are simply bound to cells, then every occurrence of a variable must be dereferenced. This, however, requires a parsing and re-structuring of the entire (lexical) variable scope. Because of this relationship, we have chosen to adopt a simple assignment abstraction as our basis for the programming language.

The implementation strategy for single-value cells generalizes to full Lisp-cons cells. The functions set-car! and set-cdr! are represented by σ-capabilities and permit the assignment of a new value to the selected field of a cons-cell. These operations differ from set-cell! in that they return the new cell upon completion. This self-

[4] This solution is indicated in Steele and Sussman's treatise on interpreters and modularity [65:32].

referential effect is accomplished with recursion:

$$\begin{aligned}
\mathsf{NIL} &\stackrel{df}{\equiv} [\mathsf{True}, \mathsf{False}],\\
\mathsf{cons} &\stackrel{df}{\equiv} \lambda vl.\mathsf{Y}_v(\lambda c.[\mathsf{False}, [v, l, (\sigma v.c), (\sigma l.c)]_4]),\\
\mathsf{car} &\stackrel{df}{\equiv} \lambda l.(l)_2\pi^4_1,\ \mathsf{set\text{-}car!} \stackrel{df}{\equiv} \lambda l.(l)_2\pi^4_3,\\
\mathsf{cdr} &\stackrel{df}{\equiv} \lambda l.(l)_2\pi^4_2,\ \mathsf{set\text{-}cdr!} \stackrel{df}{\equiv} \lambda l.(l)_2\pi^4_4,\\
\mathsf{null?} &\stackrel{df}{\equiv} \lambda l.(l)_1.
\end{aligned}$$

The implementation of cells and cons-cells indicates how arbitrary data objects with internal state may be built in Idealized Scheme. Furthermore, it is well-known that a programmer can create circular lists with such cons-cells. These structures may serve an interesting purpose and their existence underlies our concern about store-comparisons as raised in the previous section.

The existence of circular or self-referential structures also offers an implementation technique for recursive functions. The essence of a recursive function is—as discussed in Section 2.5—the self-reference of the function body (via a name). One way to achieve this effect is by self-application as in the Y_v-combinator. Assignments offer another possibility. Suppose the defining function is evaluated in a block where the function name is bound to a dummy value. If we immediately assign the resulting function to the function name, the references to the function name in the function body are the required self-references. The imperative implementation of the **rec**-form thus becomes:

$$(\textbf{rec!}\ (f\ x)\ = Body_{fx}) \stackrel{df}{\equiv} (\textbf{let}\ (f\ \mathsf{I})(\textbf{begin}\ f := \lambda x.Body_{fx}\ \textbf{result}\ f))$$

or, without additional syntax,

$$(\textbf{rec!}\ (f\ x)\ = Body_{fx}) \stackrel{df}{\equiv} (\lambda f.(\sigma f.f)(\lambda x.Body_{fx}))\mathsf{I}.$$

Alternatively, a different, imperative fixpoint combinator can be defined. It also takes a defining functional and returns the defined recursive function. With the new **rec**-form, we can define this combinator as

$$\mathsf{Y}_! \stackrel{df}{\equiv} \lambda f.(\mathbf{rec!}\ (g\ x)\ =\ (fgx)) \equiv \lambda f.(\lambda g.(\sigma g.g)(\lambda x.fgx))|.$$

When applied to a functional, $\mathsf{Y}_!$ constructs a pseudo-fixpoint of the function f with the imperative version of **rec**, calls it g, and passes g to the defining functional f. On an ordinary sequential machine, building and using recursive functions is faster with $\mathsf{Y}_!$ than with Y_v, because variable lookups and assignments are cheaper than function calls. Since the application of f to its fixpoint produces the required recursive function, the $\mathsf{Y}_!$-combinator should behave like Y_v, *i.e.*, for an operational equivalence $\simeq$, $\mathsf{Y}_!$ should satisfy

$$\mathsf{Y}_! F x \simeq F(\mathsf{Y}_! F)x.$$

We shall demonstrate in Chapter 6 that the calculus for Idealized Scheme can indeed prove this equation.

A noticeable difference between the $\mathsf{Y}_!$-example and the cell-example is the use of assignable variables. For cells and objects in general, the variable binds together a set of functions that share the state of the variable. That is, the assignable variable is hidden in the common lexical scope of some functions. The value associated with the variable determines the state of the functions,[5] and a change of the value that belongs to the variable signals a state transition. This is particularly convenient for modeling mathematical state variables or real-world objects. We therefore call these variables *state variables.*

[5] This makes them "non-mathematical" functions.

The $Y_!$-combinator uses assignments in a different way. For the single assignment in the λ-body, the assignable variable mimics a certain relationship for some expression, but this relationship is only established *after* the assignment takes place. Once the assignment has happened, the value association of the variable is fixed. This use of assignable variables and assignment reflects a lack of facilities for expressing certain syntactic relationships. Since there are an indefinite number of such relationships, it is better to accept the basic language and to require a facility for extending language syntax. We refer to this type of variable and assignment as *single-assignment variable.*[6]

Up to this point, all example programs have taken advantage of either assignments or $\mathcal{F}$-applications but not of both. There are also many examples where assignable variables play an important role in conjunction with continuation-accessing operations. Prominent representatives are implementations of backtracking as in logic programming [27], of co-routines [29], and of intelligent backtracking [22]. They all share a common pattern, namely, the need of a variable that contains the current and possibly changing control-state of the respective object, *e.g.*, a clause or a co-routine.

Unfortunately, while being practical and elegant at the same time, these examples are still too big for a demonstration of this technique to a novice. To avoid size problems, we discuss the implementation of two simple constructs, which have short, continuation-based implementations. The first example is a looping facility that has both a straightforward functional equivalent and an imperative realization. Given both in the same formalism, it is possible to compare programming styles, efficiency, and correctness proofs. The second example is derived from the

[6] Although this name is consistent with the literature, it is inaccurate considering such examples as import-by-need [16] where *two* assignments are needed for establishing the correct relationship.

generator concept in the programming language Icon [24]. The two examples reflect the above discussion on assignable variables. Whereas the looping facility needs a single-assignment variable, the generator abstraction requires a true state variable in order to model the changing control-state of the generator object.

The chosen looping example is the **iterate-until**-loop for the iteration of a function over a value until a certain condition holds of the current value. The syntax of this loop is

$$(\textbf{iterate } F \textbf{ over } V \textbf{ until } P).$$

F is a pure function and V a value such that F^iV is defined for all $i \geq 0$. P is a pure predicate that is defined on F^iV for all $i \geq 0$. Given this, the result of the construct should be

$$F^mV \qquad \text{where } m = min\{i \geq 0 \mid P(F^iV) = \mathsf{True}\}.$$

A recursive implementation of the **iterate-until**-loop is only a minor variation on the implementation of **iterate-times** and we state it without further explanation:

$$(\textbf{iterate } F \textbf{ over } V \textbf{ until } P) \stackrel{df}{\equiv} (\textbf{rec } (l\ v)\ (\textbf{if } (Pv)\ v\ l(Fv)))V.$$

This implementation has an important characteristic: the application of the loop-function l is a tail-call. This means, that with respect to this construct, there is nothing left to do after the application is evaluated. A good compiler translates this into a fast label-**goto**-statement à la

$$\ldots\ x := V;\ L : \textbf{if } (Px) \textbf{ then skip else begin } x := Fx;\ \textbf{goto } L \textbf{ end};\ldots,$$

where x is the result register.

The **goto**-translation provides the basic idea for an implementation with continuations. Semantically, labeling a statement is equivalent to accessing and remembering the current continuation; a **goto** to a label is an invocation of the respective

continuation with a simultaneous elimination of the current continuation. Our transliteration of this into $\Lambda_{\mathcal{F}\sigma}$ is

$$(\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P) \stackrel{df}{\equiv} (\textbf{let}\ (l\ \mathsf{I})\ (\textbf{let}\ (x\ \mathcal{F}(\sigma l.lV))\ (\textbf{if}\ (Px)\ x\ (\textbf{throw}\ l\,(Fx))))).$$

The label l is a variable that contains the continuation representing the label. The continuation rebinds x to a new value, initially V, then tests this value on the property P, and finally, depending on the outcome of the test, returns the value x as the result or invokes itself on Fx. During the i–th invocation of the continuation, x is bound to F^iV, and hence, this implementation should return the correct value. This argument will be the basis of our correctness proof in Chapter 6.

Generators are less common than looping facilities in traditional programming languages, but they become increasingly popular in special purpose high-level languages. One example is Icon [24], which provides the possibility of generating consecutively the elements of a list or the components of a vector. Take, for example, the vector $vec \equiv [1, 2, 3]$. If we let G stand for a generator of vec, then three calls to G return 1, 2, and 3. More generally, if (**generator** n vec) returns a generator G for a vector vec of length n, then for all vectors $[V_1, \ldots, V_n]$ the facility must satisfy

$$[V_1, \ldots, V_n]_n \simeq (\textbf{let}\ (G\ (\textbf{generator}\ n\ [V_1, \ldots, V_n]_n))\ [(G\mathsf{I}), \ldots, (G\mathsf{I})]_n).$$

The sequence of applications $(G\mathsf{I})$ indicates that G's result is independent of its dummy argument.

The independence of the generator from its argument makes it obvious that it is not an ordinary (mathematical) function. Still, there are many different and feasible implementation strategies for generators in an imperative higher-order language.

Since we are interested in the general technique, we present a continuation-based solution that is easily adapted to similar constructs.

The idea behind our solution is simple. From the perspective of the internal control flow, a generator is not called like a function, but is resumed at the place where it had been at the end of the last resumption. It follows that a generator return not only provides a value to the caller, but that it also remembers the current control context of the generator. If we assume the existence of a syntactic form

$$(\textbf{tocaller } V),$$

which accomplishes a generator return with value V, the core of the generator code is rather simple:

$$(\textbf{begin } (\textbf{tocaller } (\pi_1^n \; vec)) \ldots (\textbf{tocaller } (\pi_n^n \; vec))).$$

It is a sequence of n returns, each picking the correct vector component.

For the rest of the program, the generator must behave like a resume function:

$$\lambda d.(\textbf{resume } \mathsf{l}),$$

where **resume** is the dual syntactic form to **tocaller**. Also, if we assume an appropriate initialization, the code for creating a generator can return this resume function as the 0-th result. Given that the **begin**-expression already accomplishes the appropriate sequencing for the **tocaller**-expressions, we can insert this 0-th call at the beginning:

$$(\textbf{begin } (\textbf{tocaller } \lambda d.(\textbf{resume } \mathsf{l}))(\textbf{tocaller } (\pi_1^n \; vec)) \ldots (\textbf{tocaller } (\pi_1^n \; vec))).$$

The implementation problem is now reduced to the realization of two syntactic forms: **resume** and **tocaller**. Both must remember a control-state, and hence,

must be associated with two state variables: c for the control-state of the caller, g for the control-state of the generator. Assuming the existence of the two state variables, the two forms are almost obvious: each grabs the current control context, assigns it to its control-state variable, and finally resumes the thread of control associated with the other control-state variable. In Idealized Scheme this is expressed as

$$(\textbf{tocaller } V) \stackrel{df}{\equiv} \mathcal{F}(\sigma g.cV)$$

$$(\textbf{resume } V) \stackrel{df}{\equiv} \mathcal{F}(\sigma c.gV).$$

Because of our previous assumption that **tocaller** can be used at definition time, the generator must accordingly set up the control-state variable c. The generator's control-state variable is uninteresting at definition time and is initialized by the **resume**-construct. Putting things together, we obtain the code for a generator abstraction:

$$\begin{aligned}(\textbf{generator } n\ vec) \stackrel{df}{\equiv} \mathcal{F}(\lambda c.(\textbf{let } (g\ \mathsf{I})\\ (\textbf{begin}\\ (\textbf{tocaller } \lambda d.(\textbf{resume } \mathsf{I}))\\ (\textbf{tocaller } (\pi_1^n\ vec)) \ldots (\textbf{tocaller } (\pi_n^n\ vec)))))),\end{aligned}$$

where g and c are bound variables for the **tocaller** and **resume** abbreviation.

It is again relatively easy to convince oneself that at the i-th **resume**-switch to a generator, the control variable g continues with the i-th step in the **begin**-sequence. Since the i-th step is a **tocaller**-switch with the i-th component of the vector, the correctness of this implementation is almost obvious.

The preceding correctness arguments for the imperative implementation of the **iterate-until**-loop and the **generator**-abstraction sound convincing. Yet, they are not formal, testable proofs, and proofs are what we are really asking for. This demand for formal proofs of imperative higher-order programs makes the motivation for a calculus more concrete. Its development is the topic of the next two chapters.

4. From the CESK-Machine to a Program Rewriting System

The CESK-machine uses environments, stores, and continuation codes in addition to terms for the evaluation of programs. A reasoning system for Idealized Scheme programs, on the other hand, should relate programs directly to each other. Hence, our first goal should be to construct a program-oriented rewriting semantics that is extensionally equivalent to an appropriately instantiated CESK-evaluation function. It should then be possible to extend this rewriting semantics into an equational theory.

The construction of a rewriting semantics and the method underlying the construction is the result of this chapter. The central idea behind the construction is to eliminate one machine component at a time. The first two steps are devoted to environments and continuation codes. With some modifications to the transition function, they can be removed easily. Although the complete elimination of stores is impossible, a merging of the term and the store component solves the "garbage"- and "isomorphism"-problems from the preceding chapter in a rather natural manner.

4.1. Eliminating the Environment

The environment of a CESK-machine state maps free variables in the control string to their meaning. The meaning of an identifier is a location, which in turn associates

the identifier to a (current) value. Environments play a similar role in the evaluation of logical formulae. However, in logic it is well-known that environments are actually unnecessary for determining the value of a term or formula. The process of quasi-substitution[1] works equally well. When quasi-substitution is used, the meaning function replaces free identifiers by their meanings and evaluates this semantically enriched term instead of remembering the associations in a separate environment argument.

An adaptation of this idea to the CESK-machine means that transition steps which extend the environment must place locations in the control string. This naturally requires some changes in the computational domains. All control string-environment pairs are merged into control strings that contain locations. The control string domain is appropriately extended to the language $\Lambda_{\mathcal{F}\sigma Locs}$. The substitution function is also adapted to work on the new control language. Since the machine states are now triples of control strings, stores, and continuations, the resulting machine is called CSK-machine. Its computational domains are formalized in Definition 4.1, part I; its transition and evaluation function are defined in part II.

The comparison of the CESK- and the CSK-machine is based on translations that map states and state components from the old domains to the new ones. The principal translation takes control string-environment pairs to $\Lambda_{\mathcal{F}\sigma Locs}$-terms by replacing all free variables with their meaning:

$$\mathsf{R}(\langle M, \rho\rangle) = M[x_1 := \rho(x_1)] \ldots [x_n := \rho(x_n)] \quad \text{where } FV(M) = \{x_1, \ldots, x_n\},$$

$$\mathsf{R}(\langle \ddagger, \emptyset\rangle) = \ddagger;$$

the second clause is added for technical reasons.

Furthermore, since control string-environment pairs also occur in continuation

[1] The term *quasi-substitution* was coined by Michael Dunn; the concept of value-containing syntactic entities, so-called U-formulae, was invented by Raymond Smullyan [58].

Definition 4.1: The CSK-machine, part I: the computational domains

Computational Domains:

$$
\begin{array}{lll}
s \in \mathit{States} & = \mathit{Controls} \times \mathit{Stores} \times \mathit{Conts} & \text{(machine states)} \\
c \in \mathit{Controls} & = \Lambda_{\mathcal{F}\sigma\mathit{Locs}} + \ddagger & \text{(control strings)} \\
\theta \in \mathit{Stores} & = \mathit{Locs} \multimap \mathit{Closures} & \text{(stores)} \\
V \in \mathit{Closures} & = \mathit{Values} + \mathbf{p} \times \mathbf{p}\text{-}\mathit{Conts} & \text{(closures)} \\
\kappa \in \mathit{Conts} & = \mathbf{ret}\text{-}\mathit{Conts} + \mathbf{p}\text{-}\mathit{Conts} & \text{(continuation codes)}
\end{array}
$$

p-continuations and **ret**-continuations are

$$
\begin{aligned}
\mathbf{p}\text{-}\mathit{Conts} &= \mathbf{stop} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{fun} \times \mathit{Closures} + \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{arg} \times \Lambda_{\mathcal{F}\sigma\mathit{Locs}} \\
\mathbf{ret}\text{-}\mathit{Conts} &= \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{ret} \times \mathit{Closures},
\end{aligned}
$$

and $\Lambda_{\mathcal{F}\sigma\mathit{Locs}}$ is the following variation of $\Lambda_{\mathcal{F}\sigma}$:

$$M ::= a \mid x \mid n \mid (\lambda x.M) \mid (MN) \mid (\mathcal{F}M) \mid (\sigma X.M),$$

where $n \in \mathit{Locs}$. X ranges over both *Vars* and *Locs*.

Values contains constants, variables, abstractions, and capabilities.

Auxiliary Functions:

$$
\begin{aligned}
\cdot \otimes \cdot &: \mathbf{p}\text{-}\mathit{Conts} \times \mathbf{p}\text{-}\mathit{Conts} \longrightarrow \mathbf{p}\text{-}\mathit{Conts} \\
\kappa \otimes (\mathbf{stop}) &= \kappa \\
\kappa \otimes (\kappa' \ \mathbf{arg}\ N) &= (\kappa \otimes \kappa' \ \mathbf{arg}\ N) \\
\kappa \otimes (\kappa' \ \mathbf{fun}\ V) &= (\kappa \otimes \kappa' \ \mathbf{fun}\ V)
\end{aligned}
$$

$$
\begin{aligned}
\cdot[\cdot := \cdot] &: \Lambda_{\mathcal{F}\sigma\mathit{Locs}} \times \mathit{Vars} \times \mathit{Locs} \longrightarrow \Lambda_{\mathcal{F}\sigma\mathit{Locs}} \\
a[x := n] &\equiv a, \\
x[x := n] &\equiv n, \quad y[x := n] \equiv y \qquad (x \not\equiv y), \\
n[x := n] &\equiv n, \\
(\lambda y.M)[x := n] &\equiv (\lambda y.M[x := n]), \\
(LM)[x := n] &\equiv (L[x := n]M[x := n]), \\
(\mathcal{F}M)[x := n] &\equiv (\mathcal{F}M[x := n]), \\
(\sigma X.M)[x := n] &\equiv (\sigma X[x := n].M[x := n]).
\end{aligned}
$$

Definition 4.1: The CSK-machine, part II: the transition function

The CSK-transition function maps states to states:

$$States \overset{CSK}{\longmapsto} States,$$

according to the following cases:

$$\langle a, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa\ \mathbf{ret}\ a)\rangle \tag{0}$$

$$\langle n, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa\ \mathbf{ret}\ \theta(n))\rangle \tag{1}$$

$$\langle \lambda x.M, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa\ \mathbf{ret}\ \lambda x.M)\rangle \tag{2}$$

$$\langle MN, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle M, \theta, (\kappa\ \mathbf{arg}\ N)\rangle \tag{3}$$

$$\langle \ddagger, \theta, ((\kappa\ \mathbf{arg}\ N)\ \mathbf{ret}\ V)\rangle \overset{CSK}{\longmapsto} \langle N, \theta, (\kappa\ \mathbf{fun}\ V)\rangle \tag{4}$$

$$\langle \ddagger, \theta, ((\kappa\ \mathbf{fun}\ \lambda x.M)\ \mathbf{ret}\ V)\rangle \overset{CSK}{\longmapsto} \langle M[x := n], \theta[n := V], \kappa\rangle \quad \text{where } n \notin Dom(\theta) \tag{5}$$

$$\langle \ddagger, \theta, ((\kappa\ \mathbf{fun}\ f)\ \mathbf{ret}\ a)\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa\ \mathbf{ret}\ \delta(f, a))\rangle \tag{6}$$

$$\langle \mathcal{F}M, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle Mn, \theta[n := \langle \mathbf{p}, \kappa\rangle], (\mathbf{stop})\rangle \quad \text{where } n \notin Dom(\theta) \tag{7}$$

$$\langle \ddagger, \theta, ((\kappa\ \mathbf{fun}\ \langle \mathbf{p}, \kappa_0\rangle)\ \mathbf{ret}\ V)\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa \otimes \kappa_0\ \mathbf{ret}\ V)\rangle \tag{8}$$

$$\langle \sigma n.M, \theta, \kappa\rangle \overset{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa\ \mathbf{ret}\ \sigma n.M)\rangle \tag{9}$$

$$\langle \ddagger, \theta, ((\kappa\ \mathbf{fun}\ \sigma n.M)\ \mathbf{ret}\ V)\rangle \overset{CSK}{\longmapsto} \langle M, \theta[n := V], \kappa\rangle \tag{10}$$

The *eval*-function is

$$eval_{CSK,\mathsf{Unload}}(M) = \mathsf{Unload}(V, \theta) \text{ iff}$$
$$\langle M, \emptyset, (\mathbf{stop})\rangle \overset{CSK^+}{\longmapsto} \langle \ddagger, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle.$$

codes and stores, we need auxiliary translations between CESK-continuations and CSK-continuations:

$$\begin{aligned}
\mathsf{R}_k((\mathbf{stop})) &= (\mathbf{stop}),\\
\mathsf{R}_k((\kappa\ \mathbf{arg}\ N\rho)) &= (\mathsf{R}_k(\kappa)\ \mathbf{arg}\ \mathsf{R}(\langle N,\rho\rangle)),\\
\mathsf{R}_k((\kappa\ \mathbf{fun}\ V)) &= (\mathsf{R}_k(\kappa)\ \mathbf{fun}\ \mathsf{R}(V)),\\
\mathsf{R}_k((\kappa\ \mathbf{ret}\ V)) &= (\mathsf{R}_k(\kappa)\ \mathbf{ret}\ \mathsf{R}(V));
\end{aligned}$$

and between CESK-stores and CSK-stores:

$$\mathsf{R}_s(\theta) = \{\langle n, \mathsf{R}(\theta(n))\rangle\}.$$

And finally, in order to take care of **p**-closures in the store, we must add the clause

$$\mathsf{R}(\langle \mathbf{p}, \kappa\rangle) = \langle \mathbf{p}, \mathsf{R}_k(\kappa)\rangle$$

to the definition of R.

Together, these definitions yield a map from CESK-states to CSK-states that preserve initial, final, and stuck states. With these, we can express in what sense the CESK-machine is equivalent to the CSK-machine:

Lemma 4.2 (CSK-simulation). *For any program $M \in \Lambda_{\mathcal{F}\sigma}$,*

$$eval_{CESK,\mathsf{UnloadR}}(M) = eval_{CSK,\langle\cdot,\cdot\rangle}(M),$$

where $\mathsf{UnloadR}(V,\theta) = \langle \mathsf{R}(V), \mathsf{R}_s(\theta)\rangle$ *and* $\langle\cdot,\cdot\rangle$ *is the pairing function.*

Proof. The proof is an induction on the number of transition steps in an evaluation. The basic claim is that the i-th state in a CESK-evaluation maps to the i-th state in the corresponding CSK-evaluation via the translations R, R_k, and R_s. This is true for initial states where $\mathsf{R}(\langle M, \emptyset\rangle) = M$. For the induction step, consider the CESK-transition step:

$$\langle c_1, \rho_1, \theta, \kappa_1\rangle \overset{CESK}{\longmapsto} \langle c_2, \rho_2, \theta, \kappa_2\rangle.$$

By case analysis, it is easy to show that

$$\langle \mathsf{R}(\langle c_1, \rho_1 \rangle), \mathsf{R}_s(\theta), \mathsf{R}_k(\kappa_1) \rangle \stackrel{CSK}{\longmapsto} \langle \mathsf{R}(\langle c_2, \rho_2 \rangle), \mathsf{R}_s(\theta), \mathsf{R}_k(\kappa_2) \rangle$$

and thus, the claimed relationship is an invariant of transition steps. The result follows from the property that R preserves final and stuck states. □

Before leaving this section, we simplify the transition function in order to shorten the following machine definitions. All transition rules that evaluate syntactic values, *i.e.*, (CSK0), (CSK2), and (CSK9), are now of the same form

$$\langle V, \theta, \kappa \rangle \stackrel{CSK}{\longmapsto} \langle \ddagger, \theta, (\kappa \, \mathbf{ret} \; V) \rangle. \qquad \text{(CSK. } \textit{Values}\text{)}$$

Consequently, we can replace the three separate rules in the definition of the transition function with this new rule.

4.2. Eliminating the Continuation Code

The expected next step should be an incorporation of continuation codes into the control string components. However, the notion of CSK-continuation codes is too machine-oriented for a direct transition. The first transformation is therefore a replacement of continuation codes by a more familiar, term-related concept. Then the elimination of machine continuations follows quite naturally.

4.2.1. Contexts as Continuations

An inspection of the sample CESK-evaluations in Chapter 3 reveals that the continuation code is a control string memory. Those pieces of the control string that are of no interest to the current computation phase are shifted to the continuation stack until a need arises for reconsidering them. From the perspective of the program as a whole, the machine searches through applications until it finds a value in the left part of an application. This is also true for $\mathcal{F}$-applications which are immediately

replaced by proper applications. When the machine discovers a value, it remembers this on the continuation stack and concentrates its search on the argument position of the respective application. Once an application with two values—a *redex*[2]—has been found, a transition step replaces the redex by a new term. If the new term is a value, the machine inspects the continuation for further instructions and eventually continues the search. Otherwise, if the new term is an application, the cycle repeats immediately.

Definition 4.3: The CSC-machine, part I: the computational domains

Computational Domains:

$$V \in \mathit{Closures} = \mathit{Values} + \mathbf{p} \times \mathit{AppSConts} \quad \text{(closures)}$$
$$\kappa \in \mathit{Conts} \quad = \mathbf{ret}\text{-}\mathit{Conts} + \mathit{AppSConts} \quad \text{(continuation codes)}$$

where $\mathit{AppSConts}$ is the set of applicative standard contexts:

$$C[\ ] ::= [\ ] \mid VC[\ ] \mid C[\ ]M$$

with M ranging over $\Lambda_{\mathcal{F}\sigma\mathit{Locs}}$, V over closures,
and **ret**-continuations are

$$\mathbf{ret}\text{-}\mathit{Conts} = \mathit{AppSConts} \times \mathbf{ret} \times \mathit{Closures}.$$

All other CSK-domains become CSC-domains without further changes.

While searching through the term for the next redex, the machine partitions the term into a *context* and a current search area. Eventually, this search area will be narrowed down to a redex. In the mean time, the context of this redex is shifted

[2] We do not refer to the entire machine state as redex, but to parts of the program component. This should not cause any confusion and is in accordance with the calculus terminology.

to the continuation component, and hence, term contexts may as well represent continuation codes. Not surprisingly, the concept of an applicative standard context captures the dynamic character of the search phase in the correct way. The context hole corresponds to the current search area. Initially, the entire term is the search area and the empty context [ ] can serve as the respective continuation. If a value V is in the left part of an application VN, the search moves to the right, *i.e.*, the current context $C[\ ]$ becomes $C[V[\ ]]$. Finally, if an application MN with an application M in the left position is encountered, the search space is narrowed down to the left part: $C[[\ ]N]$ represents the new continuation where $C[\ ]$ is the old one.

The informal explanation of the correspondence between **p**-continuations and applicative standard contexts demonstrates that an inside-out definition of contexts is more appropriate for the use of applicative standard contexts as continuations:

$$C[\ ] ::= [\ ] \mid C[[\ ]M] \mid C[V[\ ]],$$

but, of course, an inductive argument immediately proves this modification equivalent to the original definition. We shall use whichever definition is preferable.

The definition of applicative standard context requires two extensions. First, since the control string language of the CSK-machine consists of expressions in $\Lambda_{\mathcal{F}\sigma Locs}$, all terms in a context must be in this term set. Second, values to the left of the path from the context-root to the context-hole may be **p**-closures because the machine evaluates syntactic entities to closures by putting them into the continuation. The formal definition of applicative contexts, the new value and continuation domains, and the transition function are displayed in Definition 4.3.

Roughly speaking, the CSC-machine is like the CSK-machine except for the replacement of **p**-continuations by contexts. A formalization of the correspondence is almost straightforward and is primarily based on a translation from continuation codes to applicative standard contexts:

Definition 4.3: The CSC-machine, part II: the transition function

The CSC-transition function maps states to states:

$$States \overset{CSC}{\longmapsto} States,$$

according to the following cases:

$$\langle V, \theta, C[\ ]\rangle \overset{CSC}{\longmapsto} \langle \ddagger, \theta, (C[\ ]\,\mathbf{ret}\ V)\rangle \qquad (0,2,9)$$

$$\langle n, \theta, C[\ ]\rangle \overset{CSC}{\longmapsto} \langle \ddagger, \theta, (C[\ ]\,\mathbf{ret}\ \theta(n))\rangle \qquad (1)$$

$$\langle MN, \theta, C[\ ]\rangle \overset{CSC}{\longmapsto} \langle M, \theta, C[[\ ]N]\rangle \qquad (3)$$

$$\langle \ddagger, \theta, (C[[\ ]N]\,\mathbf{ret}\ V)\rangle \overset{CSC}{\longmapsto} \langle N, \theta, C[V[\ ]]\rangle \qquad (4)$$

$$\langle \ddagger, \theta, (C[(\lambda x.M)[\ ]]\,\mathbf{ret}\ V)\rangle \overset{CSC}{\longmapsto} \langle M[x := n], \theta[n := V], C[\ ]\rangle \quad \text{where } n \notin Dom(\theta) \qquad (5)$$

$$\langle \ddagger, \theta, (C[f[\ ]]\,\mathbf{ret}\ a)\rangle \overset{CSC}{\longmapsto} \langle \ddagger, \theta, (C[\ ]\,\mathbf{ret}\ \delta(f,a))\rangle \qquad (6)$$

$$\langle \mathcal{F}M, \theta, C[\ ]\rangle \overset{CSC}{\longmapsto} \langle Mn, \theta[n := \langle \mathbf{p}, C[\ ]\rangle], [\ ]\rangle \quad \text{where } n \notin Dom(\theta) \qquad (7)$$

$$\langle \ddagger, \theta, (C[\langle \mathbf{p}, C_0[\ ]\rangle[\ ]]\,\mathbf{ret}\ V)\rangle \overset{CSC}{\longmapsto} \langle \ddagger, \theta, (C[C_0[\ ]]\,\mathbf{ret}\ V)\rangle \qquad (8)$$

$$\langle \ddagger, \theta, (C[(\sigma n.M)[\ ]]\,\mathbf{ret}\ V)\rangle \overset{CSC}{\longmapsto} \langle M, \theta[n := V], C[\ ]\rangle. \qquad (10)$$

The *eval*-function is

$$eval_{CSC,\mathsf{Unload}}(M) = \mathsf{Unload}(V,\theta) \text{ iff } \langle M, \emptyset, [\ ]\rangle \overset{CSC^+}{\longmapsto} \langle \ddagger, \theta, ([\ ]\,\mathbf{ret}\ V)\rangle.$$

$$\left.\begin{aligned} \mathsf{S}((\mathbf{stop})) &= [\ ] \\ \mathsf{S}((\kappa\,\mathbf{arg}\ N)) &= C[[\ ]N] \\ \mathsf{S}((\kappa\,\mathbf{fun}\ V)) &= C[\mathsf{S}_c(V)[\ ]] \end{aligned}\right\} \text{ where } \mathsf{S}(\kappa) = C[\ ].$$

The auxiliary translation S_c is needed to replace **p**-continuation codes by contexts

within **p**-closures:

$$\mathsf{S}_c(\langle \mathbf{p}, \kappa \rangle) = \langle \mathbf{p}, \mathsf{S}(\kappa) \rangle, \text{ and } \mathsf{S}_c(V) = V \qquad \text{if } V \in \mathit{Values}.$$

In stores, **p**-closures must be transformed by a pointwise application of S_c:

$$\mathsf{S}_s(\theta) = \{\langle n, \mathsf{S}_c(\theta(n)) \rangle\}.$$

The correctness lemma is of the familiar form:

Lemma 4.4 (CSC-simulation). *For any program* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$\mathit{eval}_{CESK,\mathsf{UnloadS}}(M) = \mathit{eval}_{CSC,\langle\cdot,\cdot\rangle}(M),$$

where $\mathsf{UnloadS}(V, \theta) = \langle \mathsf{S}(V), \mathsf{S}_s(\theta) \rangle$.

Proof. The proof follows the same pattern as the one for Lemma 4.2, with the difference that we must show:

$$\langle c_1, \theta_1, \kappa_1 \rangle \overset{CSK}{\longmapsto} \langle c_2, \theta_2, \kappa_2 \rangle$$

implies

$$\langle c_1, \mathsf{S}_s(\theta_1), \mathsf{S}(\kappa_1) \rangle \overset{CSK}{\longmapsto} \langle c_2, \mathsf{S}_s(\theta_2), \mathsf{S}(\kappa_2) \rangle.$$

This part is a case-by-case comparison of (CSKi) with (CSCi). □

4.2.2. Merging Control Strings with Contexts

The key idea for merging context-continuations and control strings into a single component is simple. If we ignore the CSC-store for a moment, contexts and control strings are always paired in each clause of the transition function. More precisely, on the left-hand and right-hand sides of all CSC-transition rules there are only two classes of states:

$$\langle M, \cdot, C[\ ] \rangle \text{ and } \langle \ddagger, \cdot, (C[\ ]\ \mathbf{ret}\ V) \rangle.$$

It is therefore quite natural to fold the two components together by filling the context holes with the respective terms, *i.e.*, the two prototypical states become

$$\langle C[M], \cdot \rangle \text{ and } \langle C[V], \cdot \rangle.$$

A naïve transformation of the CSC-transition function according to this idea only produces a transition *relation* because the left-hand sides of (CSC.*Values*), (CSC3), and (CSC4) become identical to the right-hand sides. The reason is that these rules are context search rules. Whereas a continuation-free transition function assumes an *a priori* partitioning of a control string into a redex and an applicative standard context, the CSC-function is explicitly searching for the next redex. This latter strategy is closer to an actual machine implementation, but inappropriate for human consumption. Hence, this kind of rule is rendered superfluous during a transformation of a machine into a program-oriented rewriting system. Eliminating them from the naïvely constructed transition system yields the desired continuation-free machine.

The new CS-machine and its transition function are formalized in Definition 4.5. Since the set of semantic values includes **p**-closures, the merging of the control string and continuation components of CSC-states requires the inclusion of **p**-closures in the (syntactic) value set of the control string language. With this modification, the store is simplified to a finite map from locations to syntactic values.

The simulation translation is a formal expression of the informal key observation. It maps control strings and continuation contexts to terms in the new control string language:

$$\mathsf{T}(\ddagger, (C[\ ]\ \mathbf{ret}\ V)) = C[V],$$
$$\mathsf{T}(M, C[\ ]) = C[M].$$

The correctness of this transformation is expressed in

Definition 4.5: The CS-machine

Computational Domains:

$s \in States = \Lambda_{\mathcal{F}\sigma Locs\, p} \times Stores$ (machine states)
$\theta \in Stores = Locs \multimap Values_p$ (stores)

where $\Lambda_{\mathcal{F}\sigma Locs\, p}$ is

$$M ::= a \mid x \mid n \mid (\lambda x.M) \mid (MN) \mid (\mathcal{F}M) \mid (\sigma x.M) \mid \langle \mathbf{p}, C[\ ] \rangle,$$

and $Values_p$ includes constants, variables, abstractions, capabilities, and **p**-closures.

The CS-transition function maps states to states:

$$States \xmapsto{CS} States,$$

according to the following cases where applicative standard contexts are defined over $\Lambda_{\mathcal{F}\sigma Locs\, p}$:

$$\langle C[n], \theta \rangle \xmapsto{CS} \langle C[\theta(n)], \theta \rangle \tag{1}$$

$$\langle C[(\lambda x.M)V], \theta \rangle \xmapsto{CS} \langle C[M[x := n]], \theta[n := V] \rangle \quad \text{where } n \notin Dom(\theta) \tag{5}$$

$$\langle C[fa], \theta \rangle \xmapsto{CS} \langle C[\delta(f, a)], \theta \rangle \tag{6}$$

$$\langle C[\mathcal{F}M], \theta \rangle \xmapsto{CS} \langle Mn, \theta[n := \langle \mathbf{p}, C[\ ] \rangle] \rangle \quad \text{where } n \notin Dom(\theta) \tag{7}$$

$$\langle C[\langle \mathbf{p}, C_0[\ ] \rangle V], \theta \rangle \xmapsto{CS} \langle C[C_0[V]], \theta \rangle \tag{8}$$

$$\langle C[(\sigma n.M)V], \theta \rangle \xmapsto{CS} \langle C[M], \theta[n := V] \rangle \tag{10}$$

The *eval*-function is

$$eval_{CS,\mathsf{Unload}}(M) = \mathsf{Unload}(V, \theta) \text{ iff } \langle M, \emptyset \rangle \xmapsto{CS}{}^{*} \langle V, \theta \rangle.$$

Lemma 4.6 (CS-simulation). *For any program* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$eval_{CSC,\langle\cdot,\cdot\rangle}(M) = eval_{CS,\langle\cdot,\cdot\rangle}(M).$$

Proof. The proof idea is again the same as in the previous simulation lemmas. However, this time the lengths of the two evaluations differ by the number of search steps in the CSC-transition sequence. For the search rules, it is now the case that the CSC-machine performs a transition step:

$$\langle c_1, \theta_1, \kappa_1\rangle \overset{CSC}{\longmapsto} \langle c_2, \theta_2, \kappa_2\rangle,$$

while the CS-machine rests in the state

$$\langle \mathsf{T}(c_1, \kappa_1), \theta_1\rangle = \langle \mathsf{T}(c_2, \kappa_2), \theta_2\rangle.$$

For the remaining transitions, we show that

$$\langle c_1, \theta_1, \kappa_1\rangle \overset{CSC}{\longmapsto} \langle c_2, \theta_2, \kappa_2\rangle$$

implies

$$\langle \mathsf{T}(c_1, \kappa_1), \theta_1\rangle \overset{CS}{\longmapsto} \langle \mathsf{T}(c_2, \kappa_2), \theta_2\rangle.$$

Otherwise, the proof remains the same. □

At first glance, the simplicity of the CS-semantics versus the CESK-semantics is the result of an elimination of machine continuations in exchange for an extension of the control string language. However, a further simplification shows that this is not true. In the current version of the CS-system, an $\mathcal{F}$-application grabs the current context and packages it up into a **p**-closure; a continuation application places the argument back into the respective context hole. Yet, a context can be perceived as a term that is a function of its hole, and indeed, a **p**-closure of the form $\langle \mathbf{p}, C[\ ]\rangle$

behaves in the same way as the term $\lambda x.C[x]$. When applied to a value within some other applicative context, the value is labeled, placed into the hole of $C[\ ]$, and is then immediately delabeled. Hence, the transition rules (CS7) and (CS8) can be replaced by

$$\langle C[\mathcal{F}M], \theta\rangle \longrightarrow \langle Mn, \theta[n := (\lambda x.C[x])]\rangle. \qquad \text{(CS'7)}$$

The introduction of this rule not only reduces the number of rules, but it also eliminates the need for **p**-closures in the control string language. This shows that $\Lambda_{\mathcal{F}\sigma}$ suffices for a syntactic treatment of programming language control.

A translation T_p from $\Lambda_{\mathcal{F}\sigma Locs\,p}$ to $\Lambda_{\mathcal{F}\sigma Locs}$ is simple. T_p parses a term until it encounters a **p**-closure. Then the **p**-closure is replaced by the respective abstraction and the translation continues:

$$\mathsf{T}_p(\langle \mathbf{p}, C[\ ]\rangle) = \lambda x.\mathsf{T}_p(C[x]).$$

We omit a further formalization of this rather straightforward change in the CS-transition function and state the correctness lemma for the modified CS-machine, called CS'-machine:

Lemma 4.7 (CS'-simulation). *For any program $M \in \Lambda_{\mathcal{F}\sigma}$,*

$$eval_{CS,\mathsf{Unload}\mathsf{T}_p}(M) = eval_{CS',\langle\cdot,\cdot\rangle}(M),$$

where $\mathsf{Unload}\mathsf{T}_p(V, \theta) = \langle \mathsf{T}_p(V), \mathsf{T}_s(\theta)\rangle$ and T_s is the pointwise application of T_p to a store.

The proof of this lemma is also omitted. It essentially formalizes the above, informal argument on the equivalence of **p**-closures and terms.

After the elimination of environments and continuations we are left with a machine that is solely based on control strings and stores. This is close to the desired

rewriting semantics, but the realization of side-effects still relies on the existence of cells in the underlying implementation machine. In the following section we demonstrate that assignment can be interpreted in the syntactic world, and that this interpretation is indeed an abstraction from the cell-concept of state in programs.

4.3. Replacing the Store by Sharing Relations

The role of the store is characterized by the three transition rules (CS1), (CS5), and (CS10),[3] which use, extend, and modify the store. The crucial rule is (CS5). It replaces all bound variables of a λ-abstraction by a new, distinct location and thus gradually builds up the store. All future references to a bound variable are resolved via the always-present store: a request for the current value of a variable causes a store lookup; an instruction to change the current value results in a modified store. Hence, a deeper understanding of the store requires a closer look at the nature of bound variables.

At this point we must recall the α-congruence convention about bound variables in terms. According to this convention, the name of a bound variable is irrelevant (up to uniqueness). Abstractions like $\lambda x.x$ and $\lambda y.y$ are considered equal. This actually means that the programming language is the quotient of Λ over $\equiv_\alpha$. From this perspective, a λ-abstraction is an expression together with a relation that determines which parts of the expression are equivalent. The relationship is displayed by occurrences of the bound variable.

A unification of our ideas on the role of the CS-store and the nature of bound variables directly leads to an abstracted view of the store. The intention behind the replacement of bound variables by unique locations in the bodies of λ-abstractions is to retain the *static* α-equivalence for the rest of the computation, *i.e.*, as a *dynamic*

[3] The extension of the store in (CS'7) is an artifact of our choice of putting continuations in the store instead of the environment.

sharing relation for term positions,[4] even after the $\lambda x.$-part has disappeared. From this argument it follows that an integration of the store into the control string language necessitates a way of expressing this dynamic relationship syntactically.

The most natural solution is a labeling scheme. Instead of placing a location into the program text and the value in the store, the value could be labeled in a unique way and placed into the text as a *labeled value*.[5] With respect to the transition relation, we would like to replace

$$\langle C[(\lambda x.M)V], \theta \rangle \xrightarrow{CS} \langle C[M[x := l]], \theta[l := V] \rangle \qquad \text{where } l \notin Dom(\theta)$$

by a rule like

$$C[(\lambda x.M)V] \longmapsto C[M[x := V^l]] \text{ where } l \text{ is not used in the rest of the program.}$$

V^l is the labeled version of the value V. With the labeling technique it is indeed possible to re-interpret lookups and assignments as term manipulations.

The emulation of a variable lookup apparently strips off the label from the labeled value since the value is already sitting in the right position. The effect of an assignment relies on the introduction of a new substitution operation. In the extended term language, a σ-application looks like

$$(\sigma U^l.M)V$$

when it is about to be evaluated. The assignable variable has been replaced by a labeled value; all other related variable positions carry the same label. When it

[4] The idea of modeling the store as a sharing relation was already known to Landin [38], but his sharing relations defined equivalent machine positions, not relationships of term positions.

[5] Labeled terms have also been used for the investigation of the regular λ-calculus [5:353]. Although our problem is unrelated to these investigations we have adopted the notation in order to avoid the introduction of an entirely new concept.

is time to perform the above σ-application, all these occurrences of an l-labeled value must be replaced by l-labeled values V. To implement this, we introduce the labeled-value substitution $M[\bullet^l := V^l]$. The result of $M[\bullet^l := V^l]$ is a term that is like M except that all l-labeled subterms are replaced by V^l. With this new operation, the assignment transition is definable as

$$C[(\sigma U^l.M)V] \longmapsto C[M][\bullet^l := V^l].$$

Unfortunately, the new semantics for assignments and lookups has a minor flaw: thus far, it cannot deal with circular or self-referential assignments. When the label l appears not only in $C[M]$ but also in V, the equivalence-positions in V are not affected by the labeled-value substitution. As an illustration, let us recall our prototypical example for self-referential assignments from Section 3.2. The program $(\lambda x.(\sigma x.x)(\lambda y.x))0$ yields a circular closure on the CESK-machine. A term evaluation according to the above rules proceeds as follows:

$$(\lambda x.(\sigma x.x)(\lambda y.x))0 \longmapsto (\sigma 0^1.0^1)(\lambda y.0^1) \longmapsto (\lambda y.0^1)^1.$$

The last term should represent this circular structure and in some sense it does: an interpretation of the label 1 requires that the position occupied by 0^1 is in the same sharing equivalence class as the entire expression $(\lambda y.0^1)^1$. However, a lookup that simply strips off the label produces a non-circular object, namely, $(\lambda y.0^1)$. For a correct implementation this problem needs a solution.

There are two obvious ways out of our dilemma. First, we could blame the assignment statement and require that a self-referential assignment builds a truly circular or infinite term. Although this approach has been successfully employed in other circumstances [56], we believe that it unnecessarily complicates a symbolic rewriting semantics. Second, we can try to find a more intelligent lookup transition

that is knowledgeable about circularities. In other words, the assignment transition only updates the equivalence classes within the program and leaves the equivalence positions within the new value alone. The complementary lookup transition is then something like a by-need continuation of the latest assignment transition:

$$C[V^l] \longmapsto C[V[\bullet^l := V^l]].$$

This version of the lookup transition simultaneously strips off a label and updates all equivalence positions within the value. For the above example, this leads to the correct final value of $(\lambda y.(\lambda y.0^1)^1)$.

The intuitive reason for the correctness of the new transition rules is the following invariant: every outermost occurrence of a label is associated with the correct current value. When the label is taken off, some inner occurrences may become outermost, but they are immediately updated with the correct value. Assignments place a label on the value, and hence, all self-referential labels within the value are not outermost.

Given the informal descriptions and correctness arguments, we proceed to define the final machine in this chapter. The machine is a control string rewriting system. Its only state components are expressions in the language Λ_{rew}, which is a proper extension of $\Lambda_{\mathcal{F}\sigma}$. There are two new categories of terms: labeled values and σ-capabilities with labeled values in the variable position. Whereas the latter kind is included in the set of values, labeled values are *not*. Labeled values approximately correspond to store cells, but they are more abstract. Unlike cells, labels may be tagged at different values. For example, $\mathsf{K}^1\mathsf{I}^1\mathsf{S}^1$ is a legitimate expression in Λ_{rew} with the unique value I. No such expression represents a legal state in the evaluation of $\Lambda_{\mathcal{F}\sigma}$-programs, but its existence illustrates the difference between labels and cells: labels only denote an equivalence class for assignments, cells are one out of many possible implementation techniques for evaluations of $\Lambda_{\mathcal{F}\sigma}$-programs.

Definition 4.8: The C-rewriting system, part I: the language

The term language Λ_{rew}:

$$M ::= MN \mid \mathcal{F}M \mid V \mid V^l \qquad V ::= a \mid x \mid \lambda x.M \mid \sigma X.M$$

where l ranges over *Labels*, X over *Vars* and *LabeledValues*.
Labs, the set of labels, is an arbitrary, infinite set, *e.g.*, *Locs*.

Auxiliary Operations:

$$Lab\colon \Lambda_{rew} \longrightarrow \mathcal{P}(Labs)$$

$$Lab(MN) = Lab(M) \cup Lab(N) \qquad Lab(a) = Lab(x) = \emptyset$$
$$Lab(\mathcal{F}M) = Lab(M) \qquad Lab(\lambda x.M) = Lab(M)$$
$$Lab(V^l) = Lab(V) \cup \{l\} \qquad Lab(\sigma X.M) = Lab(X) \cup Lab(M)$$

$$\cdot[\cdot := \cdot]\colon \Lambda_{rew} \times Vars \times Values \cup LabeledValues \longrightarrow \Lambda_{rew}$$

(substitution)

$$x[x := L] = L,\ y[x := L] = y \text{ if } y \neq x,$$
$$U^m[x := L] = U[x := L]^m,$$
$$(\lambda y.M)[x := L] = \lambda y.M[x := L],$$
$$(MN)[x := L] = (M[x := L]N[x := L]),$$
$$(\mathcal{F}M)[x := L] = (\mathcal{F}M[x := L]),$$
$$(\sigma x.M)[x := V^l] = (\sigma V^l.M[x := V^l]),$$
$$(\sigma y.M)[x := L] = (\sigma y.M[x := L]) \text{ if } x \neq y,$$
$$(\sigma U^m.M)[x := L] = (\sigma U[x := L]^m.M[x := L]).$$

$$\cdot[\cdot := \cdot]\colon \Lambda_{rew} \times Labs \times Values \longrightarrow \Lambda_{rew}$$

(labeled-value substitution)

$$x[\bullet^l := V^l] = x,\ U^l[\bullet^l := V^l] = V^l,\ U^m[\bullet^l := V^l] = U[\bullet^l := V^l]^m \text{ if } l \neq m,$$
$$(\lambda x.M)[\bullet^l := V^l] = \lambda x.M[\bullet^l := V^l],$$
$$(MN)[\bullet^l := V^l] = (M[\bullet^l := V^l]N[\bullet^l := V^l]),$$
$$(\mathcal{F}M)[\bullet^l := V^l] = (\mathcal{F}M[\bullet^l := V^l]),$$
$$(\sigma X.M)[\bullet^l := V^l] = (\sigma X[\bullet^l := V^l].M[\bullet^l := V^l]).$$

The C-transition function is conventional and uses the partitioning of control strings into contexts and redexes that is known from the CS-machine. The details of the formal machine description may be found in Definition 4.8, parts I and II.

Definition 4.8: The C-rewriting system, part II: the transition function

The transition function maps programs to programs:

$$\Lambda_{rew} \overset{C}{\longmapsto} \Lambda_{rew}$$

according to the following cases:

$$C[V^l] \overset{C}{\longmapsto} C[V[\bullet^l := V^l]] \tag{1}$$

$$C[(\lambda x.M)V] \overset{C}{\longmapsto} C[M[x := V^l]] \quad \text{where } l \notin Lab(C[MV]) \tag{5}$$

$$C[fa] \overset{C}{\longmapsto} C[\delta(f,a)] \tag{6}$$

$$C[\mathcal{F}M] \overset{C}{\longmapsto} M(\lambda x.C[x])^l \quad \text{where } l \notin Lab(C[M]) \tag{7}$$

$$C[(\sigma U^l.M)V] \overset{C}{\longmapsto} C[M][\bullet^l := V^l] \tag{10}$$

The *eval*-function is

$$eval_{C,\mathsf{Unload}}(M) = \mathsf{Unload}(V) \text{ iff } M \overset{C}{\longmapsto}{}^{*} V.$$

Before we can prove the correctness of the C-rewriting system, we must explore the extension of the α-congruence relation to the new term language. In Chapter 2, we defined α-equivalence in an informal manner. Formally, α-equivalence can be characterized[6] as a compatible equivalence relation that is generated by

$$\lambda x.M \equiv_\alpha \lambda y.M[x := y] \qquad \text{if } y \notin FV(M).$$

6 See Barendregt's exposition on this topic for a fuller treatment [5:Appendix].

An extension of this relation to labeled values should be based on a simple compatibility rule that compares the value parts of labeled values. However, this is insufficient because the very same label can occur within these values. Such a self-referential occurrence indicates that the value part of the respective position is equivalent to the entire value, but for the comparison of binding relations, it is only important that the inner label occurrences mark the same value. Hence, stripping-off l and replacing inner occurrences with l—or any other arbitrary value—guarantees a well-founded and correct definition:

$$V^l \equiv_\alpha U^l \qquad \text{iff} \qquad V[\bullet^l := \mathsf{l}^{l'}] \equiv_\alpha U[\bullet^l := \mathsf{l}^{l'}].$$

An example of this extended α-equivalence relation is

$$(\lambda y.y0^l)^l \equiv_\alpha (\lambda x.x1^l)^l.$$

It simultaneously shows the irrelevance of variable names and inner label occurrences. The soundness of the extended α-equivalence is captured in the following proposition, which says that evaluating equivalent terms produces equivalent values:

Proposition 4.9. *Let $C[\ ]$ be an arbitrary context. If $M \equiv_\alpha N$, $eval_C(C[M]) = U$, and $eval_C(C[N]) = V$, then $U \equiv_\alpha V$.*

Proof. The proof is an induction on the transition sequence and reflects the above informal argument. □

In order to be useful, a translation from the CS-system to the C-system must produce a term in the appropriate α-equivalence class. It is therefore natural to incorporate the idea of neglecting self-referential cell occurrences during the translation. Thus, the translation U from CS-control string/store pairs to C-control strings

changes the contents of a cell to I once the cell is dereferenced:

$$\mathsf{U}(x,\theta) = x,$$

$$\mathsf{U}(l,\theta) = \mathsf{U}(\theta(l),\theta[l := \mathsf{I}])^{l},$$

$$\mathsf{U}(\lambda x.M,\theta) = \lambda x.\mathsf{U}(M,\theta),$$

$$\mathsf{U}(MN,\theta) = \mathsf{U}(M,\theta)\mathsf{U}(N,\theta),$$

$$\mathsf{U}(\mathcal{F}M,\theta) = \mathcal{F}\mathsf{U}(M,\theta)$$

$$\mathsf{U}(\sigma X.M,\theta) = \sigma\mathsf{U}(X,\theta).\mathsf{U}(M,\theta)$$

U is the required unload function for the statement of the correctness lemma:

Lemma 4.10 (C-simulation). *For all programs* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$eval_{CS,\mathsf{U}}(M) = eval_{C}(M).$$

Proof. The proof plan for this last machine-simulation lemma is again the same as in the four preceding ones. The central claim is that a transition step

$$\langle c_1,\theta_1\rangle \xmapsto{CS} \langle c_2,\theta_2\rangle$$

on the CS-machine implies a transition step of the form

$$\mathsf{U}(c_1,\theta_1) \xmapsto{C} \mathsf{U}(c_2,\theta_2)$$

in the rewriting system. However, unlike in previous simulation proofs, we must also address the issue of choosing new locations and labels. In the previous proofs, we could rely on the convention that

$$l \text{ such that } l \notin D$$

uniquely determined the new element, and that this choice algorithm could never interfere with the simulation because the respective stores always had the same domain.

In the present simulation of the CS-machine via the C-rewriting system, the domain of a CS-store in $\langle c, \theta \rangle$ and the set of labels in $\mathsf{U}(c, \theta)$ are not necessarily the same. The C-machine only maintains labels that are reachable from the program and automatically eliminates all others by (vacuous) substitutions. It follows that we can add the clause

$$Lab(\mathsf{U}(c_i, \theta_i)) \dot{\subseteq} Dom(\theta_i) \text{ for } i = 1, 2$$

to the induction invariant. A verification of this condition is straightforward. The condition implies that the simulation of the transition rules (CS5) and (CS'7) can use the new location as a label on the respective values. In other words, given that the choice algorithm for new labels accounts for previously used labels, the simulation of the CS-machine by the C-rewriting system is perfect. □

The construction of the C-rewriting system and its correctness lemma point out a crucial difference between program control and program state. Whereas for the former $\Lambda_{\mathcal{F}\sigma}$ suffices for formulating a symbolic rewriting system, the latter requires an extension. The store (in a denotational or operational semantics) establishes a relation among subterm occurrences in a program that is not expressible with the syntactic constructions of $\Lambda_{\mathcal{F}\sigma}$. Nevertheless, as the above proof indicates, the syntactic-symbolic model of the store has two crucial advantages over a semantic model.

First, all locations that are decidably connected with the rest of the computation are located in the program text, *i.e.*, they are in the set $Lab(\cdot)$. Due to the replacement of environments and stores with substitutions, there is no accumulation of garbage in C-terms as there is in CESK-stores. For example, $(\lambda x.M)V$ where x is not free in M behaves equivalently to M in all computational contexts $C[\ ]$:

$$C[(\lambda x.M)V] \xrightarrow{C} C[M[x := V']] \equiv_\alpha C[M].$$

Similarly, if all occurrences of l are in labeled values V^k, an assignment $(\sigma V^k.M)U$ erases all occurrences of l provided that U does not contain l. The equivalent construction to $Lab(\cdot)$ in denotational frameworks is based on the notion of a program cover [25]. It allows the same kind of equality arguments, but the required mathematical machinery is complicated and not intended for use by programmers.

Second, with the elimination of garbage, the comparison of states in evaluation sequences can be reduced to finding an isomorphism between label sets of programs. The basis of the isomorphism-argument is an extension of α-equivalence. The central idea behind the extension is that the identity of a label should be irrelevant for a program just like the identity of a variable name is irrelevant for a λ-body. In short, we should identify programs that are equal modulo some renaming of labels. Formally, we express this as *label-equality*:

Definition 4.11. *Two programs M and N are* label-equal, $M \equiv_{lab} N$, *if there exists a bijection ϕ between the label sets $Lab(M)$ and $Lab(N)$:*

$$\phi\colon Lab(M) \longrightarrow Lab(N)$$

such that $\phi(M) \equiv_\alpha N$.

The extension of ϕ on the entire term is:

$$\phi(MN) = \phi(M)\phi(N) \qquad \phi(a) = a,\ \phi(x) = x$$

$$\phi(\mathcal{F}M) = \mathcal{F}\phi(M) \qquad \phi(\lambda x.M) = \lambda x.\phi(M)$$

$$\phi(V^l) = \phi(V)^{\phi(l)} \qquad \phi(\sigma X.M) = \sigma\phi(X).\phi(M)$$

Warning, part I. *Label-equivalence resembles an ordinary term relation, but it is a program relation. Put differently, label-equivalence only applies to terms that are used as programs. It makes no sense to embed related terms in contexts. For example, it is true that the result of*

$$(\lambda xm.mx(\sigma x.x))0$$

as a program is label-equal to

$$\lambda m.m0^l(\sigma 0^l.0^l),$$

but that does not imply that the first expression evaluates to the second one in arbitrary contexts: sharing relations always concern the program as a whole and cannot be considered in isolation. We therefore recommend imagining a unique marker at the root of every program.[7] **End**

The soundness statement for label-equivalence is an adapted version of Proposition 4.9:

Proposition 4.12. *If* $M \equiv_{lab} N$, $eval_C(M) = U$, *and* $eval_C(N) = V$, $U \equiv_{lab} V$.

Proof. The core of the proof is an induction on the number of transition steps in the evaluation. It depends on the following two claims about substitution and labeled-value substitution:

(1) $MV \equiv_{lab} NU$ implies $M[x := V] \equiv_{lab} N[x := U]$, and

(2) $MV \equiv_{lab} NU$ implies $M[\bullet^l := V^l] \equiv_{lab} N[\bullet^l := U^l]$.

These claims express that sharing relationships commute with substitutions if the relationships hold for the program. □

The relevance of label-equivalence to the comparison problem is captured in:

Corollary 4.13. *Suppose the C-transition function can pick new labels randomly. Furthermore, suppose that* $M \equiv_\alpha N$, $eval_C(M) = U$, $eval_C(N) = V$, *and* $U \not\equiv_\alpha V$. *Then there is also a value* V' *such that* $eval_C(N) = V'$ *and* $U \equiv_\alpha V' \equiv_{lab} V$.

Proof. Clearly, the random choice of labels for C-transitions instead of a deterministic choice does not invalidate Proposition 4.12. Hence, $U \equiv_{lab} V$. But then

[7] Yet another way to see this problem is to say that programs *implicitly* delimit the scope of labels. Hence, the introduction of an explicit *first-class* marker for program-terms would resolve this dilemma, but it is not clear what such a marker would mean on an intuitive level.

there is an isomorphism between the label sets of U and V, and we can use this isomorphism to rearrange the allocation of labels in the C-transition sequence from N to V. This rearrangement yields the desired transition sequence from N to V'.

□

Informally, even if we relax the constraint on the label-choice algorithm, we can always reconstruct appropriate evaluation sequences that simulate the CS-machine correctly. As a consequence, we do relax the constraint on the condition $l \notin Lab(M)$ and admit *any fresh* element outside of the label set. Furthermore, we adapt the two conventions concerning bound variables to labels:

- Label-equal *programs* are identified [*label-equivalence convention*];
- Different label-names always denote different sharing relationships [*label hygiene convention*].

From a practical point of view, this is a solution to the program/store-comparison problem. Recall the appropriate example from Section 3.2:

$$(\lambda xy.M)01 \text{ and } (\lambda yx.M)10.$$

In the rewriting system, the two terms yield the same result for all evaluation contexts:

$$\begin{aligned} C[(\lambda xy.M)01] &\xrightarrow{C} C[M[x := 0^k][y := 1^l]] \\ &\equiv_{lab} C[M[x := 0^m][y := 1^n]] \equiv_{lab} \\ C[(\lambda xy.M)01] &\xrightarrow{C} C[M[y := 1^n][x := 0^m]], \end{aligned}$$

and can thus be considered interchangeable as desired. In semantic store models this kind of reasoning is more difficult because of the mutual interference of garbage and isomorphic re-allocations.

The result of the three sections in this chapter is summarized in a theorem that connects the CESK-machine with the C-rewriting system:

Theorem 4.14 (C-Simulation). *For all programs* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$eval_{CESK,\mathsf{Unload}}(M) = eval_C(M),$$

where $\mathsf{Unload}(V, \theta) = \mathsf{U}(\mathsf{S}(\mathsf{T}_p(\mathsf{R}(V))), \mathsf{S}_s(\mathsf{T}_s(\mathsf{R}_s(V))))$.

The C-Simulation Theorem says that, provided Unload is used as an interface, an outside observer cannot perceive any differences between the CESK-evaluation function and the C-evaluation function. The CESK-Unload function is natural since it coincides with the CEK-Unload function (see Chapter 2) on Λ-values.

The differences between the semantic frameworks are important, too. On one hand, the control string language Λ_{rew} is only a minor extension of the original programming language; on the other, the C-transition function solely relies on terms in Λ_{rew}, has fewer rules than the CESK-machine, automatically eliminates useless garbage, and solves the program/store-comparison problem. We believe that reasoning about Idealized Scheme programs in the C-system is a more viable alternative.

Based on this assessment, we shall use the C-rewriting system instead of the CESK-machine in all future discussions. Theorem 4.14 guarantees that all results carry over to the original CESK-semantics. In the next chapter the C-rewriting system serves as a starting point for the design of reductions for an extended λ_v-calculus. Together with the reductions, the rewriting semantics will then constitute a reasoning system for Idealized Scheme.

5. The λ_v-CS-Calculus

A closer look at the program rewriting system for Idealized Scheme should remind the reader of the alternative characterization of the λ_v-standard reduction function in Proposition 2.8. According to this proposition, the standard reduction evaluation first partitions a program into an evaluation context and a β_v- or δ-redex. Then, an appropriate contraction replaces the redex by a new term, and finally, the standard evaluation resumes the cycle. The C-rewriting system works in the same way, except that the context-redex partitioning plays a different role. All but one of the transition rules crucially depend on the uniqueness of the partitioning, and furthermore, two of the rules not only alter the redex but the entire program. We characterize this property of transitions as *context-sensitivity*.

We begin the first section of this chapter with a study of the nature of context-sensitivity. Based on this, we design a set of term relations that achieve the same effects. While most are notions of reduction, some relations are special, context-sensitive *computation relations*. This set of term relations forms the basis of the λ_v-CS-calculus. The second section of this chapter is devoted to the problems of consistency and standardization. The last section contains the correspondence theorems, which connect the machine semantics with the calculus and *vice versa*. The main result of the last section is an equational theory for reasoning about imperative higher-order programs.

5.1. Reductions and Computations

The transition rules of the C-rewriting system fall into three different classes: context-insensitive rules, those that leave the context intact, and those that rewrite the context or displace it. The first category comprises just (C6), the second (C1) and (C5), the third (C7) and (C10). The division implies two reasons for context-sensitivity.

First, partitioning programs into C-redexes and evaluation contexts uniquely determines *when* a transition is to occur. This is obviously the case for the delabeling rule. Assignments and delabeling steps must happen in an exact order such that the extensional sequentiality constraints of the CESK-machine are satisfied. A freely applicable reduction of the form

$$V^l \longrightarrow V[\bullet^l := V^l]$$

would interfere with this ordering. Similarly, a modified β-reduction à la

$$(\lambda x.M)V \longrightarrow M[x := V^l]$$

would establish sharing relations at the wrong time. Since this is a rather subtle point, we demonstrate the problem with an example. Consider the expression

$$(\lambda f.(f\,\mathsf{I})(f\,\mathsf{K}))(\lambda x.(\lambda y.(\sigma y.(\lambda u.yu))x)0).$$

When evaluated on the CESK-machine, it yields K. If we use the above pseudo-relation to simplify the argument, we first obtain

$$(\lambda f.(f\,\mathsf{I})(f\,\mathsf{K}))(\lambda x.(\sigma 0^l.(\lambda u.0^l u))x),$$

and this expression in turn produces $\lambda xu.\mathsf{K}^l u$.

Second, the transition rules for $\mathcal{F}$- and σ-applications not only need to be timed correctly, they also must manipulate the entire context in order to achieve the appropriate effect. $\mathcal{F}$-applications remove and give their argument control over the current context; σ-applications replace a number of subterms in the context.

Considering these observations, there is no hope of finding an equivalent set of context-insensitive reduction relations: by their very nature, imperative effects must happen in a certain order. However, the second kind of context-sensitivity suggests a partial solution. It indicates that if imperative transitions were only discharged at the root of a term, there would be no extent problem: all effects would be concerned with proper subterms of the redex. A restriction of imperative transitions to the root has the additional advantage that it naturally coordinates the timing of effects.

Following this line of reasoning, we design two sets of term relations: reductions and *computations*. The former have the same status as ordinary notions of reduction, and their task is to *bubble* a C-redex to the root of an evaluation context. Once the redex has reached the root, a computation relation performs the proper imperative effect. In order to keep this system consistent, computation relations must have a sub-privileged status. Unlike notions of reduction, they cannot be applied to subterms. We indicate this difference by using $\triangleright$ instead of $\longrightarrow$ for denoting computations.

An evaluation in the new system resembles a race. All imperative redexes in a term simultaneously start bubbling up towards the root of a term. Whichever redex gets there first, performs an imperative effect. Sequentiality is preserved and coordinated by a proper scheduling of arrivals. This certainly differs from a traditional calculus, but as we shall demonstrate in the sequel, it is an acceptable generalization of the notion of a calculus.

One disadvantage of our plan is that it is not conservative with respect to the λ_v-

calculus. Because of the timing constraints, even λ-redexes must apparently move to the root before being contracted. Fortunately, we can recover the β_v-relation with a simple observation. Thus far, we have followed the implicit convention that all variables are the same kind of objects, but this is only partially true. Whereas variables in general manifest an α-equivalence, it is only the *assignable* kind that necessitates a dynamic continuation of this relationship. Non-assignable variables do not require a labeling step. Accordingly, our solution is to divide the set of variables into two disjoint subsets, one for each category. We call these subsets Var_λ for non-assignable and Var_σ for assignable variables.

Given this division of the variable set, we split the transition rule for the application of λ-abstractions into two *sub*-rules, each of them accounting for a variable category:

$$C[(\lambda x.M)V] \xrightarrow{C} C[M[x := V]],\ x \in Var_\lambda, \tag{C5a}$$

and

$$C[(\lambda x.M)V] \xrightarrow{C} C[M[x := V^l]],\ x \in Var_\sigma,\ l \text{ is fresh.} \tag{C5b}$$

The first is clearly context-insensitive and represents β_v. Consequently, β_v and δ remain the basic notions of reduction for the functional subset of Idealized Scheme.

The second sub-rule, (C5b), is equivalent to the original one. Hence, we must design reduction and computation relations to simulate it. The computation relation is only used at the root of a term, *i.e.*, in the empty context. In short, it has the same form as the transition rule without $C[\ ]$:

$$(\lambda x.M)V \triangleright M[x := V^l],\ x \in Var_\sigma,\ l \text{ is fresh.} \tag{β_σ}$$

The non-deterministic choice of the new label l is possible because we identify label-equivalent *programs*, and this rule explicitly applies to programs only.

Otherwise, if the C-redex for (C5b) appears nested in an evaluation context, it must bubble up to the root. Since the computation relation takes care of the empty context, the inductive definition of evaluation contexts requires consideration of two more cases: the embedding of a C-redex to the left of an arbitrary term N and to the right of a value U. In the first case, the C-transition rule says that the modified λ-body and the term N form a new application. Thus, the way to move the C-redex up is to include N in the λ-body:

$$C[((\lambda x.M)V)N] \longrightarrow C[(\lambda x.MN)V],\ x \in Var_\sigma.$$

This new rule is clearly independent of $C[\ ]$ and therefore we adopt the context-free variant as another notion of reduction:

$$((\lambda x.M)V)N \longrightarrow (\lambda x.MN)V,\ x \in Var_\sigma. \qquad (\beta_L)$$

Based on the symmetry of the two cases, we suggest for the second one:

$$(U((\lambda x.M)V)) \longrightarrow (\lambda x.UM)V,\ x \in Var_\sigma. \qquad (\beta_R)$$

Like the β-rule itself, these reductions rely on the hygiene convention and assume that the set of free variables in N and U, respectively, does not contain x.

The simulation of assignment with reduction and computation relations is now straightforward. The computation relation is

$$(\sigma U^l.M)V \triangleright M[\bullet^l := V^l]; \qquad (\sigma_T)$$

an embedded σ-application can work its way to the root of a term with rules similar to those for a λ-application:

$$U((\sigma X.M)V) \longrightarrow (\sigma X.(UM))V \qquad (\sigma_R)$$

$$((\sigma X.M)V)N \longrightarrow (\sigma X.(MN))V \qquad (\sigma_L)$$

Unfortunately, the bubbling-up technique cannot be applied as easily to the delabeling rule for labeled values. Unlike a λ- or a σ-capability, a labeled value does not contain a subterm that can be used to gradually incorporate the term context. A possible solution is introduce a *delabel application* of the form

$$(\mathcal{D}\ MV^l)$$

where $\mathcal{D}$ is an improper symbol, V^l the labeled value, and M is an arbitrary term which is to consume the value of V^l.

Assuming that all labeled values occur within $\mathcal{D}$-applications, we can design an appropriate set of reductions and computations. A $\mathcal{D}$-application at the root of a term simply delabels the labeled value—according to the C-transition rules—and applies M to the resulting value:

$$(\mathcal{D}\ MV^l) \triangleright M(V[\bullet^l := V^l]). \qquad (\mathcal{D}_T)$$

If $(\mathcal{D}\ MV^l)$ is to the left of some expression N, the $\mathcal{D}$-application must somehow incorporate the argument N into the consumer expression in order to move closer to the top. In a modified C-rewriting system, the entire application would first delabel V^l, then apply M to the resulting value, and finally, the result of this application would absorb N. Abstracting from the value of V^l, this informal description is captured in the λ-abstraction $\lambda v.MvN$, and hence, the two reductions for $\mathcal{D}$-applications are:

$$(\mathcal{D}\ MX)N \rightarrow (\mathcal{D}\ (\lambda x.MxN)X), \qquad (\mathcal{D}_L)$$

$$U(\mathcal{D}\ MX) \rightarrow (\mathcal{D}\ (\lambda x.U(Mx))X). \qquad (\mathcal{D}_R)$$

Now that we have explored our solution for the bubbling-up of delabeling redexes, we must clarify how to get all labeled values into $\mathcal{D}$-applications. Two

Definition 5.1: The calculus language Λ_{CS}

Syntactic Domains:

a	$\in Const$	(constants)
x	$\in Vars$	(variables), $Vars = Var_\lambda \cup Var_\sigma$
x_λ	$\in Var_\lambda$	(non-assignable variables)
x_σ	$\in Var_\sigma$	(assignable variables)
V	$\in Values$	(values)
l	$\in Labels$	(labels)
M, N	$\in \Lambda$	(Λ-terms)

Abstract Syntax:

$$M ::= a \mid x_\lambda \mid (\lambda x.M) \mid (MN) \mid (\mathcal{F}M) \mid (\sigma x_\sigma.M) \mid (\sigma V^l.M) \mid (\mathcal{D}\, Mx_\sigma) \mid (\mathcal{D}\, MV^l).$$

Constants, non-assignable variables, abstractions, and capabilities are referred to as values.

X is subsequently used for assignable variables and labeled values.

Programs are closed terms possibly including labeled values.

Λ_{CS}-contexts are defined by two clauses:

$$C[\ ] ::= [\ ] \mid \mid (\lambda x.C[\ ]) \mid (C[\ ]M) \mid (MC[\ ]) \mid (\mathcal{F}C[\ ]) \mid (\sigma V[\ ]^l.M \mid (\sigma X.C[\ ]) \mid (\mathcal{D}\, MV[\ ]^l) \mid (\mathcal{D}\, C[\ ]X)$$

$$V[\ ] ::= [\ ] \mid (\lambda x.C[\ ]) \mid (\sigma V[\ ]^l.M) \mid (\sigma X.C[\ ])$$

The separation of contexts and value-contexts is necessary to avoid category conflicts in the value positions of σ-capabilities and $\mathcal{D}$-applications.

Φ injects $\Lambda_{\mathcal{F}\sigma}$- and Λ_{rew}-terms into Λ_{CS}:

$$\Phi(x_\lambda) = x_\lambda,\ \Phi(x_\sigma) = \mathcal{D}\, !x_\sigma,\ \Phi(V^l) = \mathcal{D}\, !\Phi(V)^l,$$
$$\Phi(a) = a,\ \Phi(\lambda x.M) = \lambda x.\Phi(M),\ \Phi(MN) = \Phi(M)\,\Phi(N),$$
$$\Phi(\mathcal{F}M) = \mathcal{F}\Phi(M),\ \Phi(\sigma x.M) = \sigma\Phi(x).\Phi(M),\ \Phi(\sigma V^l.M) = \sigma\Phi(V)^l.\Phi(M).$$

possibilities are feasible: the β_σ-computation could actually substitute assignable variables with $\mathcal{D}$-applications, or the language could be defined such that assignable variables only occur within $\mathcal{D}$-applications. The former alternative has the advantage that the calculus-language directly subsumes the programming language, but it also has the disadvantage of unnecessarily enlarging the language. Furthermore, since assignable variables represent labeled values, they cannot be included in the set of values. Therefore, we choose the second solution because it syntactically highlights this fact. Definition 5.1 contains the new language, the definition of Λ_{CS}-contexts, and a map that injects $\Lambda_{\mathcal{F}\sigma}$ (and Λ_{rew}).

Finally, we deal with $\mathcal{F}$-applications. The respective C-transition constructs a function from the context of the $\mathcal{F}$-application, labels it, and then applies the $\mathcal{F}$-argument to this labeled value. The labeling is due to the design of the CESK-machine, which contains all values in the store. Since this is unnecessary, we require the $\mathcal{F}$-reduction and computation relations to satisfy a modified rule:

$$C[\mathcal{F}M] \longrightarrow M(\lambda x.C[x]). \tag{C7}$$

Accordingly, the computation relation must apply the $\mathcal{F}$-argument to the initial continuation represented by the empty context:

$$\mathcal{F}M \triangleright M(\lambda x.x). \tag{$\mathcal{F}_T$}$$

For reductions, (C7) implies that they must simultaneously bubble up an $\mathcal{F}$-redex and encode the term context on their way up. Again, we simply investigate the two inductive cases of a context definition. Consider the expression $(\mathcal{F}M)N$ and suppose it is embedded in a context $C[\ ]$. Clearly, the function $\lambda f.fN$ is a correct encoding of the immediate context of the $\mathcal{F}$-application. The rest of the context, *i.e.*, $C[\ ]$, can be encoded by another $\mathcal{F}$-application. Putting these two parts of the

continuation together is straightforward: fN represents the immediate continuation of the $\mathcal{F}$-application, κ the one that follows this application, and hence, $\kappa(fN)$ is the entire continuation. In short, these ideas suggest the following notions of reductions:

$$(\mathcal{F}M)N \longrightarrow \mathcal{F}(\lambda\kappa.M(\lambda f.\kappa(fN))), \qquad (\mathcal{F}_L)$$

$$U(\mathcal{F}M) \longrightarrow \mathcal{F}(\lambda\kappa.M(\lambda v.\kappa(Uv))). \qquad (\mathcal{F}_R)$$

The definition of reduction and computation relations for $\mathcal{F}$ ends the design phase of the λ_v-CS-calculus. The notions of reduction and computation relations are collected in Definition 5.2. The next steps in our development are determined by the calculus-construction procedure outlined in Section 2.3. According to this procedure, we first collect all notions of reduction in a single term relation **cs**. Based on this reduction, a compatible closure is formed, yielding the one-step reduction $\longrightarrow_{cs}$. The compatible closure must account for five different syntactic constructions and two *value*-ness constraints on subterms in σ-capabilities and $\mathcal{D}$-applications. These considerations are built into the definition of Λ_{CS}-contexts, and, for simplicity's sake, we exploit this in the formal specification of the calculus. Finally, we define the reduction and congruence relation of the one-step reduction relation: see Definition 5.3.

Thus far, our calculus is a simple, conventional extension of the traditional λ_v-calculus. However, since our goal is a calculus for Idealized Scheme, we must go beyond conventional constructions and somehow include the computation relations in order to emulate imperative effects. To avoid any interference of computations with the compatibility construction, we have chosen to define a *computation* as the union of the reduction $\longrightarrow\!\!\!\!\!\rightarrow_{cs}$ and the five computation relations. The motivation behind this step is that one computation step can simulate one C-transition step: the reduction can bubble up a redex to the root and a computation relation can

Definition 5.2: Reductions and Computations

The notions of reduction are

$$f a \longrightarrow V \text{ if } \delta(f,a) = V, V \in \Lambda \qquad (\delta)$$
$$(\lambda x.M)V \longrightarrow M[x := V] \qquad (\beta_v)$$
$$U((\lambda x_\sigma.M)V) \longrightarrow (\lambda x_\sigma.(UM))V \qquad (\beta_R)$$
$$((\lambda x_\sigma.M)V)N \longrightarrow (\lambda x_\sigma.(MN))V \qquad (\beta_L)$$
$$U(\mathcal{F}M) \longrightarrow \mathcal{F}\lambda\kappa.M(\lambda v.\kappa(Uv)) \qquad (\mathcal{F}_R)$$
$$(\mathcal{F}M)N \longrightarrow \mathcal{F}\lambda\kappa.M(\lambda f.\kappa(fN)) \qquad (\mathcal{F}_L)$$
$$U((\sigma X.M)V) \longrightarrow (\sigma X.(UM))V \qquad (\sigma_R)$$
$$((\sigma X.M)V)N \longrightarrow (\sigma X.(MN))V \qquad (\sigma_L)$$
$$U(\mathcal{D}\ MX) \longrightarrow (\mathcal{D}\ (\lambda v.U(Mv))X) \qquad (\mathcal{D}_R)$$
$$(\mathcal{D}\ MX)N \longrightarrow (\mathcal{D}\ (\lambda v.MvN)X) \qquad (\mathcal{D}_L)$$

where $x_\sigma \in \mathit{Var}_\sigma$, $U, V \in \mathit{Values}$, and X ranges over Var_σ and labeled values. Non-indexed variables are from Var_λ.

The computation relations are

$$(\mathcal{F}M) \triangleright M(\lambda x.x) \qquad (\mathcal{F}_T)$$
$$(\lambda x_\sigma.M)V \triangleright M[x_\sigma := V^l] \text{ where } l \text{ is fresh} \qquad (\beta_\sigma)$$
$$(\sigma U^l.M)V \triangleright M[\bullet^l := V^l] \qquad (\sigma_T)$$
$$(\mathcal{D}\ MV^l) \triangleright M(V[\bullet^l := V^l]) \qquad (\mathcal{D}_T)$$

with the above provisions. We refer to the left-hand sides of computations as (computational) redexes.

Substitution and labeled-value substitution are adapted *mutatis mutandis* to the new syntax:

$$(\mathcal{D}\ MX)[y := V] \equiv (\mathcal{D}\ M[y := V]X[y := V])$$
$$(\mathcal{D}\ MX)[\bullet^l := V^l] \equiv (\mathcal{D}\ M[\bullet^l := V^l]X[\bullet^l := V^l])$$

perform the imperative action. On top of this computation, we define computational equality as the smallest equivalence relation that encloses the computation. These relations are also summarized in Definition 5.3.

Definition 5.3: The λ_v-CS-calculus

The basic notion of reduction is

$$\mathbf{cs} = \beta_v \cup \delta \cup \mathcal{F}_L \cup \mathcal{F}_R \cup \beta_L \cup \beta_R \cup \sigma_L \cup \sigma_R \cup \mathcal{D}_L \cup \mathcal{D}_R.$$

The *one-step* **cs**-*reduction* $\longrightarrow_{cs}$ is the compatible closure of **cs**:

$M \longrightarrow_{cs} N$
if there are P, Q, and an *arbitrary* context $C[\ ]$ such that $(P, Q) \in \mathbf{cs}$, $M \equiv C[P]$, and $N \equiv C[Q]$.

The **cs**-*reduction* is denoted by $\longrightarrow\!\!\!\!\rightarrow_{cs}$ and is the reflexive, transitive closure of $\longrightarrow_{cs}$. We denote the smallest equivalence relation generated by $\longrightarrow_{cs}$ with $=_{cs}$ and call it **cs**-*equality*.

The **cs**-*computation* $\rhd_{cs}$ is defined by:

$$\rhd_{cs} = \mathcal{F}_T \cup \beta_\sigma \cup \sigma_T \cup \mathcal{D}_T \cup \longrightarrow\!\!\!\!\rightarrow_{cs}.$$

The relation $\stackrel{\rhd}{=}_{cs}$ is the smallest equivalence relation generated by $\rhd_{cs}$. We refer to it as *computational equality*.

The result of our calculus design is an untraditional two-level system: on the lower level, it is a conventional congruence, on the upper one, a simple equivalence relation. When we talk about the λ_v-CS-calculus, we refer to the relation $\stackrel{\rhd}{=}_{cs}$. We write $M \stackrel{\rhd}{=}_{cs} N$ or $\lambda_v\text{-CS}^{\rhd} \vdash M = N$ for derivations on this level. Although weaker, the congruence relation $=_{cs}$ is traditional and interesting in its own right. We consider it as a sub-calculus and use the notation $\lambda_v\text{-CS} \vdash M = N$.

5.2. Consistency and Standardization

Given the definition of a calculus, the question arises how this calculus compares with others. For our work there are two problems of immediate concern, namely, consistency and standardization. However, before we turn to these, we take a brief look at the nature of variables in our system. In an ordinary logic or calculus, variables do not play an active role in proofs; they are simply placeholders for values and nothing else. Although Λ_{CS} contains the more sophisticated category of assignable variables, we can still prove a comparable substitution property:

Theorem 5.4. *Let $x_\lambda \in Var_\lambda$, $x_\sigma \in Var_\sigma$, and let l be a label such that $l \notin Lab(MV) \cup Lab(NV)$:*

(i) $\lambda_v\text{-}CS^{\triangleright} \vdash MV = NV$ *implies* $\lambda_v\text{-}CS^{\triangleright} \vdash M[x_\lambda := V] = N[x_\lambda := V]$;

(ii) $\lambda_v\text{-}CS^{\triangleright} \vdash MV = NV$ *implies* $\lambda_v\text{-}CS^{\triangleright} \vdash M[x_\sigma := V^l] = N[x_\sigma := V^l]$;

(iii) $\lambda_v\text{-}CS \vdash M = N$ *implies* $\lambda_v\text{-}CS \vdash M[x_\lambda := V] = N[x_\lambda := V]$;

(iv) $\lambda_v\text{-}CS \vdash M = N$ *implies* $\lambda_v\text{-}CS \vdash M[x_\sigma := V^l] = N[x_\sigma := V^l]$.

Remark. The antecedents in statements (i) and (ii) are equivalent to the condition that the label set for V is disjoint from the symmetric difference of the label sets of M and N, *i.e.*, the labels that a proof of $M \overset{\triangleright}{=}_{cs} N$ introduces in M and N cannot interfere with the labels in V. **End**

Proof. The claims follow from an induction on the structure of the proof and Proposition 4.12. The induction relies on a generalized version of the substitution lemma:

$$M[x := X][y := Y[x := X]] \equiv M[y := Y][x := X] \qquad \text{if } y \notin FV(X);$$

and on a commutation lemma for substitution and labeled-value substitution:

$$M[x := X][\bullet^l := U[x := X]^l] \equiv M[\bullet^l := U^l][x := X] \qquad \text{if } l \notin Lab(X). \qquad \square$$

We shall use the theorem in the following section on the correctness of the calculus. It is stated here because of its logical nature.

As mentioned in Chapter 2, the consistency of traditional calculi is implied by a Church-Rosser theorem. This theorem shows the confluence of two reduction paths that proceed in two different directions from the same term. For our two-level calculus, however, this is insufficient. We must prove in addition that a computation step cannot interfere with the Church-Rosser property of the reduction system, *i.e.*, that it cannot cause a divergence of derivation paths in the upper-level equivalence relation.

From these preliminary remarks it follows that our generalized Church-Rosser Theorem must be of the form

Theorem 5.5 (Consistency).

(i) The notion of reduction **cs** *is Church-Rosser.*

(ii) The **cs***-computation satisfies the diamond property.*

The proof of the first part is a modification of the Martin-Löf proof for the original Church-Rosser Theorem; the second part is a simple case analysis and relies on the first part. Details can be found in the subsection.

The theorem implies the important

Corollary 5.6. *If* $M \stackrel{\triangleright}{=}_{cs} N$ *then there exists an* L *such that* $M \triangleright^*_{cs} L$ *and* $N \triangleright^*_{cs} L$.

This corollary guarantees that the calculus cannot prove every arbitrary equation because there are distinct normal forms, and that a program reduces to a value if and only if it has a value.

The second basic question about calculi is whether there are standardized derivation sequences. Standardized derivation sequences constitute the basis for a semi-decision procedure for equalities and are thus crucial for finding values and normal-

forms of programs. We again proceed in two steps. First, we must show that there are standard *reduction* sequences in the reduction sub-calculus. Second, we must extend the notion of standard reduction sequences to standard computation sequences and prove their adequacy.

The actual definition of standard reduction sequences follows Plotkin's strategy for the corresponding work on the λ_v-calculus. In order to facilitate the presentation in Section 5.3, we use the technique of Proposition 2.8 to define the concept of a standard reduction and a standard computation function:

Definition 5.7.

(i) *The* standard reduction function *maps* M *to* N, $M \longmapsto_{scs} N$, *if there are* P, Q, *and an evaluation context* $C[\ ]$ *such that* $(P, Q) \in \mathbf{cs}$, $M \equiv C[P]$, *and* $N \equiv C[Q]$.

(ii) *The* standard computation function *is an extension of the standard reduction function with computation relations:*

$$\overset{\triangleright}{\longmapsto}_{scs} = \longmapsto_{scs} \cup \mathcal{F}_T \cup \beta_\sigma \cup \sigma_T \cup \mathcal{D}_T.$$

By adding in the computation after extending the notion of reduction **cs** to a standard reduction function, we ensure that root-level computation steps cannot cause inconsistencies.

The step from standard reduction and computation functions to standard derivation sequences—CS-SRS-s and CS-SCS-s—is a straightforward generalization of the respective step in the λ_v-calculus: see Definition 5.8. Care is taken to permit computations only at the root of a term. The Standardization Theorem is stated for the calculus and the sub-calculus:

Definition 5.8: Standard reduction and computation sequences

Standard reduction sequences, abbreviated CS-SRS-s, are defined by:

1. all constants and non-assignable variables are CS-SRS-s;
2. if $M_1, \ldots, M_m$, $N_1, \ldots, N_n$, and $V_1, \ldots, V_j$ are CS-SRS-s, then

$$\lambda x.M_1, \ldots, \lambda x.M_m,$$
$$M_1N_1, \ldots, M_mN_1, \ldots, M_mN_n,$$
$$\mathcal{F}M_1, \ldots, \mathcal{F}M_m,$$
$$\sigma x.M_1, \ldots, \sigma x.M_m \text{ and } \sigma V_1^l.M_1, \ldots, \sigma V_j^l.M_1, \ldots, \sigma V_j^l.M_m$$
$$\mathcal{D}\ M_1x, \ldots, \mathcal{D}\ M_mx \text{ and } \mathcal{D}\ M_1V_1^l, \ldots, \mathcal{D}\ M_mV_1^l, \ldots, \mathcal{D}\ M_mV_j^l$$

   are CS-SRS-s;

3. if $M \longmapsto_{scs} M_1$ and $M_1, \ldots, M_m$, then $M, M_1, \ldots, M_m$ is a CS-SRS.

All standard reduction sequences are also *standard computation sequences*, CS-SCS-s, and if $M \overset{\triangleright}{\longmapsto}_{scs} M_1$ and $M_1, \ldots, M_k$ is a CS-SCS, then $M, M_1, \ldots, M_k$ is a CS-SCS.

Theorem 5.9 (Standardization).

(i) $M \longmapsto^*_{scs} N$ *if and only if there is a CS-*SRS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$;

(ii) $M \overset{\triangleright}{\longmapsto}{}^*_{scs} N$ *if and only if there is a CS-*SCS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$.

The proof of the theorem is presented in the subsection.

With the Consistency and Standardization Theorems in place, we are on firm ground. Consistency gives us the security that equations in the calculus make sense, Standardization provides an effective procedure for finding values of programs. We are now ready to investigate the correspondence between the λ_v-CS-calculus and Idealized Scheme.

5.2.1. Proofs for the Consistency and Standardization Theorems

The proofs of the Consistency and Standardization Theorems follow the same pattern. For both theorems, the validity of the statement about the sub-calculus implies the claim about the entire λ_v-CS-calculus. Furthermore, traditional techniques are sufficient in both cases for proving the statements on the sub-calculus. The presentation closely follows Barendregt's [5:59–63] and Plotkin's [47:136–142] expositions. In many cases we simply state the necessary lemmas and show some prototypical proof steps. It should be no problem to fill in the remaining gaps.

We first treat the Consistency Theorem:

Theorem 5.5 (Consistency).

(i) The notion of reduction **cs** *is Church-Rosser.*

(ii) The **cs***-computation satisfies the diamond property.*

Proof. Let us assume the correctness of point (i). Then we need to show that whenever $M \rhd_{cs} L_i$ for $i = 1, 2$ and $L_1 \not\equiv_{lab} L_2$, there exists a term N such that $L_i \rhd_{cs} N$. We proceed by a case analysis on $M \rhd_{cs} L_1$:

1. $M \equiv (\mathcal{F}P) \rhd P(\lambda x.x) \equiv L_1$.

   The only possibility for a truly distinct reduction is $M \longrightarrow\!\!\!\!\!\rightarrow_{cs} (\mathcal{F}P_2) \equiv L_2$ where $P \longrightarrow\!\!\!\!\!\rightarrow_{cs} P_2$. This immediately implies that

   $$(\mathcal{F}P_2) \rhd P_2(\lambda x.x).$$

   Since we also have

   $$P(\lambda x.x) \longrightarrow\!\!\!\!\!\rightarrow_{cs} P_2(\lambda x.x),$$

   we can take $N \equiv P_2(\lambda x.x)$.

2. $M \equiv (\lambda x_\sigma.P)V \rhd P[x_\sigma := V^l] \equiv L_1$ where l is fresh.

   First, we must consider the case

   $$M \equiv (\lambda x_\sigma.P)V \rhd P[x_\sigma := V^m]$$

where $m \neq l$. At this point, we can exploit the knowledge that computations can only be applied to an entire program. Hence, it follows that the label sets of

$$P[x_\sigma := V^m] \text{ and } P[x_\sigma := V^l]$$

are isomorphic, *i.e.*,

$$L_1 \equiv_{lab} L_2.$$

Since we identify label-equal *programs*, this case degenerates to

$$L_1 \equiv_{lab} L_2 \equiv_{lab} N.$$

Next, we must analyze two subcases because M contains two subterms:

a) $L_2 \equiv (\lambda x_\sigma.P_2)V$ because $P \longrightarrow\!\!\!\!\!\rightarrow_{cs} P_2$.

But then

$$(\lambda x_\sigma.P_2)V \triangleright P_2[x_\sigma := V^l]$$

and the Substitution Theorem (5.4) induces

$$P[x_\sigma := V^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} P_2[x_\sigma := V^l].$$

The label l can hereby be reused because a reduction cannot introduce new labels and therefore l is still available for the computation step. Hence, $N \equiv P_2[x_\sigma := V^l]$.

b) $L_2 \equiv (\lambda x_\sigma.P)V_2$ because $V \longrightarrow\!\!\!\!\!\rightarrow_{cs} V_2$.

This second alternative requires a variant of the Substitution Theorem, namely, that

$$P[x_\sigma := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} P[x_\sigma := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_{cs} V.$$

Given this, the rest is the same as in subcase a).

3. $M \equiv (\sigma W^l.P)V \triangleright P[\bullet^l := V^l] \equiv L_1$.

   There are again two relevant cases: a reduction of M either transforms P or V. The claim follows from calculations like the preceding ones provided that we can prove the following two properties of labeled-value substitution:

$$P[\bullet^l := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} P[\bullet^l := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_{cs} V,$$

$$P[\bullet^l := V^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} Q[\bullet^l := V^l] \text{ if } P \longrightarrow\!\!\!\!\!\rightarrow_{cs} Q.$$

4. $M \equiv (\mathcal{D}\ MV^l) \triangleright M(V[\bullet^l := V^l]) \equiv L_1$.

   For this case we need the property

$$U[\bullet^l := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} V[\bullet^l := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_{cs} V.$$

   Otherwise, it does not add any new problems.

5. $M \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_1$.

   There are two subcases possible: the second step can be a computation or a reduction. In the former case, we have a situation that is symmetric to a previous one; in the latter, $M \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_2$, and the consequence holds because of the assumed Church-Rosser property for **cs**.

In order to complete the proof, we must still show the following four properties of substitution and labeled-value substitution:

$$P[x_\sigma := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} P[x_\sigma := V^l]$$

$$P[\bullet^l := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} P[\bullet^l := V^l]$$

$$P[\bullet^l := V^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} Q[\bullet^l := V^l]$$

$$U[\bullet^l := U^l] \longrightarrow\!\!\!\!\!\rightarrow_{cs} V[\bullet^l := V^l]$$

if $P \longrightarrow\!\!\!\!\!\rightarrow_{cs} Q$ and $U \longrightarrow\!\!\!\!\!\rightarrow_{cs} V$. The first two claims follow from straightforward inductions on the structure of P. The third uses a structural induction on the reduction from P to Q and is based on the commutativity of substitution and labeled-value

substitution (see Theorem 5.4). Finally, the proof of the fourth claim is a simple combination of the second and third statement. □

With the preceding proof we have reduced the consistency problem of the λ_v-CS-calculus to the consistency problem of the sub-calculus, *i.e.*, the Church-Rosser property of **cs**. The proof of this is an application of Martin-Löf's method for showing the corresponding result for β. The first step is to define a version of the parallel reduction relation $\xrightarrow[1]{}\!\!\!\!\!\rightarrow$ for **cs**. For the proof of the Standardization Theorem we also define a notion of the size of a parallel reduction: see Definition 5.10. It follows from this definition that a value can only parallel reduce to another value. The relationship between $\longrightarrow\!\!\!\!\!\rightarrow_{cs}$ and $\xrightarrow[1]{}\!\!\!\!\!\rightarrow$ is shown by the following lemma:

Lemma 5.11. $\mathbf{cs} \subset \longrightarrow\!\!\!\!\!\rightarrow_{cs} \subset \xrightarrow[1]{}\!\!\!\!\!\rightarrow \subset \longrightarrow\!\!\!\!\!\rightarrow_{cs}$.

Next we show that unlike the **cs**-reductions, the parallel reduction relation transforms the expression $M[x := N]$ into $M'[x := N']$ in one step if M and N parallel reduce to M' and N' in one step, respectively. That is, two β_v-contractums reduce to each other if the subterms do. Furthermore, following Plotkin [47], we simultaneously prove a statement that is needed for the proof of the Standardization Theorem, namely, that this new reduction is shorter than the one from $(\lambda x.M)N$ to $M'[x := N']$:

Lemma 5.12. *Suppose* $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'$, $N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'$, *and* N *is a value. Then the following holds:*

$$M[x := N] \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'[x := N'] \text{ and } s_{M[x:=N] \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'[x:=N']} < s_{(\lambda x.M)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'[x:=N']}.$$

Proof. The proof is a structural induction on the reduction $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'$. For each possible group in Definition 5.10 of parallel reduction steps, we explain the major proof steps with a typical example. In order to shorten the calculations for

Definition 5.10: The parallel reduction

The *parallel reduction* over Λ_{CS} is denoted by $\xrightarrow[1]{}\!\!\!\!\!\rightarrow$ and is defined as follows, where $s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow N}$ or just s is the function which measures the *size of the derivation* $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow N$ and $n(x, M)$ is the *number of free occurrences of* x *in* M:

1. $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M, \quad s = 0$
2. if $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'$, $N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'$, $U \xrightarrow[1]{}\!\!\!\!\!\rightarrow U'$, $V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'$, and $W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'$ then

$$(\lambda x.M)V \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'[x := V'],\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + n(x, M')s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + 1$$
$$(\mathcal{F}M)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow \mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN'))),\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + 1$$
$$V(\mathcal{F}M) \xrightarrow[1]{}\!\!\!\!\!\rightarrow \mathcal{F}(\lambda\kappa.M'(\lambda v.\kappa(V'v))),\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$((\lambda x_\sigma.M)V)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\lambda x_\sigma.M'N')V',\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$U((\lambda x_\sigma.M)V) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\lambda x_\sigma.U'M')V',\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{U \xrightarrow[1]{}\!\!\!\!\!\rightarrow U'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$((\sigma x.M)V)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma x.M'N')V',\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$U((\sigma x.M)V) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma x.U'M')V',\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{U \xrightarrow[1]{}\!\!\!\!\!\rightarrow U'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$((\sigma W^l.M)V)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma W'^l.M'N')V',$$
$$s = s_{W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'} + s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$U((\sigma W^l.M)V) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma W'^l.U'M')V',$$
$$s = s_{W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'} + s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{U \xrightarrow[1]{}\!\!\!\!\!\rightarrow U'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$(\mathcal{D}\ Mx)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ (\lambda x.M'xN')x),\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + 1$$
$$V(\mathcal{D}\ Mx) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ (\lambda x.V'(M'x))x),\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$
$$(\mathcal{D}\ MW^l)N \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ (\lambda x.M'xN')W'^l),$$
$$s = s_{W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'} + s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'} + 1$$
$$V(\mathcal{D}\ MW^l) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ (\lambda x.V'(M'x))W'^l),$$
$$s = s_{W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'} + s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + 1$$

3. if $M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'$, $N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'$, and $V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'$ then

$$\lambda x.M \xrightarrow[1]{}\!\!\!\!\!\rightarrow \lambda x.M',\ (\mathcal{F}M) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{F}M'),$$
$$(\sigma x.M) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma x.M'),\ (\mathcal{D}\ Mx) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ M'x),$$
$$\text{where } s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'}$$
$$(\sigma V^l.M) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\sigma V'^l.M'),\ (\mathcal{D}\ MV^l) \xrightarrow[1]{}\!\!\!\!\!\rightarrow (\mathcal{D}\ M'V'^l),$$
$$\text{where } s = s_{V \xrightarrow[1]{}\!\!\!\!\!\rightarrow V'} + s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'}$$
$$MN \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'N',\ s = s_{M \xrightarrow[1]{}\!\!\!\!\!\rightarrow M'} + s_{N \xrightarrow[1]{}\!\!\!\!\!\rightarrow N'}$$

the second claim, we introduce two abbreviations:

$$s_L = s_{M[x:=N] \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} M'[x:=N']},$$

$$s_R = s_{(\lambda x.M)N \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} M'[x:=N']}.$$

Furthermore, we use the fact that

$$s_R = s_{M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} M'} + n(x, M')s_{N \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N'} + 1$$

and that therefore, the second claim is equivalent to

$$s_L \leq s_{M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} M'} + n(x, M')s_{N \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N'}.$$

1. $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} M \equiv M'$. The result follows from an induction on the structure of M.

2. A typical case from the second group in Definition 5.10 is:

$$M \equiv V(\mathcal{F}P) \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \mathcal{F}(\lambda\kappa.P'(\lambda v.\kappa(V'v))) \equiv M',$$

based on $V \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} V'$ and $P \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} P'$.

The first claim follows from:

$$\begin{aligned}
M[x := N] &\equiv (V(\mathcal{F}P))[x := N] \\
&\equiv V[x := N](\mathcal{F}P[x := N]) \\
&\underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} \mathcal{F}(\lambda\kappa.P'[x := N'](\lambda v.\kappa(V'[x := N']v))) \\
&\qquad \text{by inductive hypothesis for} \\
&\qquad\quad P[x := N] \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} P'[x := N'] \text{ and} \\
&\qquad\quad V[x := N] \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} V'[x := N'] \\
&\equiv \mathcal{F}(\lambda\kappa.P'(\lambda v.\kappa(V'v)))[x := N'] \\
&\equiv M'[x := N'].
\end{aligned}$$

The second claim is verified by:

$$\begin{aligned}
s_L &= s_{V[x:=N]\xrightarrow[1]{}V'[x:=N']} + s_{P[x:=N]\xrightarrow[1]{}P'[x:=N']} \\
&\leq s_{V\xrightarrow[1]{}V'} + n(x,V')s_{N\xrightarrow[1]{}N'} + s_{P\xrightarrow[1]{}P'} + n(x,P')s_{N\xrightarrow[1]{}N'} \\
&\quad \text{by inductive hypothesis for} \\
&\qquad s_{P[x:=N]\xrightarrow[1]{}P'[x:=N']} \text{ and} \\
&\qquad s_{V[x:=N]\xrightarrow[1]{}V'[x:=N']} \\
&= [s_{V\xrightarrow[1]{}V'} + s_{P\xrightarrow[1]{}P'}] + [n(x,P') + n(x,V')]s_{N\xrightarrow[1]{}N'} \\
&< s_{M\xrightarrow[1]{}M'} + n(x,M')s_{N\xrightarrow[1]{}N'} + 1 \\
&= s_R.
\end{aligned}$$

3. For the third group we pick the case

$$M \equiv PQ \xrightarrow[1]{} P'Q' \equiv M',$$

where $P \xrightarrow[1]{} P'$ and $Q \xrightarrow[1]{} Q'$.

But then by inductive hypothesis it follows that

$$P[x := N] \xrightarrow[1]{} P'[x := N'] \text{ and } Q[x := N] \xrightarrow[1]{} Q'[x := N'],$$

and

$$\begin{aligned}
s_{P[x:=N]\xrightarrow[1]{}P'[x:=N']} &\leq s_{P\xrightarrow[1]{}P'} + n(x,P')s_{N\xrightarrow[1]{}N'} \\
s_{Q[x:=N]\xrightarrow[1]{}Q'[x:=N']} &\leq s_{Q\xrightarrow[1]{}Q'} + n(x,Q')s_{N\xrightarrow[1]{}N'}.
\end{aligned}$$

From this we immediately get

$$\begin{aligned}
M[x := N] &\equiv P[x := N]Q[x := N] \\
&\xrightarrow[1]{} P'[x := N']Q'[x := N'] \\
&\equiv M'[x := N']
\end{aligned}$$

and

$$\begin{aligned} s_L &= s_{P[x:=N]\xrightarrow[1]{}P'[x:=N']} + s_{Q[x:=N]\xrightarrow[1]{}Q'[x:=N']} \\ &\leq s_{P\xrightarrow[1]{}P'} + n(x,P')s_{N\xrightarrow[1]{}N'} + s_{Q\xrightarrow[1]{}Q'} + n(x,Q')s_{N\xrightarrow[1]{}N'} \\ &< [s_{P\xrightarrow[1]{}P'} + s_{Q\xrightarrow[1]{}Q'}] + [n(x,P') + n(x,Q')]s_{N\xrightarrow[1]{}N'} + 1 \\ &= s_{M\xrightarrow[1]{}M'} + n(x,M')s_{N\xrightarrow[1]{}N'} = s_R. \qquad \square \end{aligned}$$

For our purposes, Lemma 5.12 requires a simple extension showing that the two contractums of $\mathcal{F}$- and $\mathcal{D}$-redexes also reduce to each other in one $\xrightarrow[1]{}$ -step if the respective subterms do. As before, we add secondary claims for the proof of the Standardization Theorem:

Lemma 5.12 (part II). *Suppose* $M \xrightarrow[1]{} M'$, $N \xrightarrow[1]{} N'$, $V \xrightarrow[1]{} V'$, *and* $W \xrightarrow[1]{} W'$. *Then the following statements hold:*

(i) $\mathcal{F}(\lambda\kappa.M(\lambda f.\kappa(fN))) \xrightarrow[1]{} \mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))$

$s_{\mathcal{F}(\lambda\kappa.M(\lambda f.\kappa(fN)))\xrightarrow[1]{}\mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))} < s_{(\mathcal{F}M)N\xrightarrow[1]{}\mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))}$

(ii) $\mathcal{F}(\lambda\kappa.M(\lambda v.\kappa(Vv))) \xrightarrow[1]{} \mathcal{F}(\lambda\kappa.M'(\lambda v.\kappa(V'v)))$

$s_{\mathcal{F}(\lambda\kappa.M(\lambda v.\kappa(Vv)))\xrightarrow[1]{}\mathcal{F}(\lambda\kappa.M'(\lambda v.\kappa(V'v)))} < s_{V(\mathcal{F}M)\xrightarrow[1]{}\mathcal{F}(\lambda\kappa.M'(\lambda v.\kappa(V'v)))}$

(iii) $(\mathcal{D}\ (\lambda x.MxN)x) \xrightarrow[1]{} (\mathcal{D}\ (\lambda x.M'xN')x)$

$s_{(\mathcal{D}\ (\lambda x.MxN)x)\xrightarrow[1]{}(\mathcal{D}\ (\lambda x.M'xN')x)} < s_{(\mathcal{D}\ Mx)N\xrightarrow[1]{}(\mathcal{D}\ (\lambda x.M'xN')x)}$

(iv) $(\mathcal{D}\ (\lambda x.V(Mx))x) \xrightarrow[1]{} (\mathcal{D}\ (\lambda x.V'(M'x))x)$

$s_{(\mathcal{D}\ (\lambda x.V(Mx))x)\xrightarrow[1]{}(\mathcal{D}\ (\lambda x.V'(M'x))x)} < s_{V(\mathcal{D}\ Mx)\xrightarrow[1]{}(\mathcal{D}\ (\lambda x.V'(M'x))x)}$

(v) $(\mathcal{D}\ (\lambda w.MwN)W') \xrightarrow[1]{} (\mathcal{D}\ (\lambda w.M'wN')W'')$

$s_{(\mathcal{D}\ (\lambda w.MwN)W')\xrightarrow[1]{}(\mathcal{D}\ (\lambda w.M'wN')W'')} < s_{(\mathcal{D}\ MW')N\xrightarrow[1]{}(\mathcal{D}\ (\lambda w.M'wN')W'')}$

(vi) $(\mathcal{D}\ (\lambda w.V(Mw))W') \xrightarrow[1]{} (\mathcal{D}\ (\lambda w.V'(M'w))W'')$

$s_{(\mathcal{D}\ (\lambda w.V(Mw))W')\xrightarrow[1]{}(\mathcal{D}\ (\lambda w.V'(M'w))W'')} < s_{V(\mathcal{D}\ MW')\xrightarrow[1]{}(\mathcal{D}\ (\lambda w.V'(M'w))W'')}$

Proof. We show (i) with two straightforward calculations:

$N \xrightarrow[1]{\twoheadrightarrow} N'$ hence $\lambda f.(\kappa(fN)) \xrightarrow[1]{\twoheadrightarrow} \lambda f.(\kappa(fN'))$,

$M \xrightarrow[1]{\twoheadrightarrow} M'$ hence $M(\lambda f.(\kappa(fN))) \xrightarrow[1]{\twoheadrightarrow} M'(\lambda f.(\kappa(fN')))$,

and $\lambda\kappa.(M(\lambda f.(\kappa(fN)))) \xrightarrow[1]{\twoheadrightarrow} \lambda\kappa.(M'(\lambda f.(\kappa(fN'))))$,

and therefore $\mathcal{F}(\lambda\kappa.M(\lambda f.\kappa(fN))) \xrightarrow[1]{\twoheadrightarrow} \mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))$.

Similarly,

$$\begin{aligned} s_{\mathcal{F}(\lambda\kappa.M(\lambda f.\kappa(fN)))\xrightarrow[1]{}\mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))} &= s_{M\xrightarrow[1]{\twoheadrightarrow}M'} + s_{N\xrightarrow[1]{\twoheadrightarrow}N'} \\ &< s_{M\xrightarrow[1]{\twoheadrightarrow}M'} + s_{N\xrightarrow[1]{\twoheadrightarrow}N'} + 1 \\ &= s_{(\mathcal{F}M)N\xrightarrow[1]{\twoheadrightarrow}\mathcal{F}(\lambda\kappa.M'(\lambda f.\kappa(fN')))} \end{aligned}$$

The proofs for the other statements follow the same pattern. □

Remark. Similar statements also hold for λ_L-, λ_R-, σ_L, and σ_R-contractums, but are omitted because of their simplicity. **End**

Everything is now in place to state and prove the diamond lemma for the parallel reduction:

Lemma 5.13. *The relation* $\xrightarrow[1]{\twoheadrightarrow}$ *satisfies the diamond property, i.e., if* $M \xrightarrow[1]{\twoheadrightarrow} L_i$ *then there exists an* N *such that* $L_i \xrightarrow[1]{\twoheadrightarrow} N$ *for* $i = 1, 2$.

Proof. The proof is an induction on the structure of the reduction $M \xrightarrow[1]{\twoheadrightarrow} L_1$. Again, we discuss three prototypical cases:

1. $M \xrightarrow[1]{\twoheadrightarrow} M \equiv L_1$. Pick $N \equiv L_2$.
2. $M \equiv U((\sigma x.P)V) \xrightarrow[1]{\twoheadrightarrow} (\sigma x.U_1P_1)V_1 \equiv L_1$.

   Since U is a value, there are only two possible subcases for the reduction from M to L_2:

   a) $L_2 \equiv U_2((\sigma x.P_2)V_2)$ because $U \xrightarrow[1]{\twoheadrightarrow} U_2$, $P \xrightarrow[1]{\twoheadrightarrow} P_2$, and $V \xrightarrow[1]{\twoheadrightarrow} V_2$.

   An invocation of the inductive hypothesis yields terms U_3, V_3, and P_3 such that $U_i \xrightarrow[1]{\twoheadrightarrow} U_3$, $V_i \xrightarrow[1]{\twoheadrightarrow} V_3$, and $P_i \xrightarrow[1]{\twoheadrightarrow} P_3$ for $i = 1, 2$. The claim then follows from setting $N \equiv (\sigma x.U_3P_3)V_3$.

b) $L_2 \equiv (\sigma x.U_2 P_2)V_2$ because $U \xrightarrow[1]{} U_2$, $P \xrightarrow[1]{} P_2$, and $V \xrightarrow[1]{} V_2$.

As in the preceding case, the term $N \equiv (\sigma x.U_3 P_3)V_3$ constitutes the missing piece and can be derived with the help of the inductive hypothesis.

Other cases in this group make also use of Lemma 5.12.

3. $M \equiv (\mathcal{D}\ Px) \xrightarrow[1]{} (\mathcal{D}\ P_1 x) \equiv L_1$ because $P \xrightarrow[1]{} P_1$.

But then the only alternative for L_2 is $(\mathcal{D}\ P_2 x)$ where $P \xrightarrow[1]{} P_2$. Again, by inductive hypothesis there must be a P_3 such that $P_i \xrightarrow[1]{} P_3$ for $i = 1, 2$ and consequently $N \equiv (\mathcal{D}\ P_3 x)$. □

From this lemma, the Church-Rosser property of **cs** follows:

Corollary 5.14. *The notion of reduction* **cs** *is Church-Rosser.*

Proof. Since $\xrightarrow{}_{cs}$ is the transitive closure of **cs**, it is also the transitive closure of $\xrightarrow[1]{}$. As a result, the reduction $\xrightarrow{}_{cs}$ inherits the diamond property from the parallel reduction relation. □

With the Church-Rosser theorem in place, we can tackle the Standardization Theorem:

Theorem 5.9 (Standardization).

(i) $M \xrightarrow{*}_{scs} N$ *if and only if there is a CS-*SRS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$;

(ii) $M \xrightarrow{\triangleright *}_{scs} N$ *if and only if there is a CS-*SCS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$.

Proof. For both parts, the direction from right to left is trivial. The other directions need some elaboration:

(i) For the standardization property of the sub-calculus, we follow Plotkin's plan for the corresponding theorem about the λ_v-calculus. First, the sequence of $\xrightarrow{}_{cs}$-steps is replaced by a sequence of steps using the parallel reduction $\xrightarrow[1]{}$. This

follows from Lemma 5.11. Then we iteratively transform the parallel reduction sequence into a CS-SRS. The basis for this step is developed in Lemma 5.16.

(ii) Assuming that the proof of part (i) can be completed, we can prove (ii) with an induction on the number of computation steps in the sequence from M to N. If there are no computation steps, we can use (i) for forming a CS-SRS from M to N. Otherwise, there is at least one computation step and we must have the following situation:

$$L_1 \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_m \rhd L_{m+1} \rhd_{cs}^{*} L_n.$$

Again by (i), we can form a CS-SRS $K_1, \ldots, K_l$ for the reduction $L_1 \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_m$ where $K_1 \equiv L_1$ and $K_l \equiv L_m$. Since L_m is a computational redex, there is a maximal prefix of this CS-SRS whose terms are related via the standard reduction function, *i.e.*, for some j, $1 \leq j \leq l$ there is a computational redex K_j of the same kind as L_m such that

$$L_1 \longmapsto_{scs}^{*} K_j \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_m \rhd L_{m+1} \rhd_{cs}^{*} L_n$$

and it is not the case that $K_j \longmapsto_{scs} K_{j+1}$. At this point, we can employ the proof of the Consistency Theorem, part (ii) and interchange the computation step with the reduction sequence:

$$L_1 \longmapsto_{scs}^{*} K_j \rhd P \longrightarrow\!\!\!\!\!\rightarrow_{cs} L_{m+1} \rhd_{cs}^{*} L_n,$$

for some term P. But this can be recast as

$$L_1 \overset{\rhd}{\longmapsto}_{scs}^{+} P \rhd_{cs}^{*} L_n$$

and, because there is one less computation step in the sequence from P to L_n, an application of the inductive hypothesis produces a CS-SCS for this second half

of the reduction. Together with the first half, this yields the desired CS-SCS from M to N. □

For completeness sake, we include a precise formulation of the prefix-property for CS-SRS-s:

Lemma 5.15. *If $N_1, \ldots, N_k$ is a CS-SRS where N_k is a computational redex, then there exists a j, $1 \leq j < k$ such that for all $i, 1 \leq i < j, N_i \longmapsto_{scs} N_{i+1}$, and for all i, $j \leq i < k, N_i \longrightarrow_{cs} N_{i+1}$ and it is not the case that $N_i \longmapsto_{scs} N_{i+1}$.*

Proof. By case analysis on N_k and induction on k. □

We can now turn to the completion of part (i) of the Standardization Theorem. What we must prove is that sequences of parallel reductions can be transformed into standard reduction sequences:

Lemma 5.16. *If $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_1$ and $N_1, \ldots, N_j$ is a CS-SRS then there exists a CS-SRS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N_j$.*

Proof. The proof is a lexicographic induction on j, on the size of the proof $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_1$, and on the structure of M. We proceed by case analysis on the last step in $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_1$ and treat three prototypical examples.

1. For the second possible group of parallel reductions we treat the case:

$$M \equiv (\mathcal{D}\ PV^l)Q \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} (\mathcal{D}\ (\lambda x.P_1 x Q_1)V_1^l) \equiv N_1$$

where $P \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} P_1$, $Q \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} Q_1$, and $V \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} V_1$. But then, M can immediately be standard reduced to $(\mathcal{D}\ (\lambda x.PxQ)V^l)$. Furthermore, from Lemma 5.12 we know that

$$(\mathcal{D}\ (\lambda x.PxQ)V^l) \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} (\mathcal{D}\ (\lambda x.P_1 x Q_1)V_1^l)$$

by a derivation that is shorter than the one for $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_1$. Therefore, by inductive hypothesis, we can find a CS-SRS from $(\mathcal{D}\ (\lambda x.P_1 x Q_1)V_1^l)$ to N_j from which we build the required CS-SRS from M to N_j.

2. The first half of the third group in Definition 5.10 is represented by

$$M \equiv (\sigma x.P) \xrightarrow[1]{} (\sigma x.P_1) \equiv N_1,$$

where $P \xrightarrow[1]{} P_1$. This implies that all $N_i \equiv (\sigma x.P_i)$. Thus, it is natural to consider the sequence

$$P \xrightarrow[1]{} P_1, P_2, \ldots, P_j.$$

Since P is a proper subterm of M, the inductive hypothesis applies and there is a CS-SRS from P to P_j. Building a σ-capability from every element in this sequence with $\sigma x. \ldots$ yields the required reduction sequence.

3. The third example is taken from the second half of the third group:

$$M \equiv PQ \xrightarrow[1]{} P_1 Q_1 \equiv N_1,$$

where $P \xrightarrow[1]{} P_1$ and $Q \xrightarrow[1]{} Q_1$. In this case, we must distinguish the two alternatives of forming the sequence $N_1, \ldots N_j$:

a) $N_1, \ldots, N_j$ is the result of merging two sequences $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_l$: Definition 5.8 (2). As in the preceding case, the inductive hypothesis immediately produces two CS-SRS-s $P, \ldots, P_k$ and $Q, \ldots, Q_l$ and merging the two sequences results in a CS-SRS from M to N.

b) $N_1, \ldots, N_j$ is the extension of a sequence $N_2, \ldots, N_j$ where $N_1 \longmapsto_{scs} N_2$: Definition 5.8 (3). Now, suppose we can show that

$$M \xrightarrow[1]{} N_1 \longmapsto_{scs} N_2$$

commutes into

$$M \longmapsto^{+}_{scs} K \xrightarrow[1]{} N_2$$

for some K. Then the result immediately follows from an application of the inductive hypothesis. □

The last lemma in this subsection shows that parallel reduction steps and standard reduction steps indeed commute as required by the preceding lemma:

Lemma 5.17. *If*

$$M \xrightarrow[1]{\twoheadrightarrow} L_1 \longmapsto_{scs} N$$

then there exists an L_2 *such that*

$$M \longmapsto^{+}_{scs} L_2 \xrightarrow[1]{\twoheadrightarrow} N.$$

Proof. An extension of Plotkin's proof of his Lemma 8 [47:140] goes through with almost no change. It is a lexicographic induction on the size of the reduction $M \xrightarrow[1]{\twoheadrightarrow} L_2$ and the size of M and is divided according to the last parallel reduction step. The last case deserves some explanation.

Assume that $M \equiv PQ \xrightarrow[1]{\twoheadrightarrow} P_1Q_1 \equiv L_1$ because $P \xrightarrow[1]{\twoheadrightarrow} P_1$, $Q \xrightarrow[1]{\twoheadrightarrow} Q_1$, and $L_1 \longmapsto_{scs} N$. We proceed by case analysis on the standard reduction step and consider a typical subcase:

$$L_1 \equiv (\mathcal{F}P_1')Q_1 \longmapsto_{scs} \mathcal{F}(\lambda\kappa.P_1'(\lambda f.\kappa(fQ_1))) \equiv N.$$

But, given that $P \xrightarrow[1]{\twoheadrightarrow} \mathcal{F}P_1'$, we claim that there is an R such that

$$P \longmapsto^{*}_{scs} (\mathcal{F}R) \xrightarrow[1]{\twoheadrightarrow} \mathcal{F}P_1'.$$

With this, we can derive

$$PQ \longmapsto^{*}_{scs} (\mathcal{F}R)Q \longmapsto_{scs} \mathcal{F}(\lambda\kappa.R(\lambda f.\kappa(fQ))) \xrightarrow[1]{\twoheadrightarrow} \mathcal{F}(\lambda\kappa.P_1'(\lambda f.\kappa(fQ_1))),$$

and hence, $L_2 \equiv \mathcal{F}(\lambda\kappa.R(\lambda f.\kappa(fQ)))$.

The preceding analysis shows that the claim of the lemma reduces to the following two statements:

(i) if $M \xrightarrow[1]{} N$ where M is an application and N is a λ-abstraction or a computational redex, then there is a λ-abstraction or computational redex L such that $M \longmapsto^*_{scs} L \xrightarrow[1]{} N$;

(ii) if an application M parallel reduces to a constant a or a variable x, then M standard reduces to a and x, respectively.

Both statements can be verified with a straightforward induction on the size of the parallel reduction. □

5.3. Correspondence

As discussed in Chapter 2, the correctness proof of a calculus requires answers to two questions:

1. Do the machine and the calculus compute the same answer for a program?

and

2. Does equality in the calculus imply operational equality on the machine?

Since we have generalized the concept of a calculus, we cannot expect that the respective theorems hold without modification. Our task in this section is to reformulate the relationships and to reassess the impact of the modifications on the reasoning system.

For the functional subset of Idealized Scheme, the CEK-evaluation function and the standard reduction function produce the same value for a given program. To prove the equivalent result for the λ_v-CS-calculus, we must show that the standard computation function satisfies the C-transition rules. The proof statement is familiar from the simulation lemmas in the preceding chapter:

$$\text{if } M \xmapsto{C} N \text{ then } \Phi(M) \xmapsto{\triangleright}{}^{+}_{scs} \Phi(N).$$

This obviously holds for β_v- and δ-contractions. Furthermore, the design of the reduction and computation relations for λ-, σ-, and $\mathcal{D}$-applications is so closely

oriented at the respective C-rules that a proof of the above correspondence is practically built into the definition. Only the relations for $\mathcal{F}$-applications cause some problems.

An $\mathcal{F}$-application in the C-system applies its argument to $\lambda x.C[x]$ where $C[\ ]$ is the context of the application. In the calculus, however, the continuation function is gradually constructed from the context pieces. If $\mathcal{F}M$ is embedded in $C[[\ ]N]$ and K represents $C[\ ]$, M is eventually applied to $\lambda f.K(fN)$, which is equivalent to $\lambda f.K([\ ]N)$ filled with f. Similarly, an $\mathcal{F}$-application that sits to the left of a value V in some context $C[\ ]$ represents the continuation as the function $\lambda v.K(Vv)$. In short, an $\mathcal{F}$-application in the calculus applies its argument to a functional encoding of the context that is formed according to the following algorithm:

$$[\![\mathcal{F}P]\!]_c \equiv \lambda x.x,$$

$$[\![C[(\mathcal{F}P)M]]\!]_c \equiv \lambda f.[\![C[\mathcal{F}P]]\!]_c(fM),$$

$$[\![C[V(\mathcal{F}P)]]\!]_c \equiv \lambda v.[\![C[\mathcal{F}P]]\!]_c(Vv).$$

Given this characterization, we can formally state how the calculus computes imperative actions:

Proposition 5.18. *Let $C[\ ]$ be an evaluation context over Λ_{CS}. Then the following relationships hold:*

$$C[(\mathcal{D}\ MV^l)] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ C[M(V[\bullet^l := V^l])],$$

$$C[(\lambda x_\sigma.M)V] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ C[M[x := V^l]],\ \textit{where } l \textit{ is fresh},$$

$$C[(\mathcal{F}M)] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ M[\![C[\mathcal{F}P]]\!]_c,$$

$$C[(\sigma U^l.M)V] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ C[M][\bullet^l := V^l].$$

Proof. The claims follow from an induction on the structure of evaluation contexts. □

The major consequence of this proposition is that a naïve version of the simulation theorem fails. Whereas in the λ_v-CS-calculus continuations are constructed

to *simulate* the behavior of contexts, continuations in the rewriting system *are* contexts. Thus, when a continuation is to be captured after another one is invoked, the transition in the machine and the one via the standard computation function diverge: the rewriting system simply captures the current context, the standard computation sequence encodes the term that simulates the former continuation for a second time.

Let us illustrate the nature of the problem with an example. Suppose the continuation $\lambda f.D[fV]$ is invoked on the value U in the context $C[\ ]$:[1]

$$C[(\lambda f.D[fV])U] \stackrel{C}{\longmapsto} C[D[UV]].$$

Furthermore, assume that the application UV evaluates to $E[\mathcal{F}M]$ after some β_v-steps. Then the C-transition reaches the term $M(\lambda f.C[D[E[f]]])$. According to Proposition 5.18, the corresponding reduction sequence in the λ_v-CS-calculus begins with:

$$C[[\![D[(\mathcal{F}M)V]]\!]_c U] \equiv C[(\lambda f.[\![D[\mathcal{F}M]]\!]_c(fV))U] \stackrel{\triangleright}{\longmapsto}{}^*_{scs}\ C[[\![D[\mathcal{F}P]]\!]_c(U\ V)].$$

The next few β_v-steps are correctly simulated by the standard computation function:

$$C[[\![D[\mathcal{F}P]]\!]_c(U\ V)] \stackrel{\triangleright}{\longmapsto}{}^*_{scs}\ C[[\![D[\mathcal{F}P]]\!]_c E[\mathcal{F}M]].$$

Now, this last term also constructs a continuation—just like $C[D[E[\mathcal{F}M]]]$—except that the continuation encodes the term $[\![D[\mathcal{F}P]]\!]_c$ instead of the context $D[\ ]$:

$$C[[\![D[\mathcal{F}P]]\!]_c E[\mathcal{F}M]] \stackrel{\triangleright}{\longmapsto}{}^*_{scs}\ M[\![C[[\![D[\mathcal{F}P]]\!]_c E[\mathcal{F}M]]]\!]_c.$$

But clearly, $[\![C[D[E[\mathcal{F}P]]]]\!]_c$ is not equal to $[\![C[[\![D[\mathcal{F}P]]\!]_c E[\mathcal{F}M]]]\!]_c$ and therefore the two evaluation procedures proceed in different directions.

---

[1] For simplicity, we assume that none of the involved terms contains assignable variables and that the terms exist in both languages.

The best we can hope for is that the standard computation simulation of the C-transition function preserves a fixed relation between contexts-as-continuations and terms-as-continuations. From the above proposition and the example one may suspect that a continuation like $\lambda f.C[D[f]]$ is related to the terms $[\![C[D[\mathcal{F}P]]]\!]_c$ and $[\![[\![C[\mathcal{F}P]]\!]_c D[\mathcal{F}P]]\!]_c$. However, the situation in our example could recur many times. Instead of having two contexts composing a new one, we would have several of them: some representing former continuations, others newly generated evaluation contexts. A formal definition of the relationship must account for all possible finite decompositions of a given context into smaller contexts and for recursive instances of the above example. This leads to the formalization of the relation $\approx_p$ in Definition 5.19, which connects continuation representations in Λ_{rew} to continuation representations in Λ_{CS}.

Definition 5.19: The correspondence of continuations in Λ_{rew} and Λ_{CS}

The relation $\approx_p$ compares continuations in Λ_{rew} to those in Λ_{CS}:

$$\lambda x.C[x] \approx_p [\![C'_1[K'_2 C'_3[\ldots K'_{n-1} C'_n[\mathcal{F}P]\ldots]]]\!]_c$$

for all finite sequences $C_1[\ ], \ldots, C_n[\ ]$ of evaluation contexts such that

- $n \geq 1$,
- $C[\ ] \equiv C_1[C_2[\ldots C_n[\ ]\ldots]]$,
- and for all $i \geq 1$, $C_{2i-1}[\ ] \approx_p C'_{2i-1}[\ ]$ and $\lambda x.C_{2i}[x] \approx_p K'_{2i}$.

for arbitrary terms, we add

$$x_\lambda \approx_p x_\lambda,\ \lambda x.P \approx_p \lambda x.P',\ PQ \approx_p P'Q',\ \mathcal{F}P \approx_p \mathcal{F}P',$$
$$\sigma x.P \approx_p \sigma x.P',\ \sigma V'.P \approx_p \sigma V''.P',\ x_\sigma \approx_p \mathcal{D}\ |x_\sigma,\ V' \approx_p \mathcal{D}\ |V''|,$$
$$\text{if } P \approx_p P',\ Q \approx_p Q',\ \text{and } V \approx_p V';$$

and finally, we extend the relation to contexts by setting $[\ ] \approx_p [\ ]$.

After formalizing the correspondence of continuation representations, we can finally state a simulation theorem that correctly connects the C-evaluation function with the standard computation function:

Theorem 5.20 (Simulation). *For all programs $M \in \Lambda_{\mathcal{F}\sigma}$,*

$$eval_C(M) = V \text{ iff } \Phi(M) \overset{\triangleright}{\longmapsto}{}^{*}_{scs} U \text{ for some } U \text{ such that } V \approx_p U.$$

The proof of this theorem is rather technical and may be found in the subsection. Since the $\approx_p$-relation relates a constant only to itself, the theorem immediately implies

Corollary 5.21. *For all programs $M \in \Lambda_{\mathcal{F}\sigma}$ and $a \in BConst$,*

$$eval_C(M) = a \qquad \text{iff} \qquad \Phi(M) \overset{\triangleright}{\longmapsto}{}^{*}_{scs} a.$$

A consequence of the corollary is the correctness of the mathematical program interpretation. Recall that for a given arity n, the machine and the calculus each assign a function to a program M: $\mathcal{M}^n_M$ and $\mathcal{I}^n_M$, respectively. Their definitions are:

$$\mathcal{M}^n_M = \{\langle a_1, \ldots, a_n, c\rangle \mid eval_C(M a_1 \ldots a_n) = c\}$$
$$\mathcal{I}^n_M = \{\langle a_1, \ldots, a_n, c\rangle \mid \lambda_v\text{-CS}^{\triangleright} \vdash M a_1 \ldots a_n = c\},$$

where the a_i's and c stand for basic constants. The following corollary shows that the two functional interpretations always agree:

Corollary 5.22. *For all $n \geq 0$, $\mathcal{I}^n_M = \mathcal{M}^n_{\Phi(M)}$.*

Proof. The two function definitions are only concerned with basic, observable constants. Therefore, the corollary is a simple consequence of Corollary 5.21, which says that the two systems—rewriting system and calculus—produce the same basic constants. □

Another, more important implication of the Simulation Theorem and the associated corollary is that continuations in the C-transition system and a $\approx_p$-related function in Λ_{CS} are behaviorally equivalent. No matter which one occurs in a program, the program loops forever, gets stuck, or terminates; if the result is a basic constant, this constant is uniquely determined. Because of this behavioral equivalence, the definition of an operational equivalence relation for Idealized Scheme is independent of the particular evaluation mechanism. In order to express this relationship, we first restate the definition of operational equivalence in a generic form:

Definition 5.23. *$M, N \in L$ are* operationally equivalent, *$M \simeq N$, if for any arbitrary context $C[\ ]$ over L such that $C[M]$ and $C[N]$ are closed, the evaluation is undefined for both, or it is defined for both and if one of the programs yields a basic constant, then the value of the other is the same basic constant.*

The two instantiations of this definition that are of interest to us are $\simeq_{C,rew}$ and $\simeq_{C,CS}$. The first is based on Λ_{rew} and $eval_C$, the second on Λ_{CS} and the standard computation function. Since both subsume Idealized Scheme, we automatically have an operational equivalence for it. We must only show that the two definitions are equivalent:

Proposition 5.24. *$M \simeq_{C,rew} N$ iff $\Phi(M) \simeq_{C,CS} \Phi(N)$.*

Proof. From the Simulation Theorem and its corollary we know that—provided termination—a program M and its injected counterpart $\Phi(M)$ produce $\approx_p$-related values. Since $\approx_p$ relates a basic constant to itself, the result is immediate. □

Proposition 5.24 is particularly important for the second half of our investigations into the correctness of the λ_v-CS-calculus. The general objective of this phase is to establish a link between the calculi and the operational equivalence relation. However, unlike for functional languages, the calculus and the evaluation function

are defined over two different languages. Without the above proposition, this could cause some technical difficulties, but the proposition implies that comparisons can be based on $\simeq_{C,CS}$. Any equation that is provable for Λ_{CS}-programs also holds for the source programs in Λ_{rew} (or $\Lambda_{\mathcal{F}\sigma}$ if they do not contain labels). Because of this, we henceforth use $\simeq_C$, omitting the qualification.

The most important property of an operational equivalence relation is that it is a fully substitutional theory. In other words, it satisfies Leibnitz's Law that equals can be substituted for equals:

$$M \simeq_C N \Rightarrow C[M] \simeq_C C[N].$$

For reasoning about programs, this is desirable: to be reusable, verifications and transformations of program pieces should be independent of the context. Since this latter condition holds for the reduction-based sub-calculus, we can expect that **cs**-equality implies operational equality. Because of the computation relations, however, the context-insensitivity condition rules out that operational equality subsumes computational equality. Our first result is thus of mixed nature:

Theorem 5.25.

(i) If λ_v*-CS* $\vdash M = N$*, then* $M \simeq_C N$*. The converse is false.*

(ii) λ_v*-CS*$^{\triangleright} \vdash M = N$ *does* **not** *imply* $M \simeq_C N$*, nor vice versa.*

Proof. The proof of (i) and the right-to-left direction of (ii) are adaptations and generalizations of Plokin's corresponding theorem [47]: see the discussion following Theorem 2.14. For the left-to-right direction of (ii), consider proof steps like

$$(\mathcal{F}\lambda d.0) \stackrel{\triangleright}{=}_{cs} 0$$

in $M \stackrel{\triangleright}{=}_{cs} N$. They are specifically restricted to the root of a term and thus cannot be built into a congruence relation. The term $(\mathcal{F}\lambda d.0)$ is clearly different from 0 in all contexts except the empty one. □

According to this theorem, equational reasoning in the calculus is admissible as long as the equations do not include computations. But these relations are precisely the basis for the simulation of imperative effects and an equational theory for these effects is our central goal. Consequently, we must try to exploit computational equality and computations in a different way so that we can establish valid conclusions about operational equalities.

As pointed out, the major discrepancy between computational and operational equality is the context-sensitivity of the former: computation steps are only applicable in the empty context. On the other hand, computation steps are required because reductions alone can only perform the first part of a C-transition step. Together, the two classes of term relations form a program relation which proves equations of the form:

$$C[M] \stackrel{\triangleright}{=}_{cs} K.$$

In these equations, the context-sensitivity is represented by the evaluation context $C[\ ]$. This context appears in a possibly altered form in K. Hence, it is natural to wonder whether a universal quantification over this evaluation context implies a context-insensitive equivalence for M and N, *i.e.*, whether

$$C[M] \stackrel{\triangleright}{=}_{cs} C[N]$$

implies

$$M \simeq_C N.$$

Unfortunately, the answer is no.

Although the suggested condition is quite strong, it is not sufficient. Let us first illustrate some of its benefits. The condition clearly rules out arbitrary σ-contractions. If $C[\ ]$ contains a label l, a redex of the form $(\sigma U^l.M)V$ clearly

interferes with the context. In the derivation

$$C[(\sigma U^l.M)V] \stackrel{\triangleright}{=}_{cs} C[M][\bullet^l := V^l],$$

the replacement algorithm on l affects all occurrences of l-labeled values in $C[\ ]$. Similarly, but in a more subtle manner, the condition forbids arbitrary β_σ-type contractions of the form

$$(\lambda x.M)V = M[x := V^l].$$

The reason is again that some $C[\ ]$ may already contain the label l in which case l is not fresh.

Warning, part II. *These arguments should recall that programs, but* **not** *terms are considered modulo* $\equiv_{lab}$. **End**

The failure of the current proposal is caused by admitting unrestricted $\mathcal{D}$-redexes. Delabeling transitions never depend on the particular context, but on the *evaluation*-ness of the context. Arbitrary $\mathcal{D}$-contractions may interfere with pending assignments. The crucial part of this observation is that bad timing of a $\mathcal{D}$-contraction collides with the specific value parts of labeled values. This problem can be avoided, if we prohibit a theorem of the above form to depend on labeled values. We call such a theorem *safe* and formalize the concept in

Definition 5.26. *A theorem* λ_v-$CS^{\triangleright} \vdash M = N$ *is* safe *if for an arbitrary evaluation context* $C[\ ]$ *and some arbitrary values* $V_1, \ldots, V_n$

$$\lambda_v\text{-}CS^{\triangleright} \vdash C[M][\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}] = C[N][\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}]$$

where $Lab(M) \cup Lab(N) = \{l_1, \ldots, l_n\}$.

The adequacy of the safety condition is encapsulated in the central theorem of our work:

Theorem 5.27 (Safety). *If λ_v-CS$^{\triangleright} \vdash M = N$ is safe, then $M \simeq_C N$.*

Proof. Let $D[\ ]$ be an arbitrary context and assume that $D[M]$ evaluates to a basic constant a:

$$\lambda_v\text{-CS}^{\triangleright} \vdash D[M] \overset{\triangleright}{\longmapsto}{}^{*}_{scs}\ a.$$

If M plays an active role in this derivation, a closed and possibly side-effected version M' must occur in an evaluation context $C[\ ]$:

$$\lambda_v\text{-CS}^{\triangleright} \vdash D[M] \overset{\triangleright}{\longmapsto}{}^{*}_{scs}\ C[M'] \overset{\triangleright}{\longmapsto}{}^{*}_{scs}\ a.$$

Without loss of generality, we can say that $M' \equiv M[x := U][y := V^m][\bullet^l := W^l]$ where x and y are the free variables of M, and l is the affected label that occurs in M. From the principal assumption that there is a safe derivation for

$$\lambda_v\text{-CS}^{\triangleright} \vdash M = N,$$

it follows that

$$\lambda_v\text{-CS}^{\triangleright} \vdash C[M'] = C[N']$$

where $N' \equiv N[x := U][y := V^m][\bullet^l := W^l]$. The Safety Condition guarantees that control- and side-effects cannot interfere with the proof. The Substitution Theorem provides for orthogonality of variable substitution: its antecedent is satisfied because U and V were a part of the program all along.

Given this, we can replace the above derivation by

$$\lambda_v\text{-CS}^{\triangleright} \vdash D[M] = C[N'] = C[M'] \overset{\triangleright}{\longmapsto}{}^{*}_{scs}\ a.$$

This can be done for every occurrence of M in an evaluation context in the rest of the standard computation sequence and therefore the entire derivation is independent of M:

$$\lambda_v\text{-CS}^{\triangleright} \vdash D[N] = a.$$

By the Consistency, the Standardization Theorem, and the corollary to the Simulation Theorem, it follows that

$$\lambda_v\text{-CS}^{\triangleright} \vdash D[N] \xrightarrow{\triangleright}{}^{*}_{scs} \; a$$

as desired. □

Theorem 5.27 provides the basis for a useful equational theory about imperative effects. To this end, we define the set of safe theorems

$$Th_0^{safe} = \{M = N \mid \lambda_v\text{-CS}^{\triangleright} \vdash M = N \text{ and } M = N \text{ is safe}\}$$

and form its compatible closure

$$Th^{safe} = \{C[M] = C[N] \mid M = N \in Th_0^{safe} \text{ and } C[\ ] \text{ is arbitrary}\}.$$

This theory can now be added to the reduction calculus and we obtain a quasi-calculus of safe-theorems. Theoremhood in this extended calculus is denoted by

$$\lambda_v\text{-CS} \cup Th^{safe} \vdash M = N;$$

we also use the abbreviations

$$M =_{cs,safe} N \text{ and } \lambda_v\text{-CS}^{safe} \vdash M = N$$

This level of the calculus comprises all lower levels, *i.e.*, **v**- and **cs**-equality, as well as imperative derivations whose effect is invisible to an outside observer. This and other facts about the extended calculus are gathered in a corollary to the above theorem:

Corollary 5.28.

(i) $\lambda_v \vdash M = N$ *implies* $\lambda_v\text{-CS} \vdash M = N$ *implies* $\lambda_v\text{-CS}^{safe} \vdash M = N$;

(ii) $\lambda_v\text{-}CS^{safe} \vdash M = N$ *implies* $M \simeq_C N$;

(iii) $\simeq_C$ *is the largest, consistent extension of* $=_{cs,safe}$ *and* $=_{cs}$ *that respects equality on basic constants, and that satisfies*

— $M \simeq_C N$ *implies* $C[M] \simeq_C C[N]$,

— $M \simeq_C N$ *implies* $C[M]$ *has a value iff* $C[N]$ *has a value for all (closing) program contexts* $C[\ ]$.

This last corollary is a good starting point for a summary of our development. Altogether we have defined 5 new calculus-related comparison relations. On the static side, there are the extended α-congruence relation: $\equiv_\alpha$ on terms and the label-equivalence: $\equiv_{lab}$ on programs. On the dynamic side, we have the reduction-based **cs**-equality: $=_{cs}$ on terms, computational **cs**-equality: $\stackrel{\triangleright}{=}_{cs}$ on programs, and the quasi-calculus of safe **cs**-equality, all of which are conservative extensions of the λ_v-calculus.

The mutual relationship of the term relations is simple: α-congruence, **v**-equality, **cs**-equality, and safe **cs**-equality form an ascending chain of equivalence relations. Similarly, α-equivalence, $\equiv_{lab}$-equivalence, and computational **cs**-equality form a chain of program comparisons. However, whereas the former imply operational equivalence on the CESK-machine, the latter do not. The purpose of program relations is to compare labeled programs and to determine the values of programs. It is therefore natural to call the system of term relations the reasoning part of the calculus and the system of computation relations the evaluation part.

The disappointing part of our work is that, unlike in the pure framework, the evaluation part of the calculus is not equal to the reasoning part and *vice versa*. That is, we need two different systems for manipulating imperative programs, depending on whether we want to evaluate programs or compare expressions. This is the major difference between functional and state-of-the-art imperative programming

languages. Although this is disappointing, we shall show in the next chapter that it is possible to live with the available tools: reasoning about imperative programs can proceed in almost the same algebraic manner that we are used to from the functional world.

5.3.1. Proof for the Simulation Theorem

Due to the mismatch between continuation representations in the C-rewriting system and the λ_v-CS-calculus, the proof of the Simulation Theorem requires two parts. The first part shows that the standard computation function correctly simulates single C-transitions steps on related terms as long as the program does not invoke a continuation. This part directly follows from Proposition 5.18. In the second part, we prove that the calculus also handles the invocation of continuations in the right way. More precisely, when the C-rewriting system evaluates a continuation invocation, then sooner or later the standard computation function transforms a related continuation invocation into a related term. Together the two parts suffice to prove that related programs have related results.

The following first lemma is a partial simulation result for the non-continuation related sub-calculus. Since the result is later applied in situations where the respective evaluation contexts are not related, we do not include any assumptions about the evaluation contexts. If we do know that the evaluation contexts are related as continuation representations, we can also show that the grabbing of continuations yields related terms:

Lemma 5.29. *Let $C[\ ]$ and $C'[\ ]$ be arbitrary evaluation contexts over Λ_{rew} and Λ_{CS}, respectively. Then, for $M \approx_p M'$ and $V \approx_p V'$, the following relationships hold:*

(i) if $C[(\mathcal{D}\ MV^l)] \xrightarrow{C} C[MV[\bullet^l := V^l]]$ *then*

$$C'[(\mathcal{D}\ M'V'^l)] \overset{+}{\underset{scs}{\rhd\!\!\longmapsto}}\ C'[M'V'[\bullet^l := V'^l]];$$

(ii) if $C[(\lambda x.M)V] \xrightarrow{C} C[M[x := V]]$ *then*

$$C'[(\lambda x.M')V'] \overset{+}{\underset{scs}{\rhd\!\!\longmapsto}}\ C'[M'[x := V']];$$

(iii) if $C[(\lambda x_\sigma.M)V] \xrightarrow{C} C[M[x := V^l]]$ *then*

$$C'[(\lambda x_\sigma.M')V'] \overset{+}{\underset{scs}{\rhd\!\!\longmapsto}}\ C'[M'[x := V'^l]];$$

(iv) if $C[(\sigma U^l.M)V] \xrightarrow{C} C[M][\bullet^l := V^l]$ *then*

$$C'[(\sigma U'^l.M')V'] \overset{+}{\underset{scs}{\rhd\!\!\longmapsto}}\ C'[M'][\bullet^l := V'^l];$$

and, furthermore, since $M \approx_p M'$ *and* $V \approx_p V'$, *we also have*

$$M[x := V] \approx_p M'[x := V'],$$
$$M[x := V^l] \approx_p M'[x := V'^l],$$
$$M[\bullet^l := V^l] \approx_p M'[\bullet^l := V'^l].$$

(v) Finally, if $C[(\mathcal{F}M)] \xrightarrow{C} M(\lambda x.C[x])$ *and* $\lambda x.C[x] \approx_p [\![C'[\mathcal{F}P]]\!]_c$, *then*

$$C'[(\mathcal{F}M')] \overset{+}{\underset{scs}{\rhd\!\!\longmapsto}}\ M'[\![C'[\mathcal{F}P]]\!]_c$$

and $M(\lambda x.C[x]) \approx_p M'[\![C'[\mathcal{F}P]]\!]_c$.

Proof. Parts (i) through (v) are consequences of Proposition 5.18. For the statements about substitution and labeled-value substitution we observe that occurrences of free variables and labels are orthogonal to the relation $\approx_p$. □

The most important consequence of the lemma is that we can henceforth ignore the issue of side-effects. The standard computation function simulates side-effects and delabeling steps correctly, and we consider it therefore unnecessary to overburden the rest of the proofs with respective clauses. Strictly speaking, the subsequent statements are wrong in the sense that they do not subsume the possibility of side-effects, but it is clear that they can easily be fixed with the preceding lemma. We exemplify the abbreviated statements with the following summary of the first four parts of the lemma:

Corollary 5.30. *Let $C[\ ]$ and $C'[\ ]$ be arbitrary evaluation contexts and let M, M' be related terms, i.e., $M \approx_p M'$. Then, a rewriting sequence from M to a term N in some evaluation context $D[\ ]$:*

$$C[M] \overset{C}{\longmapsto}^{+} C[D[N]]$$

without use of the transition rule (C7) or invocation of a continuation implies that there are N' and $D'[\ ]$ such that

$$C'[M'] \overset{\triangleright}{\longmapsto}^{+}_{scs} C'[D'[N']],$$

$N \approx_p N'$ and $D[\ ] \approx_p D'[\ ]$.

Next we must consider the invocation of continuations. From the discussion in the main body of this section and Definition 5.19, it immediately follows that the standard computation function does *not* simulate every C-rewriting step of such an invocation on a step-by-step basis. On the other hand, we know that there are only three possibilities for the outcome of such a rewriting sequence within a given context: it may terminate with a value, it may yield a (first) $\mathcal{F}$-application, or it may diverge. The following lemma treats the first two of these cases. The proof requires a measure of the relationships of the involved continuations: we call *the*

degree of the relationship $K \approx_p K'$ *simple* if

$$K \equiv \lambda x.C[x] \approx_p [\![C[\mathcal{F}P]]\!]_c \equiv K';$$

the relationship

$$K \equiv \lambda x.C_1[C_2[\ldots C_n[x]\ldots]] \approx_p [\![C'_1[K'_2C'_3[\ldots K'_{n-1}C'_n[\mathcal{F}P]\ldots]]]\!]_c \equiv K'$$

is of higher degree than all of the relationships $\lambda x.C_{2i}[x] \approx_p K'_{2i}$. With this definition, we can perform inductions on the relationships between continuations:

Lemma 5.31. *Let* $C[\ ]$ *and* $C'[\ ]$ *be arbitrary evaluation contexts and let* K, K', V, *and* V' *be related continuations and values, respectively, i.e.,* $K \approx_p K'$, $V \approx_p V'$. *Then,*

(i) a rewriting sequence from KV *to a value* W *within the context* $C[\ ]$:

$$C[KV] \overset{C}{\longmapsto}^{+} C[W]$$

without use of the (C7) transition rule (grabbing of a continuation) implies that there is a value W' *such that*

$$C'[K'V'] \overset{\triangleright}{\longmapsto}^{+}_{scs} C'[W']$$

and $W \approx_p W'$;

(ii) a rewriting sequence from some M *to* V:

$$C[M] \overset{C}{\longmapsto}^{+} C[V]$$

without use of the transition rule (C7) implies that for every related M', *i.e.,* $M \approx_p M'$ *there is a value* V' *such that*

$$C'[M'] \overset{\triangleright}{\longmapsto}^{+}_{scs} C'[V']$$

and $V \approx_p V'$;

(iii) a rewriting sequence from KV *to a first* $\mathcal{F}$*-application* $\mathcal{F}M$ *within the context* $C[\ ]$:

$$C[KV] \overset{C}{\longmapsto}^{+} C[D[\mathcal{F}M]]$$

for some term M *and context* $D[\ ]$ *implies that there are* M' *and* $D'[\ ]$ *such that*

$$C'[K'V'] \overset{\triangleright}{\longmapsto}^{+}_{scs} C'[D'[\mathcal{F}M']],$$

and $M \approx_p M'$ *and* $\lambda x.D[x] \approx_p [\![D'[\mathcal{F}P]]\!]_c$.

Proof. The proof of the first part is a simplified version of the proof of the following third part. In order to avoid repetitions, we omit it. The second claim is a simple consequence of the first part.

The proof of claim (iii) is a lexicographic induction on the number of rewriting steps and the degree of $K \approx_p K'$. Thus, suppose the relationship is simple, that is,

$$K \equiv \lambda x.E[x] \text{ and } K' \equiv [\![E'[\mathcal{F}P]]\!]_c.$$

Since $E[\ ] \equiv [\ ]$ is impossible—otherwise the invocation KV would immediately return the value V—we must consider two cases:

a) $E[\ ] \equiv F[U[\ ]]$ for some context $F[\ ]$ and value U. Then $E'[\ ] \equiv F'[U'[\ ]]$ and the two evaluation sequences proceed as follows:

$$C[KV] \overset{C}{\longmapsto} C[F[UV]]$$

and

$$C'[K'V'] \overset{\triangleright}{\longmapsto}_{scs} C'[[\![F[\mathcal{F}P]]\!]_c(U'V')].$$

Depending on the outcome of the evaluation of UV, there are three possible subcases:

a1) the evaluation of UV yields a value W without grabbing a continuation:

$$C[F[UV]] \stackrel{C}{\longmapsto}^{+} C[F[W]].$$

According to part (ii), the standard computation function produces a related value W':

$$C'[[F'[\mathcal{F}P]]_c(U'V')] \stackrel{\triangleright}{\longmapsto}^{+}_{scs} C'[[F'[\mathcal{F}P]]_c W'].$$

Now, we can interpret the term $F[W]$ as a continuation invocation:

$$C[(\lambda x.F[x])W] \stackrel{C}{\longmapsto} C[F[W]],$$

and, furthermore, we know that this continuation invocation rewrites to the assumed $\mathcal{F}$-application in fewer steps than the original continuation invocation KV. Hence, an application of the inductive hypothesis yields the desired result:

$$C'[[F'[\mathcal{F}P]]_c W'] \stackrel{\triangleright}{\longmapsto}^{+}_{scs} C'[D'[\mathcal{F}M']].$$

a2) the application UV directly results in an $\mathcal{F}$-application without invoking a continuation:

$$C[F[UV]] \stackrel{C}{\longmapsto}^{+} C[F[D_1[\mathcal{F}M]]].$$

Corollary 5.30 implies that the standard reduction sequence leads to a related $\mathcal{F}$-application:

$$C'[[F'[\mathcal{F}P]]_c(U'V')] \stackrel{\triangleright}{\longmapsto}^{+}_{scs} C'[[F'[\mathcal{F}P]]_c D_1'[\mathcal{F}M']]$$

where $D_1[\ ] \approx_p D_1'[\ ]$. It follows that the contexts

$$D[\ ] \equiv F[D_1[\ ]]$$

and

$$D'[\ ] \equiv [\![F[\mathcal{F}P]]\!]_c D_1'[\ ]$$

satisfy the required relationship.

a3) UV rewrites to a (first) continuation invocation K_1V_1 that in turn leads to an $\mathcal{F}$-application:

$$C[F[UV]] \overset{C}{\longmapsto}{}^{*}\ C[F[C_1[K_1V_1]]] \overset{C}{\longmapsto}{}^{+}\ C[F[C_1[D_1[\mathcal{F}M]]]].$$

Once again, we apply Corollary 5.30 and accordingly claim that

$$C'[[\![F'[\mathcal{F}P]]\!]_c(U'V')] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ C'[[\![F'[\mathcal{F}P]]\!]_c C_1'[K_1'V_1']]$$

where $C_1[\ ] \approx_p C_1'[\ ]$, $K_1 \approx_p K_1'$, and $V_1 \approx_p V_1'$. Since the rewriting sequence

$$C[F[C_1[K_1V_1]]] \overset{C}{\longmapsto}{}^{+}\ C[F[C_1[D_1[\mathcal{F}M]]]]$$

is at least one step shorter than the original one, we can invoke the inductive hypothesis and get

$$C'[[\![F'[\mathcal{F}P]]\!]_c C_1'[K_1'V_1']] \overset{\triangleright}{\longmapsto}{}^{+}_{scs}\ C'[[\![F'[\mathcal{F}P]]\!]_c C_1'[D_1'[\mathcal{F}M']]].$$

The induction hypothesis also yields that

$$\lambda x.D_1[x] \approx_p [\![D'[\mathcal{F}P]]\!]_c$$

and therefore

$$\lambda x.F[C_1[D_1[x]]] \approx_p [[\![F'[\mathcal{F}P]]\!]_c C_1'[D_1'[\mathcal{F}P]]]\!]_c$$

as required.

b) $E[\ ] \equiv F[[\ ]M]$ for some context $F[\ ]$ and expression M. This case is treated like a).

The preceding arguments mostly carry over to the case when the relationship between K and K' is more complex, *i.e.*,

$$K \equiv \lambda x.E[x] \approx_p [\![E'_1[K'_2E'_3[\ldots K'_{n-1}E'_n[\mathcal{F}P]\ldots]]]\!]_c \equiv K'$$

for some finite decomposition $E_1[\ ], \ldots, E_n[\ ]$ of $E[\ ]$. The only interesting difference is the case $E_1[\ ] \equiv [\ ]$. Then, the two transition sequences develop as follows:

$$C[KV] \stackrel{C}{\longmapsto} C[E_1[\ldots E_{n-1}[V]\ldots]]$$

and

$$C'[K'V'] \stackrel{\triangleright}{\longmapsto}_{scs} C'[[\![E'_1[\ldots E'_n[\mathcal{F}P]\ldots]]\!]_c(K'_{n-1}V')].$$

At this point, we can again apply our trick and consider $E_{n-1}[V]$ as the result of applying the continuation $\lambda x.E_{n-1}[x]$ to V:

$$C[E_1[\ldots((\lambda x.E_{n-1}[x])V)\ldots]] \stackrel{C}{\longmapsto} C[E_1[\ldots E_{n-1}[V]\ldots]].$$

If this continuation invocation yields a value W without grabbing a continuation, we have

$$C[KV] \stackrel{C}{\longmapsto}^{+} C[E_1[\ldots E_{n-2}[W]\ldots]]$$

with at least one step, and, by part (i),

$$C'[K'V'] \stackrel{\triangleright}{\longmapsto}^{+}_{scs} C'[[\![E'_1[\ldots E'_{n-2}[\mathcal{F}P]\ldots]]\!]_c W']$$

such that $W \approx_p W'$. Applying the same trick a second time, we see that

$$C[(\lambda x.E_1[\ldots E_{n-2}[x]\ldots])W] \stackrel{C}{\longmapsto}^{+} C[D[\mathcal{F}M]].$$

Even though this rewriting sequence may have the same number of steps as the original invocation, the relationship

$$\lambda x.E_1[\ldots E_{n-2}[x]\ldots] \approx_p [\![E'_1[\ldots E'_{n-2}[\mathcal{F}P]\ldots]]\!]_c$$

is clearly of lesser degree than $K \approx_p K'$. Given this, we can apply the inductive hypothesis and get the desired conclusion.

If the continuation invocation $(\lambda x.E_{n-1}[x])V$ directly leads to an $\mathcal{F}$-application:

$$C[E_1[\ldots((\lambda x.E_{n-1}[x])V)\ldots]] \stackrel{C}{\longmapsto}{}^{+} C[E_1[\ldots E_{n-2}[D_1[\mathcal{F}M]]\ldots]],$$

then we can apply the inductive hypothesis: even though we may not have reduced the number of rewriting steps as compared to the original rewriting sequence, the relationship $\lambda x.E_{n-1}[x] \approx_p K'_{n-1}$ is of lesser degree than $K \approx_p K'$. Hence,

$$C'[[\![E'_1[\ldots E'_{n-2}[\mathcal{F}P]\ldots]]\!]_c(K'_{n-1}V')] \stackrel{\triangleright}{\longmapsto}{}^{+}_{scs} C'[[\![E'_1[\ldots E'_{n-2}[\mathcal{F}P]\ldots]]\!]_c D'_1[\mathcal{F}M]]$$

such that $\lambda x.D_1[x] \approx_p [\![D'_1[\mathcal{F}P]]\!]_c$, and, therefore,

$$D[\ ] \equiv E_1[\ldots E_{n-2}[D_1[\ ]]\ldots]$$

and

$$D'[\ ] \equiv [\![E'_1[\ldots E'_{n-2}[\mathcal{F}P]\ldots]]\!]_c D'_1[\ ]$$

are appropriately related.

The remaining subcases where $E[\ ]$ is non-empty are essentially treated like the subcases a) and b) of the first half of the proof. Since there are no other cases, this completes the proof. □

We are now ready to prove the main lemma of this subsection:

Lemma 5.32. *$M \approx_p M'$ and M rewrites to V:*

$$M \xmapsto{C}{}^{*} V$$

if and only if the standard computation function maps M' to a value V':

$$M' \xmapsto{\triangleright}{}^{*}_{scs} V'$$

and $V \approx_p V'$.

Proof. With the help of the preceding lemmas, we can now prove that rewriting sequences and related standard computation sequences proceed in a synchronized manner. Given Lemma 5.29 and Corollary 5.30, we must only consider the case

$$M \xmapsto{C}{}^{*} C[KV] \text{ and } M' \xmapsto{\triangleright}{}^{*}_{scs} C'[K'V'],$$

where $C[\ ] \approx_p C'[\ ]$, $K \approx_p K'$, and $V \approx_p V'$. If KV yields a value U, then, by Lemma 5.31 (i), $K'V'$ produces a related value U' and

$$C[U] \approx_p C'[U'].$$

Otherwise, if KV rewrites into an $\mathcal{F}$-application, then, by Lemma 5.31 (iii), $K'V'$ computes to an $\mathcal{F}$-application such that the two terms build related continuations:

$$C[KV] \xmapsto{C}{}^{+} P(\lambda x.D[x]) \text{ and } C'[K'V'] \xmapsto{\triangleright}{}^{+}_{scs} P'[\![D'[\mathcal{F}M]]\!]_c$$

such that

$$P(\lambda x.D[x]) \approx_p P'[\![D'[\mathcal{F}M]]\!]_c.$$

Finally, if KV starts an infinite loop within $C[\ ]$, then $K'V'$ will also diverge. Although the standard computation function may build a different evaluation context,[2] the reduction starting in $K'V'$ will sooner or later produce related redexes. This is an indirect consequence of Lemma 5.31.

[2] The respective evaluation redexes are always related when interpreted as continuations, but may not be related as contexts.

It follows from this analysis by an inductive argument that the termination of the rewriting process implies the termination of the standard computation procedure, that two related sequences end in related values, and that, furthermore, non-termination of a rewriting process implies non-termination for all related standard computations. □

The Simulation Theorem follows directly from this lemma:

Theorem 5.20 (Simulation). *For all programs* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$eval_C(M) = V \text{ iff } \Phi(M) \overset{\triangleright}{\longmapsto}{}^{*}_{scs} U \text{ for some } U \text{ such that } V \approx_p U.$$

Proof. Since $\Phi(M)$ is related to M, *i.e.*, $M \approx_p \Phi(M)$, the antecedent of the preceding lemma is satisfied and the conclusion is immediate. □

6. Reasoning with the λ_v-CS-Calculus

In the two preceding chapters we have developed an equational theory for imperative higher-order languages. Now we must demonstrate that reasoning with the λ_v-CS-calculus is a viable endeavor. The key to this experiment is an interpretation of the theorems and propositions in the preceding chapter. Four factors play a major role.

First, our goal is to demonstrate the use of the calculus in conjunction with correctness proofs. Since correctness proofs concern program pieces and their operational equivalences to other program pieces, it is natural to work with the theory of safe **cs**-equality. This means that our proofs will generally be a mixture of reduction-based proofs and safe derivations in the computational **cs**-equality. In many circumstances, we will use the terminology "M is operationally indistinguishable from N" after proving that M is safely equivalent to N. This should not be confused with the statement "M evaluates to N": because of the division of the calculus into two levels, the latter no longer implies the former unlike in the functional world.

Second, the λ_v-CS-calculus subsumes the λ_v-calculus. Hence, we can carry over all conventions, theorems, and proof techniques from the functional calculus. For example, we can prove the equivalence of two total functions by recursion induction; we can treat an expression M as if it were a value when it is operationally indistinguishable ($=_s$) from a value; or, we can call a function F *pure* when FV is

operationally indistinguishable from a value U for every argument V—even if the function is not a Λ-expression. However, the subsumption does *not* mean that every statement in $\simeq_{CEK}$ is true for $\simeq_C$. Whereas the first relation compares results of terminating computations, the second relation compares results and *effects*. It is therefore important that *statements about functions are verified in the calculus.*

Third, since the calculus is an equational extension of the C-rewriting system (Theorem 5.20), we can freely mix reductions and C-transition rules in equivalence proofs. Rewriting rules are of particular importance when we employ Theorem 5.27 for the construction of correctness proofs. We shall use this in many of the following examples.

The fourth and final point is not a conclusion from the theorems but is a statement about them. As pointed out at various places, the programming language and the calculus have two independent fragments: the control and the assignment part. Appropriate reformulations of the theorems hold for both subsystems. Indeed, the two were developed separately [15, 18, 20] and only merged into a single system afterwards [17]. This natural fragmentation is welcome because it allows us to study the two classes of programs in isolation.

The first two sections of this chapter are dedicated to the control and the assignment fragment. Each section has two parts. The first is a discussion of the specific properties and principles of the respective system, the second a collection of examples. The third section contains two examples that make use of the full language and calculus.

6.1. Reasoning with Control

The control fragment of the λ_v-CS-calculus is rather simple. The language is that of the classical λ-calculus, enriched with $\mathcal{F}$-applications. This means in particular that the programming language and the reasoning language are the same. The

axioms of the system are the δ-, the β_v-relation, and the three $\mathcal{F}$-rules. For the reader's convenience, we have collected these notions in Definition 6.1.

Definition 6.1: The control fragment of λ_v-CS

Term language $\Lambda_{\mathcal{F}}$:

$$M ::= a \mid x \mid \lambda x.M \mid MN \mid \mathcal{F}M$$

Reductions and Computations:

$$\begin{aligned}
fa &\longrightarrow \delta(f,a) && (\delta)\\
(\lambda x.M)V &\longrightarrow M[x := V] \text{ provided } V \text{ is a value} && (\beta_v)\\
(\mathcal{F}M)N &\longrightarrow \mathcal{F}(\lambda k.M(\lambda m.k(mN))) && (\mathcal{F}_L)\\
V(\mathcal{F}M) &\longrightarrow \mathcal{F}(\lambda k.N(\lambda n.k(Vn))) \text{ provided } V \text{ is a value} && (\mathcal{F}_R)\\
(\mathcal{F}M) \triangleright M(\lambda x.x) && && (\mathcal{F}_T)
\end{aligned}$$

Meta-rule:

$$C[\mathcal{F}M] \longrightarrow M(\lambda x.C[x])$$

for every evaluation context $C[\ ]$.

The absence of labeled values greatly facilitates reasoning in this fragment. Without labels, safety considerations about derivations become superfluous. We capture this in a corollary to Theorem 5.27:

Corollary 6.2. *Let M and N be in $\Lambda_{\mathcal{F}}$. Then $M \simeq_C N$ if $\lambda_v\text{-CS}^{\triangleright} \vdash C[M] = C[N]$ for all evaluation contexts $C[\ ]$ over $\Lambda_{\mathcal{F}}$.*

Equipped with these general observations, we continue the $\mathcal{F}$-related programming examples from Section 3.3. **throw**-expressions play a central role in many of these examples. Intuitively, (**throw** $L\,V$) eliminates the current continuation and continues with the evaluation of $L\,V$. From the preceding chapters we know that a

continuation of a program piece is its evaluation context. Hence, we can formalize the behavior of **throw**-expressions by analyzing their actions in these contexts:

Proposition 6.3. *Suppose F is a value and M an arbitrary expression.*

(i) $\lambda_v\text{-}CS^{safe} \vdash F(\mathbf{throw}\ L\ V) = (\mathbf{throw}\ L\ V)$;

(ii) $\lambda_v\text{-}CS^{safe} \vdash (\mathbf{throw}\ L\ V)M = (\mathbf{throw}\ L\ V)$.

Proof. The proofs are trivial, *e.g.*,

$$\begin{aligned} \lambda_v\text{-CS} \vdash F(\mathbf{throw}\ L\ V) &= F(\mathcal{F}(\lambda d.LV)) \qquad \text{by definition} \\ &= (\mathcal{F}\lambda k.(\lambda d.LV)(\lambda v.k(Fv)) \\ &= (\mathcal{F}\lambda k.LV) = (\mathbf{throw}\ L\ V). \qquad \square \end{aligned}$$

Next we apply this first proposition to the correctness proof of Σ_0^* which is our prototypical example of a function with exceptional flow of control. According to its specification, the function is to sum up the numbers in a binary tree unless the tree contains 0, in which case the function must produce 0. Assuming the existence of a predicate occur0? that tests the presence of a 0, the behavior of Σ_0^* on a tree T can be specified by

$$(\Sigma_0^* T) \simeq_C (\mathbf{if}\ (\mathsf{occur0?}\ T)\ 0\ (\Sigma^* T)).$$

The implementation of Σ_0^* relies on the **exit**-facility, which in turn is based on **throw**. It is therefore natural that a correctness proof for Σ_0^* uses Proposition 6.3:

Proposition 6.4. *For all binary number-trees T, Σ_0^* satisfies:*

$$\lambda_v\text{-}CS^{safe} \vdash (\Sigma_0^* T) = (\mathbf{if}\ (\mathsf{occur0?}\ T)\ 0\ (\Sigma^* T)).$$

Proof. Let us recall Σ_0^* in a slightly de-sugared form:

$$\Sigma_0^* \equiv \lambda t.\mathcal{F}(\lambda\epsilon.\epsilon((\mathbf{rec}\ (s\ t)\ =$$
$$(\mathbf{if}\ (\mathsf{empty?}\ t)\ 0$$
$$(\mathbf{if}\ (\mathsf{zero?}(\mathsf{info}\ t))\ (\mathbf{throw}\ \epsilon\ 0)$$
$$(+(\mathsf{info}\ t)(+(s(\mathsf{lson}\ t))(s(\mathsf{rson}\ t)))))))$$
$$t))\ .$$

Since this function obviously manipulates its entire calling context, we use Corollary 6.2 for the proof.

Let $C[\ ]$ be an arbitrary evaluation context. Then an application of Σ_0^* to a tree T in $C[\ ]$ proceeds as follows:

$$\lambda_v\text{-CS}^{\ \triangleright} \vdash C[\Sigma_0^* T] = C[\mathcal{F}\lambda\epsilon.\epsilon(\mathsf{S}_\epsilon T)] = (\lambda x.C[x])(S_\epsilon[\epsilon := (\lambda x.C[x])]\ T),$$

where

$$\mathsf{S}_\epsilon \equiv (\mathbf{rec}\ (s\ t)\ =$$
$$(\mathbf{if}\ (\mathsf{empty?}\ t)\ 0$$
$$(\mathbf{if}\ (\mathsf{zero?}(\mathsf{info}\ t))\ (\mathbf{throw}\ \epsilon\ 0)$$
$$(+(\mathsf{info}\ t)(+(s(\mathsf{lson}\ t))(s(\mathsf{rson}\ t)))))).$$

Now we must show that S_ϵ behaves correctly. Given that $(\mathsf{occur0?}\ T)$ is either True or False, we can split the claim into two sub-claims:

(i) $\lambda_v\text{-CS}^{safe} \vdash(\mathsf{occur0?}\ T) = \mathsf{False}$ implies $\lambda_v\text{-CS}^{safe} \vdash(\mathsf{S}_\epsilon T) = (\Sigma^* T)$

(ii) $\lambda_v\text{-CS}^{safe} \vdash(\mathsf{occur0?}\ T) = \mathsf{True}$ implies $\lambda_v\text{-CS}^{safe} \vdash(\mathsf{S}_\epsilon T) = (\mathbf{throw}\ \epsilon\ 0)$

Assuming that both hold, the rest of the proposition follows easily:

$$\lambda_v\text{-CS}^{\ \triangleright} \vdash(\lambda x.C[x])(\Sigma^* T) = C[\Sigma^* T] = C[\mathbf{if}\ \mathsf{False}\ 0\ (\Sigma^* T)],$$

$$\lambda_v\text{-CS}^{\ \triangleright} \vdash(\lambda x.C[x])(\mathbf{throw}\ (\lambda x.C[x])\ 0) = C[0] = C[\mathbf{if}\ \mathsf{True}\ 0\ (\Sigma^* T)].$$

The first derivation uses the convention that expressions with an operationally indistinguishable value are treated as if they were a value.

For the proof of part (i) observe that $(\mathsf{occur0?}\, T) = \mathsf{False}$ implies

$$(\mathsf{zero?}(\mathsf{info}\, t)) = \mathsf{False}$$

for all subtrees t of T. Hence,

$$\lambda_v\text{-CS}^{safe} \vdash (\textbf{if}\ (\mathsf{zero?}(\mathsf{info}\, t))\ (\textbf{throw}\ \epsilon\, 0)\ (+(\mathsf{info}\, t)(+(s(\mathsf{lson}\, t))(s(\mathsf{rson}\, t)))))$$
$$= (+(\mathsf{info}\, t)(s(\mathsf{lson}\, t))(s(\mathsf{rson}\, t))))$$

for all recursive stages in the evaluation of S_ϵ. By induction, the consequence of (i) holds for the entire tree T.

The proof of part (ii) is also an induction on the structure of T but with the hypothesis:

$$(\mathsf{S}_\epsilon t) = (\textbf{throw}\ \epsilon\, 0)$$

for all subtrees t containing 0. First suppose 0 occurs at the root of t. Then the claim is immediate. Otherwise, 0 must be in one of the two subtrees. If the left subtree contains 0, the inductive hypothesis yields:

$$\lambda_v\text{-CS}^{safe} \vdash (\mathsf{S}_\epsilon t) = (+(\mathsf{info}\, t)(+(\textbf{throw}\ \epsilon\, 0)(\mathsf{S}_\epsilon(\mathsf{rson}\, t))).$$

But by Proposition 6.3 **throw** eliminates this kind of context—$(\mathsf{info}\, t)$ represents a value—and therefore (ii) follows. If the left tree does not contain 0, then $(\mathsf{S}_\epsilon(\mathsf{lson}\, t))$ is equivalent to $(\Sigma_0^*(\mathsf{lson}\, t))$ by (i). From this, we finally get

$$\lambda_v\text{-CS}^{safe} \vdash (\mathsf{S}_\epsilon t) = (+(\mathsf{info}\, t)(+(\Sigma^*(\mathsf{lson}\, t))(\textbf{throw}\ \epsilon\, 0))) = (\textbf{throw}\ \epsilon\, 0). \qquad \square$$

Beyond its immediate result, the preceding proof offers a strategy for similar **exit**-programs. We call this strategy **exit**-induction. It applies to all programs that are regularly recursive, except for some finite number n of exception conditions. The strategy requires $n+1$ routine inductions on the primary information-structure: one

for the regular case, one for each exception. This is quite natural and corresponds to the folk wisdom that calls for a similar testing strategy.

For a further illustration of **exit**-induction, consider the function Π, which returns the product of a list of numbers:

$$\Pi \stackrel{df}{\equiv} \lambda l.(\textbf{rec}\ (p\ l)\ =\ (\textbf{if}\ (\mathsf{null?}\ l)\ 1\ (*(\mathsf{car}\ l)(p(\mathsf{cdr}\ l))))))\ l.$$

Although the function is correct, it is inefficient. Given that 0 collapses the product, Π must satisfy

$$\lambda_v\text{-CS}^{safe} \vdash (\Pi L) = (\textbf{if}\ (\mathsf{occur0?}\ L)\ 0\ (\Pi L)).$$

But this looks almost like the specification of Σ_0^*. A straightforward inversion of the principle of **exit**-induction leads to the specification of the function Π_0 by

$$\lambda_v\text{-CS}^{safe} \vdash (\Pi_0 L) = (\textbf{if}\ (\mathsf{occur0?}\ L)\ 0\ (\Pi L)).$$

and an implementation as

$$\begin{aligned}\Pi_0 \stackrel{df}{\equiv} (\textbf{function}\ L\ ((\textbf{rec}\ (p\ l)\ = \\ (\textbf{if}\ (\mathsf{null?}\ l)\ 1 \\ (\textbf{if}\ (\mathsf{zero?}(\mathsf{car}\ l))\ (\textbf{exit}\ 0) \\ (*(\mathsf{car}\ l)(p(\mathsf{cdr}\ l))))))\ L)).\end{aligned}$$

In exchange for at most n extra tests, the function avoids all multiplications if the list contains 0. The correctness of Π_0 is captured in

Proposition 6.5. *For all lists of numbers L,*

$$\lambda_v\text{-}CS^{safe} \vdash (\Pi_0 L) = (\textbf{if}\ (\mathsf{occur0?}L)\ 0\ (\Pi L)) = (\Pi L).$$

From the perspective of program development, this second example is more relevant. Instead of writing down a program and independently verifying its correctness,

we have used the **exit**-induction principle for the construction of the program. As usual, the proof is then a mere exercise.

Besides the use of $\mathcal{F}$-applications for loop- and function-exits, there are few other examples that use control information and solely rely on functions. In particular, continuations that are passed out of their original context—first-class continuations—are mostly useful in conjunction with state variables. We resume this topic in the last section after investigating the use of side-effects in the next.

6.2. Reasoning with State

As pointed out at the end of Chapter 4, the assignment fragment as depicted in Definition 6.6 is more complicated than the control fragment. The major difference between the two systems is that the addition of assignment to a functional language is insufficient for reasoning about programs in the extended language: another necessary addition is a linguistic facility for expressing sharing relations, *e.g.*, labels. However, the presence of labels makes proofs of operational equalities more difficult. To be useful, the resulting theorem must respect the full *safe*-ness condition as formulated in Definition 5.26. Although this formal definition is easy to understand and well-suited for the proof of Theorem 5.27, it is impractical for real work in the calculus. It is therefore our first priority to develop more insight into the nature of safe theorems.

A safe theorem is the result of some derivation in the top-level of the λ_v-CS-calculus and a *safe*-ness check for the resulting theorem. The check consists of verifying a set of derived equations as theorems. The easiest way to perform this second step is to see whether modifications of the original derivation prove the modified equations. From the converse perspective, we could say that if a derivation automatically verifies derived equations, then its resulting theorem is safe. Accordingly, we call such derivations *safe*.

Definition 6.6: The assignment fragment of λ_v-CS

Term language $\Lambda_{\sigma\mathcal{D}}$:

$$M ::= a \mid x_\lambda \mid \lambda x.M \mid MN \mid \sigma x_\sigma.M \mid \mathcal{D}\, M\, x_\sigma \mid \sigma V^l.M \mid \mathcal{D}\, M\, V^l.$$

Reductions and Computations:

$$\begin{aligned}
fa &\longrightarrow V \text{ if } \delta(f,a) = V, V \in \Lambda && (\delta)\\
(\lambda x.M)V &\longrightarrow M[x := V] && (\beta_v)\\
U((\lambda x_\sigma.M)V) &\longrightarrow (\lambda x_\sigma.(UM))V && (\beta_R)\\
((\lambda x_\sigma.M)V)N &\longrightarrow (\lambda x_\sigma.(MN))V && (\beta_L)\\
(\lambda x_\sigma.M)V &\triangleright M[x_\sigma := V^l] \text{ where } l \text{ is fresh} && (\beta_\sigma)\\
U((\sigma X.M)V) &\longrightarrow (\sigma X.(UM))V && (\sigma_R)\\
((\sigma X.M)V)N &\longrightarrow (\sigma X.(MN))V && (\sigma_L)\\
(\sigma U^l.M)V &\triangleright M[\bullet^l := V^l] && (\sigma_T)\\
U(\mathcal{D}\, MX) &\longrightarrow (\mathcal{D}\, (\lambda v.U(Mv))X) && (\mathcal{D}_R)\\
(\mathcal{D}\, MX)N &\longrightarrow (\mathcal{D}\, (\lambda v.MvN)X) && (\mathcal{D}_L)\\
(\mathcal{D}\, MV^l) &\triangleright MV[\bullet^l := V^l] && (\mathcal{D}_T)
\end{aligned}$$

Meta-rules:

$$\begin{aligned}
C[(\lambda x_\sigma.M)V] &\longrightarrow C[M[x_\sigma := V^l]]\\
C[(\sigma U^l.M)V] &\longrightarrow C[M][\bullet^l := V^l]\\
C[\mathcal{D}\ M\ V^l] &\longrightarrow C[MV[\bullet^l := V^l]]
\end{aligned}$$

for every evaluation context $C[\ ]$.

A derivation is obviously safe if it transforms an Idealized Scheme expression into another one: since such expressions do not contain any labels, they can neither affect nor be affected by their context. This means that derivations can safely establish sharing relationships if they are guaranteed to disappear in the theorem-terms and that they can freely assign to and delabel such *derivation-local* sharing relations. Although this sheds some light on safe derivations and is highly useful as

we will see later, it is not always possible to rely on theorems in Idealized Scheme. Many of the intermediate steps in such proofs involve statements about terms that contain labels, and it is therefore necessary to look more closely at what can happen to a non-local label and its associated value during a derivation.

Due to the Consistency Theorem and its corollary, we can fortunately restrict our attention to derivations of the form

$$M \triangleright^* L \ {}^*\!\triangleleft N.$$

According to this corollary, an equation $M = N$ must have such a derivation. Furthermore, because of the transitivity of safe **cs**-equality, we can always restrict our attention to such fragments in a given derivation: if each piece is safe, the composition is safe as well.

Next, we must analyze the effect of creating, assigning to, or delabeling labeled values in such derivations. Thus, suppose a sharing relation is created during a derivation and persists in only one of the terms in the resulting theorem. As mentioned in the discussions preceding the design of the β_σ-relation (Section 5.1) and the definition of *safe*-ness (Definition 5.26), such a derivation cannot be safe because it may establish a sharing relation too early. Put differently, a sharing relation of a safe theorem may not originate within the derivation: an arbitrary replacement of the labeled value will always interfere with the β_σ-step. Thus, if a sharing relation is created during one half of a derivation, there must be a symmetric β_σ-expansion step in the other half.

Assignments (on non-local labels) corrupt the *safe*-ness of a derivation in a cruder way. If assignments are discharged arbitrarily, they can only affect the currently visible labels:

$$(\sigma \mathsf{False}^l.(\mathcal{D} \mid \mathsf{False}^l))\mathsf{True} = (\mathcal{D} \mid \mathsf{True}^l).$$

The resulting theorems depend on their specific context and are incorrect in others, *e.g.*, replacing

$$(\sigma \mathsf{False}^l.(\mathcal{D} \mid \mathsf{False}^l))\mathsf{True}$$

with

$$(\mathcal{D} \mid \mathsf{True}^l)$$

in an expression like

$$(\lambda x.(\mathcal{D} \mid \mathsf{False}^l))\underline{((\sigma \mathsf{False}^l.(\mathcal{D} \mid \mathsf{False}^l))\mathsf{True})}$$

would lead to the contradiction

$$\mathsf{True} = \mathsf{False}.$$

It follows that assignments—just like β_σ-steps—must come in do-undo pairs. That is, an assignment in one derivation half is undone by re-assigning the old value, or if there are assignments with a lasting effect in one half, there must be at least one un-assignment in the other half to undo the effect. For example,

$$(\sigma V^l.(\sigma V^l.M))UW = M[\bullet^l := U^l][\bullet^l := W^l] = (\sigma V^l.M)W$$

is a perfectly safe derivation: the final assignment to l will be performed by either term independently of the context.

Finally, delabeling steps need a more subtle analysis. Again, a single delabeling step is unsafe, but consider the following situation:

$$\mathsf{K0}(\mathcal{D} \mid V^l) = \mathsf{K0}(V[\bullet^l := V^l]) = 0.$$

No matter what the value V is, this equivalence holds in any context because the value simply disappears. On the other hand, if the value is actively used in some subsequent derivation step as in

$$(\mathbf{if}\ (\mathcal{D} \mid \mathsf{True}^l)\ \mathsf{True}\ \mathsf{False}) = (\mathbf{if}\ \mathsf{True}\ \mathsf{True}\ \mathsf{False}) = \mathsf{True},$$

the derivation becomes unsafe: in a context where True^l is first changed to False^l this equality cannot hold. A delabeling step like this can only be safe if it is preceded by an assignment in the same derivation:

$$(\sigma x^l.(\mathbf{if}\ (\mathcal{D}\ \mathsf{I}\ \mathsf{True}^l)\ \mathsf{True}\ \mathsf{False}))\mathsf{True} = (\mathbf{if}\ \mathsf{True}\ \mathsf{True}\ \mathsf{False}) = (\sigma x^l.\mathsf{True})\mathsf{True} :$$

the value part of the labeled value is then known, no matter what the use-context will assign to it.

At this point, a clarification of the terminology "use of a value" is necessary. In the course of an derivation, two things can happen to a value: it may or it may not be a direct part of a redex (in the sense of a C-redex). That is, a value may either occur in the function or in the argument position of an application; or, it is only a proper sub-part of a redex. In the first case, we say that *the value is used* because the value *may* impact the derivation. Furthermore, we distinguish the notion of *actively used* when a value actually does have an impact on the derivation. For a constant this means that it must be used in a δ-transition; for abstractions *actively used* means that they are in function position during a β_v- or β_σ-step. Other uses of values are called *passive*.

Based on the preceding arguments, we can now precisely characterize safe derivations:

Proposition 6.7. *A derivation*

$$M \triangleright^* L\ {}^*\!\triangleleft N.$$

is safe iff it satisfies the following conditions on labeled values in M and N:

1. *none of the labels in M and N originates from a β_σ-step in the derivation;*
2. *if there are assignments to a label l in one half of the derivation such that the value of a labeled value part is changed, there must be at least one assignment to this label in the other half;*

3. *if a labeled value is delabeled and actively used thereafter, the delabeling step must be preceded by an assignment;*

4. *if a labeled value V^l is delabeled, occurs in L, but is only used passively, then there must be a delabeling step in the other half that yields this $V[\bullet^l := V^l]$ in L.*

Proof. The direction from left to right is trivial. If suffices to show that the negation of any point leads to unsafe theorems. This can be done with the counterexamples from above.

For the opposite direction, we assume that the four conditions hold and show that modified versions of the derivation also prove all theorems of the form:

$$C[M][\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}] = C[N][\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}]$$

for an arbitrary evaluation context $C[\ ]$ and arbitrary values $V_1 \ldots V_n$. First, since $C[\ ]$ is an evaluation context, reductions carry over directly, computation steps must be pre- and post-fixed with an appropriate series of reductions. Second, by assumption 2 every assignment step with an impact on $C[L]$ is undone by the other half of the derivation. It follows from these two arguments that the derivation can be embedded in $C[\ ]$ without problem.

Third, we must trace all values V_i^l through the derivation and show that the replacement cannot interfere with the derivation steps. Three cases are possible:

a) V^l is part of a β_σ-expansion so that the label disappears. However, this is ruled out by assumption 1 which says that none of the labels in M and N originates from such steps.

b) V^l is delabeled and actively used thereafter. This contradicts assumption 3, which requires that an assignment precedes such a delabeling step. But given the assignment, the replacement of the original value with V is irrelevant.

c) V^l is delabeled, but not actively used. If L does not contain $V[\bullet^l := V^l]$, the replacement is again irrelevant; otherwise, by assumption 4, there is a (n inverse) $\mathcal{D}_T$-step in the other half of the derivation that reconstructs the labeled value V^l and thus satisfies the replacement of the l-labeled value in the second term.

Since there are no other cases, this concludes the proof. □

Subsequently, we will rely on this proposition whenever we use computational equality for the derivation of safe theorems. As a direct consequence, the proposition yields a first meta-rule on stating safe theorems. The proposition requires that all transitions with globally visible effects come in do-undo pairs. Thus, a derivation of the form

$$C[M] \stackrel{\triangleright}{=}_{cs} D[V] \stackrel{\triangleright}{=}_{cs} C[M]$$

is automatically safe: the left part performs the effects, the right one undoes them. Of course, this is a useless derivation. What we would really like to see is an equivalence between non-identical terms. Yet, the idea behind this case is important. Suppose we can prove

$$\lambda_v\text{-CS}^{\triangleright} \vdash C[x] = D[x]$$

for some evaluation contexts $C[\ ]$ and $D[\ ]$. Is it then possible to replace x by an arbitrary term M since M is guaranteed to affect both sides of the equation during its evaluation? The answer is positive provided M evaluates to some value regardless of pending assignments:

Proposition 6.8. *Let M be an expression, V a value; let $Lab(M) = \{l_1, \ldots, l_m\}$ and let $\{k_1, \ldots, k_n\} \subseteq Lab(M)$. If for some evaluation contexts $C_1[\ ]$ and $C_2[\ ]$*

$$\lambda_v\text{-}CS^{safe} \vdash C_1[x] = C_2[x] \tag{1}$$

and if for all values $V_1, \ldots, V_m$ and for some values $U_1, \ldots, U_n$

$$\lambda_v\text{-}CS^{\triangleright} \vdash M[\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_m} := {V_m}^{l_m}] =$$

$$V[\bullet^{l_1} := V_1{}^{l_1}] \ldots [\bullet^{l_m} := V_m{}^{l_m}][\bullet^{k_1} := U_1{}^{k_1}] \ldots [\bullet^{k_n} := U_n{}^{k_n}] \quad (2)$$

for $i = 1, 2$ *then*

$$\lambda_v\text{-}CS^{safe} \vdash C_1[M] = C_2[M].$$

Proof. The proof is simply a matter of checking the *safe*-ness of the consequence according to Definition 5.26:

$$\begin{aligned}
&\lambda_v\text{-CS}^{\triangleright} \vdash D[C_1[M]][\bullet^{l_1} := V_1{}^{l_1}] \ldots [\bullet^{l_m} := V_m{}^{l_m}] \\
&\quad = D[C_1[V]][\bullet^{l_1} := V_1{}^{l_1}] \ldots [\bullet^{l_m} := V_m{}^{l_m}][\bullet^{k_1} := U_1{}^{k_1}] \ldots [\bullet^{k_n} := U_n{}^{k_n}] \text{ by (2)} \\
&\quad = D[C_2[V]][\bullet^{l_1} := V_1{}^{l_1}] \ldots [\bullet^{l_m} := V_m{}^{l_m}][\bullet^{k_1} := U_1{}^{k_1}] \ldots [\bullet^{k_n} := U_n{}^{k_n}] \\
&\qquad\qquad \text{by substitution and (1)} \\
&\quad = D[C_2[M]][\bullet^{l_1} := V_1{}^{l_1}] \ldots [\bullet^{l_m} := V_m{}^{l_m}] \text{ by (2).} \qquad \square
\end{aligned}$$

Together with the Substitution Theorem, this proposition provides the foundation for stating operational equivalences with implicitly universally quantified (non-assignable) variables. Since these equations hold for all values, we are free to replace all variables by *labeled values*—they automatically satisfy the antecedent—or by *arbitrary expressions with values* if the variable occurs on both sides in an evaluation context. A useful example is:

$$\lambda_v\text{-CS}^{safe} \vdash \pi_i^n x_1 \ldots x_n = x_i \qquad \text{for } 1 \leq i \leq n.$$

This equation means that for all (labeled) values $V_1, \ldots, V_n$ the selection function π_i^n picks the i-th value, and, by the second argument, that this i-th expression can be any expression if it has a value, *i.e.*,

$$\lambda_v\text{-CS}^{safe} \vdash \pi_i^n x_1 \ldots M_i \ldots x_n = M_i \qquad \text{for } 1 \leq i \leq n.$$

The importance of this particular statement is that **begin**-expressions are abbreviations for π-applications, and that side-effects in expressions can easily be

characterized as effect sequences in **begin**-blocks. It follows, for example, that if an expression M_1 in a **begin**-block is operationally equivalent to a value, then

$$\lambda_v\text{-CS}^{safe} \vdash (\textbf{begin}\ M_1 M_2 \ldots M_n) = (\textbf{begin}\ M_2 \ldots M_n).$$

Convention. *In the rest of this chapter we use assignable variables and labeled values as abbreviations for delabeling applications of the form:* $(\mathcal{D} \mid \cdot)$. *The motivation behind this is that equivalence proofs never use* $\mathcal{D}$*-reductions, but delabeling C-transitions instead.* **End**

After this theoretical, in-depth consideration of *safe*-ness, let us now apply the fresh insight to the first assignment-programming example from Section 3.3: the implementation of cells with higher-order functions and assignment abstractions. Recall that the three major operations on cells are:

$$\begin{aligned}
\mathsf{mk\text{-}cell} &\stackrel{df}{\equiv} \lambda x_\sigma.\lambda m.m x_\sigma(\sigma x_\sigma.x_\sigma),\\
\mathsf{deref} &\stackrel{df}{\equiv} \lambda c.c(\lambda x s.x) \equiv \lambda c.c\pi_2^1,\\
\mathsf{set\text{-}cell!} &\stackrel{df}{\equiv} \lambda c.c(\lambda x s.s) \equiv \lambda c.c\pi_2^2.
\end{aligned}$$

The result of a call to mk-cell is a functional value with a new sharing relation:

$$\lambda_v\text{-CS}^{\triangleright} \vdash \mathsf{mk\text{-}cell}\, x = \lambda m.m x^l(\sigma x^l.x^l).$$

The value is not operationally indistinguishable from the application because of the newly introduced label l. Put differently, the calculus respects that every call to mk-cell creates a new, distinct cell-object. To avoid some notational overhead, we use the abbreviation

$$(\mathsf{mk\text{-}cell}\, x)^l.$$

It denotes a cell-object, indicating the sharing relationship l as a superscript to the entire expression and the current contents as x. When necessary, we expand the abbreviation to the above abstraction.

The effect of a set-cell!-operation can be characterized by an operational equivalence:

$$\lambda_v\text{-CS}^{safe} \vdash (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y) = (\pi_2^2 x^l(\sigma x^l.x^l))y = (\sigma x^l.x^l)y.$$

In the same manner, we can specify the result of a deref-operation on a cell:

$$\lambda_v\text{-CS}^{safe} \vdash (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l) = (\pi_1^2 x^l(\sigma x^l.x^l)) = x^l.$$

The disadvantage of these equations is that they are of a rather low-level nature. By connecting specific operations on a cell with facilities in the underlying language, they uncover too much about how cells are structured. More abstract equations in the style of algebraic specifications are preferable. In the functional fragment—see Section 2.5—such equations specify operations by showing their mutual interaction without relying on the coding. For example, the effect of car is related to cons by

$$(\mathsf{car}(\mathsf{cons}\ x\ y)) = x.$$

A transliteration of this algebraic style and the particular example to the cell-world means finding an operational equivalent of

$$(\mathsf{deref}(\mathsf{mk\text{-}cell}\ x)).$$

With the above characterization of deref and mk-cell, this is rather simple:

$$\lambda_v\text{-CS}^{\triangleright} \vdash (\mathsf{deref}(\mathsf{mk\text{-}cell}\ x)) = (\mathsf{deref}(\mathsf{mk\text{-}cell}\ x)^l) = x^l = x.$$

The last step is valid because l is introduced by the derivation and cannot be part of x. However, the essence of a cell is that it can change its contents with set-cell!, and that deref can fetch this new value. In other words, we should be able to show

$$\lambda_v\text{-CS}^{safe} \vdash (\mathbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l))$$
$$= (\mathbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ y).$$

Because set-cell! does not return the cell, the **begin** is necessary in order to express the appropriate sequencing of actions.[1] The proof is a simple calculation:

$$\lambda_v\text{-CS}^{safe} \vdash (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l) \tag{1}$$

$$= \pi_2^2((\sigma x^l.x^l)y)x^l \tag{2}$$

$$= (\sigma x^l.\pi_2^2 x^l x^l)y \tag{3}$$

$$= (\sigma x^l.x^l)y = (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y) \tag{4}$$

$$= (\sigma x^l.y)y \tag{5}$$

$$= (\sigma x^l.\pi_2^2 x^l y)y \tag{6}$$

$$= \pi_2^2((\sigma x^l.x^l)y)y \tag{7}$$

$$= (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ y). \tag{8}$$

All steps in this proof, except (4) to (5), are simple reductions or safe statements about the selection function π_i^n; the transition from (4) to (5) is discussed below in Proposition 6.10. The fourth step also yields a slightly simplified version of the operational equivalence.

Continuing with the transliteration of the algebraic strategy, we ask what the effect of an immediate assignment to a cell is. At first glance, the question is about the term

$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)\ y),$$

but given this, another possibility comes to mind:

$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y).$$

[1] An appropriate theorem for cons-cells could be written more elegantly as

$$(\mathsf{car}\ (\mathsf{set\text{-}car!}\ x\ y)) = y.$$

This second term expresses the possibility that the cell survives the effect because it already exists in several places. This is, for example, the case in the expression

$$(\textbf{let } (c\ (\mathsf{mk\text{-}cell}\ x))\ (\textbf{begin } (\mathsf{set\text{-}cell!}\ c\ y)\ c)),$$

which we treat as prototypical.

Of the two possible cases, the first is the less interesting one. A simple calculation proves that the two operations cancel each other

$$\begin{aligned}\lambda_v\text{-CS}^{\triangleright} \vdash (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)\ y) &= (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\\ &= (\sigma x^l.x^l)y\\ &= y.\end{aligned}$$

The second case requires a more sophisticated derivation:

$$\begin{aligned}\lambda_v\text{-CS}^{\triangleright} \vdash (\textbf{let } (c\ (\mathsf{mk\text{-}cell}\ x))\ (\textbf{begin } (\mathsf{set\text{-}cell!}\ c\ y)\ c))&\\ = (\textbf{begin } (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ (\mathsf{mk\text{-}cell}\ x)^l) &\quad (1)\\ = (\pi_2^2((\sigma x^l.x^l)y)(\mathsf{mk\text{-}cell}\ x)^l) &\quad (2)\\ = ((\sigma x^l.\pi_2^2 x^l(\mathsf{mk\text{-}cell}\ x)^l))y &\quad (3)\\ = ((\sigma x^l.(\mathsf{mk\text{-}cell}\ x)^l))y &\quad (4)\\ = (\mathsf{mk\text{-}cell}\ y)^l &\quad (5)\\ = (\mathsf{mk\text{-}cell}\ y). &\quad (6)\end{aligned}$$

The proof step from line (4) to (5) depends on the uniqueness of l with respect to the context; the β_σ-steps in (1) and (6) are the necessary inverses of each other so that l does not survive the derivation.

Finally, we can ask what the interaction of two set-cell!-operations is. Again, there are two possible cases: the operations may affect the same cell or different cells. A typical expression of the first kind is

$$(\textbf{begin } (\mathsf{set\text{-}cell!}\ c\ x)\ (\mathsf{set\text{-}cell!}\ c\ y)).$$

Intuitively, the second operation must cancel the first, *i.e.*, the first is invisible to an outside observer. This is verified by

$$\begin{aligned}
\lambda_v\text{-CS}^{safe} \vdash (&\textbf{begin } (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ x)\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ y)) \\
&= \pi_2^2((\sigma u^l.u^l)x)((\sigma u^l.u^l)y) \\
&= (\sigma u^l.\pi_2^2 u^l((\sigma u^l.u^l)y))x \\
&= (\sigma u^l.(\sigma u^l.u^l)y)x \\
&= (\sigma u^l.u^l)y \\
= (&\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ y).
\end{aligned}$$

The second case is treated similarly. We consider the expression

$$(\textbf{begin } (\mathsf{set\text{-}cell!}\ c\ x)\ (\mathsf{set\text{-}cell!}\ d\ y)),$$

where c and d are distinct cells, and prove its equivalence to

$$(\textbf{begin } (\mathsf{set\text{-}cell!}\ d\ y)\ (\mathsf{set\text{-}cell!}\ c\ x)\ y).$$

With the following equality about the connection between deref- and set-cell!-operations:

$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y) = (\textbf{begin } (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ y)$$

and the fact that nested **begin**-expressions can be linearized, the required calculation becomes

$$\begin{aligned}
\lambda_v\text{-CS}^{safe} \vdash (&\textbf{begin } (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ x)\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ v)^k\ y)) \\
&= (\textbf{begin } (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ x)\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ v)^k\ y)\ y) \\
&= \pi_3^3((\sigma u^l.u^l)x)((\sigma v^k.v^k)y)y \\
&= (\sigma u^l.\pi_3^3 u^l((\sigma v^k.v^k)y)y)x \\
&= (\sigma u^l.(\sigma v^k.\pi_3^3 u^l v^k y)y)x
\end{aligned}$$

$$= (\sigma v^k.(\sigma u^l.\pi_3^3 u^l v^k y)x)y$$

$$= (\sigma v^k.(\sigma u^l.\pi_2^2 u^l y)x)y$$

$$= (\sigma v^k.\pi_2^2((\sigma u^l.u^l)x)y)y$$

$$= (\sigma v^k.\pi_3^3 v^k((\sigma u^l.u^l)x)y)y$$

$$= \pi_3^3((\sigma v^k.v^k)y)((\sigma u^l.u^l)x)y$$

$$= (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ v)^k\ y)\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ u)^l\ x)\ y).$$

For the convenience of the reader, we have collected all of the above statements about the three major cell operations in

Proposition 6.9. *Let c range over cells. Then*

(i) $\lambda_v\text{-}CS^{safe} \vdash (\mathsf{deref}(\mathsf{mk\text{-}cell}\ x)) = x;$

(ii) $\lambda_v\text{-}CS^{safe} \vdash (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)\ y) = y;$

(iii) $\lambda_v\text{-}CS^{safe} \vdash (\textbf{let}\ (c\ (\mathsf{mk\text{-}cell}\ x))\ (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ c\ y)\ c)) = (\mathsf{mk\text{-}cell}\ y);$

(iv) $\lambda_v\text{-}CS^{safe} \vdash (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ c\ x)\ (\mathsf{deref}\ c)) = (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ c\ x)\ x);$

(v) $\lambda_v\text{-}CS^{safe} \vdash (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ c\ x)\ (\mathsf{set\text{-}cell!}\ c\ y)) = (\mathsf{set\text{-}cell!}\ c\ y);$

(vi) Let d be a cell that is distinct from c:

$$\lambda_v\text{-}CS^{safe} \vdash (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ c\ x)\ (\mathsf{set\text{-}cell!}\ d\ y))$$

$$= (\textbf{begin}\ (\mathsf{set\text{-}cell!}\ d\ y)\ (\mathsf{set\text{-}cell!}\ c\ x)\ y).$$

The proofs of the above statements share several characteristics. First, they all rely on a partial expansion of syntactic abstractions into Λ_{CS}-expressions. This is bearable for small examples, but for larger ones we must develop strategies that correspond to the right level of syntactic abstraction. Second, once the syntactic abstractions are eliminated, the proofs are mostly manipulations in the reduction system. This makes them automatically safe. Finally, the central parts of the proof are some universal statements about σ-capabilities similar to the one on the selection functions π_i^n. We have also collected these statements:

Proposition 6.10.

(i) $\lambda_v\text{-}CS^{safe} \vdash (\sigma u^l.((\sigma u^l.M)x))y = (\sigma u^l.M)x;$

(ii) $\lambda_v\text{-}CS^{safe} \vdash (\sigma u^l.((\sigma v^k.M)x))y = (\sigma v^k.((\sigma u^l.M)y))x;$

(iii) $\lambda_v\text{-}CS^{safe} \vdash (\sigma u^l.u^l)x = (\sigma u^l.x)x.$

Proof.

$$
\begin{aligned}
(i) \qquad \lambda_v\text{-CS}^{\triangleright} \vdash\ & (\sigma u^l.((\sigma u^l.M)x))y \\
&= (\sigma u^l.M)x[\bullet^l := y^l] \\
&= M[\bullet^l := y^l][\bullet^l := x^l] \\
&= M[\bullet^l := x[\bullet^l := y^l]^l] \equiv_\alpha M[\bullet^l := x^l] \qquad (*) \\
&= (\sigma u^l.M)x;
\end{aligned}
$$

$$
\begin{aligned}
(ii) \qquad \lambda_v\text{-CS}^{\triangleright} \vdash\ & (\sigma u^l.((\sigma v^k.M)x))y \\
&= (\sigma v^k.M)x[\bullet^l := y^l] \\
&= M[\bullet^l := y^l][\bullet^k := x[\bullet^l := y^l]^k] \\
&\equiv_\alpha M[\bullet^k := x^k][\bullet^l := y[\bullet^k := x^k]^l] \quad (*) \\
&= (\sigma v^k.((\sigma u^l.M)y))x;
\end{aligned}
$$

$$
\begin{aligned}
(iii) \qquad \lambda_v\text{-CS}^{\triangleright} \vdash\ & (\sigma u^l.u^l)x \\
&= u^l[\bullet^l := x^l] \\
&= x^l[\bullet^l := x^l] = x[\bullet^l := x^l][\bullet^l := x^l] \equiv_\alpha x[\bullet^l := x^l] \qquad (*) \\
&= (\sigma u^l.x)x.
\end{aligned}
$$

The three steps marked with (*) should recall that α-congruence ignores inner occurrences of labels. Also, by writing

$$x[\bullet^l := y^l]$$

we indicate that x could be a value with occurrences of the label l. This is *possible* because l is not created during the derivation and *desirable* because x may represent self-referential values. □

As a further illustration of the above principle, we prove a property of the eq?-operation on cells

$$\begin{aligned}\mathsf{eq?} \stackrel{df}{\equiv} \lambda c_1 c_2.(&\textbf{let } ((x_1\ (\mathsf{deref}\, c_1))(x_2\ (\mathsf{deref}\, c_2)))\\ &(\textbf{begin}\\ &\quad(\mathsf{set\text{-}cell!}\, c_1\ 1)\ \ (\mathsf{set\text{-}cell!}\, c_2\ 2)\\ &\quad(\textbf{let } (e\ (\stackrel{?}{=}\ (\mathsf{deref}\, c_1)\, 2))\\ &\qquad(\textbf{begin}\\ &\qquad\quad(\mathsf{set\text{-}cell!}\, c_1\ x_1)\ \ (\mathsf{set\text{-}cell!}\, c_2\ x_2)\\ &\qquad\quad e)))).\end{aligned}$$

The operation compares the identity of cells as cells as opposed to contents. Hence, eq? is characterized by:

Proposition 6.11. *Let c and d be distinct cells. Then,*

$$\lambda_v\text{-}CS^{safe} \vdash (\mathsf{eq?}\, c\, c) = \mathsf{True} \text{ and } \lambda_v\text{-}CS^{safe} \vdash (\mathsf{eq?}\, c\, d) = \mathsf{False}.$$

Note. A consequence of this statement is

$$\lambda_v\text{-}\mathrm{CS}^{safe} \vdash (\mathsf{eq?}\ (\mathsf{mk\text{-}cell}\, x)\ (\mathsf{mk\text{-}cell}\, x)) = \mathsf{False}.$$

Proof. We prove the first half of the statement, the second half being similar. By a minor generalization of Proposition 6.9, we can reduce the problem:

$$\lambda_v\text{-CS}^{safe} \vdash (\mathsf{eq?}\ c\ c) = (\mathbf{let}\ (x\ (\mathsf{deref}\ c))$$
$$(\mathbf{begin}$$
$$(\mathsf{set\text{-}cell!}\ c\ 2)$$
$$(\mathbf{let}\ (e\ (\stackrel{?}{=}\ (\mathsf{deref}\ c)\ 2))$$
$$(\mathbf{begin}$$
$$(\mathsf{set\text{-}cell!}\ c\ x)$$
$$e)))).$$

The rest is a simple calculation:

$$\lambda_v\text{-CS}^{\triangleright} \vdash (\mathbf{let}\ (x\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l)) \tag{1}$$
$$(\mathbf{begin}$$
$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ 2)$$
$$(\mathbf{let}\ (e\ (\stackrel{?}{=}\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l)\ 2))$$
$$(\mathbf{begin}$$
$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ x)$$
$$e))))$$
$$= (\mathbf{begin} \tag{2}$$
$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ 2)$$
$$(\mathbf{let}\ (e\ (\stackrel{?}{=}\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l)\ 2))$$
$$(\mathbf{begin}$$
$$(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ x)$$
$$e)))$$
$$= (\mathbf{let}\ (e\ (\stackrel{?}{=}\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ 2)^l)\ 2)) \tag{3}$$
$$(\mathbf{begin}$$

$$
\begin{aligned}
&(\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ 2)^{l}\ x) \\
&e))[\bullet^{l} := 2^{l}] \\
=\ &(\mathbf{begin} && (4)\\
&\quad (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ 2)^{l}\ x) \\
&\quad \mathsf{True})[\bullet^{l} := 2^{l}] \\
=\ &\mathsf{True}[\bullet^{l} := 2^{l}][\bullet^{l} := x[\bullet^{l} := 2^{l}]^{l}] && (5)\\
\equiv_{\alpha}\ &\mathsf{True} && (6)
\end{aligned}
$$

The derivation is a degenerated instance of the safe derivations described by Proposition 6.7. A term is simply reduced to a value. The *safe*-ness of the assignment steps is guaranteed because the second one undoes the effect of the first, the *safe*-ness of the delabeling step is based on the embedding between the two assignments. □

The lesson of this proof is simple. Proofs should rely as much as possible on abstract specifications à la Proposition 6.9, but we cannot expect that these specifications are always sufficient. If they are not, we must be ready to use a less abstract way of reasoning and to generalize the low-level proofs to new high-level characterizations. We return to this problem in the last chapter when we discuss related and future research.

Up to this point, none of the discussed specifications concerned higher-order functions. Although the underlying computations rely on function-valued functions for modeling state variables, the abstract equations are independent of their existence. This is different for the $\mathsf{Y}_!$-combinator, which we discuss next.

The combinator

$$\mathsf{Y}_! \equiv \lambda f.(\lambda g.(\sigma g.g)(\lambda x.fgx))\mathsf{I}$$

is an alternative means for creating recursive functions. The claim is that, given

a defining functional F for some recursively specified function, $\mathsf{Y}_!F$ returns the fixpoint of F and hence the appropriate meaning of the definition. With all the practice in correctness proofs about state variables, this claim is easy to prove:

Proposition 6.12. *Let F be a value. Then,*

$$\lambda_v\text{-}CS^{safe} \vdash \mathsf{Y}_!Fx = F(\mathsf{Y}_!F)x.$$

Proof. We calculate:

$$\begin{aligned}
\lambda_v\text{-CS}^{\triangleright} \vdash \mathsf{Y}_!Fx &= (\lambda g_\sigma.(\sigma g_\sigma.g_\sigma)(\lambda x.Fg_\sigma x))!x && (1)\\
&= (\sigma !^l.!^l)(\lambda x.F!^l x)x && (2)\\
&= (\lambda x.F!^l x)^l x && (3)\\
&= (\lambda x.F(\lambda x.F!^l x)^l x)x && (4)\\
&= F(\lambda x.F!^l x)^l x && (5)\\
&= F((\sigma !^l.!^l)(\lambda x.F!^l x))x && (6)\\
&= F((\lambda g_\sigma.(\sigma g_\sigma.g_\sigma)(\lambda x.Fg_\sigma x))!)x && (7)\\
&= F(\mathsf{Y}_!F)x. && (8)
\end{aligned}$$

The strategy of the calculation is simple. Steps (1) through (5) unfold the term $\mathsf{Y}_!Fx$ until there is the term of the structure FMx. Since $F[\ ]x$ forms an evaluation context, the rest is equally simple: steps (6) through (8) invert the first four steps, thus undoing the effects that make the first half of the derivation unsafe. □

The proposition finally verifies some old folklore among compiler builders. As mentioned in Section 3.3, $\mathsf{Y}_!$ builds faster recursive functions than the functional fixpoint combinator Y_v on ordinary machines. Therefore, compilers build recursive functions with truly self-referential closures instead of self-application as in Y_v. For this reason, even implementations of functional languages provide recursion as a

built-in syntactic facility so that it can be realized with this imperative strategy [36, 43, 71]. Since the two levels are separate, correctness arguments are generally informal. The advantage of Idealized Scheme is that the two aspects can be treated in the same language and reasoning system, and that the reasoning system is capable of proving the fixpoint-property of $\mathsf{Y}_!$.

A problem with our proposition is that it does not prove the uniqueness of the fixpoint. Since in general there are many fixpoints of functions, it is not clear which solution is produced by a fixpoint combinator. For the functional fragment it is possible to prove that Y_v produces the unique, minimal fixpoint with respect to an approximation ordering [70]. Again, for $\mathsf{Y}_!$ this is an open question.

6.3. Reasoning with Control-State

The two fragments of the λ_v-CS-calculus almost cover the entire programming language Idealized Scheme. Programs requiring the expressiveness of the entire language are those that must store control contexts in variables. In Section 3.3, we discussed two such cases: **iterate-until** and **generator**. The first construct uses a single-assignment variable, the second a true control-state variable. Both are characterized by operational equivalences; we prove the correctness of these.

The purpose of the **iterate-until**-loop is defined by the following equation:

$$(\textbf{iterate } F \textbf{ over } V \textbf{ until } P) \simeq_C F^m V$$
$$\text{where } m = min\{i \geq 0 \mid P(F^i V) = \mathsf{True}\},$$

for all pure functions F, predicates P, and values V. That is, the loop computes the values $V, (FV), (F(FV)), \ldots$ and returns the first that satisfies P. The original specification also requires that F be defined on all values $V, (FV), (F(FV)), \ldots$ An immediate consequence is that

$$P(F^i V) =_v \mathsf{False} \qquad \text{for } 0 \leq i < m,$$

where m is as above.

A functional implementation of **iterate-until** relies on recursion:

$$\begin{aligned}(\textbf{iterate } F \textbf{ over } V \textbf{ until } P) &\stackrel{df}{\equiv} (\textbf{rec } (l\ v)\ (\textbf{if } (Pv)\ v\ l(Fv)))V \\ &\equiv \mathsf{Y}_v(\lambda lv.(\textbf{if } (Pv)\ v\ l(Fv)))V.\end{aligned}$$

The correctness proof for this version is a simple induction:

Proposition 6.13. *Let F, V, P, and m be as specified above. Then*

$$\lambda_v\text{-}CS^{safe} \vdash (\textbf{iterate } F \textbf{ over } V \textbf{ until } P) = F^m V.$$

Proof. We first prove the following invariant:

$$\begin{aligned}\lambda_v\text{-}\mathrm{CS}^{safe} \vdash &(\textbf{iterate } F \textbf{ over } V \textbf{ until } P) \\ &= (\textbf{if } (PV)\ V\ (\textbf{iterate } F \textbf{ over } V \textbf{ until } P)).\end{aligned}$$

This part uses the fixpoint property of Y_v:

$$\begin{aligned}\lambda_v\text{-}\mathrm{CS}^{safe} \vdash &(\textbf{iterate } F \textbf{ over } V \textbf{ until } P) \\ &= (\textbf{rec } (l\ v)\ (\textbf{if } (Pv)\ v\ l(Fv)))V \\ &= \mathsf{Y}_v(\lambda lv.(\textbf{if } (Pv)\ v\ l(Fv)))V \\ &= (\textbf{if } (PV)\ V\ (\mathsf{Y}_v(\lambda lv.(\textbf{if } (Pv)\ v\ l(Fv)))(FV))) \\ &= (\textbf{if } (PV)\ V\ (\textbf{rec } (l\ v)(\textbf{if } (Pv)\ v\ l(Fv)))(FV)) \\ &= (\textbf{if } (PV)\ V\ (\textbf{iterate } F \textbf{ over } (FV) \textbf{ until } P)).\end{aligned}$$

The rest is an induction on m. Clearly, if m is 0, the invariant shows that **iterate-until** yields V. Otherwise, m is greater than 0. But then (PV) is False, and (**iterate** F **over** (FV) **until** P) yields $F^{m-1}(FV)$ by the inductive hypothesis.

□

Our imperative version of **iterate-until** is an Idealized Scheme version of a good compiler's output. We refer to it as **iterate!**-loop:

$$(\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P) \stackrel{df}{\equiv} (\textbf{let}\ (l_\sigma\ \mathsf{I})\ (\textbf{let}\ (x\ \mathcal{F}(\sigma l_\sigma.l_\sigma V))\ (\textbf{if}\ (Px)\ x\ (\textbf{throw}\ l_\sigma\ (Fx))))).$$

For the correctness proof we can actually follow the strategy of Proposition 6.13. The important part is to re-prove the invariant:

$$(\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P) = (\textbf{if}\ (PV)\ V\ (\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P)).$$

In order to show this, we use the upper level of the λ_v-CS-calculus:

$$\begin{aligned}
\lambda_v\text{-CS}^{\triangleright} \vdash & C[(\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P)] \\
&= C[(\textbf{let}\ (l_\sigma\ \mathsf{I})\ (\textbf{let}\ (x\ \mathcal{F}(\sigma l_\sigma.l_\sigma V))\ (\textbf{if}\ (Px)\ x\ (\textbf{throw}\ l_\sigma\ (Fx)))))] \\
&= C[(\textbf{let}\ (x\ \mathcal{F}(\sigma \mathsf{I}'.\mathsf{I}' V))\ (\textbf{if}\ (Px)\ x\ (\textbf{throw}\ \mathsf{I}'\ (Fx))))] \\
&= C[(\textbf{if}\ (PV)\ V\ (\textbf{throw}\ (\lambda x.C[\textbf{if}\ (Px)\ x\ (\textbf{throw}\ \mathsf{I}'\ (Fx))])^{l}\ (FV))))]
\end{aligned}$$

At this point, we must use the assumption that (PV) either holds or doesn't. In the second case, the result is obvious; in the first, we need a few more steps:

$$\begin{aligned}
\lambda_v\text{-CS}^{\triangleright} \vdash \ldots &= C[(\textbf{throw}\ (\lambda x.C[\textbf{if}\ (Px)\ x\ (\textbf{throw}\ \mathsf{I}'\ (Fx))])^{l}\ (FV))] \\
&= C[(\textbf{if}\ (P(FV))\ (FV)\ (\textbf{throw}\ (\lambda x.C[\textbf{if}\ (Px)\ x\ (\textbf{throw}\ \mathsf{I}'\ (Fx))])^{l}\ (F(FV)))))] \\
&= C[(\textbf{iterate!}\ F\ \textbf{over}\ (FV)\ \textbf{until}\ P)]
\end{aligned}$$

Together, the two cases imply the above invariant. An appropriate proposition follows:

Proposition 6.14. *Let F, V, P, and m be as specified above. Then*

$$\lambda_v\text{-}CS^{safe} \vdash (\textbf{iterate!}\ F\ \textbf{over}\ V\ \textbf{until}\ P) = F^m V.$$

The **iterate!**-example shares the folklore-property with the $\mathsf{Y}_!$-example. The recursive version is a special case of a tail-recursive function, and good compilers should eliminate tail-recursion in favor of **goto**-s and register assignments, *i.e.*, they should generate the imperative version of **iterate!**. Once again, we have verified (a particular instance of) a well-known optimization technique.

The generator example is more complicated than the imperative **iterate-until**-loop. It is an inherently imperative construct and cannot as easily benefit from proof techniques for some functional counterpart. The equational specification of (**generator** n vec) is given in Section 3.3 as

$$[V_1, \ldots, V_n]_n \simeq_C (\textbf{let}\ (G\ (\textbf{generator}\ n\ [V_1, \ldots, V_n]_n))\ [(G\mathsf{I}), \ldots, (G\mathsf{I})]_n).$$

An expanded implementation of the **generator**-form is

$$\begin{aligned}(\textbf{generator}\ n\ vec) \equiv \mathcal{F}(\lambda c_\sigma.(\textbf{let}\ (g_\sigma\ \mathsf{I})\\ (\textbf{begin}\\ (\mathcal{F}(\sigma g_\sigma.c_\sigma\ \lambda d.(\mathcal{F}(\sigma c_\sigma.g_\sigma\ \mathsf{I}))))\\ (\mathcal{F}(\sigma g_\sigma.c_\sigma\ (\pi_1^n\ vec)))\ldots(\mathcal{F}(\sigma g_\sigma.c_\sigma\ (\pi_n^n\ vec)))))).\end{aligned}$$

A first attempt at a proof of the above statement results in the following situation. For every evaluation context $C[\ ]$, we must show that the vector

$$[G\mathsf{I}, \ldots, G\mathsf{I}]_n$$

yields the above vector if G stands for the generator. Performing the calculation in a naïve way, we realize that every position of the vector function becomes the hole of an evaluation context—from left to right, that is—and that in the end we really obtain the desired vector. However, this method is too low-level, and it is indeed possible to deduce a more abstract characterization.

A generalization of the generator specification directly leads to the envisioned abstract generator theorem. Instead of considering the entire vector, we assume that we only look at some evaluation context $D[\ ]$ with possibly free G-s in the body of the **let**-expression:

$$(\textbf{let}\ (G\ (\textbf{generator}\ n\ [V_1,\ldots,V_n]_n))\ D[(G\mathsf{I})]).$$

The very same method now yields a more interesting result. Let us first introduce the abbreviation

$$(\textbf{generator}\ n\ [V_1,\ldots,V_n]_n)^{k,l}$$

for the value

$$\lambda d.\mathcal{F}(\sigma c^k.((\lambda x.\textbf{begin}\ \mathcal{F}(\sigma g^l.c^k V_1)\ \ldots \mathcal{F}(\sigma g^l.c^k V_n))^l\mathsf{I})),$$

which for some arbitrary c and g and some fresh labels l and k, is the result of

$$(\textbf{generator}\ n\ [V_1,\ldots,V_n]_n).$$

It is hereby important that c stands for an arbitrary value, *i.e.*,

$$\lambda_v\text{-CS}^{safe} \vdash \mathcal{F}(\sigma c^k.M) = \mathcal{F}(\sigma d^k.M).$$

This follows from

$$\begin{aligned}\lambda_v\text{-CS}^{\triangleright} \vdash C[\mathcal{F}(\sigma c^k.M)] &= ((\sigma c^k.M)(\lambda x.C[x])) \\ &= M[\bullet^k := \lambda x.C[x]^k] = \\ &= ((\sigma d^k.M)(\lambda x.C[x])) = C[\mathcal{F}(\sigma d^k.M)].\end{aligned}$$

Under these provisions, we can derive the following:

$$
\begin{aligned}
\lambda_v\text{-CS}^{\triangleright} \vdash & C[(\textbf{let } (G\ (\textbf{generator } n\ [V_1,\ldots,V_n]_n))\ D[(G\mathsf{I})])] \\
&= C[D[G\mathsf{I}][G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]] \\
&= C[D[(\lambda d.\mathcal{F}(\sigma c^k.(\lambda x.\textbf{begin } \mathcal{F}(\sigma g^l.c^k V_1)\ \ldots \mathcal{F}(\sigma g^l.c^k V_n))^l\mathsf{I})\mathsf{I})] \\
&\qquad [G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]] \\
&= ((\sigma c^k.(\lambda x.\textbf{begin } \mathcal{F}(\sigma g^l.c^k V_1)\ \ldots \mathcal{F}(\sigma g^l.c^k V_n))^l\mathsf{I}) \\
&\qquad (\lambda x.C[D[x]])[G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]) \\
&= (\textbf{begin} \\
&\qquad \mathcal{F}(\sigma g^l.(\lambda x.C[D[x][G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]])^k V_1)\ldots \\
&\qquad \mathcal{F}(\sigma g^l.(\lambda x.C[D[x][G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]])^k V_n)) \\
&= ((\sigma g^l.(\lambda x.C[D[x][G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]])^k V_1) \\
&\qquad (\lambda x.\textbf{begin } \mathcal{F}(\sigma g^l.c^k V_2)\ldots\mathcal{F}(\sigma g^l.c^k V_n))) \\
&= C[D[V_1][G := (\textbf{generator } n\ [V_1,\ldots,V_n]_n)^{k,l}]] \\
&\qquad [\bullet^l := \lambda x.\textbf{begin } \mathcal{F}(\sigma g^l.c^k V_2)\ldots\mathcal{F}(\sigma g^l.c^k V_n)^l] \\
&= C[D[V_1][G := \lambda d.\mathcal{F}(\sigma c^k.((\lambda x.\textbf{begin } \mathcal{F}(\sigma g^l.c^k V_2)\ldots\mathcal{F}(\sigma g^l.c^k V_n))^l\mathsf{I}))]] \\
&= C[D[V_1][G := (\textbf{generator } n-1\ [V_2,\ldots,V_n]_{n-1})^{k,l}]]
\end{aligned}
$$

This last step is justified by the above observation that c in the abbreviation for the **generator**-value is arbitrary. As before, the transition has again led to a term from which we can extract a variant of the starting term by undoing one of the previous steps:

$$\ldots = C[(\textbf{let } (G\ (\textbf{generator } n-1\ [V_2,\ldots,V_n]_{n-1}))\ D[V_1])].$$

Clearly, the derivation is safe:

Proposition 6.15. *Let $C[\ ]$ be an evaluation context. Then*

(i) for $n > 1$ and G possibly free in $C[\ ]$,

$$\lambda_v\text{-}CS^{safe} \vdash (\textbf{let } (G\ (\textbf{generator } n\ [V_1, \ldots, V_n]_n))\ C[(G\,I)]) = (\textbf{let } (G\ (\textbf{generator } n-1\ [V_2, \ldots, V_n]_{n-1}))\ C[V_1])$$

(ii) for $n \geq 1$ and G not free in $C[\ ]$,

$$\lambda_v\text{-}CS^{safe} \vdash (\textbf{let } (G\ (\textbf{generator } n\ [V_1, \ldots, V_n]_n))\ C[(G\,I)]) = C[V_1].$$

Proof Note. Part (ii) is a consequence of the proof of part (i). □

This proposition is a rather general statement about generators and can serve as a specification. One of its consequences is the original generator-equation:

Corollary 6.16.

$$\lambda_v\text{-}CS^{safe} \vdash [V_1, \ldots, V_n]_n = (\textbf{let } (G\ (\textbf{generator } n\ [V_1, \ldots, V_n]_n))\ [(G\,I), \ldots, (G\,I)]_n).$$

Proof. The proof is a simple calculation in the safe level of the λ_v-CS-calculus:

$$\begin{aligned}
\lambda_v\text{-CS}^{safe} \vdash & (\textbf{let } (G\ (\textbf{generator } n\ [V_1, \ldots, V_n]_n))\ [(G\,I), \ldots, (G\,I)]_n) \\
= & (\textbf{let } (G\ (\textbf{generator } n-1\ [V_2, \ldots, V_n]_{n-1}))\ [V_1, (G\,I), \ldots, (G\,I)]_n) \\
= & \ \ldots \\
= & (\textbf{let } (G\ (\textbf{generator } 1\ [V_n]_1))\ [V_1, \ldots, V_{n-1}, (G\,I)]_n) \\
= & [V_1, \ldots, V_n]. \qquad \square
\end{aligned}$$

This last example precisely illustrates what we mean by "[generalizing] low-level proofs to new high-level characterizations." A calculation in the λ_v-CS-calculus would have been a perfectly valid proof for the corollary, but such a proof contains recurring patterns. By extracting these patterns and unifying them into a general form, we were able to prove an abstract generator theorem. The theorem itself is probably more useful in other correctness proofs than the original specification.

At the end of this small feasibility study, it is appropriate to crystallize the major points. First, we have treated two different categories of examples and two different kinds of proofs. The first we call *horizontal* and it comprises the Σ_0^*-example, the cell-example, and the generator example. For these examples, we prove properties of and equivalence between programs, which could have been written by an ordinary programmer. The point of this kind of proof is to show that a program satisfies certain requirements, or that some more expressively written or more efficient program is equivalent to some obviously correct, but in-efficient or less expressive program. The second category is appropriately referred to as *vertical*. Such a vertical proof compares two unequal programs: one is written by a programmer, the other is generated by a compiler (or meta-programmer). The $Y_!$-combinator and the **iterate!**-loop are typical cases. Whereas the first category is the well-known kind of correctness proof that is found in the functional world, the second one is made possible by the introduction of imperative facilities. To some extent, these proofs capture the meaning of compilation, and we speculate that this is a fruitful field for future research.

Second, a comparison of the proof and programming styles for the functional and imperative world points to an interesting trade-off. In the introductory chapter we argued that functional programs for modeling control and state transitions contain repetitive patterns and are less modular than their imperative counterparts. The inverse apparently holds for correctness proofs. A proof about functional programs can formalize most conditions about its use-context as simple constraints on its free variables. In many cases a correctness proof for an imperative program must account for its use-context in the form of a universal quantification over evaluation contexts. However, we hope that the preceding examples have shown that this is not as stringent as it appears. Furthermore, a functional reformulation of imperative

programs also imposes an additional burden on the verification. It is not only necessary to validate the correctness of the functionality, the proof must also show that the representations of control and state information are treated and modified appropriately in the rest of the program. Finally, given the choice of bad programs with good proofs versus good programs with bad proofs, we opt for the latter.

Third and last, our set of examples is small and possibly non-representative. Nevertheless, we believe that these examples are specialized instances of common programming patterns, and that the theorems and proof techniques are general enough to be carried over to related problems. This point requires an extensive treatment of programming examples and certainly remains a topic for future research.

7. Summary and Perspective

After developing a syntactic theory of control and state in imperative higher-order programming languages, and after demonstrating the theory's usefulness, we have reached a first milestone in our project. It is time to look back, to compare, to integrate, and to project. The following sections are devoted to these tasks.

7.1. Results and Limitations

The goal of our work was to extend the λ_v-calculus to a calculus for an *imperative* higher-order programming language. Beyond higher-order functions, the envisioned system was to include first-class access to a functional abstraction of the current continuation and unrestricted, lexical assignment. This would ensure that we could syntactically express a broad variety of syntactic forms.

To understand the connection between programming languages and calculi, we studied the λ_v-calculus and its relationship to AE/ISWIM. We then defined an imperative extension of AE/ISWIM, Idealized Scheme, with the two required imperative abstractions: $\mathcal{F}$-applications for the manipulation of program control and σ-capabilities for the manipulation of program state. The programming style was illustrated with a set of meta-programs that embedded some commonly available

facilities. Based on this preliminary work, we developed a program rewriting semantics for this class of languages, extended it to an equational calculus, and worked out a set of example theorems in this theory.

The program rewriting semantics for imperative languages is an important step towards understanding the class of imperative languages on a purely symbolic level. With this semantics, a programmer can check the effect and result of an imperative program by simple, algebra-like program manipulations. A program is rewritten into another program until a value is reached. This semantics thus replaces other models that require auxiliary means for the explanation of imperative effects.

The calculus for Idealized Scheme is an equational extension of the rewriting system. It is a two-level system of reductions and computations. The reductions are freely applicable term relations, the computations are program relations. This division represents the context-sensitivity of imperative constructs. The calculus subsumes the rewriting equations (Proposition 5.18), and it determines a set of safe derivations that imply operational equivalences of programs (Theorem 5.27). The advantage of the calculus over the rewriting system is that equivalence proofs can generally rely on simple and safe reductions, but that rewriting rules are also available when needed.

The example proofs in the preceding chapter illustrate that the mixture of reductions and rewriting rules works well. The proofs reveal that there are some recurring principles and proof techniques. Most of these are generalizations of well-known counterparts in the functional world. A typical example of such a principle is **exit**-induction, which in its simplest form is recursion induction. The technique of undoing imperative, un-safe effects by inverting proof steps plays an important role in proofs of properties of assignment-based programs. Even though the set of examples is a good starting point, this area requires more investigation.

The major limitation of our approach is its concentration on computational abstractions of sequential imperative programming languages. We have neither addressed the issue of strong typing, which is a part of many available programming languages, nor the set of fundamental abstractions of less traditional languages. Although the typing of variables is only remotely related to computations, it is an important part of correctness proofs: getting the types correct often eliminates the majority of problems. Our set of fundamental abstractions covers the standard variety but neglects such concepts as quotation and un-quotation, explicit parallelism, or real-time constraints. In order to understand interpretation and compilation, operating systems, and real-time control within a single computational framework, these abstractions must be incorporated in future research.

7.2. Related Work

The rewriting semantics and λ_v-CS-calculus constitute one possible symbolic-equational reasoning system for imperative abstractions in an extended functional language. A different solution was worked out in two related dissertations at Stanford University. These are Talcott's thesis on continuations in higher-order functional languages [70] and Mason's thesis on the semantics of destructive first-order Lisp [41, 42]. Both formulate reasoning systems within the framework of operational equivalences—appropriately restricted versions of $\simeq_C$— but each has a slightly different emphasis. Whereas Mason is concerned with an equational theory for reasoning about operational equivalences, Talcott primarily focuses on the intensional semantics of programs and its relationship to extensional semantics. Neither works out a purely symbolic semantics or a syntactic calculus in the sense of the λ_v-CS-calculus.

Mason's theory is closely related to the assignment fragment of λ_v-CS. His programming language is first-order Lisp, but for the purpose of a comparison, this

is only an infringement of expressiveness. The idea behind the theory is relatively straightforward.

In principle, operational equivalence—referred to as strong isomorphism—is an operationally defined calculus. That is, while the relation satisfies the substitutional inference rule:

$$e_1 \simeq e_2 \Rightarrow C[e_1] \simeq C[e_2],$$

it lacks an axiomatic basis. Consequently, Mason develops a set of self-evident operational equivalences, which he calls axioms. The method for developing this set is simple. Each of the equations captures the mutual interaction of two syntactic forms. For example, the axioms for side-effects on cons-cells are appropriate modifications of our cell-theorems in Proposition 6.9. The correctness of these axioms is almost always obvious, but can also be derived from the formal definition of strong isomorphism.

In subsequent chapters Mason explores the practicality of his axioms. He develops notation and terminology for reasoning about lists, and with a "plethora of examples," he illustrates that the approach is well-suited for correctness proofs of list-processing programs. The examples range from a simple eq?-program to a structure editor for Lisp and deserve the attribute *realistic*. Mason also develops some principles and proof techniques of general applicability for object-oriented programming.

The major deficiency of this approach is its *ad hoc*-ness. The approach attempts to give a finite, axiomatic definition of the safe theory that is induced by the λ_v-CS-calculus. The problem is that there is no sufficient finite axiom set for this relation. When a correctness proof fails because of some yet-to-be-discovered axiom, the programmer has to go back to the store machine in order to find more information. In particular, in a world where a programmer can extend the syntactic facilities of his

language, *e.g.*, with macros, such a failure is likely to occur, and a syntactic calculus should be provided so that new axiom-like characterizations may be established.

Mason's work can be considered complementary to our own. Whereas we are interested in the fundamental nature of programming languages and their symbolic semantics, Mason attempts to study practical programming with and correctness proofs in a given and fixed programming language. The importance of his work with respect to our own is that he develops and explores a practical method for correctness proofs. Together, this method and the λ_v-CS-calculus have a great potential. The λ_v-CS-calculus provides the means to prove axioms in Mason's sense about new objects and syntactic forms for specialized application areas. These axioms can then be used like algebraic specifications in the correctness proofs of programs that employ the new entities. If the axioms turn out to be inadequate, the programmer can backtrack to the calculus.

The approach of Talcott towards reasoning with continuations is more machine-oriented. She does not search for axiom-like equations over control objects, but uses a modified CEK-machine for performing program manipulations. By working out a number of examples, she also develops and illustrates notation and terminology for reasoning about special applications of control contexts. Recently, Talcott [69] has experimented with an integration of the λ_v-CS-control fragment into her system. This approach resembles our suggested integration of Mason's method with the assignment fragment, and it looks rather promising.

A more important aspect of Talcott's system is the treatment of intensional properties of computations. Such properties are considered an integral part of programs and programming. There are mathematical and computational intensional properties. The former are captured with additional equalities and inequalities that programs satisfy under the assumption of instantaneous computation; the latter

account for the consumption of time and other resources.

The analysis of mathematical properties of programs is important for semantic considerations. A classical example is the minimality of fixpoints produced by Y_v. Under an approximation ordering $\lesssim_v$ that compares the set-theoretic containment of expressions interpreted as functions, this is formulated as

$$\text{for all } f,\ Ff =_v f \Rightarrow \mathsf{Y}_v F \lesssim_v f,$$

where F is a functional. Talcott generalizes this setting and introduces a hierarchy of equivalence and approximation relations that express various other properties of functions and computations.

Computational resources are always scarce and it is therefore necessary to analyze their usage. However, the analysis of algorithms is generally not a part of correctness proofs. The two are integrated in Talcott's systems by means of derived programs. Derived programs transform intensional properties into extensional ones and measure such aspects as the number of function applications, the number of primitives in use, *etc.* For example, the program $(\Sigma_0^* t)$ and its counterpart $(\mathsf{if}\ (\mathsf{occur0?}\ t)\ 0\ (\Sigma^* t))$ can be compared with respect to the number of tree nodes visited by each. Talcott shows that for a broad variety of properties there is a transformation for mapping a program to its derived program. The derived programs that compute these properties are structurally related to the original ones and generally re-interpret some of the primitive symbols.

The λ_v-CS-calculus makes no attempt at an intensional theory of programs. Yet, if the calculus is to be used for program specifications and transformations, such a theory must become a part of the system. We return to this topic in the next section where we discuss possible avenues of future research.

7.3. Future Research

The answering of mathematical questions almost always leads to new questions. This is also true for our research. We have only begun to explore the syntactic theory of fundamental programming abstractions, yet, there is already a host of practical implications and further theoretical questions. In the following subsections, we present some of the most obvious and interesting problems and discuss possible solutions. The first four subsections are theoretical and address such issues as an extended set of fundamental abstractions, the treatment of syntactic abstractions, a systematic analysis of proof principles, and the incorporation of intensional properties. The next three subsections contain discussions of practical proposals, namely, an exploitation of the rewriting semantics for a visual display of program evaluation, two new implementation strategies, and the extension of Idealized Scheme with a new fundamental abstraction. The last subsection is a list of less developed ideas, among others the typing of the λ_{υ}-CS-calculus. We hope that this analysis somewhat clarifies the limitations of our approach and how to overcome them.

7.3.1. Fundamental Abstractions

The design of programming languages is concerned with abstractions of recurring patterns. The expressiveness of programming languages is proportional to the set of facilities that it can express as syntactic abstractions. In this sense, Idealized Scheme is highly expressive because it subsumes almost all of the traditional programming languages, however, it is not the penultimate language. It lacks some important fundamental abstractions such as the call-by-name parameter-passing technique, facilities for quotation and un-quotation, and an abstraction for trial computations, just to name a few.

In his original work on the correspondence of programming languages and cal-

culi, Plotkin [47] shows that a call-by-name based language requires a calculus about values with the original β-rule. More precisely, the interesting results of calculations are *values*—as opposed to normal forms—but the basic axiom is the β-relation:

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

for all M and N, where N is not necessarily a value.

Plotkin also investigates the relationship between call-by-value and call-by-name. There are two main techniques for simulating call-by-name with call-by-value. The first relies on freezing and thawing argument expressions, *i.e.*, the introduction and application of dummy abstractions to prevent untimely evaluations. The second is based on a transformation that employs a continuation-passing strategy. The realization of both techniques requires more than just syntactic abstraction. The same holds for the inverse direction. Hence, call-by-name is a fundamental abstraction with regard to Idealized Scheme and should be integrated to broaden the semantic basis. It is relatively easy to see that a β-rewriting rule for a λ_{name}-abstraction can fit into the rewriting system, but the open question is whether and how the β-relation meshes with the λ_v-CS-calculus.

The Lisp-quotation form poses an entirely different problem. An expression (**quote** *exp*) yields a value that represents the (abstract) syntactic counterpart of *exp*. The introduction of **quote** requires a constant set rich enough to model the syntactic variable domain and the improper symbols. The expression domain can be represented with lists and vectors, *e.g.*,

$$(\textbf{quote}\ (\lambda x.y)) \equiv (\mathsf{cons}\ (\textbf{quote}\ \lambda)[(\textbf{quote}\ x), (\textbf{quote}\ y)]).$$

It is immediately clear that **quote** maps all expressions to normal forms. Since normal forms are effectively comparable, expressibility of **quote** implies a solution

to the program comparison problem, and hence, **quote** cannot be an ordinary combinator [46:32]. On the other hand, the introduction of **quote** as a new form creates *referentially opaque* term positions [48], *i.e.*, we can no longer conclude

$$(\textbf{quote}\ x) = (\textbf{quote}\ y)$$

from

$$x = y.$$

Reasoning in a system with **quote** becomes inherently context-*sensitive*.

From the **quote**-facility it is only a short step to the more general field of *reification* and *reflection* [57]. In simple terms, reification and reflection is about quotation and un-quotation; more technically, reification and reflection refers to the capability of a program to inspect and alter its current computational state. Thus far, these capabilities have been studied in the framework of denotational semantics. In this context, reification and reflection means that programs can at any time access and change (part of) the program text, the environment, and the continuation [57, 75].[1] Even though **quote** and its generalized relatives are incompatible with the calculus, it is nevertheless an interesting question how known reification and reflection capabilities can be expressed in the rewriting system, or what reification and reflection means with regard to a rewriting semantics. A comparison of these capabilities in various frameworks could lead to a more general theory of this phenomena.

Our final example of a different fundamental abstraction is trial computation. A trial computation is a variant of McCarthy's **amb**-construct [45]. It refers to the idea that a computation is performed for a limited period instead of an indefinite amount of time. If the computation returns a value after the allotted time, the

[1] Programming with variable-value bindings as first-class objects is an interesting experiment by itself and leads to quite different language paradigms [9, 23].

trial is successful and the value is its result; otherwise, the trial is unsuccessful and returns a suspended computation. A trial is useful in circumstances where there are alternative ways to make progress on a computation, but it is unknown whether the various sub-computations terminate. Practical examples of this kind include operating systems for time-shared computation and search situations in artificial intelligence programs. Once again, we would like to have a rewriting semantics and a calculus description, which could provide new insights into the nature of these computational primitives. Whereas the first part is obviously feasible, the second part constitutes a true problem. With a trial facility added, expressions not only determine a value, but the time it takes to compute the value. Hence, a calculus for trial computations must include a facility to reason about both aspects of expressions.

7.3.2. Syntactic Abstractions

Dual to the concept of fundamental abstraction is that of syntactic abstraction. Syntactic abstractions are those linguistic facilities that can be explained as abbreviations of fundamental expression patterns. They hide details of these patterns and thus facilitate the writing and reading of programs. This, however, is contrary to the way we treat syntactic abstractions during correctness proofs. Our informal rule is to expand these abbreviations as far as possible. What we would really like to have is a system where proofs are on the same level as programs. In other words, there should be a method that seamlessly incorporates syntactic abbreviations for Idealized Scheme into its syntactic theory.

One promising approach seems to be the following two-step procedure. First, every syntactic abbreviation is analyzed as to how its sub-expressions are evaluated. Since the expansion determines a unique sequencing for the evaluation of sub-expressions, this defines when a hole at a sub-expression position in the abbre-

viation can become the hole of an evaluation context. Second, once it is known in which order a syntactic abstraction evaluates its pieces, we can try to establish general theorems about the various cases with respect to the possible set of C-redexes.

Let us illustrate these two steps with the simple example of **begin**-expressions. A **begin**-expression was defined with the following equation:

$$\textbf{begin}\ M_1 \ldots M_n \stackrel{df}{\equiv} (\lambda x_1 \ldots x_n.x_n) M_1 \ldots M_n.$$

The rewriting semantics determines that sub-expression M_i becomes the fill term of an evaluation context after M_1 through M_{i-1} are reduced to values. This can be stated on the level of a **begin**-expression as:

$$\textbf{begin}\ V_1 \ldots V_{i-1}\ [\ ]\ M_{i+1} \ldots M_n \qquad \text{for } 1 \leq i \leq n$$

are evaluation contexts. For the second step we must now analyze how a given redex in this new class of contexts behaves. Consider the case of a σ-redex, *i.e.*,

$$\textbf{begin}\ V_1 \ldots V_{i-1}\ ((\sigma x^l.M)y)\ M_{i+1} \ldots M_n.$$

A brief look at the expansion shows that this is equivalent to

$$(\sigma x^l.\textbf{begin}\ V_1 \ldots V_{i-1}\ M\ M_{i+1} \ldots M_n)y.$$

With these new equivalences, the proof of statement (iv) in Proposition 6.9 becomes more perspicuous:

$$\begin{aligned}
&\textbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ (\mathsf{deref}\ (\mathsf{mk\text{-}cell}\ x)^l) \\
&\quad = (\sigma x^l.\textbf{begin}\ x^l x^l)y \\
&\quad = (\sigma x^l.x^l)y = (\sigma x^l.y)y \\
&\quad = (\sigma x^l.\textbf{begin}\ x^l y)y \\
&= \textbf{begin}\ (\mathsf{set\text{-}cell!}\ (\mathsf{mk\text{-}cell}\ x)^l\ y)\ y.
\end{aligned}$$

More examples, where we have made implicit use of this method, can be found towards the end of the preceding chapter.

For many of the syntactic abstractions and the particular abbreviation technique that we have employed, it seems that these two steps can be mechanized. This would mean that a programmer can explore the axiom-like characterization of his syntactic extensions to a language with the help of a program. Such automatically generated theorems and rewriting rules would be of invaluable help in larger correctness proofs.

A different issue about syntactic abstractions concerns the target language. Until now, we have used Idealized Scheme for this purpose. Another possibility is Λ_{CS}. The difference between the two is that, in addition to all Idealized Scheme programs, Λ_{CS} can express all *values* and intermediate program stages. Because of this, there are syntactic abstractions that can be formulated with Λ_{CS}, but not with Idealized Scheme.

In order to illustrate the difference, we consider an abstraction called **once**. For every occurrence of (**once** *exp*) in a program, the respective expansion must ensure that *exp* is only evaluated once during the evaluation of the program. At other times, (**once** *exp*) is to return the first value of *exp*. At first glance, the solution has something to do with an expression that memorizes its result:

$$(\textbf{let } (f_\sigma\ \lambda d.exp)\ (\textbf{let } (v\ (f_\sigma\ \mathsf{I}))\ (\sigma f_\sigma.v)(\lambda d.v))).$$

However, imagine that (**once** *exp*) is embedded in a program like

$$(\lambda x.xx)(\lambda d.(\textbf{once } exp))\mathsf{I}.$$

If we β-reduce this expression after an expansion of **once** into the above expression, *exp* is replicated and evaluated *twice*. The trick is that the expansion of a **once**-expression must keep track of the sharing relation between instances of the original

occurrence of *exp*. A partially evaluated version of the above expression achieves the proper effect:

$$(\textbf{let } (v\ ((\lambda d.exp)^{l}\ 1))\ (\sigma(\lambda d.exp)^{l}.v)(\lambda d.v))).$$

The sharing relation l, which cannot be expressed in Idealized Scheme, is precisely what is needed to track the distribution of *exp*.

Unfortunately, the extension of the semantic target language interferes with the orthogonality of Idealized Scheme-based abstractions to reduction relations. Λ_{CS}-based syntactic abstractions can only be expanded at the beginning of an evaluation. We nevertheless believe that this extension is an interesting issue for further research.

7.3.3. Proof Principles, Proof Techniques, and Program Development

The preceding chapter on reasoning with the λ_v-CS-calculus and the comparison of our work with that of Mason have made it clear that we must acquire more experience on working with λ_v-CS. There are three main concerns: proof principles, proof techniques, and their impact on program development.

A correctness proof for program pieces requires two major insights: *what* the theorem should be and *how* the proof should proceed. Guidelines for answering these questions are proof principles. Two of these we have identified above, *i.e.*, **exit**-induction and object-axioms about mutual effects. However, we believe that there are many more interesting principles. For example, the generalized generator-theorem (Proposition 6.15) indicates that there is a connection between objects and immediate evaluations within a given scope. We are hopeful that this connection can be abstracted into a general principle for higher-order objects. With respect to control effects and first-class control contexts, we must experiment with $\mathcal{F}$, explore its usefulness in different situations, and derive some related proof principles.

The call for more proof techniques is related to the one for proof principles. Once it is decided what to prove and how to prove it, some general term manipulation techniques become important. One example of this is the expansion of abbreviations—which we decided to avoid; another is the inversion of proof steps as described in Proposition 6.7. Predictions on this field are difficult, but in any case, such techniques should explicitly be observed and collected.

Finally, as we have mentioned above in conjunction with **exit**-induction, the application of proof principles to program design is highly important. Few programs are actually worth proving correct, but if programs are built under guidance of some proof principle, the likelihood that they are correct improves. A good example of this is data recursion. If a recursive program considers all possible cases of the inductive data structure definition, it has a good chance of being correct. Furthermore, if every recursive function invocations works on proper sub-pieces of a data structure, we can expect that the function terminates. As we have seen in Section 6.1, this also works for **exit**-induction. Other exception-like situations should be inspected for further principles.

A similar case can be built for the implementation of abstract data types versus abstract data objects. The theory of abstract data types has gotten to a point where such systems can automatically derive efficient implementations of types: see the development of the language SETL [55]. This should also be possible for abstract data objects like cells. It would be interesting to find out to what extent an equational specification determines an implementation, and whether there is an algorithm for completing and/or enhancing such specifications.

7.3.4. Calculi for Intensions

The most important application of program calculi is the domain of compilation or, more generally, of program transformation. Program transformations rely on

observational equivalences between expressions and implementation-dependent inequivalences. The λ_v-CS-calculus is a good basis for establishing equivalences, but it lacks capabilities for deriving inequalities. Talcott's work indicates how these aspects could be combined into a single system.

There are two sides to the problem: mathematical and computational properties. Talcott has shown that the latter can be integrated into extensional relationships via derived programs. This strategy should also work for the calculus. However, since many of the derived properties depend on attributes of the underlying evaluation mechanism, the approach must be elaborated and possibly parameterized over evaluation systems. This also suggests that a set of calculi for different intensions, *i.e.*, theories of inequalities with respect to a given intension, may be a possible alternative.

Mathematical intensions in Talcott's system are expressed with approximation orderings that are based on a CEK-like machine. The maximal approximation relation $\precsim_v$ is a symmetric half of $\simeq_{CEK}$:

$$\simeq_{CEK} = \precsim_v \cap \succsim_v .$$

In order to determine operational approximation, Talcott establishes some axiom-like theorems. This is similar to Mason's approach for reasoning with strong isomorphism. Given this parallel situation, we wonder whether there is a syntactic counterpart to operational approximation that corresponds to a calculus. In other words, we are looking for a syntactic relation $\sqsubseteq_v$ such that

$$M \sqsubseteq_v N \Rightarrow M \precsim_v N.$$

Provided such calculi exist, it should be possible, for example, to prove syntactically the minimality of Y_v- and $\mathsf{Y}_!$-fixpoints.

7.3.5. Feedback Information for Language Design

Originally, the control fragment of the λ_v-CS-calculus was designed around Scheme's control operation *call-with-current-continuation*, abbreviated call/cc. As explained in Section 3.2, call/cc applies its argument to the continuation of the entire application. The difference between the continuation and the functional abstraction of the continuation is that the former eliminates the continuation of its invocation. Furthermore, call/cc does not give its argument total control over the continuation since the continuation of the call/cc-application remains the default continuation for the evaluation of the call/cc-argument. This leads to rather baroque reductions and computations. Two different sets of relations are required. The first captures the behavior of an abort-application $(\mathcal{A}M)$:

$$\begin{aligned}(\mathcal{A}M) &\triangleright M,\\ V(\mathcal{A}M) &\longrightarrow (\mathcal{A}M),\\ (\mathcal{A}M)N &\longrightarrow (\mathcal{A}M);\end{aligned}$$

the second group resembles the $\mathcal{F}$-relations and explains how call/cc provides its argument with an abstraction of its control context:

$$\begin{aligned}(\text{call/cc } M) &\triangleright M(\lambda x.\mathcal{A}x),\\ V(\text{call/cc } M) &\longrightarrow (\text{call/cc } \lambda k.V(M(\lambda m.\mathcal{A}(k(Vm))))),\\ (\text{call/cc } M)N &\longrightarrow (\text{call/cc } \lambda k.(M(\lambda m.\mathcal{A}(k(mN)))N)).\end{aligned}$$

The $\mathcal{A}$-applications on the right-hand sides are necessary to ensure that all continuation objects immediately eliminate their control contexts upon application.

The second group of reductions contains some regular patterns. The most outstanding is the replication of the call/cc-context, *i.e.*, the term N. If we omit the

occurrence that is outside of the continuation, we obtain simpler reductions:

$$(\mathcal{C}\ M) \triangleright M(\lambda x.\mathcal{A}x),$$

$$V(\mathcal{C}\ M) \longrightarrow (\mathcal{C}\ \lambda k.(M(\lambda m.\mathcal{A}(k(Vm))))),$$

$$(\mathcal{C}\ M)N \longrightarrow (\mathcal{C}\ \lambda k.(M(\lambda m.\mathcal{A}(k(mN))))),$$

and we can still implement call/cc as a syntactic abstraction:

$$(\text{call/cc}\ M) \equiv \mathcal{C}(\lambda k.k(Mk)).$$

Unlike call/cc, $\mathcal{C}$ can express $\mathcal{A}$-applications:

$$(\mathcal{A}M) \equiv \mathcal{C}(\lambda d.M) \qquad \text{where } d \notin FV(M).$$

The inverse relationship is:

$$(\mathcal{C}\ M) \equiv (call/cc\ (\lambda k.\mathcal{A}(Mk))).$$

Although the introduction of $\mathcal{C}$ is an essential improvement for the axiom system, the new reductions and the computation still share a common pattern: the occurrence of the form $(\mathcal{A}\ldots) \equiv \mathcal{C}(\lambda d.\ldots)$. It is easy to see that an omission of these additional $\mathcal{C}$-applications leads to the $\mathcal{F}$-relations in their current form. Given $\mathcal{F}$'s higher degree of expressiveness and the simplicity of the axioms, $\mathcal{F}$ is an improvement for a semantic meta-language like Scheme.

What we have just experienced is an instance of well-known law: a design process provides feedback information on the design object. In our case, the design of a calculus for control contexts has revealed that call/cc is not the most expressive fundamental control abstraction. Idealized Scheme is the product of a first iteration of the calculus design process. Since $\mathcal{F}$ replaces call/cc, we feel justified calling the result *Scheme*. There is, however, another problem that plagues all three control calculi and whose solution necessitates the introduction of a new, fundamental control construct.

The problem that we refer to is the division of the calculi into two levels. A possible and obvious fix is to introduce a top-level marker # such that the root of a program is uniquely identified. Then, the control computation relation becomes a true reduction:

$$(\#\,(\mathcal{F}M)) \longrightarrow (\#\,(M(\lambda x.x))).$$

Unfortunately, this solution eliminates the division by pushing it into the language syntax. However, the new syntactic category of #-applications can be merged into the set of expressions with a relatively simple move. Instead of allowing only one #-application at the root of a term, we introduce the #-application as a new first-class construct and interpret it as a fundamental programming abstraction.

The new grammar for the control calculus is a minor extension of the original:

$$M ::= x \mid \lambda x.M \mid MN \mid \mathcal{F}M \mid \#M.$$

Programs are identified as the set of closed #-applications. Since programs should reduce to values, we need an additional reduction relation. Once the #-argument is reduced to a value, the entire application can return this value:

$$(\#\,V) \longrightarrow V \qquad \text{if } V \text{ is a value.}$$

If we now assume that an interactive CEK-machine automatically supplies the #-part to a program, a dialogue with the machine looks like

$$\#\ M_1$$

$$V_1$$

$$\#\ M_2$$

$$V_2$$

$$\ldots$$

In other words, for the control fragment # corresponds to the prompt of an interactive machine. Our construction thus interprets the prompt as a program control delimiter and allows a programmer to define a limited control domain whenever it is appropriate.

The true challenge behind the use of the feedback information is the seamless incorporation into the original language semantics and the associated programming paradigms. For #-applications this does not seem difficult. Given that a #-application marks the extent to which a program can manipulate its control context, it is implementable with a stack marker and a knowledgeable stack copy function. A #-application also has some obvious applications in search-and-backtrack situations, where a part of the control context determines the next result. Furthermore, since #-applications delimit the scope of control operations, they are also well-suited for situations when a program must constrain dynamic control. For example, a program that works on files is generally responsible for opening and closing the file. In a world without #-applications this is hard to enforce because control jumps may get around the close operation [28]. With a #-application, the respective code becomes straightforward:

$$(\textbf{begin}\ (\mathsf{open}\ \mathit{file})(\#\ M)(\mathsf{close}\ \mathit{file})),$$

where M is code for the manipulation of *file*. We believe that this work and similar investigations for the assignment calculus will yield additional insights into the design of programming languages.

7.3.6. New Implementation Strategies for Imperative Languages

An equational semantics not only provides information on the structure of programming languages, but also on their implementation. One of the first to realize this for the compilation of AE/ISWIM was Burge [7]. He used Curry's [12, 13] λ-abstraction

algorithm for mapping Λ-expressions into combinator expressions to capture the essence of compilation. The translation eliminates bound variables, the predominant non-machine related notion. An evaluation in combinatory logic simulates parts of constant folding and extraction. The rest of a compilation process is the optimization phase, which can also be interpreted as an exploitation of combinatory equivalences.

For a long time, an application of the same technique to the compilation process of imperative languages has been considered impossible, the major obstacle being the lack of an appropriate equational calculus. Instead, people have tried to exploit various kinds of denotational semantics for the specification of code generation [3, 53, 73]. This, however, has the disadvantage that the strategy for writing denotational semantics predetermines many parameters of a translation and evaluation scheme. A direct translation from λ_v-CS to a suitable extension of combinatory logic would avoid this predetermination. It would concentrate on the essence of compilation and would leave the other parameters open. In particular, the choice of machine architecture and appropriate optimization techniques would still be undecided.

One of the open choices is the one of evaluating programs in a sequential or parallel manner. The reduction and computation system defines an evaluation function for Idealized Scheme that is extensionally equivalent to the CESK-evaluation function, but is intensionally a parallel system. This can best be seen from the reformulation of the standard evaluation function in Definition 7.1.

This function definition reveals that practically all applications, that is, the principal evaluation vehicles, can evaluate their sub-expressions in parallel. The synchronization of control- and side-effects happens naturally through the bubbling-up movement of the redexes. The only exception occurs when an $\mathcal{F}$-application be-

Definition 7.1: The *Eval*-function

Let U and V range over values; S over semi-values, that is, $\mathcal{F}$-, $\mathcal{D}$-, σ-, and λ-applications (whose variables are in Var_σ); M, N, P, and Q over arbitrary terms; and X over labeled values. Then the *Eval*-function is defined as:

$$Eval(M) \equiv \begin{cases} Eval(N[x := V^n]) & \text{if } seval(M) \equiv (\lambda x.N)V \\ Eval(N(\lambda x.x)) & \text{if } seval(M) \equiv (\mathcal{F}N) \\ Eval(N[\bullet^n := V^n]) & \text{if } seval(M) \equiv (\sigma U^n.N)V \\ Eval(NV[\bullet^n := V^n]) & \text{if } seval(M) \equiv (\mathcal{D}\,NV^n) \\ seval(M) & \text{otherwise.} \end{cases}$$

where *seval* is defined in four inductive clauses:

$$seval(V) \equiv V \text{ and } seval(S) \equiv S;$$

and, if $seval(M) \equiv U$ and $seval(N) \equiv V$, then

$$seval(MN) \equiv \begin{cases} seval(P[x := V]) & \text{if } U \equiv \lambda x.P,\ x \in Var_\lambda \\ UV & \text{otherwise;} \end{cases}$$

and, if $seval(M) \equiv U$ and $seval(N) \equiv S$ then

$$seval(MN) \equiv \begin{cases} (\lambda x.UP)V & \text{if } S \equiv (\lambda x.P)V \\ \mathcal{F}\lambda\kappa.P(\lambda v.\kappa(Uv)) & \text{if } S \equiv \mathcal{F}P \\ (\sigma X.UP)V & \text{if } S \equiv (\sigma X.P)V \\ \mathcal{D}\,(\lambda v.U(Pv)X) & \text{if } S \equiv \mathcal{D}\,PX; \end{cases}$$

and, finally, if $seval(M) \equiv S$, then

$$seval(MN) \equiv \begin{cases} (\lambda x.PN)V & \text{if } S \equiv (\lambda x.P)V \\ \mathcal{F}\lambda\kappa.P(\lambda f.\kappa(fN)) & \text{if } S \equiv \mathcal{F}P \\ (\sigma X.PN)V & \text{if } S \equiv (\sigma X.P)V \\ \mathcal{D}\,(\lambda v.PvN)X & \text{if } S \equiv \mathcal{D}\,PX. \end{cases}$$

comes the function part of an application. This indicates that the program may never want to evaluate the argument part of an application, and that an early eval-

uation of this argument may be a waste of computational resources. The situation corresponds to parallel evaluations in a call-by-name world. It constitutes the only case of speculative parallelism in such a system.

Another possibility for parallel evaluations is based on Corollary 5.22. According to this corollary we can use *any* evaluation strategy for a program if we are only interested in observable values. One conceivable strategy is to evaluate all existing reduction and computation redexes simultaneously. Naturally, this is a rather naïve and speculative scheme, but it may prove successful if the bubble movements can be contained as much as possible. Then, the implementation relies on advances in the field of so-called functional architectures. If such architectures ever realize their promised potential, it will be almost straightforward to extend the approach to the implementation of imperative higher-order languages.

7.3.7. Dynamic stepping

The rewriting semantics gives rise to another practical application of our work. It provides the theoretical underpinning for a dynamic stepper or debugger. Currently, such tools work on the static program text, leaving it to the programmer to imagine the dynamic process. With the rewriting semantics, this can be changed. The displayed program text can represent the entire machine state. A division between current redex and current context immediately clarifies the control aspects of a program, variable contents can be hidden until needed, sharing relationships displayed when required. There is no need to know the structure of the underlying machine, the organization of the run-time stack, the allocation scheme for registers, or similar things.

Since the underlying language is powerful enough to express traditional programming constructs, such a tool is language independent and can be tailored towards particular applications. In order to preserve the syntactic level of expressive-

ness, an addition of syntactic abstractions requires an extension of the rewriting rules. This is related to the problem addressed above in Subsection 7.3.2, and we hope that the discussed solution can be integrated into a dynamic stepper.

We perceive two major application areas for a dynamic stepper. On one hand, it constitutes a promising learning tool for understanding and experimenting with programs. It is operational enough to communicate the appropriate notions, but it is also machine-independent enough to teach abstract principles. On the other hand, a dynamic stepper may be the right debugging tool for professional programmers. In any case, the dynamic stepper is another example where our theoretical work may yield a practical result.

7.3.8. Miscellaneous

Beyond these problems, there is a series of interesting questions that we want to mention without further discussion. The first is an expansion on the topics of syntactic abstractions and modeling semantics. The traditional λ-calculus has become the standard target language for denotational semantics. We suggest investigating the use of λ_v-CS-calculus for this role and expect that this may offer some advantages.

This consideration immediately raises concerns about a (direct) mathematical model for the λ_v-CS-calculus. It is not clear to us what it really means to have such a model, but given the insight models have provided for the classical λ-calculus, it is certainly an interesting problem.

A related matter is the construction of a typed λ_v-CS-calculus. The typed λ-calculus was introduced and studied in order to circumvent the problems of self-application. A typed λ_v-CS-calculus would avoid self-references and perhaps would shed some light on the problem of types and type inference for imperative programs.

Finally, there is the question of other reasoning systems for imperative program-

ming languages. Those too, define a language semantics and provide a framework for inspecting and verifying programs. Noticeably the work on Floyd-Hoare logics has received a great deal of attention. The relationship to this work is not clear, but merits consideration.

7.4. Concluding Remarks

We have reached the end of our endeavor. Our contribution to the study of programming languages should be perceived as an attempt to reconcile two diverging currents in the programming language community. The syntactic theory of control and state in imperative higher-order programming languages combines the best of both worlds: capabilities for expressing control and state transitions and a symbolic semantics for direct program manipulations. The beneficiary is the programmer who can succinctly express his thoughts about problem solutions and still reason on the level of program equivalences. The contribution does not answer all open questions, indeed it opens a series of new ones. But, we sincerely hope that this is only the end of a first step, and we expect that a further exploration of the λ_v-CS-calculus will yield more fruitful insights into the nature of computation.

References

1. Abdali, S.K. A lambda-calculus model of programming languages. *Journal of Computer Languages* (Pergamon Press) **1**, 1976, 287–301; 303–320.

2. Abelson, H. and G.J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

3. Appel, A. *Semantics-directed Code Generation.* Ph.D. dissertation, Carnegie-Mellon University, 1985.

4. Böhm, C. Alcune proprietà della forma $\beta\eta$-normali nel λ-K-calcolo. Pubblicazioni dell' Istituto par le Applicazioni del Calcolo, Rome, 1968.

5. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* rev. ed. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.

6. Berry, G. Séquentialité de l'evaluation formelle des λ-expressions. In *Proc. 3rd International Colloquium on Programming*, 1978.

7. Burge, W. *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass., 1975.

8. BURSTALL, R.M. Semantics of assignment. In *Machine Intelligence* **2**, edited by E. Dale and D. Michie. American Elsevier Publishing Company, New York, 1968, 3–20.

9. BURSTALL, R.M. AND B. LAMPSON. A kernel language for abstract data types and modules. In *Proc. International Symposium on Semantics of Data Types.* Lecture Notes in Computer Science 173. Springer-Verlag, New York, 173, 1–50.

10. CHURCH, A. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, 1941.

11. CLINGER, W.D., D.P. FRIEDMAN AND M. WAND. A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M. Nivat. Cambridge University Press, London, 1985, 237–250.

12. CURRY, H.B. Grundlagen der kombinatorischen Logik. *Amer. J. Math.* **52**, 1930, 509–536; 789–834.

13. CURRY, H.B. AND R. FEYS. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1958.

14. FELLEISEN, M. Reflections on Landin's J-operator: A partly historical note. *Journal of Computer Languages* (Pergamon Press), 1987, to appear.

15. FELLEISEN, M. AND D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.

16. FELLEISEN, M. AND D.P. FRIEDMAN. A closer look at export and import statements. *Journal of Computer Languages* (Pergamon Press) **11**, 1986, 29–37.

17. FELLEISEN, M. AND D.P. FRIEDMAN. A reduction semantics for impera-

tive higher-order languages. In *Proc. Conference on Parallel Architectures and Languages Europe, Volume II: Parallel Languages.* Lecture Notes in Computer Science 259. Springer-Verlag, Heidelberg, 1987, 206–223.

18. Felleisen, M. and D.P. Friedman. A calculus for assignments in higher-order languages. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.

19. Felleisen, M., D.P. Friedman, B. Duba, and J. Merrill. Beyond continuations. Technical Report No 216, Indiana University Computer Science Department, 1987.

20. Felleisen, M., D.P. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Proc. First Symposium on Logic in Computer Science*, 1986, 131–141.

21. Fischer, M.J. Lambda calculus schemata. In *Proc. ACM Conference on Proving Assertions About Programs, SIGPLAN Notices* **7**(1), 1972, 104–109.

22. Friedman, D.P., C.T. Haynes, and E. Kohlbecker. Programming with continuations. In *Program Transformations and Programming Environments*, edited by P. Pepper. Springer-Verlag, Heidelberg, 1985, 263–274.

23. Gelernter, D., S. Jagannathan, and T. London. Environments as first class objects. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 98–110.

24. Griswold, R.E. and M.T. Griswold. *The ICON Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 1983.

25. Halpern, J.Y., A.R. Meyer, and B.A. Trakhtenbrot. The semantics of local storage, or What makes the free-list free? In *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984, 245–257.

26. Haplpern, J.Y., J.W. Williams, E.L. Wimmers, and T.C. Winkler. Denotational semantics and rewrite rules for FP. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, 1985, 108–120.

27. Haynes, C. T. Logic continuations. *J. Logic Program.* 4, 1987, 157–176.

28. Haynes, C. and D.P. Friedman. Embedding continuations in procedural objects. In *ACM Trans. Program. Lang. Syst.*, 1987, to appear.

29. Haynes, C.T., D.P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press) **11**, 1986, 143–153.

30. Henderson, P. *Functional Programming: Application and Implementation.* Prentice-Hall International, London, 1980.

31. Henson, M.C. and R. Turner. Completion semantics and interpreter generation. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, 1982, 242–254.

32. Knight, T. An architecture for mostly functional languages. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, 1986, 105–112.

33. Kohlbecker, E. *Syntactic Extensions in the Programming Language Lisp.* Ph.D. dissertation, Indiana University, 1986.

34. Kranz, D., et al. ORBIT: An optimizing compiler for Scheme. In *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices* **21**(7), 1986, 219–233.

35. Landin, P.J. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM,* 8(2), 1965, 89–101; 158–165.

36. Landin, P.J. A λ-calculus approach. In *Advances in Programming and Non-numerical Computation*, edited by L. Fox. Pergamon Press, New York, 1966,

97–141.

37. Landin, P.J. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*, edited by T.B. Steel. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1966, 266–294.

38. Landin, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.

39. Landin, P.J. An abstract machine for designers of computing languages. In *Proc. IFIP Congress*, 1965, 438–439.

40. Landin, P.J. The mechanical evaluation of expressions. *Comput. J.* **6**(4), 1964, 308–320.

41. Mason, I. A. Equivalences of first-order Lisp programs. In *Proc. First Symposium on Logic in Computer Science*, 1986, 105–117.

42. Mason, I. A. *The Semantics of Destructive Lisp*. Ph.D. dissertation, Stanford University, 1986.

43. Mauny, M. and A. Suarez. Implementing functional languages in the Categorical Abstract Machine. In *Proc. 1986 Conference on Lisp and Functional Programming*, 1986, 266-278.

44. Mazurkiewicz, A.W. Proving algorithms by tail functions. In *Inf. Control* **18**, 1971, 220–226.

45. McCarthy, J. A basis for a mathematical theory of computations. In *Computer Programming and Formal Systems*, edited by Braffort and Hershberg. North-Holland, Amsterdam, 1963, 33–70.

46. Morris, J.H. *Lambda-Calculus Models of Programming Languages*. Ph.D. dissertation, MIT, 1968.

47. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

48. QUINE, W.V. *Word and Logic*. MIT Press, Cambridge, Mass., 1960.

49. REES J. AND W. CLINGER (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

50. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.

51. REYNOLDS, J.C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM* **13**(5), 1970, 308–319.

52. ROSSER, J.B. Highlights of the lambda calculus. *Ann. Hist. Comp.* **1**(4), 1984, 337–349.

53. SCHMIDT, D.A. An implementation from a direct semantics definition. In *Proc. Workshop on Programs as Data Objects*, edited by H. Ganzinger and N.D. Jones. Lecture Notes in Computer Science 217. Springer-Verlag, Heidelberg, 1986, 222-235.

54. SCHMIDT, D.A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, Mass., 1986.

55. SCHWARTZ, J.T., R.B.K. DEWAR, E. DUBINSKY, AND E. SCHONBERG. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1986.

56. SETHI R. Circular expressions: elimination of static environments. *Sci. Comput. Program.* **1**, 1982, 203–222.

57. SMITH, B.C. Reflection and Semantics in Lisp. In *Proc. 11th ACM Symposium on Principles or Programming Languages*, 1984, 23–35.

58. SMULLYAN, R.A. *First-Order Logic.* Springer-Verlag, Heidelberg, 1968.

59. STEEL, T.B. (Ed.). *Formal Language Description Languages for Computer Programming.* Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1966.

60. STEELE, G. *Common Lisp—The Language.* Digital Press, 1984.

61. STEELE, G.L. Lambda: The ultimate declarative. Memo 379, MIT AI Lab, 1976.

62. STEELE, G.L. Macaroni is better than spaghetti. In *Proc. Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices* **12**(8), 1978, 60–66.

63. STEELE, G.L. RABBIT: A compiler for SCHEME. Memo 474, MIT AI Lab, 1978.

64. STEELE, G.L. AND G.J. SUSSMAN. Lambda: The ultimate imperative. Memo 353, MIT AI Lab, 1976.

65. STEELE, G.L. AND G.J. SUSSMAN. The art of the interpreter or, the modularity complex. Memo 453, MIT AI Lab, 1978.

66. STOY, J.E. *Denotational Semantics: The Scott-Stratchey Approach to Programming Languages.* MIT Press, Cambridge, Mass., 1981.

67. STRACHEY, C. AND C.P. WADSWORTH. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

68. SUSSMAN G.J. AND G. STEELE. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

69. TALCOTT, C. Rum: An intensional theory function and control abstractions.

In *Proc. Workshop on Foundations of Logic and Functional Programming*, 1986. to appear.

70. TALCOTT, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* Ph.D. dissertation, Stanford University, 1985.

71. TURNER, D.A. A new implementation technique for applicative languages. *Softw. Pract. Exper.* **9**, 1979, 31–49.

72. WAND, M. Continuation-based program transformation strategies. *J. ACM* **27**(1), 1980, 164–180.

73. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* 4(3), 1982, 496–517.

74. WAND, M. AND D.P. FRIEDMAN. Compiling lambda-expressions using continuations and factorizations. *Journal of Computer Languages* (Pergamon Press) **3**, 1978, 241–263.

75. WAND, M. AND D.P. FRIEDMAN. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, 1986, 298–307.

Vita

Matthias Felleisen earned a Master of Science degree in Computer Science from the University of Arizona, Tucson, in 1981 and a Diplom Wirtschaftsingenieur degree from the Universität Karlsruhe, W. Germany, in 1983. From 1981 to 1983 he was a programmer for a local software company in Karlsruhe. During this first period of his studies, Matthias Felleisen was the recipient of a Konrad-Adenauer- and a Fulbright Fellowship.

Since 1984 he has been an Associate Instructor and Research Assistant at Indiana University, Bloomington. In the summer of 1985, he visited the Software Technology Program at the MCC in Austin, Texas. For the last year of his doctoral studies, Matthias Felleisen received an IBM Graduate Fellowship. He has been a member of the Association for Computing Machinery since 1981.