A Syntactic Theory of Sequential State

by

Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251

and

Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 230

A Syntactic Theory of Sequential State

by

Matthias Felleisen and Daniel P. Friedman

Revised: January, 1989

This work was supported in part by an IBM Fellowship Award to Matthias Felleisen, and by the National Science Foundation grant number and CCR 87-02117.

To appear in *Theoretical Computer Science.*

A Syntactic Theory of Sequential State

by

Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251

and

Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 230

A Syntactic Theory of Sequential State

by

Matthias Felleisen and Daniel P. Friedman

Revised, January, 1989

This work was supported in part by an IBM Fellowship Award to Matthias Felleisen, and by the National Science Foundation grant number and DCR 87-03917.

To appear in Theoretical Computer Science

A Syntactic Theory of Sequential State

Matthias Felleisen*
Department of Computer Science, Rice University, Houston, TX 77251-1892

Daniel P. Friedman†
Computer Science Department, Indiana University, Bloomington, IN 47405

January 2, 1989

Abstract

The assignment statement is a ubiquitous building block of programming languages. In functionally-oriented programming languages, the assignment is *the* facility for modeling and expressing state changes. Given that functional languages are directly associated with the equational λ-calculus-theory, it is natural to wonder whether this syntactic proof system is extensible to imperative variants of functional languages including state variables and side-effects. In this paper, we show that such an extension exists, and that it satisfies variants of the conventional consistency and standardization theorems. With a series of examples, we also demonstrate the system's capabilities for reasoning about imperative-functional programs and illustrate some of its advantages over alternative models.

1 Assignments in Functional Languages

The assignment statement is a ubiquitous building block of programming languages. In imperative Algol-style languages, it generalizes and incorporates the capabilities of store, load, and move instructions from assembly languages. For modern, mathematically oriented programming languages, the assignment constitutes an abstraction of a different kind: it provides the means for modeling state variables and signaling state changes. The abstraction hides recurring programming patterns that would otherwise render programs unreadable.

For an illustration of our claim, we briefly compare the representation of an object in two different function-oriented languages: one with, the other without assignment. Consider the following program fragment:

```
let ...
    TransManager  =  let TransCounter = 0 in
                     function (TransType)
```

*Partly supported by NSF and an IBM Fellowship.
†Partly supported by NSF.

```
                    if TransType is counter
                      then TransCounter
                    else
                      begin TransCounter := TransCounter + 1;
                        BODY
                      end
                ...
... TransManager(t1)  ...
```

The variable *TransManager* is initialized to a function whose scope contains the assignable variable *TransCounter* with initial value 0. On every subsequent call to *TransManager* for performing a proper transaction, the routine increases *TransCounter* by 1 with an assignment. The number of past transactions can be checked with a special transaction.

In a language without assignments the above program fragment would have to be rewritten into something like:

```
let ...
  TransManager  =  function (TransType, TransCounter)
                     if TransType is counter
                       then TransCounter
                     else
                       ⟨BODY, TransCounter + 1⟩
                ...
... let ⟨result, NewCounter⟩ = TransManager(t1, 0) in ...
```

The manager routine now takes the current value of the counter as an additional argument. Upon completion of the transaction, *TransManager* returns a pair whose first component is the proper result of the transaction and whose second component is the increased counter value. All calls to *TransManager* require a modification to the end that the current value of *TransCounter* is passed as an extra argument, and that the pair of results is disassembled in the desired way.

The functional programming style exhibited in this example has two major problems. First, from the software engineering perspective, there is a loss of modularity and security. In the first program, the variable *TransCounter* is hidden in the scope of the manager routine. Its value is only accessible through appropriate calls to *TransManager* and is only changed upon calls to the routine. The functional version, however, has no such protection mechanism. The state of the counter is exposed, visible and modifiable throughout the program. The responsibility for maintaining the correct value is now distributed over the *entire* program.

Second, from the programming language perspective, the important drawback of the functional programming style is the accumulation of recurring programming patterns. Parameter lists as well as function calls must be changed uniformly: functions need additional parameters in order to account for the state of the world when called, function calls need

additional arguments to pass along the current values of the state variables and receive extended results to account for the altered state variables. The repetitive occurrences of such programming patterns clearly call for an abstraction that hides the details. The assignment is this required abstraction.

Unfortunately, the introduction of assignment into a functional language not only solves but also creates a problem. The advantage of a functional programming language is that it automatically comes with a powerful, symbolic reasoning system: the λ-calculus or a variant thereof. This connection provides an abstract understanding of programs and is the basis for algebra-like program evaluations, transformations, and verifications. The problem with imperative extensions of functional languages is that the λ-calculus no longer corresponds to the extended language, and that it is consequently impossible to reason about imperative-functional programs with the calculus.

In a companion paper [6], we solved the same problem for a functional language with imperative control operators. More specifically, we derived a calculus-like system from the abstract machine semantics and analyzed to what extent the system satisfies Plotkin's criteria of a programming language calculus. In this paper, a revised and expanded version of an earlier report [5], we apply a similar method to a functional language with assignment and develop a calculus-like equational theory for it.

In the second section, we briefly outline the general aspects of Plotkin's work on the correspondence of functional programming languages and calculi. Next, we define a λ-calculus-based programming language with an expression-oriented assignment construct. The major results are collected in Sections 4 through 7. First, we derive a rewriting semantics for the extended language that does not rely on the store. Second, we design the extended λ-calculus from the rewriting system. Third, we prove variants of the Church-Rosser Theorem, the Curry-Feys Standardization Theorem, and the Plotkin Correspondence Theorems. Section 8 is devoted to examples. The last section summarizes our development and addresses some open problems.

2 Programming Language Calculi

The basis of our development is Plotkin's work on the correspondence of programming languages and calculi [21]. The motivation for this work was the well-known observation about the mismatch between the call-by-value evaluation rule and the original β-axiom for the λ-calculus; the result was the λ-value-calculus for reasoning about call-by-value programming languages. The central message of Plotkin's work, however, is much broader. It says that calculi and logics must be constructed for or fine tuned to given programming languages. Indeed, the relationship between languages and calculi is determined by a pair of general semantic criteria, which should be applicable to a wide class of languages and equational theories.

For a programmer, a programming language is a set of syntactic phrases and an operational semantics. There are two distinguished sub-categories of phrases: *programs* and *values*. The set of *observable values* is a collection of values on which equality is decidable. The operational semantics is a partial function from programs to values. If the function is undefined on a program, we say the program diverges; otherwise, the program converges to a value. A calculus is a congruence relation on a set of syntactic phrases. It equates pro-

grams and program pieces. In addition, a calculus usually satisfies a syntactic consistency property.

A calculus for a specific programming language is supposed to capture behavioral equality among syntactic phrases. Hence, the calculus should have the same syntactic domain as the language, but it must also account for the language semantics. For this semantic comparison of languages and calculi, it is crucial that each can be interpreted as a form of the other.

The semantics-based congruence relation is called *operational equivalence* [19, 21]. Operational equivalence expresses the idea of two terms being indistinguishable by the machine and hence by the programmer. More specifically, two terms are operationally equivalent if one can replace the other in any program without changing the result: both variants of the program either diverge or converge, and if one converges to an observable value, the other converges to the same observable value. The calculus-based function from programs to values is the *standard reduction function.* It associates a program with a unique value from the set of all the values that the calculus equates with the program. The standard reduction function is a specialization of the Curry-Feys standardization procedure for calculus equations. Given (certain) equations in a calculus, the standardization procedure shows how to construct standard derivations for these equations. Since these derivations are unique, they map programs to unique values.

Equipped with these basic definitions, we can now state Plotkin's pair of correspondence criteria for languages and calculi. First, the calculus-based semantic function must be the same as the original language semantics. After all, we want to use the calculus to evaluate programs in an algebra-like manner. Second, equivalence in the calculus must imply semantic equivalence so that we can reason about the behavioral equality of programs. Since operational equivalence equates all non-terminating programs, we cannot expect that a calculus captures all operational equalities.

Plotkin's criteria work out well in the purely functional framework. For imperative extensions, however, we must generalize some of the notions. In the case of imperative control operators [6], there are two different equivalence relations: one for evaluating programs, one for proving equivalences. Although this complicates the calculus, it is still an improvement over the current practice of reasoning with the machine. Furthermore, as we shall see below, the calculus may also require a larger, syntactically richer language since a programming language may not necessarily be able to express all intermediate evaluation states. In a sense, these complications are the price that must be paid for adding imperative constructs.

3 Λ with assignment abstractions

The practical starting point of our work was the programming language Scheme [25]. For our purposes, Scheme is essentially an applicative-order language with first-class procedures, an assignment statement, and some primitive, algebraic constants and functions. The applicative-order character of the language imposes the important restrictions on the language evaluator that procedure arguments are evaluated only once before the evaluation of the instantiated body, and that assignments to a procedure parameter are only visible in the particular instantiation of the procedure body. These restrictions are reflected in

our semantics. They seem to simplify some of the development, but we believe that similar results can still be established for different settings.

We model the core of our programming language with Λ, the term set of the λK-$\beta\delta$-calculus [2] and the λ_v-calculus [21]. It contains basic and functional constants, variables, λ-abstractions, and term juxtapositions. Constants represent built-in primitive data types and their functions, *e.g.*, natural numbers (0, 1, 2, ...) and the successor function (succ). Variables are placeholders or parameters. We interpret λ-abstractions as parameterized call-by-value procedures. Juxtapositions denote procedure applications.

Λ is an entirely expression-oriented language. Adding an assignment statement to such a language confronts the language designer with a problem: if added naïvely, the language is suddenly divided into two major syntactic categories, namely commands and expressions. We avoid this by using a new kind of expression: the *σ-capability*. Its syntax is $(\sigma x.M)$ where x is a variable and M is an expression. Though it resembles a λ-abstraction, a σ-capability is *not* a binding construct—its variable is usually bound by some enclosing λ-abstraction. Instead, a σ-capability is a procedural object that abstracts the right to assign a variable a new value. When applied to a value, it assigns the σ-variable that value and evaluates the σ-body to yield the result of the application.

The alert reader may have noticed that we used "variable" in two different senses. First, there is the usual notion of a variable as a placeholder for some arbitrary, but fixed value. Second, there is the concept of an *assignable* variable, which ultimately stands for some value, but it may stand for different values at different times. This second notion is a generalization of the first. Since both kind of variables play the role of a parameter for a procedure body, we treat them as one set of variables. But, to distinguish their different nature, we subscript non-assignable and assignable variables with λ and σ, respectively.

Definition 3.1 (Λ_σ) The improper symbols are λ, (,), ., and σ. *Vars* is a countable set of variables; $x, \ldots$ ranges overs *Vars*. The set of variables is also the disjoint union of non-assignable variables, ($x_\lambda \in Var_\lambda$), and assignable variables, ($x_\sigma \in Var_\sigma$). *Const* ($c \in Const$) is the set of algebraic constants, a disjoint union of functional ($f \in FuncConst$) and basic constants ($b \in BasicConst$). The term set Λ_σ is defined inductively as:

$$M ::= c \mid x_\lambda \mid (\lambda x.M) \mid (MN) \mid x_\sigma \mid (\sigma x_\sigma.M).$$

Λ stands for Λ_σ restricted to constants, variables, applications, and abstractions.

When we write programs in Λ_σ, we follow the usual λ-calculus conventions [2]. We omit parentheses from applications and abstractions if they can be reconstructed. Applications associate to the left, λ-abstractions extend to the end of the program or some other enclosing parenthesis. In addition, we drop subscripts from variables in unambiguous contexts.

The sets of free and bound variables of a term M, $FV(M)$ and $BV(M)$, are defined as usual: the only binding construct in the language is λ-abstraction. Terms with no free variables are called *closed terms*. In order to avoid the issue of free and bound variable interference, we adopt Barendregt's [2] convention of *identifying* ($\equiv_\alpha$ or just $\equiv$) *terms that are equal modulo some renaming of bound variables* and the hygiene condition which says that *free variables are assumed to be distinct from bound ones in the various terms of*

mathematical discussions. Substitution is extended in the natural way and we use the notation $M[x := N]$ to denote the result of substituting all free variables x in M by N.

The operational semantics for Λ_σ is a partial function from Λ_σ-programs to values. Programs are closed terms, which avoids the need for an external interpretation function for free variables. Values are basic constants or procedural objects, *i.e.*, functional constants, abstractions, and capabilities. The former represent some final answer, the latter only make sense when applied to arguments. Once bound to a value, non-assignable variables always represent the same value. Hence, we include these variables in the set of values. Assignable variables, on the other hand, only indirectly represent values and do not count as values. To abstract from the concrete set of constants, we assume that the set of constants is equipped with a partial interpretation function

$$\delta\text{: } \mathit{FuncConst} \times \mathit{BasicConst} \longrightarrow \mathit{Closed\text{-}\Lambda\text{-}Values}$$

that specifies the semantics of constants.

Definition 3.2 (Λ_σ*-programs and -values*) A *program* is a closed term. Constants, non-assignable variables, λ-abstractions, and σ-capabilities are collectively referred to as Λ_σ-*values*: $U, V, W, \ldots$ stand for values.

Convention. (*Unnestable terms*). Once we have decided to use a term M as a complete, independent expression, *e.g.*, a program, we write

$$\&M$$

to indicate that this expression is not to be nested any further. The device is purely notational and has no semantic significance. **End**

We define the operational semantics via an abstract machine model. For the core language Λ, the abstract machine is usually a state transition system that manipulates states with instruction-, environment-, and, possibly, control stack-components [11]. From our previous work [6], we know, however, that this machine can equally well be specified as a contextual rewriting system on just Λ. The rewriting system proceeds by partitioning a program into a β-value- or δ-redex and an evaluation context before each transition step. A redex corresponds to the next instruction, the evaluation context to the rest of the instructions. After the partitioning, the program is transformed according to the redex, and the process repeats until a value is reached.

For Λ_σ, we must extend the rewriting system to cope with assignable variables and σ-capabilities. The usual way to model such facilities is to introduce a set of locations for representing assignable variables and a store for mapping locations to their current value. In a rewriting system, the connection between an assignable variable and its location is established at the time of procedure application. Instead of replacing the parameter by its value, the rewriting system substitutes a location in all positions where the assignable variable occurred and remembers the value of the location in the store. Upon encountering

a location in the control string, the machine derives the current value from the store; an assignment becomes a simple, functional update of the store component.[1]

Putting all of this together, we can now formalize the abstract machine for Λ_σ. The CS-machine manipulates pairs of control strings and stores. A control string is a Λ_σ-expression where all free variables are replaced by locations. Letting l range over the set of locations, we define the set of control strings (M, N, ...) by

$$M ::= c \mid x_\lambda \mid (\lambda x.M) \mid (MN) \mid x_\sigma \mid (\sigma x.M) \mid l \mid (\sigma l.M).$$

The set of *CS-values* comprises constants, non-assignable variables, abstractions, capabilities, and σ-capabilities with locations in the variable position; U and V range over CS-values.

A store is a finite map from a set of locations to a set of values. $Dom(\theta)$ denotes the domain of the store θ; if θ is a store, l a location, and V a value, $\theta[l := V]$ is the same store as θ except at l where it is V. To hide the details of the storage allocation process, we identify isomorphic stores. Two stores θ_1 and θ_2 are isomorphic if there is a bijection between their domains:

$$\varphi: \mathrm{Dom}(\theta_1) \longrightarrow \mathrm{Dom}(\theta_2),$$

and if for all $l \in \mathrm{Dom}(\theta_1)$,

$$\overline{\varphi}(\theta_1(l)) = \theta_2(\varphi(l)),$$

where $\overline{\varphi}$ is the natural extension of φ to terms. In other words, one store is equivalent to another if the contents of the first store is relocated in different locations in the second. Moreover, if θ_1 and θ_2 are isomorphic, l_1 and l_2 locations outside of their domains, and V a value without reference to either l_1 or l_2, $\theta_1[l_1 := V]$ is isomorphic to $\theta_2[l_2 := V]$. As a consequence, it is arbitrary which location we choose when extending a store.

For a transition step, a control string is partitioned into a CS-evaluation context and a redex. CS-redexes are terms of the form (fb), $(\lambda x_\lambda.M)V$, $(\lambda x_\sigma.M)V$, l, or $(\sigma l.M)V$. An evaluation context is a control string with a hole. The hole approximately corresponds to a program counter and the instructions in the hole are to be executed next. Since applications are the only syntactic construction that requires evaluation, the hole can only be nested inside of applications. Furthermore, in the presence of side-effects, we must have a fixed evaluation order to get deterministic results. Hence, we choose that all applications to the *left* of a hole must have been reduced to values before the instructions in the hole are considered. This determines an evaluation order from left to right. We let [ ] represent a hole and let $E[\ ]$ range over evaluation contexts:

$$E[\ ] ::= [\ ] \mid (VE[\ ]) \mid (E[\ ]M).$$

If $E[\ ]$ is an evaluation context, the term $E[M]$ is the result of filling the hole with the term M. It is easy to verify that the partitioning of a program into a redex and a context is unique.

[1] For the original development of the calculus [5], we *derived* the machine from the CESK-rewriting system, a variant of Landin's SECD-machine [11], which is closer to a realistic interpreter. At the same time, Mason [12] published a CS-like store-program rewriting system for Lisp-like languages and used it for proving properties of imperative programs.

A machine evaluation for a program M starts in an initial state $\langle M, \emptyset \rangle$; final states are pairs of CS-values and stores: $\langle V, \theta \rangle$. There are five clauses for the one-step CS-transition function, one for each kind of redex:

$$\langle E[fa], \theta \rangle \xrightarrow{CS} \langle E[\delta(f,a)], \theta \rangle \tag{CS1}$$

$$\langle E[(\lambda x_\lambda.M)V], \theta \rangle \xrightarrow{CS} \langle E[M[x_\lambda := V]], \theta \rangle \tag{CS2}$$

$$\langle E[(\lambda x_\sigma.M)V], \theta \rangle \xrightarrow{CS} \langle E[M[x_\sigma := l]], \theta[l := V] \rangle \quad \text{where } l \notin Dom(\theta) \tag{CS3}$$

$$\langle E[l], \theta \rangle \xrightarrow{CS} \langle E[\theta(l)], \theta \rangle \tag{CS4}$$

$$\langle E[(\sigma l.M)V], \theta \rangle \xrightarrow{CS} \langle E[M], \theta[l := V] \rangle \tag{CS5}$$

The first two clauses are the transition rules for the subset Λ. Naturally, they do not utilize the store. The third clause specifies the behavior of an application whose procedure abstracts over an assignable variable. The machine picks an arbitrary location not in the domain of the current store and replaces all occurrences of the parameter with the location. The store is updated to include a binding for the new location to the argument value. Picking a fresh location upon every application of a procedure guarantees that assignments to a parameter variable are invisible outside of the instantiation of the procedure body. This is the call-by-value rule for languages with assignments. The fourth rule specifies the machine behavior for location-redexes: the location is simply dereferenced in the current store. Finally, an application of a σ-capability proceeds as described above: the location's value is changed to the new value and the evaluation proceeds with the σ-body.

In order to clarify the abstract explanations about the CS-machine, we trace the evaluation of the program

$$\&(\lambda p.(\lambda d.(p0))(p0))(\lambda x.(\sigma x.x)(\lambda y.x)).$$

In a syntactically more elaborate language, this would be written as

$$\textbf{let } p = (\lambda x.\textbf{begin } x := (\lambda y.x) \textbf{ result } x) \textbf{ in}$$
$$\textbf{begin } p(0);\ p(0) \textbf{ end},$$

where **begin** A; B **end** stands for $(\lambda d.B)A$ with d not free in B. The evaluation trace of this program (with the initial store θ_0) is:

$$\langle (\lambda p.(\lambda d.(p0))(p0))(\lambda x.(\sigma x.x)(\lambda y.x)), \theta_0 \rangle \tag{1}$$

$$\xrightarrow{CS} \langle (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))((\lambda x.(\sigma x.x)(\lambda y.x))0), \theta_0 \rangle \tag{2}$$

$$\xrightarrow{CS} \langle (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))((\sigma l_1.l_1)(\lambda y.l_1)), \theta_0[l_1 := 0] \rangle \tag{3}$$

$$\xrightarrow{CS} \langle (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))l_1, \theta_0[l_1 := (\lambda y.l_1)] \rangle \tag{4}$$

$$\xrightarrow{CS} \langle (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))(\lambda y.l_1), \theta_0[l_1 := (\lambda y.l_1)] \rangle \tag{5}$$

$$\xrightarrow{CS} \langle ((\lambda x.(\sigma x.x)(\lambda y.x))0), \theta_0[l_1 := (\lambda y.l_1)] \rangle \tag{6}$$

$$\xrightarrow{CS} \langle ((\sigma l_2.l_2)(\lambda y.l_2)), \theta_0[l_1 := (\lambda y.l_1)][l_2 := 0] \rangle \tag{7}$$

$$\xrightarrow{CS} \langle l_2, \theta_0[l_1 := (\lambda y.l_1)][l_2 := (\lambda y.l_2)] \rangle \tag{8}$$

$$\xrightarrow{CS} \langle (\lambda y.l_2), \theta_0[l_1 := (\lambda y.l_1)][l_2 := (\lambda y.l_2)] \rangle \tag{9}$$

The trace illustrates several points about the machine evaluations. First, the two distinct calls to the procedure p cannot affect each other. As mentioned above, every call to p allocates a fresh cell for the parameter x and thus the assignment in (3) during the first invocation of p has no impact at the second invocation. Second, the result state represents a circular structure, *i.e.*, the final CS-value $(\lambda y.l_2)$ refers to the location l_2, which in turn refers to the very same value. Third, the location l_1 is irrelevant for the evaluation from transition (6) on through the end. More precisely, the location has become unreachable through a static trace from the control string component and can therefore no longer affect the evaluation. Such locations are usually called *garbage*; the others are *relevant locations.* Garbage locations cause problems in practical implementations where they require garbage collectors as well as in program equivalence proofs as we shall see below.

Equipped with the CS-machine, we are now ready to define the operational semantics of Λ_σ. We call the function $eval_\sigma$. It abstracts the details of CS-machine evaluations. For pure Λ-programs and values without reference to the final store, this is straightforward. The result is simply the value-component of the final state. The extension to the full language is more difficult due to the illustrated potential for self-referential final states. Therefore, we only provide a partial definition and parameterize $eval_\sigma$ over a yet-to-be-determined unload function U.

Definition 3.3 ($eval_{\sigma,\mathsf{U}}$, $eval_v$) The partial function $eval_{\sigma,\mathsf{U}}$ maps Λ_σ-programs to final answers iff there is a (possibly empty) CS-reduction sequence from the corresponding initial machine state to some final state:

$$eval_{\sigma,\mathsf{U}}(\&M) = \mathsf{U}(V,\theta) \text{ iff } \langle M,\emptyset\rangle \stackrel{CS}{\longmapsto}{}^* \langle V,\theta\rangle.$$

The partial function $eval_v$ maps Λ-programs to Λ-values iff there is a (possibly empty) CS-reduction sequence from the corresponding initial machine state to some final state:

$$eval_v(\&M) = V \text{ iff } \langle M,\emptyset\rangle \stackrel{CS}{\longmapsto}{}^* \langle V,\emptyset\rangle.$$

The operational semantics is the major source of intuition for a programmer. In order to test ideas about a program, he submits the program to *eval* and observes the result. If the function fails to distinguish between programs, the programs are freely interchangeable. This idea is the basis for the operational equivalence relation. Operational equivalence is omni-present in the work of a programmer. For any kind of program transformation, *e.g.*, for making a program shorter, faster, better looking, *etc.*, the operational semantics of the program must be preserved: the two versions must be operationally equivalent. Because of this, it is desirable that the relation satisfy some key characteristics:

1. that it preserve equality on basic constants since these constitute the ultimate program answers;

2. that it respect the ordinary evaluation process: after all, the program user relies on the machine for running the program;

3. that it be a congruence relation, *i.e.*, that equals can be substituted for equals in all contexts; after all, to be reusable, verifications and transformations of program pieces should be independent of the context.

Beyond this, the operational equivalence relation should be as large as possible so that every desirable equality can be expressed

The formalization of operational equivalence depends on the notion of a context. A context is a Λ_σ-term with exactly one hole in an expression position. More precisely, let $[\ ]$ again be the hole and let $C[\ ]$ range over contexts. Then, contexts are defined by

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

Clearly, contexts are a generalization of evaluation contexts. The term $C[M]$ results from filling $C[\ ]$ with M; if M contains free variables, they may become bound by the fill operation.

Given the notion of a context, we say that terms are operationally equivalent if there is no context that completes both phrases to programs, and that produces observably different results for the two programs on the evaluator.

Definition 3.4 ($\simeq_\sigma$, $\simeq_v$) Two terms $M, N \in \Lambda_\sigma$ are operationally equivalent, $M \simeq_\sigma N$, iff for any arbitrary Λ_σ context $C[\ ]$ such that $C[M]$ and $C[N]$ are programs, $eval_{\sigma,\mathsf{U}}$ 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 definition of $\simeq_\sigma$ is adapted *mutatis mutandis* for the operational equivalence relation on pure Λ. Instead of $eval_{\sigma,\mathsf{U}}$ it uses $eval_v$; all terms and contexts are restricted to Λ. The resulting relation is denoted by $\simeq_v$.

It is well-known that the by-value operational equivalence relation satisfies the above criteria on the sub-language Λ [21]. For the extended relation, we need to prove this fact.

Proposition 3.1 *$\simeq_\sigma$ is the largest consistent equivalence relation on Λ_σ that respects equality on the set of basic constants and that is also*

- *compatible: $M \simeq_\sigma N$ implies $C[M] \simeq_\sigma C[N]$ for all $C[\ ]$, and*
- *evaluating: $M \simeq_\sigma N$ 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 [19] relies on clever techniques by Böhm [3, 2:254–260] for the separation of normal forms. The proof of this proposition is similar, but much simpler, because we can rely on the separability of observable values. Thus, assume the opposite. Then there is a relation $\simeq_R$ that satisfies the antecedent, and that relates two terms M and N that are operationally distinct: $M \not\simeq_\sigma N$. Consequently, there must be a context $C[\ ]$ such that $C[M]$ and $C[N]$ are programs, the value of both exists, but the values are distinct basic constants. But then $C[M] \simeq_R C[N]$ even though $eval_\sigma(C[M]) = b_1 \neq b_2 = eval_\sigma(C[N])$. This implies that $\simeq_R$ is *not* a conservative extension of equality on basic constants contrary to the assumption about its character. This generates the desired contradiction and concludes the proof. □

On the negative side, we have two aspects. The first is rather general and concerns the relationship of operational equivalence on Λ versus Λ_σ. Since reasoning with operational equivalence on functional languages is well-established and is at the heart of proof systems [17, 20, 21], we would hope that $\simeq_\sigma$ is a conservative extension of $\simeq_v$. Unfortunately, but not surprisingly, this is wrong.

Proposition 3.2 *$\simeq_\sigma$ is not a conservative extension of $\simeq_v$.*

Proof. In Λ, we can prove that

$$\lambda p.((\lambda d.p0)(p0)) \simeq_v \lambda p.(p0),$$

or, with syntactic sugar,

$$\lambda p.(\textbf{begin } p(0);\ p(0) \textbf{ end}) \simeq_v \lambda p.p(0).$$

In essence, pure Λ-procedures cannot affect the store, and the result of their application cannot depend on the calling history. But this no longer holds in Λ_σ. When evaluated, the following expression produces a procedure that first returns some value as a result, and that diverges on subsequent calls:

$$\textbf{let } x = (\lambda z.z) \textbf{ in } (\lambda y.x(\sigma x.x)(\lambda d.(\lambda x.xx)(\lambda x.xx))).$$

It follows that the context

$$E[\ ] \equiv [\ ](\textbf{let } x = (\lambda z.z) \textbf{ in } (\lambda y.x(\sigma x.x)(\lambda d.(\lambda x.xx)(\lambda x.xx)))$$

can distinguish the above two terms with respect to $\simeq_\sigma$. □

The second negative aspect concerns the use of the CS-transition function for proofs of operational equivalences. Because of the garbage problem, it is rather cumbersome to prove some simple, intuitive operational equivalences. Consider the program

$$P \stackrel{df}{\equiv} ((\lambda x.(\sigma x.M)1)0),$$

where x is not free in M. This phrase is clearly operationally equivalent to M. In order to prove this, we compare the CS-evaluation of P and M in an arbitrary evaluation context $E[\ ]$ and an arbitrary store θ. After a few transition steps, the evaluation of P reaches the state

$$\langle M, \theta[l := 1]\rangle,$$

where l is the location that was allocated for x. If we can now prove that l is indeed garbage, we know that P and M are operationally equivalent. In trivial cases, such an auxiliary proof may be a reasonable task, but for larger, more complicated programs, it is infeasible to require both proofs.

From the description of the negative aspects of our current system, our goal should be clear. We intend to derive a programming language calculus for Λ_σ that is a conservative extension of the λ-value-calculus, and that consequently allows the re-use of many program proofs from the functional fragment. In addition, the calculus should avoid the garbage problem so that the programmer need not supply auxiliary proofs about the garbageness of locations. In the next section, we tackle the second problem by reconsidering the CS-rewriting system. From there, we proceed to develop our calculus.

4 Replacing the Store by Program Sharing Relations

The strategy for developing a semantics for Λ_σ without the garbage problem is straightforward and based on our previous experience on developing calculi: we incorporate the store into the program component of the machine. The role of the store in the CS-machine is characterized by the transition rules (CS3) through (CS5), which extend, use, and modify the store. The crucial rule is (CS3). 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 store. It follows that 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. Abstractions like $\lambda x.x$ and $\lambda y.y$ are considered the same. 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 relation 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 abstract 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 sharing relation* for term positions,[2] 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 new kind of syntactic phrase for expressing this dynamic relationship.

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*.[3] With respect to the transition rules, we would like to replace (CS3)

$$\langle E[(\lambda x_\sigma.M)V], \theta\rangle \xrightarrow{CS} \langle E[M[x_\sigma := l]], \theta[l := V]\rangle \quad \text{where } l \notin Dom(\theta)$$

by

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

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 already sits in the right position. The effect of an assignment is more complicated. 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

[2] Modeling the store as a sharing relation was already mentioned by Landin [10], but, apparently, he never formalized the idea. Indeed, as Stoy seems to stipulate [24:253,284], the store function in denotational semantics represents Landin's sharing relation, and, furthermore, a direct realization of the sharing relation idea for the SECD-machine would have led to a true store component.

[3] Both Aït-Kaci and Nasr [1] and Sethi [23] have used similar labeling schemes for terms in related contexts. The latter modeled the code of programs with gotos and labels through circular terms with term labels; the former used term labels for a generalized version of records and the unification of such records.

by a labeled value; all other related variable positions carry the same label. When it is time to perform the above σ-application, all these occurrences of l-labeled values U 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. The assignment transition is now definable as

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

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 the rest of the program $E[M]$ but also in the assigned value V, the equivalence-positions in V are not affected by the labeled-value substitution. For an illustration, consider the program $\&(\lambda x.(\sigma x.x)(\lambda y.x))0$. Its evaluation on the CS-machine yields the final state $\langle \lambda y.l, \{l \mapsto \lambda y.l\}\rangle$. A term evaluation according to the above rules proceeds as follows:

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

The last term should represent this circular structure and in some sense it does: an interpretation of the label l requires that the position occupied by 0^l is in the same sharing equivalence class as the entire expression $(\lambda y.0^l)^l$. However, a lookup that simply strips off the label produces a non-circular object, namely, $(\lambda y.0^l)$. For a correct simulation of the CS-machine within a rewriting system, we need to alter the lookup rule so that it unwinds a circular term another time:

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

For the above example, the modified rewriting system produces $(\lambda y.(\lambda y.0^l)^l)$, which is the correct final answer.

An intuitive argument for the correctness of the new transition rules is based on 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.

A clear advantage of the C-rewriting system over the CS-machine is its simplified treatment of stored values. In the CS-machine, a store location and its contents can only become unreachable from the control string through the use of some vacuous abstraction or vacuous assignment. Since the corresponding actions are now realized by substitutions as opposed to store modifications, the substituted terms including the contained locations simply disappear from the control string. Hence, relevant locations are always directly present in the control string, garbage locations are eliminated immediately.

Given the informal descriptions and correctness arguments, we proceed to formalize the final machine semantics. The machine is a control string rewriting system. Its only state components are expressions in the language Λ_S, which is a proper extension of Λ_σ with labeled values and σ-capabilities with labeled bullets in the variable position.

Definition 4.1 (Λ_S *and labeled value substitution*) Let $\bullet$ (bullet) be a new improper symbol, *Labels* an infinite set of label identifier, and l a meta-variable for *Labels*. Then, Λ_S is

the set of terms

$$\begin{array}{rcl} M & ::= & V \mid MN \mid x_\sigma \mid \bullet^l \mid V^l \\ V & ::= & c \mid x_\lambda \mid (\lambda x.M) \mid (\sigma x_\sigma.M) \mid (\sigma\, \bullet^l.M) \end{array}$$

with the following context-sensitive restrictions:

1. an l-labeled value may only contain l-labeled bullets ($\bullet^l$), and an l-labeled bullet may only occur as a subterm of an l-labeled value;
2. an abstraction $\lambda x.M$ must not contain a labeled value with a free variable x;
3. in an application MN, every l-labeled value in M must be identical to every l-labeled value in N after replacing all labeled values in these terms by the respective labels.

The meta-variables M, N, L range over Λ_S-terms, V, U, and W over Λ_S-values. Subsequently, we also use X to denote an assignable variable, or a labeled bullet, or a labeled value.

The *set of labels in a term* M is denoted by $Lab(M)$.

The definitions of substitution, free and bound variables, *contexts* and *evaluation contexts* are adopted mutatis mutandis, *e.g.*, the result of

$$(\sigma x_\sigma.M)[x_\sigma := V^l] \text{ is } \sigma\, \bullet^l.M[x_\sigma := V^l]),$$

and values in evaluation contexts may be the new form of instantiated σ-capabilities.

Programs in Λ_S are closed terms, possibly containing labels.

Finally, *labeled-value substitution* over Λ_S is defined as

$$\begin{array}{rcl} U^k[\bullet^l := L^l] & = & \begin{cases} L^l & \text{if } l = k \\ (U[\bullet^l := L[\bullet^k := \bullet^k]^l])^k & \text{if } l \neq k \end{cases} \\ c[\bullet^l := L^l] & = & c \\ x[\bullet^l := L^l] & = & x \\ (\lambda x.M)[\bullet^l := L^l] & = & \lambda x.M[\bullet^l := L^l] \\ (MN)[\bullet^l := L^l] & = & M[\bullet^l := L^l]N[\bullet^l := L^l] \\ (\sigma x.M)[\bullet^l := L^l] & = & \sigma x.M[\bullet^l := L^l] \\ (\sigma\, \bullet^k.M)[\bullet^l := L^l] & = & \sigma\, \bullet^k.M[\bullet^l := L^l]. \end{array}$$

Note: $L[\bullet^k := \bullet^k]$ denotes the term L with all occurrences of k-labeled values replaced by a k-labeled bullet.

The definition of the extended term language takes into account that occurrences of labeled values are only useful in certain positions. More specifically, sub-terms of a labeled value with the same label are irrelevant for the evaluation, and, similarly, for a σ-capability, it is only important which sharing relation it must affect, not what the current shared value is. We therefore use the auxiliary term $\bullet^l$ to eliminate such useless occurrences.

The labels of a Λ_S-program inherit an important property from CS-stores. Just like the precise identity of a location was actually irrelevant for a store, so is the identity of a label irrelevant for a program. As long as the same sharing relationships are identified in a program, the names of the labels do not affect the evaluation. Indeed, identifying label-equivalent programs like this corresponds to a "dynamic" extension of α-equivalence, which is in agreement with our motivational remarks at the outset of this section. Consequently, we adopt a variant of Barendregt's bound variable convention for labels in programs.

Convention. (*Label convention*)

label-equivalence convention: *Programs* that are equivalent modulo the names of labels are identified; we denote this relation with $\equiv_{lab}$;

label hygiene convention: Different label-names always denote different sharing relationships.

Warning. *Label-equivalence resembles an ordinary term relation, but it is a relation on* programs. **End**

To avoid some trivial complications with the context-sensitive restrictions on terms, we adopt a further convention on the labeling of values. As indicated above during the informal motivation, the new transition rules often require that an unlabeled value becomes a labeled value. If done naïvely, this can easily conflict with the restriction that a labeled value must not contain a labeled value with the same label.

Convention. If V is a value and l a label, then V^l denotes the labeled value $(V[\bullet^l := \bullet^l])^l$. That is, we assume that all l-labeled sub-values in the newly labeled value are automatically replaced with labeled bullets. **End**

Finally, we are ready to define the operational semantics of Λ_S. As usual, we base this function on an abstract machine: the C-machine. Its function uses the partitioning of control strings into contexts and redexes that is known from the CS-machine.

Definition 4.2 (*$eval_S$, the C-transition function*)

$$E[fa] \xmapsto{C} E[\delta(f,a)] \tag{C1}$$

$$E[(\lambda x_\lambda.M)V] \xmapsto{C} E[M[x_\lambda := V]] \tag{C2}$$

$$E[(\lambda x_\sigma.M)V] \xmapsto{C} E[M[x_\sigma := V^l]] \quad \text{where } l \notin Lab(E[(\lambda x_\sigma.M)V]) \tag{C3}$$

$$E[V^l] \xmapsto{C} E[V[\bullet^l := V^l]] \tag{C4}$$

$$E[(\sigma\, \bullet^l .M)V] \xmapsto{C} E[M][\bullet^l := V^l] \tag{C5}$$

The semantics of Λ_σ is captured in the partial function $eval_S$ from Λ_S-programs to Λ_S-values:

$$eval_S(\&M) = V \text{ iff } M \xmapsto{C}{}^{*} V.$$

To strengthen the intuition into the C-evaluation process and to provide a basis for comparison with the CS-machine evaluation, we retrace the rewriting steps from Section 3. Recall the sample program:

$$\&(\lambda p.(\lambda d.(p0))(p0))(\lambda x.(\sigma x.x)(\lambda y.x)).$$

On the C-machine, the evaluation of this program proceeds as follows:

$$\begin{aligned}
& (\lambda p.(\lambda d.(p0))(p0))(\lambda x.(\sigma x.x)(\lambda y.x)) && (1)\\
\xrightarrow{C}\ & (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))((\lambda x.(\sigma x.x)(\lambda y.x))0) && (2)\\
\xrightarrow{C}\ & (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))((\sigma 0^{l_1}.0^{l_1})(\lambda y.0^{l_1})) && (3)\\
\xrightarrow{C}\ & (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))(\lambda y.\bullet^{l_1})^{l_1} && (4)\\
\xrightarrow{C}\ & (\lambda d.((\lambda x.(\sigma x.x)(\lambda y.x))0))(\lambda y.(\lambda y.\bullet^{l_1})^{l_1}) && (5)\\
\xrightarrow{C}\ & ((\lambda x.(\sigma x.x)(\lambda y.x))0) && (6)\\
\xrightarrow{C}\ & ((\sigma 0^{l_2}.0^{l_2})(\lambda y.0^{l_2})) && (7)\\
\xrightarrow{C}\ & (\lambda y.\bullet^{l_2})^{l_2} && (8)\\
\xrightarrow{C}\ & (\lambda y.(\lambda y.\bullet^{l_2})^{l_2}) && (9)
\end{aligned}$$

The most important difference between the C-machine and the CS-machine becomes visible in the transition from line 5 to line 6: the label l_1 completely disappears. This example thus validates our claim that garbage locations cannot play a role in an evaluation.

Based on the semantics $eval_S$, we can now also define an extended notion of operational equivalence for Λ_S. The definition has a non-standard syntactic consistency component because of the context-sensitive restrictions on the programming language.

Definition 4.3 ($\simeq_S$) Two terms $M, N \in \Lambda_S$ are operationally equivalent, $M \simeq_S N$, iff for any arbitrary Λ_S-context $C[\ ]$, $C[M]$ is a program iff $C[N]$ is a program, and, if $C[M]$ and $C[N]$ are programs, then $eval_S$ 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.

At this point the natural question arises how Λ_σ and Λ_S are related. Given a program, both the CS- and the C-machine step through a number of states and stop upon reaching a final state. It is intuitively clear that the two machines go through the same number of steps, and that the i-th state in the CS-evaluation closely corresponds to the i-th state in the C-evaluation. The relationship between corresponding states formalizes the two important attributes of the C-machine: first, locations in the CS-control string are explicitly replaced by labeled values, and, second, locations in the domain of the store that are unreachable through this process are irrelevant for the rest of the evaluation process. Formally, the relationship is expressed as a function U from CS-control strings to Λ_S-expressions:

$$\begin{aligned}
\mathsf{U}(\sigma l.M, \theta) &= \sigma\, \bullet^{l}.\mathsf{U}(M, \theta)\\
\mathsf{U}(l, \theta) &= (\mathsf{U}(\theta(l), \theta[l := \bullet]))^{l}
\end{aligned}$$

$$\begin{aligned}
\mathsf{U}(c,\theta) &= c \\
\mathsf{U}(x,\theta) &= x \\
\mathsf{U}(\lambda x.M,\theta) &= \lambda x.\mathsf{U}(M,\theta) \\
\mathsf{U}(MN,\theta) &= \mathsf{U}(M,\theta)\mathsf{U}(N,\theta) \\
\mathsf{U}(\sigma x.M,\theta) &= \sigma x.\mathsf{U}(M,\theta).
\end{aligned}$$

On Λ_σ and Λ, U is the identity function; value-store pairs are mapped to Λ_S-values. In other words, initial states for the CS-machine become initial states for the C-machine, final states become final states.

With U as the unload function for the CS-machine, the two evaluation functions $eval_{\sigma,\mathsf{U}}$ and $eval_S$ become the same.

Theorem 4.1 (C-simulation) *For all programs $M \in \Lambda_\sigma$, $eval_{\sigma,\mathsf{U}}(M) = eval_S(M)$.*

Proof. The proof is an induction on the number of transition steps in an evaluation with the basic claim that the translation U maps the i-th state in a CS-evaluation to the i-th state in the corresponding C-evaluation. This is clearly true for initial and final states. For the induction step, consider the CS-transition:

$$\langle c_1,\theta_1\rangle \stackrel{CS}{\longmapsto} \langle c_2,\theta_2\rangle.$$

Assuming that we directly use locations as labels, it is easy to show by case analysis that

$$\mathsf{U}(c_1,\theta_1) \stackrel{C}{\longmapsto} \mathsf{U}(c_2,\theta_2)$$

and thus, the claimed relationship is an invariant of transition steps. The result follows from the fact that U also preserves stuck states, where a functional constant is applied to a term, or the interpretation function δ for constants is undefined on some application.

The only remaining point to be shown is that the choice of the correct new label is always possible. After all, 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. But then, on the other hand, it follows that we can add the clause

$$Lab(\mathsf{U}(c_i,\theta_i)) \subseteq Dom(\theta_i) \text{ for } i = 1,2$$

to the induction invariant. A verification of this condition is straightforward. It reassures us that at (C3)-rewriting steps it is always possible to choose the same label as the CS-machine in its corresponding transition. Consequently, the two machines always maintain the same store-equivalence relation. □

As an almost immediate consequence we get that the restriction of S-operational equivalence to Λ_σ-terms is σ-operational equivalence.

Corollary 4.2 *For all $M,N \in \Lambda_\sigma$, $M \simeq_\sigma N$ iff $M \simeq_S N$.*

Proof. The proof is easy and relies on the preceding theorem. For the direction from left to right, we must show in addition that every Λ_S-program is either a Λ_σ-program or is an intermediate evaluation stage on the C-machine for some Λ_σ-program. Clearly, we only need to consider programs with labeled values as subterms. Without loss of generality, consider a program with a single such term:

$$C[V^l]$$

for some arbitrary value V (possibly containing l-labeled bullets). As can easily be checked out, such a program is the result of evaluating

$$(\lambda x_\sigma.(\sigma x_\sigma.C[x_\sigma])V)0$$

Hence, S-operational equivalence does not gain any additional power from using contexts that already contain labeled terms. □

As a consequence of Theorem 4.1 and Corollary 4.2, we can completely replace the CS-based system by the C-machine. Most importantly, we can use the C-transition rules for reasoning about operational equivalences between Λ_σ-terms. The advantage of this can easily be demonstrated with our prototypical example $((\lambda x.(\sigma x.M)1)0)$ where x is not free in M. For any arbitrary evaluation context $E[\ ]$ this term reduces in a few steps to M, and is hence operationally equivalent; no auxiliary proof is needed. We have thus accomplished the first goal of eliminating garbage from the underlying semantics, and can now tackle the second one of designing a calculus for reasoning about operational equivalence.

5 The λ_v-S-calculus

A closer look at the C-rewriting system for Λ_S should remind the reader of the standard reduction sequences for λ-calculi [2]. A derivation according to standard rules first partitions a program into a context and a β-redex, the redex being the leftmost-outermost one. Next, an appropriate contraction replaces the redex by a new term. After that, the standard evaluation resumes the cycle. The rewriting system works in the same way, except that the uniqueness of the context-redex partitioning plays a more crucial role for the timing and the effect of the transition rules (C3) through (C5). We therefore characterize these transitions as *context-sensitive*.

The context-insensitive rules of the C-rewriting system are (C1) and (C2). They give rise to Curry's δ- and Plotkin's β_v-relation:

$$(fa) \longrightarrow \delta(f,a) \qquad (\delta)$$
$$(\lambda x_\lambda.M)V \longrightarrow M[x_\lambda := V]. \qquad (\beta_v)$$

Both relations are ordinary *notions of reduction* in Barendregt's terminology, that is, they are applicable to any instance of a redex at any position in a term. Together, the relations form the basis of Plotkin's λ_v-calculus.

Definition 5.1 (λ_v*-calculus*) The basic notion of reduction is

$$\mathbf{v} = \delta \cup \beta_v.$$

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

$$M \longrightarrow_s N \text{ if } (P,Q) \in \mathbf{v}, M \equiv C[P], \text{ and } N \equiv C[Q] \text{ for some } P,\ Q, \text{ and } C[\ ] \text{ in } \Lambda.$$

The **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 $=_v$.

Remarks. First, we use contexts to formalize the idea of applying a redex at any arbitrary term position. Second, the compatible equivalence (congruence) relation $=_v$ is the λ_v-calculus, but, sometimes, we refer to the entire system of relations as the calculus. **End**

The context-sensitive transition rules of the C-rewriting system naturally divide into two different classes: (C3) and (C4), which leave their context intact, and (C5), which modifies the context. This division implies two reasons for context-sensitivity.

The first aspect of context-sensitivity concerns the timing of C-transitions. The partitioning of programs into C-redexes and evaluation contexts uniquely determines *when* a transition is to occur. This correct timing is obviously necessary for the delabeling rule and the assignment rule in order to preserve the determinicity of the semantics. Freely applicable reduction versions of these transitions would clearly interfere with the correct ordering of side-effects (according to the C-machine semantics). On the other hand, it is less obvious why a modified β-reduction à la

$$(\lambda x_\sigma.M)V \longrightarrow M[x_\sigma := V^l]$$

would establish sharing relations at the wrong time. This is a more subtle point and deserves an example. Consider the following expression:

$$\begin{array}{l}\textbf{let } f = (\lambda x.\underline{(\textbf{let } y = 0 \textbf{ in } ((\sigma y.y)(\textsf{succ } y)))}) \textbf{ in}\\ \quad \textbf{let } d = f(0) \textbf{ in}\\ \quad\quad f(0).\end{array}$$

Clearly, this program should yield 1. Furthermore, the (expansion of the) underlined subterm matches the left-hand side of the above term relation on λx_σ-applications. If we apply the reduction, we get:

$$\begin{array}{l}\textbf{let } f = (\lambda x.\underline{((\sigma \bullet^l .0^l)(\textsf{succ } 0^l))}) \textbf{ in}\\ \quad \textbf{let } d = f(0) \textbf{ in}\\ \quad\quad f(0).\end{array}$$

However, the evaluation of this expression on the C-machine yields 2. As mentioned above, the problem with the suggested notion of reduction is that it establishes the sharing relation too early.

The second aspect of context-sensitivity concerns the extent to which a transition can affect a program. This is only visible in the rule (CS5), which not only needs to be timed correctly, but must also manipulate the entire context. In other words, (C5) specifies *how*

a redex transforms its context. This is certainly unusual and does not exist in regular, mathematical calculi.

Considering the indicated problems, there is no hope of finding a set of context-insensitive reduction relations that are equivalent to the C-rewriting rules. 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: *notions of reduction* and *computations*. The former are ordinary reduction rules that are applicable to any position within a term. 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 and by explicitly depicting their arguments as complete programs.

An evaluation in the new system resembles a race. All imperative redexes in a program simultaneously start bubbling up towards the root. Whichever redex gets there first performs an imperative effect. The appropriate arrival of an imperative redex at the top is determined by arbitration conditions in the reduction relations. Although such a system radically differs from a traditional calculus, it is an acceptable generalization of the notion of a calculus as we shall demonstrate in the next few sections.

We start the design of our reductions and computations with the C-transition rule (C3). As mentioned above, the computation relation is only used at the root of a term. That means, it is applied in the empty context, and therefore, it has the same form as the transition rule without an evaluation context $E[\ ]$:

$$\&(\lambda x_\sigma.M)V \ \triangleright \ \&M[x_\sigma := V^l],\ l \notin Lab(M,V). \qquad (\beta_\sigma)$$

Otherwise, if the C-redex for (C3) 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. But this is equivalent to forming the application of the abstraction body to the argument first and modifying the body afterwards:

$$E[((\lambda x_\sigma.M)V)N] \longrightarrow E[(\lambda x_\sigma.MN)V].$$

Since this modified transition rule is clearly independent of the evaluation context, we can adopt the context-free variant as another notion of reduction:

$$((\lambda x_\sigma.M)V)N \longrightarrow (\lambda x_\sigma.MN)V \qquad (\beta_L)$$

Based on the symmetry of the definition of evaluation contexts, we suggest the following rule for the case where the C-redex is to the right of a value U:

$$U((\lambda x_\sigma.M)V) \longrightarrow (\lambda x_\sigma.UM)V \qquad (\beta_R)$$

Like the β-rule itself, *these reductions rely on the hygiene convention and assume that the sets of free variables in N and U do not contain x_σ.*

From the treatment of λx_σ-applications, it is clear how to deal with the simulation of assignments. The computation relation for σ-applications is

$$\&(\sigma \bullet^l .M)V \triangleright \&M[\bullet^l := V^l]. \qquad (\sigma_T)$$

An embedded σ-application can work its way to the root of a term with rules similar to those for a λ-application. This process is applicable to both instantiated and uninstantiated σ-capabilities:

$$((\sigma X.M)V)N \longrightarrow (\sigma X.(MN))V, \qquad (\sigma_L)$$
$$U((\sigma X.M)V) \longrightarrow (\sigma X.(UM))V. \qquad (\sigma_R)$$

Unfortunately, the bubbling-up technique cannot be applied as easily to the delabeling of labeled values. Unlike a λ-abstraction or a σ-capability, a labeled value does not contain a subterm that can be used for the gradual incorporation of the term context. The key insight[4] is that, except for two cases, labeled values are only delabeled once they become the argument of some function. The first exception is the simple occurrence of a labeled value as a program by itself. Although this is only possible as the final step of an evaluation, we still need a (computation) rule to account for this case:

$$\&V^l \triangleright \&V[\bullet^l := V^l]. \qquad \text{(Stop)}$$

The second exception is the occurrence of a labeled value in function position, but it is easy to see that this case can be transformed into an application of a function to a labeled value:

$$V^l M \longrightarrow (\lambda v.vM)V^l. \qquad (D_{sym})$$

Once the labeled value is delabeled, the resulting function is immediately applied to the proper argument after a simple β_v-step.

Given the assumption that labeled values always occur to the right of a value, we can proceed with our analysis in the usual way. First, consider the occurrence of such a term in the empty context. Then it is time to strip off the label and to perform the application:

$$\&(UV^l) \triangleright \&U(V[\bullet^l := V^l]). \qquad (D_T)$$

Second, if (UV^l) is nested in an evaluation context and, say, is to the left of some expression M, the application must somehow incorporate the additional argument M into the function U in order to move closer to the top. The resulting application should first delabel V^l, then apply U to the resulting value, and finally, the result of this application should absorb M. Abstracting from the value of V^l, this informal description leads to the λ-abstraction $\lambda v.UvM$ for the merger function of U and M. As in the above case of σ-capabilities, this same argument also holds for an application of a function U to an assignable variable. Putting this together, the appropriate reduction is defined by:

$$(UX)M \rightarrow (\lambda x_\lambda.Ux_\lambda M)X. \qquad (D_L)$$

[4]This solution was suggested by Robert Hieb, Indiana University; it greatly improves our own from earlier reports [4, 5].

By symmetry, we get a similar rule for the case where the application is to the right of some value:

$$U(VX) \rightarrow (\lambda x.U(Vx))X. \qquad (D_R)$$

The definition of reduction and computation relations for assignable variables and labeled values ends the design phase of the λ_v-CS-calculus. For convenience, we have collected the relations in a separate definition.

Definition 5.2 (*Reductions and computations*) Recall that U and V denote values, and that X ranges over assignable variables and labeled values. The notions of reduction are

$$\begin{aligned}
fa &\longrightarrow \delta(f,a) && (\delta)\\
(\lambda x_\lambda.M)V &\longrightarrow M[x_\lambda := V] && (\beta_v)\\
U((\lambda x_\sigma.M)V) &\longrightarrow (\lambda x_\sigma.(UM))V && (\beta_R)\\
((\lambda x_\sigma.M)V)N &\longrightarrow (\lambda x_\sigma.(MN))V && (\beta_L)\\
U((\sigma X.M)V) &\longrightarrow (\sigma X.(UM))V && (\sigma_R)\\
((\sigma X.M)V)N &\longrightarrow (\sigma X.(MN))V && (\sigma_L)\\
V^lM &\longrightarrow (\lambda v.vM)V^l && (D_{sym})\\
(VX)M &\longrightarrow (\lambda x.VxM)X && (D_L)\\
U(VX) &\longrightarrow (\lambda x.U(Vx))X && (D_R)
\end{aligned}$$

The computation relations are

$$\begin{aligned}
\&(\lambda x_\sigma.M)V &\triangleright \&M[x_\sigma := V^l] \text{ where } l \text{ is fresh} && (\beta_\sigma)\\
\&(\sigma \bullet^l.M)V &\triangleright \&M[\bullet^l := V^l] && (\sigma_T)\\
\&(UV^l) &\triangleright \&U(V[\bullet^l := V^l]) && (D_T)\\
\&V^l &\triangleright \&V[\bullet^l := V^l] && (\text{Stop})
\end{aligned}$$

We refer to the left-hand sides of computations as *computational redexes.*

The next steps in the development of our calculus are straightforward. As usual, we collect all notions of reductions into a single relation, **s**, and form a congruence relation. As in the definition of the λ_v-calculus, we use an arbitrary Λ_S-context to formalize the idea of applying a reduction to a redex at an arbitrary point in a term. Thus far our calculus would be a simple extension of the traditional λ_v-calculus. However, since our goal is a calculus for simulating the rewriting semantics, we must go beyond conventional constructions and somehow include the computation relations in order to cope with imperative effects. To avoid any interference of computations with the compatibility construction, we have chosen to define a *computation* as the union of the (transitive) **s**-reduction and the three 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 perform the imperative action. On top of this computation, we define computational equality as the smallest equivalence relation that encloses the computation.

Definition 5.3 (*The λ_v-S-calculus*) The basic notion of reduction is

$$\mathbf{s} = \delta \cup \beta_v \cup \beta_L \cup \beta_R \cup \sigma_L \cup \sigma_R \cup D_{sym} \cup D_L \cup D_R.$$

The *one-step* **s**-*reduction* $\longrightarrow_s$ is the compatible closure of **s**:

$$M \longrightarrow_s N \text{ if } (P,Q) \in \mathbf{s}, M \equiv C[P], \text{ and } N \equiv C[Q] \text{ for some } P,\ Q, \text{ and } C[\ ] \text{ in } \Lambda_S.$$

The **s**-*reduction* is denoted by $\longrightarrow\!\!\!\!\rightarrow_s$ and is the reflexive, transitive closure of $\longrightarrow_s$. We denote the smallest equivalence relation generated by $\longrightarrow_s$ with $=_s$ and call it **s**-*equality.*

The **s**-*computation* $\triangleright_s$ is defined by:

$$\triangleright_s = \longrightarrow\!\!\!\!\rightarrow_s \cup \beta_\sigma \cup \sigma_T \cup D_T \cup \text{Stop}.$$

The relation $\stackrel{\triangleright}{=}_s$ is the smallest equivalence relation generated by $\triangleright_s$. We refer to it as *computational equality.*

The result of our calculus design is an unorthodox 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-S-calculus, we refer to the relation $\stackrel{\triangleright}{=}_s$. We write $M \stackrel{\triangleright}{=}_s N$ or $\lambda_v\text{-S}^{\triangleright} \vdash M = N$ for theorems and derivations on this level. Although weaker, the congruence relation $=_s$ is traditional and interesting in its own right. We consider it as a sub-calculus and use the notation $\lambda_v\text{-S} \vdash M = N$.

6 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_S contains the more sophisticated category of assignable variables, we can still prove a comparable substitution property.

Theorem 6.1 *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) $\lambda_v\text{-}S^{\triangleright} \vdash \&MV = \&NV$ *implies* $\lambda_v\text{-}S^{\triangleright} \vdash \&M[x_\lambda := V] = \&N[x_\lambda := V]$

(ii) $\lambda_v\text{-}S^{\triangleright} \vdash \&MV = \&NV$ *implies* $\lambda_v\text{-}S^{\triangleright} \vdash \&M[x_\sigma := V^l] = \&N[x_\sigma := V^l]$

(iii) $\lambda_v\text{-}S \vdash M = N$ *implies* $\lambda_v\text{-}S \vdash M[x_\lambda := V] = N[x_\lambda := V]$

(iv) $\lambda_v\text{-}S \vdash M = N$ *implies* $\lambda_v\text{-}S \vdash M[x_\sigma := V^l] = N[x_\sigma := V^l]$

Remark. The antecedents in statements (i) and (ii) imply 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 $M \stackrel{\triangleright}{=}_s N$ 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 proofs. The induction relies on a generalized version of the substitution lemma [2]:

$$M[v := V][u := U[v := V]] \equiv M[u := U][v := V] \qquad \text{if } u \notin FV(V);$$

and on a commutation lemma for substitution and labeled-value substitution:

$$M[v := V][\bullet^l := (U[v := V])^l] \;\equiv\; M[\bullet^l := U^l][v := V] \qquad \text{if } l \notin Lab(V). \qquad \square$$

We shall use the theorem in the following section on the correctness of the calculus. It is stated here because of its logical nature.

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.

Theorem 6.2 (Consistency)

(i) The notion of reduction **s** *is Church-Rosser.*

(ii) The **s***-computation satisfies the diamond property.*

Proof. The proof of the first part is a modification of the Tait/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 Appendix. □

The theorem implies an important corollary.

Corollary 6.3 *If $M \stackrel{\triangleright}{=}_s N$ then there exists an L such that $M \triangleright_s^* L$ and $N \triangleright_s^* L$.*

This corollary and the existence of distinct, irreducible terms guarantee that the calculus cannot prove an equation between all terms. It furthermore shows 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 derivations and are thus crucial for finding values and normal-forms of programs. In the λ-calculus and the λ_v-calculus, standard derivations are formed by reducing the leftmost-outermost redex or, if a leftmost-outermost redex is not reduced, it is excluded from any further consideration in the rest of the derivation. The λ_v-S-calculus also has such standard derivations, but, as above, this again requires the consideration of two levels. For the lower level, we must show that there are standard *reduction* sequences in the reduction sub-calculus; for the upper level, we must extend the notion of standard reduction sequences to standard *computation* sequences and prove their adequacy.

The definition of standard reduction and computation sequences for proving a Curry-Feys Standardization Theorem proceeds in two steps—a slick strategy due to Plotkin [21]. First, we define the standard reduction and the standard computation function. These partial functions provide the proper means for contracting exactly one redex at a time, namely, the leftmost-outermost redex or computational redex that is not embedded in a value. *Both functions are undefined on values.* By adding in the computation after extending the notion of reduction **s** to a standard reduction function, we ensure that top-level computation steps cannot cause inconsistencies.

Definition 6.1 (*The standard reduction and standard computation functions*)

(i) The standard reduction function maps M to N, $M \longmapsto_{ss} N$, if there are P, Q, and an evaluation context $C[\ ]$ such that $(P, Q) \in \mathbf{s}$, $M \equiv C[P]$, and $N \equiv C[Q]$.

(ii) The standard computation function is an extension of the standard reduction function with computation relations:

$$\overset{\triangleright}{\longmapsto}_{ss} = \longmapsto_{ss} \cup \beta_\sigma \cup \sigma_T \cup \mathcal{D}_T.$$

Second, we use these functions to define two notions of term sequences: the standard reduction and the standard computation sequences, respectively. 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 be 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. Standard *computations* sequences extend reduction sequences with computations.

Definition 6.2 (*Standard sequences*) *Standard reduction sequences*, abbreviated SR-sequences, are defined by:

- all constants and variables are SR-sequences;
- if $M_1, \ldots, M_m$, $N_1, \ldots, N_n$, and $V_1, \ldots, V_j$ are SR-sequences, then

$$\begin{array}{l} \lambda x.M_1, \ldots, \lambda x.M_m, \\ M_1N_1, \ldots, M_mN_1, \ldots, M_mN_n, \\ \sigma X.M_1, \ldots, \sigma X.M_m, \text{ and} \\ (V_1[\bullet^l := \bullet^l])^l, \ldots, (V_j[\bullet^l := \bullet^l])^l \end{array}$$

  are SR-sequences;

- if $M \longmapsto_{ss} M_1$ and $M_1, \ldots, M_m$ is an SR-sequence, then $M, M_1, \ldots, M_m$ is an SR-sequence.

All standard reduction sequences are also *standard computation sequences*, SC-sequences, and if

$$M \overset{\triangleright}{\longmapsto}_{ss} M_1 \text{ and } M_1, \ldots, M_k \text{ is an SC-sequence,}$$

then

$$M, M_1, \ldots, M_k$$

is an SC-sequence.

The Standardization Theorem states—for the calculus and the sub-calculus—that every one-way derivation can be reformulated as a standard derivation.

Theorem 6.4 (Standardization)

(i) $M \longrightarrow\!\!\!\!\!\rightarrow_s N$ *if and only if there is an* SR*-sequence* $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$*;*

(ii) $M \triangleright_s^* N$ *if and only if there is an* SC*-sequence* $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$*.*

Proof. The proof of the theorem is an extension of Plotkin's proof of the Standardization Theorem for the λ_v-calculus. It is presented in the Appendix. □

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 correctness of the λ_v-S-calculus.

7 Correctness

After completing the design and logical analysis of the λ_v-S-calculus, we need to address the important question of whether it is the *right* calculus. Our goal was to derive a conservative extension of the λ_v-calculus for Λ_σ. To verify this, we must check that the extension is conservative, and that the calculus is a programming language calculus according to the criteria outlined in Section 2. In other words, we must answer three questions:

1. Is the λ_v-S-calculus a conservative extension of the λ_v-calculus?
2. Do the machine and the calculus compute the same answer for a program?
3. Does equality in the calculus imply operational equality on the machine?

The first question is the easiest one to answer, given that the definition of the new calculus subsumes the reduction relation of the λ_v-calculus, and given the corollary of the Consistency Theorem about the shape of derivations.

Theorem 7.1 *For* $M, N \in \Lambda$, $\lambda_v \vdash M = N$ *iff* $\lambda_v\text{-}S^{\triangleright} \vdash M = N$.

Proof. The direction from left to right is obvious since the definition of the notion of reduction **s** includes the notion of reduction **v** as a subrelation. For the opposite direction, assume that λ_v-S $^{\triangleright} \vdash M = N$. By the Consistency Theorem (6.2) and its corollary, we know that for some L, $M \triangleright_s^* L$ and $N \triangleright_s^* L$. But, given that both terms are pure Λ-terms, none of the steps in either computation sequence can actually be a computation or reduction step involving assignable variables or labels, which the left-hand sides of these rules require. Hence, all steps must be β_v- or δ-reductions and are therefore also valid in the λ_v-calculus. □

For the second question, we simply need to check that the standard function is an alternative formulation of the C-rewriting system. After all, we started the design of the λ_v-S-calculus based on the idea that the C-rewriting system works like a standard reduction system, and we defined the standard computation function as *the* function that always reduces the leftmost-outermost redex in an application-term.

Theorem 7.2 *For all programs $M \in \Lambda_\sigma$ (or Λ_S), $eval_S(\&M) = V$ iff $M \overset{\triangleright}{\longmapsto}{}_{ss}^{*}$ V for some value V.*

Proof. Recall that the standard computation function reduces either a computational redex at the root of a program or a simple redex in an evaluation context, and that it never reduces a redex inside of a value. Hence, we can show by induction on the structure of an evaluation context that

$$\begin{aligned} E[V^l] &\overset{\triangleright}{\longmapsto}{}_{ss}^{+} E[V[\bullet^l := V^l]] \\ E[(\lambda x_\sigma.M)V] &\overset{\triangleright}{\longmapsto}{}_{ss}^{+} E[M[x := V^l]] \text{ where } l \text{ is fresh,} \\ E[(\sigma\, \bullet^l .M)V] &\overset{\triangleright}{\longmapsto}{}_{ss}^{+} E[M][\bullet^l := V^l]. \end{aligned}$$

This means that the standard computation function correctly simulates the imperative actions; the simulation of β_v and δ-steps is obvious. □

The last correctness question has a curious answer. Equality of Λ_σ-terms in the calculus indeed implies operational equality on the machine, but, on one hand, this statement is almost useless and, on the other, it is difficult to prove. To understand this, recall the purpose of operational equivalence. It is to equate terms that are interchangeable in *all* contexts, that is, the context-dependencies of both terms are the same. It is clear that the context-dependencies of Λ_σ-terms are their *free* variables. And this is the point of contention. In the λ_v-S-calculus, free assignable variables stand for and are replaced by labeled values, and, only when they are replaced by values, can we know how a derivation of the term proceeds. Yet, such labeled values are *not* a part of Λ_σ, but of the larger language Λ_S. Hence, we must use this larger language and reason about *its* operational equivalence relation. This, however, causes a major problem: because of the context-sensitive computations, equalities of Λ_S-terms in the calculus do not necessarily hold on the machine.

Proposition 7.3 *(i) λ_v-$S \vdash M = N$ implies $M \simeq_S N$.*

(ii) λ_v-$S^{\triangleright} \vdash M = N$ does **not** *imply $M \simeq_S N$.*

Proof. Point (i) is easy. The reduction sub-calculus is a full congruence relation. By Theorem 7.2, we also know that we can standard reduce to a value inside the calculus, and, by the Consistency Theorem, we know that this can only be a unique basic constant. For point (ii), consider instantiations of the computations (σ_T) and (D_T):

$$\lambda_v\text{-S}^{\triangleright} \vdash U0^l = U0 \quad (1)$$

$$\lambda_v\text{-S}^{\triangleright} \vdash (\sigma\, \bullet^l.1)0 = 1 \quad (2)$$

It is obvious that 1 and $(\sigma \bullet^l.1)0$ behave differently in most contexts, and that $U0$ is generally not the same as $U0^l$. The use of computations is restricted to the top of a program. It is consequently impossible that such equations are valid in more general contexts as it would be the case with a congruence relation like S-operational equivalence. □

The proof of Proposition 7.3 reveals that there are two distinct troublesome aspects of computations concerning their use in proofs about operational equivalences. On one hand, they rely on the specific value parts of labeled values, but this part of course depends on the assignments that precede the delabeling. On the other hand, terms on one side of such a computation equation are about to perform an imperative effect whereas the term on the other side has just performed such an effect; in other words, the two terms of a computation equation have different effects on their potential contexts.

If we want to use the derivations and equations of the calculus for reasoning about operational equivalence, we must factor out those that contain these prototypical problems. The first problem of relying on specific labeled values could be avoided by requiring that an equation be independent of the labeled values in the two terms. More formally, if

$$\lambda_v\text{-S}^{\triangleright} \vdash M[\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}] = N[\bullet^{l_1} := {V_1}^{l_1}] \ldots [\bullet^{l_n} := {V_n}^{l_n}],$$

where $l_1, \ldots, l_n$ are all the labels in M and N, then this equation must not only hold for the specific values $V_1, \ldots, V_n$, but for all arbitrary values in their places. This restriction would certainly rule out equation (2) in the above proof as an admissible equation.

For the second problem, we need a restriction that rules out that the two equation-terms have different effects. The solution relies on two simple ideas. First, a difference in effect can only be observed when terms are evaluated. Second, terms are only evaluated in evaluation contexts. Put together, the second restriction says that an equation is only admissible when it holds in all evaluation contexts. A unification of the two restrictions yields the formal definition of admissible or *safe* calculus-equations.

Definition 7.1 (*Safe theorems*) Let $M, N \in \lambda_S$ and let $Lab(M) \cup Lab(N) = \{l_1, \ldots, l_n\}$. The theorem

$$\lambda_v\text{-S}^{\triangleright} \vdash M = N$$

is *safe* if for all evaluation contexts $E[\ ]$ and for n non-assignable variables $v_1, \ldots, v_n$

$$\lambda_v\text{-S}^{\triangleright} \vdash E[M][\bullet^{l_1} := {v_1}^{l_1}] \ldots [\bullet^{l_n} := {v_n}^{l_n}] = E[N][\bullet^{l_1} := {v_1}^{l_1}] \ldots [\bullet^{l_n} := {v_n}^{l_n}].$$

The adequacy of safe theorems is encapsulated in the central theorem of this section.

Theorem 7.4 (*Safe*-ness) *If λ_v-$S^{\triangleright} \vdash M = N$ is safe, then $M \simeq_S N$.*

Proof. Without loss of generality, we assume that M and N contain one free non-assignable variable (x), one free assignable variable (y), and one labeled value (with label l). Now, let $D[\ ]$ be an arbitrary context and assume that $D[M]$ evaluates to a basic constant a:

$$\lambda_v\text{-S}^{\triangleright} \vdash D[M] \stackrel{\triangleright}{\longmapsto}_{ss}^{*} a.$$

If M plays an active role in this derivation, a closed and possibly side-effected version M' must occur in an evaluation context $E[\ ]$:

$$\lambda_v\text{-S}^{\triangleright} \vdash D[M] \stackrel{\triangleright}{\longmapsto}_{ss}^{*} E[M'] \stackrel{\triangleright}{\longmapsto}_{ss}^{*} a,$$

where

$$M' \equiv M[x := U][y := V^m]][\bullet^l := W^l].$$

From the assumption that $M = N$ is a safe theorem, it follows that

$$\lambda_v\text{-S}^{\triangleright} \vdash E[M'] = E[N']$$

where

$$N' \equiv N[x := U][y := V^m][\bullet^l := W^l].$$

The *safe*-ness of the theorem guarantees that side-effects cannot interfere with the proof; the Substitution Theorem provides for orthogonality of simple variable substitutions: its antecedent is satisfied because U, V, and W were a part of the program all along.

Given this, we can replace the above derivation by

$$\lambda_v\text{-S}^{\triangleright} \vdash D[M] = E[N'] = E[M'] \stackrel{\triangleright}{\longmapsto}_{ss}^{*} a.$$

This can be done for every occurrence of a version of M in an evaluation context in the rest of the standard computation sequence. Consequently, the entire derivation is independent of M:

$$\lambda_v\text{-S}^{\triangleright} \vdash D[N] = a.$$

By the Consistency and the Standardization Theorem, it follows that

$$\lambda_v\text{-S}^{\triangleright} \vdash D[N] \stackrel{\triangleright}{\longmapsto}_{ss}^{*} a$$

as desired. □

An immediate consequence of this theorem is the third correctness theorem for the λ_v-S-calculus. The central idea is that equations between Λ_σ-terms are automatically safe as opposed to equations between Λ_S-terms, which may contain labeled values.

Theorem 7.5 *For $M, N \in \Lambda_\sigma$, λ_v-S$^{\triangleright} \vdash M = N$ implies $M \simeq_\sigma N$.*

Proof. Since neither M nor N contain labeled values, they automatically satisfy one half of the *safe*-ness condition. Moreover, the absence of labels also prohibits any immediate imperative effects on an evaluation context. Hence, equations between Λ_σ-terms are safe and imply σ-operational equivalence by the *Safe*-ness theorem. □

Remark. The theorem does not imply that λ_v-S$^\triangleright \vdash M = N$ implies λ_v-S$^\triangleright \vdash C[M] = C[N]$ for arbitrary contexts. Indeed, this is wrong as the following example demonstrates. Consider the equation

$$\lambda_v\text{-S}^\triangleright \vdash (\lambda x.(\sigma x.2)1)0 = 2.$$

From this equation it does *not* follow that

$$\lambda_v\text{-S}^\triangleright \vdash \lambda y.(\lambda x.(\sigma x.2)1)0 = \lambda y.2.$$

Since the right-hand side is an irreducible term, the left-hand side would have to be reducible to the right-hand side. But it is easy to see that there is no standard reduction sequence from left to right and hence the two terms are not equivalent in the calculus. **End**

The upshot of our development and the correctness theorems is that imperative extensions of functional languages with assignments have equational theories. Based on the conservative extension property, we can re-use all the equations and proofs about functional programs in the λ_v-calculus. The Simulation Theorem gives us the right to rely on C-rewriting rules for derivations in the λ_v-S-calculus. This is of great interest, given the idea of safe equations. The *Safe*-ness theorem provides the recipe for using the full calculus in this process, and safe theorems heavily rely on the idea of performing a complete, C-rule like derivation inside of an evaluation calculus. The last correctness theorem says that the λ_v-S-calculus is indeed the right equational tool for reasoning about operational equivalences of Λ_σ-terms. In the next section, we show how all the pieces fit together.

8 Examples

In the preceding sections we have developed an equational theory for higher-order functional languages with assignments. Next, we must demonstrate the feasibility of reasoning with this theory, *i.e.*, we must show how easy or how difficult it is to derive operational equivalences about expressions. The correctness theorems of the previous section provide the receipe for such proofs. Our plan is to investigate this receipe with some simple examples, to apply these results to some typical problem cases for denotational semantics [7, 18, 16], and to conclude with a brief investigation of an imperative version of the recursion combinator.

In order to understand the connection between operational equivalences, equations about Λ_σ-terms, and safe equations, we begin with some examples on the effect of simple assignments of values to variables. Consider the program fragment

$$\textbf{begin var } x; \ldots x := U; x := V \ldots \textbf{end},$$

where U and V are some arbitrary values. Clearly, the first assignment of U to the variable x cannot have any impact on the evaluation and is thus eliminable. The formal counterpart to this idea is that the fragment is operationally equivalent to the block

$$\textbf{begin var } x; \ldots x := V \ldots \textbf{end}.$$

With Λ_σ-terms, we can express this as the equation

$$(\lambda x_\sigma.C[(\sigma x_\sigma.(\sigma x_\sigma.M)V)U])0 \simeq (\lambda x_\sigma.C[(\sigma x_\sigma.M)V])0,$$

where 0 is an arbitrary initial value for x_σ and $C[\ ]$ and M represent the rest of the block.

How can we prove this equation? First, we know that the block can only affect the result if it is evaluated. Hence, we can apply the transition rule (C3) to both sides:

$$C[(\sigma\bullet^l.(\sigma\bullet^l.M)V)U][x_\sigma := 0^l] \simeq C[(\sigma\bullet^l.M)V][x_\sigma := 0^l].$$

In this new proof goal, the crucial terms are the fill-ins of the context $C[\ ]$. The goal holds if we can show that these two subterms are operationally equivalent:

$$(\sigma\bullet^l.(\sigma\bullet^l.M)V)U[x_\sigma := 0^l] \simeq (\sigma\bullet^l.M)V[x_\sigma := 0^l].$$

Second, in order to prove this last equation, we apply the Safe-ness theorem. In other words, we must show that

$$\lambda_v\text{-S}^{\triangleright}\vdash(\sigma\bullet^l.(\sigma\bullet^l.M)V)U[x_\sigma := 0^l] = (\sigma\bullet^l.M)V[x_\sigma := 0^l],$$

and that this equation is safe. To this end, we assume without loss of generality that no label other than l occurs in M, U, or V, and prove the above equation in the λ_v-S-calculus. Indeed, to avoid duplication of work, we not only develop a derivation for the equation but a *derivation schema* that proves the equation in all evaluation contexts and for arbitrary l-labeled values:

$$\begin{aligned}
\lambda_v\text{-S}^{\triangleright}\vdash\quad & E[(\sigma\bullet^l.((\sigma\bullet^l.M)V))U][\bullet^l := {x_\lambda}^l] \\
= \ & E[(\sigma\bullet^l.M)V][\bullet^l := U^l] \\
= \ & E[M][\bullet^l := U^l][\bullet^l := V^l] \\
\equiv \ & E[M][\bullet^l := V^l] \\
= \ & E[(\sigma\bullet^l.M)V][\bullet^l := {x_\lambda}^l].
\end{aligned}$$

Clearly, this derivation schema establishes the original equation and in addition proves all relevant instantiations of this equation that we need to prove in order to verify its safe-ness. The safe-ness of the equation implies the proof goal.

Another property of simple assignments is the interchangeability of such assignments to distinct variables (in a call-by-value language). More precisely, we should be able to prove equations like

$$\ldots x := U; y := V \ldots \simeq \ldots y := V; x := U \ldots.$$

Applying the same analysis as above, this equation holds if we can prove the following safe equation about Λ_S-terms:

$$\lambda_v\text{-S}^{\triangleright}\vdash(\sigma\bullet^l.((\sigma\bullet^k.M)V))U = (\sigma\bullet^k.((\sigma\bullet^l.M)U))V.$$

The appropriate derivation schema proceeds as follows:

$$\begin{aligned}
\lambda_v\text{-S}^{\triangleright}\vdash\quad & E[(\sigma\bullet^l.((\sigma\bullet^k.M)V))U][\bullet^l := {x_\lambda}^l][\bullet^k := {y_\lambda}^k] \\
= \ & E[(\sigma\bullet^k.M)V][\bullet^l := U^l] \\
= \ & E[M][\bullet^l := U^l][\bullet^k := (V[\bullet^l := U^l])^k] \\
\equiv \ & E[M][\bullet^k := V^k][\bullet^l := (U[\bullet^k := V^k])^l] \\
= \ & E[(\sigma\bullet^l.M)U][\bullet^k := V^k][\bullet^l := V^l] \\
= \ & E[(\sigma\bullet^k.((\sigma\bullet^l.M)U))V][\bullet^k := {y_\lambda}^k][\bullet^l := {x_\lambda}^l]
\end{aligned}$$

As in the first example, this proves the desired goal.

An interesting generalization of this second equation is the case where the second simple assignment is replaced by an arbitrary computation without reference to the first variable:

$$\ldots x := U;\ M \ldots \simeq \ldots M;\ x := U \ldots, \qquad (*)$$

or

$$(\sigma \bullet^l .((\lambda d.N)M))U \simeq (\lambda d.((\sigma \bullet^l .N)U))M,$$

where $l \notin Lab(M)$. For the proof of this, we distinguish two cases. First, suppose that M diverges. Then it is clear that both expressions also diverge: the right one uses M immediately, the left one after a simple assignment that cannot affect M. Second, assume that M terminates in all evaluation contexts, possibly with effects on the context. Assuming, without loss of generality, that M can only affect and can only be affected through the label k, we can state this property as an equation:

$$\lambda_v\text{-S}^{\triangleright} \vdash E[M[\bullet^k := y_\lambda{}^k]] = E[W][\bullet^k := V^k].$$

Given this, we claim that

$$\lambda_v\text{-S}^{\triangleright} \vdash (\sigma \bullet^l .((\lambda d.N)M))U = (\lambda d.((\sigma \bullet^l .N)U))M, \text{where } l \notin Lab(M),$$

and, furthermore, that this equation is safe. The following derivation schema proves the claim:

$$\begin{aligned}
\lambda_v\text{-S}^{\triangleright} \vdash \quad & E[(\sigma \bullet^l .(\lambda d.N))U][\bullet^l := x_\lambda{}^l][\bullet^k := y_\lambda{}^k] \\
= \ & E[(\lambda d.N)M][\bullet^l := U^l] \\
= \ & E[(\lambda d.N)W][\bullet^l := U^l][\bullet^k := (V[\bullet^l := U^l])^k] \\
= \ & E[N][\bullet^k := V^k][\bullet^l := (U[\bullet^k := V^k])^l] \\
= \ & E[(\sigma \bullet^l .N)U][\bullet^k := V^k][\bullet^l := x_\lambda{}^l] \\
= \ & E[(\lambda d.((\sigma \bullet^l .N)U))M][\bullet^l := x_\lambda{}^l][\bullet^k := y_\lambda{}^k].
\end{aligned}$$

Besides elimination and interchangeability, simple assignments also have directly observable effects. By this we mean that an immediately following reference to an assigned variable can be replaced by the right-hand value, *i.e.*,

$$(\sigma \bullet^l .E[x_\lambda^l])V \simeq (\sigma \bullet^l .E[V])V$$

for all evaluation contexts $E[\ ]$. We prove this in the usual manner:

$$\begin{aligned}
\lambda_v\text{-S}^{\triangleright} \vdash E'[(\sigma \bullet^l .E[x_\lambda^l])V] \ &= \ E'[E[x_\lambda^l]][\bullet^l := V^l] \\
&= \ E'[E[V^l]][\bullet^l := V^l] \\
&= \ E'[E[V[\bullet^l := V^l]]][\bullet^l := V^l] \\
&= \ E'[E[V]][\bullet^l := V^l] \\
&= \ E'[(\sigma \bullet^l .E[V])V]
\end{aligned}$$

In Algol-style syntax, this equation is expressible as

$$\ldots x := V;\ E[x] \ldots \simeq \ldots x := V;\ E[V] \ldots \qquad (\dagger)$$

We are now ready to apply the above results to a typical example from Algol-like languages that can only be treated with non-trivial denotational store models [18]. The example concerns the following operational equivalence of Algol-like statements:

$$\textbf{begin var } x;\ x := 0;\ P();\ \textbf{if } x = 0 \textbf{ then } \Omega \textbf{ end} \simeq \Omega$$

The variable P is a procedure variable, $P()$ a procedure call, the symbol Ω stands for some infinite loop. This immediately implies that (the expression for) P cannot refer to x and hence, by $(*)$, we can exchange the call to P and the simple assignment to x:

$$\begin{array}{c}\textbf{begin var } x;\ x := 0;\ P();\ \textbf{if } x = 0 \textbf{ then } \Omega \textbf{ end} \\ \simeq \\ \textbf{begin var } x;\ P();\ x := 0;\ \textbf{if } x = 0 \textbf{ then } \Omega \textbf{ end.}\end{array}$$

Next we know that the reference to x in the comparison of the **if**-statement follows a simple assignment in an evaluation context and thus the rule $(\dagger)$ applies:

$$\ldots \simeq \textbf{begin var } x;\ P();\ x := 0;\ \textbf{if } 0 = 0 \textbf{ then } \Omega \textbf{ end.}$$

An application of the appropriate δ-rules for comparisons and for **if**, together with a second application of $(*)$, yields:

$$\ldots \simeq \textbf{begin var } x;\ P();\ \Omega;\ x := 0 \textbf{ end.}$$

A simple case analysis on P's termination behavior now shows the above equivalence. If P terminates, the block will start the infinite loop Ω; otherwise, it is in an infinite loop already. In either case, the block is operationally equivalent to an infinite loop.

For another illustration of the power of the λ_v-S-calculus, we take a second example from the same report that cannot be dealt with in the currently available denotational models. Consider the program fragment

$$\textbf{begin var } x;\ \textbf{procedure } Q(y);\ \textbf{begin } x := y \textbf{ end};\ P(Q) \textbf{ end.}$$

Since there are no further references to x, only assignments, and since Algol-languages do not permit reference to local storage outside of its lexical scope, this block is operationally equivalent to the statement

$$\textbf{begin var } x;\ \textbf{procedure } Q(y);\ \textbf{begin } \textit{noop} \textbf{ end};\ P(Q) \textbf{ end.}$$

In Meyer-Siebert store model, however, these two fragments are not semantically equivalent.

With our calculus, we can at least prove the equivalence when the call $P(Q)$ terminates. Let us first translate this equation into a suitable form:

$$(\lambda x.P(\sigma x.0))0 \simeq P(\lambda d.0).$$

The block on the left side reduces to $P(\sigma \bullet^l .0)$ for some $l \notin Lab(P)$:

$$\lambda_v\text{-S}^{\triangleright} \vdash (\lambda x.P(\sigma x.0))0 = P(\sigma \bullet^l .0).$$

The assumption that the call $P(Q)$ terminates can be translated into the fact that there is a finite derivation in the calculus from the call to some value:

$$\lambda_v\text{-S}^{\triangleright} \vdash P(\sigma \bullet^l .0) = V.$$

Since the underlying language is Algol-like, that is, stack-based, we also know that l and $\sigma \bullet^l .0$ cannot occur in V. During the evaluation, P may actually invoke $\sigma \bullet^l .0$ several times. This means some steps in the derivation are applications of the σ-capability to some value U in an evaluation context $E[\ ]$:

$$\lambda_v\text{-S}^{\triangleright} \vdash P(\sigma \bullet^l .0) = \ldots = E[(\sigma \bullet^l .0)U] = \ldots = V.$$

On the other hand, it follows for every single invocation of P's argument that it has no effect on its evaluation context and the rest of the derivation:

$$\lambda_v\text{-S}^{\triangleright} \vdash \ldots = E[(\sigma \bullet^l .0)U] = E[0][\bullet^l := U^l] = E[(\lambda d.0)U] = \ldots.$$

Since the derivation is finite, we can show by induction on the length of the derivation that $\sigma \bullet^l .0$ can consistently be replaced by $\lambda d.0$:

$$\lambda_v\text{-S}^{\triangleright} \vdash V = \ldots = E[(\lambda d.0)U] = \ldots = P(\lambda d.0).$$

Putting the first evaluation step and the two derivations together, we get that

$$\lambda_v\text{-S}^{\triangleright} \vdash (\lambda x.P(\sigma x.0))0 = P(\lambda d.0).$$

Finally, it is easy to see that this equation is also safe. Unfortunately, if $P(Q)$ does not terminate, we cannot prove this equality. We return to this point in the last section.

The final example differs from the previous ones in several ways. Thus far, the programs have been of first- or second-order flavor and have not dealt with self-referential assignments and values. Now we drop all these restrictions. The program that we treat is an imperative version of the recursion operator. The usual way of realizing recursive function definitions in the λ-calculus framework is based on self-application. That is, if the functional F defines a function f by self-reference:

$$(fx) = (Ff)x,$$

then f is represented in Λ_v as

$$f \stackrel{df}{\equiv} \mathsf{Y}_v F.$$

Y_v is the call-by-value recursion combinator [22, 26]. It is defined as

$$\mathsf{Y}_v \stackrel{df}{\equiv} (\lambda f.(\lambda x.(\lambda g.f(\lambda x.(gg)x))(\lambda g.f(\lambda x.(gg)x))))$$

and satisfies the fixpoint equation

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

In real systems, however, recursive functions are not implemented through self-application. Instead, these implementations use self-reference of function variables. In Λ_σ, this can be expressed as a different version of the Y_v-combinator. This imperative $\mathsf{Y}_!$-combinator also takes a defining functional as an argument, then sets up a new variable binding, and finally assigns to this variable a function that refers to this variable:

$$\mathsf{Y}_! \stackrel{df}{\equiv} \lambda f.(\lambda g.(\sigma g.g)(\lambda x.fgx))0.$$

Although this imperative recursion combinator is a generally accepted means for implementing recursion [9, 15, 27], its correctness proof has been ignored. With the λ_v-S-calculus, it is not only possible to express the combinator, but it is also easy to verify the fixpoint property of $\mathsf{Y}_!$ with a simple calculation:

$$\begin{array}{rcll}
\lambda_v\text{-S}^{\triangleright} \vdash \mathsf{Y}_! Fx & = & (\lambda g_\sigma.(\sigma g_\sigma.g_\sigma)(\lambda x.Fg_\sigma x))0x & \text{by } (\beta_v) \\
& = & (\sigma \bullet^l .0^l)(\lambda x.F0^l x)x & \text{by } (C3) \\
& = & (\lambda x.F0^l x)^l x & \text{by } (C5) \\
& = & (\lambda x.F(\lambda x.F0^l x)^l x)x & \text{by } (C4) \\
& = & F(\lambda x.F0^l x)^l x & \text{by } (\beta_v) \\
& = & F((\sigma \bullet^l .0^l)(\lambda x.F0^l x))x & \text{by } (C5) \\
& = & F((\lambda g_\sigma.(\sigma g_\sigma.g_\sigma)(\lambda x.Fg_\sigma x))0)x & \text{by } (C3) \\
& = & F(\mathsf{Y}_! F)x. & \text{by } (\beta_v)
\end{array}$$

Since this derivation yields a theorem between Λ_σ-terms, the theorem automatically is an operational equivalence.

Naturally, we would also like to show that the imperative recursion combinator produces a *minimal* fixpoint. This corresponds to our intuitive understanding of recursion and would establish an equivalence between Y_v and $\mathsf{Y}_!$. However, with our simple syntactic calculus, it is impossible to state or to prove such a statement. This is another drawback of our work, and we address this point in the last section.

9 Summary and Perspective

The goal of our work was to extend the λ_v-calculus to a calculus about a higher-order programming language with imperative assignment statements. For the implementation of our goal, we formalized an abstract machine semantics for an extended Λ-programming language with an assignment expression and discussed the implied operational equivalence as the basis of correctness proofs and program transformations. Next, we transformed the abstract machine semantics into a more text-oriented rewriting system and derived a calculus-like theory from it. A central advantage of both over conventional models of higher-order languages is the avoidance of the infamous "garbage" and "free list" problem [7, 18]. The proof of the correctness theorem for the λ_v-S-calculus revealed that the relationship between the extended calculus and the underlying programming language is of complex nature, yet, with a short series of examples, we were able to show that this does not impair the calculus's capabilities. In summary, the equational theory is a good basis for an almost algebraic style of reasoning about imperative programs.

During the feasibility study, we indicated some problems with our calculus. Put into slogan form, the λ_v-S-calculus is a calculus in Church's sense, but not in Scott's. More technically, the theory is a syntactic system and lacks semantic ordering relationships, limits, and an induction principle for infinite computations. One possible alternative without these problems is to work directly with operational equivalence. This line of research is being explored by Mason and Talcott [12, 13, 14].[5] Their basic idea is that operational equivalence already is a congruence relation, and that all we need to develop are strategies for reasoning with this relation. To this end, Mason and Talcott develop a set of primitive *laws*, which are frequently used operational equivalences. With these laws—there are some sixty for first-order Lisp—it is possible to prove the correctness of many interesting transformations. Since operational equivalence is closely related to operational approximation, it is also straightforward with this approach to establish a fixpoint induction principle about infinite computations.

Although convincing at first glance, the direct approach is also loaded with problems. First, given that operational equivalence is a congruence relation, but that the value of a program is often operationally distinct from the program, it is impossible to use the operational equivalence relation for evaluating programs. Second, as a consequence of this, there cannot be a finite number of laws that yield all equivalence proofs; the collection of laws will have to be expanded over time. Third, and probably most important, unlike programming language calculi, operational equivalence relations are sensitive to the principal building blocks of the underlying programming language. As a consequence, theorems about some functional programs, for example, cannot be re-used in an imperative setting: such theorems have to be re-established from scratch. From this perspective, the calculus is simply interesting because it identifies equalities that hold across a broad class of programming languages. Working with both the calculus and the operational equivalence should facilitate the work with multi-paradigm programming languages.

The development of the λ_v-S-calculus is only a first step towards an improved understanding of imperative programming languages. The problems that it poses do not have trivial solutions. Still, we believe that the system is a good starting point for further research about the nature of assignment operations and state variables in programming languages.

Acknowledgement. Sussman and Steele's paper on Scheme as an interpreter for *extended* λ-calculus [25] stimulated us to consider how such a calculus for assignments in higher-order languages could be constructed. Discussions with Robert Hieb provided the key insight for the elimination of D-applications from our previous report. Carolyn Talcott and Albert Meyer clarified some of the strengths and weaknesses of calculi systems. Bruce Duba, Shinnder Lee, Uday Reddy, Carolyn Talcott, and Mitchell Wand provided helpful comments on earlier drafts of the paper. The anonymous referees pointed out several inaccuracies and suggested many improvements to the presentation.

[5] Their work is mostly concerned with Lisp-like languages and relies on an operational semantics in the style of our CS-machine with the garbage problem. For the purpose of this comparison, we simply ignore these differences.

References

1. AÏT-KACI, H. AND R. NASR. Logic and inheritance. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, 219–228.
2. 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.
3. BÖHM, C. Alcune proprietà della forma $\beta\eta$-normali nel λ-K-calcolo. Pubblicazioni dell' Istituto par le Applicazioni del Calcolo, Rome, 1968.
4. FELLEISEN, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. dissertation, Indiana University, 1987.
5. 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.
6. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control, *Theor. Comput. Sci.***52**(3), 1987, 205–237.
7. 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.
8. LANDIN, P.J. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM* **8**(2), 1965, 89–101; 158–165.
9. 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.
10. LANDIN, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.
11. LANDIN, P.J. The mechanical evaluation of expressions. *Comput. J.* **6**(4), 1964, 308–320.
12. MASON, I.A. Equivalences of first-order Lisp programs. In *Proc. Symposium on Logic in Computer Science*, 1986, 105–117.
13. MASON, I.A. *The Semantics of Destructive Lisp.* Ph.D. dissertation, Stanford University, 1986.
14. MASON, I.A. AND C. TALCOTT. Programming, transforming, and proving with function abstractions and memories. Unpublished manuscript. 1988.
15. 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.
16. MEYER, A.R. Thirteen puzzles in programming logic. In *Proc. Workshop Formal Software Development: Combining Specification Methods*, edited by D. Bjørner. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 1984.
17. MEYER, A.R. Semantical Paradigms. In *Symp. on Logic in Computer Science*, 1988, 236–255.

18. MEYER, A. R. AND K. SIEBERT. Towards a fully abstract semantics for local variables. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 191–203.

19. MORRIS, J.H. *Lambda-Calculus Models of Programming Languages.* Ph.D. dissertation, MIT, 1968.

20. PLOTKIN, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.

21. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

22. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.

23. SETHI R. Circular expressions: elimination of static environments. *Sci. Comput. Program.* **1**, 1982, 203–222.

24. STOY, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* The MIT Press, Cambridge, Mass., 1981.

25. SUSSMAN, G.J. AND G.L. STEELE JR. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

26. TALCOTT, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* Ph.D. dissertation, Stanford University, 1985.

27. TURNER, D.A. A new implementation technique for applicative languages. *Softw. Pract. Exper.* **9**, 1979, 31–49.

Appendix 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-S-calculus. Furthermore, traditional techniques are sufficient in both cases for proving the statements on the sub-calculus [2:59–63, 21:136–142]. Whenever a proof is a simple (often tedious) re-working of a conventional proof, we omit it. We first treat the Consistency Theorem.

Theorem 6.2. *(Consistency)*

(i) The notion of reduction **s** *is Church-Rosser.*

(ii) The **s***-computation satisfies the diamond property.*

Proof. Let us assume the correctness of point (i). Then we need to show that whenever $M \rhd_s L_i$ for $i = 1, 2$ and $L_1 \not\equiv_{lab} L_2$, there exists a term N such that $L_i \rhd_s N$. We proceed by case analysis on $M \rhd_s L_1$:

1. $M \equiv \&(\lambda x_\sigma.P)V \rhd \&P[x_\sigma := V^l] \equiv L_1$ 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 are only applied to complete programs. Hence, it follows that

   $$\&L_1 \equiv_{lab} P[x_\sigma := V^m] \equiv_{lab} P[x_\sigma := V^l] \equiv_{lab} \&L_2.$$

   Since we identify label-equivalent 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 $P \longrightarrow\!\!\!\!\!\rightarrow_s P_2$.
   But then

   $$\&(\lambda x_\sigma.P_2)V \rhd \&P_2[x_\sigma := V^l]$$

   and the Substitution Theorem (6.1) induces

   $$P[x_\sigma := V^l] \longrightarrow\!\!\!\!\!\rightarrow_s P_2[x_\sigma := V^l].$$

   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 $V \longrightarrow\!\!\!\!\!\rightarrow_s V_2$.
This second alternative requires a variant of the Substitution Theorem, namely, that

$$P[x_\sigma := U^l] \longrightarrow\!\!\!\!\!\rightarrow_s P[x_\sigma := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_s V.$$

Given this, the rest is the same as in subcase (a).

2. $M \equiv \&(\sigma\, \bullet^l.P)V \triangleright \&P[\bullet^l := (V[\bullet^l := V^l])^l] \equiv L_1$.
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:

$$\begin{aligned} P[\bullet^l := U^l] &\longrightarrow\!\!\!\!\!\rightarrow_s P[\bullet^l := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_s V, \\ P[\bullet^l := V^l] &\longrightarrow\!\!\!\!\!\rightarrow_s Q[\bullet^l := V^l] \text{ if } P \longrightarrow\!\!\!\!\!\rightarrow_s Q. \end{aligned}$$

3. $M \equiv \&(UV^l) \triangleright \&U(V[\bullet^l := V^l]) \equiv L_1$.
For this case we need the property

$$U[\bullet^l := U^l] \longrightarrow\!\!\!\!\!\rightarrow_s V[\bullet^l := V^l] \text{ if } U \longrightarrow\!\!\!\!\!\rightarrow_s V.$$

Otherwise, it does not add any new problems.

4. $M \equiv \&V^l \triangleright \&V[\bullet^l := V^l] \equiv L_1$. Like 3.

5. $M \longrightarrow\!\!\!\!\!\rightarrow_s L_1$.
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, $M \longrightarrow\!\!\!\!\!\rightarrow_s L_2$, and the consequence holds because of the assumed Church-Rosser property for **s**.

In order to complete the proof, we must still show the following four properties of substitution and labeled-value substitution:

$$\begin{aligned} P[x_\sigma := U^l] &\longrightarrow\!\!\!\!\!\rightarrow_s P[x_\sigma := V^l] \\ P[\bullet^l := U^l] &\longrightarrow\!\!\!\!\!\rightarrow_s P[\bullet^l := V^l] \\ P[\bullet^l := V^l] &\longrightarrow\!\!\!\!\!\rightarrow_s Q[\bullet^l := V^l] \\ U[\bullet^l := U^l] &\longrightarrow\!\!\!\!\!\rightarrow_s V[\bullet^l := V^l] \end{aligned}$$

if $P \longrightarrow\!\!\!\!\!\rightarrow_s Q$ and $U \longrightarrow\!\!\!\!\!\rightarrow_s V$. 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 6.1). 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-S-calculus to the consistency problem of the sub-calculus, *i.e.*, the Church-Rosser property of **s**. This proof, in turn, is an application of Tait/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 **s**. For the proof of the Standardization Theorem we also define a notion of the size of a parallel reduction.

Definition Appendix.1 (*The parallel reduction* $\xrightarrow[1]{}$) The *parallel reduction* over Λ_S is denoted by $\xrightarrow[1]{}$ and is defined as follows, where $s_{M \xrightarrow[1]{} N}$ or just s is the function which measures the *size of the derivation* $M \xrightarrow[1]{} N$ and $n(x, M)$ is the *number of free occurrences of* x *in* M:

1. $M \xrightarrow[1]{} M, \quad s = 0$

2. if $M \xrightarrow[1]{} M'$, $N \xrightarrow[1]{} N'$, $U \xrightarrow[1]{} U'$, and $V \xrightarrow[1]{} V'$ then

$$\begin{aligned}
(\lambda x.M)V &\xrightarrow[1]{} M'[x := V'], \\
s &= s_{M \xrightarrow[1]{} M'} + n(x, M') s_{N \xrightarrow[1]{} N'} + 1 \\
((\lambda x_\sigma.M)V)N &\xrightarrow[1]{} (\lambda x_\sigma.M'N')V', \\
s &= s_{M \xrightarrow[1]{} M'} + s_{N \xrightarrow[1]{} N'} + s_{V \xrightarrow[1]{} V'} + 1 \\
U((\lambda x_\sigma.M)V) &\xrightarrow[1]{} (\lambda x_\sigma.U'M')V', \\
s &= s_{M \xrightarrow[1]{} M'} + s_{U \xrightarrow[1]{} U'} + s_{V \xrightarrow[1]{} V'} + 1 \\
((\sigma X.M)V)N &\xrightarrow[1]{} (\sigma X.M'N')V', \\
s &= s_{M \xrightarrow[1]{} M'} + s_{N \xrightarrow[1]{} N'} + s_{V \xrightarrow[1]{} V'} + 1 \\
U((\sigma X.M)V) &\xrightarrow[1]{} (\sigma X.U'M')V', \\
s &= s_{M \xrightarrow[1]{} M'} + s_{U \xrightarrow[1]{} U'} + s_{V \xrightarrow[1]{} V'} + 1 \\
(Ux_\sigma)M &\xrightarrow[1]{} ((\lambda x_\lambda.U'x_\lambda M')x_\sigma), \\
s &= s_{M \xrightarrow[1]{} M'} + s_{U \xrightarrow[1]{} U'} + 1 \\
V(Ux_\sigma) &\xrightarrow[1]{} ((\lambda x_\lambda.V'(U'x_\lambda))x_\sigma), \\
s &= s_{U \xrightarrow[1]{} U'} + s_{V \xrightarrow[1]{} V'} + 1 \\
(UW^l)M &\xrightarrow[1]{} ((\lambda x_\lambda.U'x_\lambda M')W^l), \\
s &= s_{M \xrightarrow[1]{} M'} + s_{U \xrightarrow[1]{} U'} + s_{W \xrightarrow[1]{} W'} + 1 \\
V(UW^l) &\xrightarrow[1]{} ((\lambda x_\lambda.V'(U'x_\lambda))W^l), \\
s &= s_{U \xrightarrow[1]{} U'} + s_{V \xrightarrow[1]{} V'} + s_{W \xrightarrow[1]{} W'} + 1 \\
V^l M &\xrightarrow[1]{} ((\lambda v.vM')V'^l), \\
s &= s_{M \xrightarrow[1]{} M'} + s_{V \xrightarrow[1]{} V'} + 1
\end{aligned}$$

3. if $M \xrightarrow[1]{} M'$, $N \xrightarrow[1]{} N'$, and $V \xrightarrow[1]{} V'$ then

$$\begin{aligned}
(\lambda x.M) &\xrightarrow[1]{} (\lambda x.M'), \ s = s_{M \xrightarrow[1]{} M'} \\
(\sigma X.M) &\xrightarrow[1]{} (\sigma X.M'), \ s = s_{M \xrightarrow[1]{} M'} \\
V^l &\xrightarrow[1]{} V'^l, \ s = s_{V \xrightarrow[1]{} V'} \\
MN &\xrightarrow[1]{} M'N', \ s = s_{M \xrightarrow[1]{} M'} + s_{N \xrightarrow[1]{} N'}
\end{aligned}$$

It follows from this definition that a value can only parallel reduce to another value. Moreover, it is obvious that parallel reduction is an extension of the single step s-reduction

and is a subset of its transitive closure:

$$\mathbf{s} \subset \longrightarrow_s \subset \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow \subset \longrightarrow\!\!\!\!\!\!\rightarrow_s .$$

Next we show that unlike the s-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, we simultaneously prove a statement that is needed for the Standardization Theorem, namely, that this new reduction is shorter than the one from $(\lambda x.M)N$ to $M'[x := N']$:

Lemma Appendix.1 *Suppose* $M \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow M'$, $N \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow N'$, *and* N *is a value. Then the following holds:*

$$M[x := N] \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow M'[x := N'] \text{ and } s_{M[x:=N]\xrightarrow[1]{}\!\!\!\!\!\!\rightarrow M'[x:=N']} < s_{(\lambda x.M)N\xrightarrow[1]{}\!\!\!\!\!\!\rightarrow M'[x:=N']}.$$

For the actual proof, we need similar lemmas for the other kind of primitive parallel reductions (group 2.), but, given their simplicity, we omit them. Everything is now in place to state and prove the diamond lemma for parallel reduction.

Lemma Appendix.2 *The relation* $\xrightarrow[1]{}\!\!\!\!\!\!\rightarrow$ *satisfies the diamond property, i.e., if* $M \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow L_i$ *then there exists an* N *such that* $L_i \xrightarrow[1]{}\!\!\!\!\!\!\rightarrow N$ *for* $i = 1, 2$.

From this lemma, the Church-Rosser property of s follows.

Corollary Appendix.3 *The notion of reduction* s *is Church-Rosser.*

Proof. Since $\longrightarrow\!\!\!\!\!\!\rightarrow_s$ is the transitive closure of s, it is also the transitive closure of $\xrightarrow[1]{}\!\!\!\!\!\!\rightarrow$. As a result, the reduction $\longrightarrow\!\!\!\!\!\!\rightarrow_s$ inherits the diamond property from the parallel reduction relation. □

With the Church-Rosser theorem in place, we can tackle the Standardization Theorem.

Theorem 6.4. *(Standardization)*

(i) $M \longrightarrow\!\!\!\!\!\!\rightarrow_s N$ *if and only if there is an* SR*-sequence* $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$;

(ii) $M \rhd_s^* N$ *if and only if there is an* SC*-sequence* $L_1, \ldots, L_n$ *with* $M \equiv L_1$ *and* $L_n \equiv N$.

Proof. For both parts, the direction from right to left is trivial. The other direction needs 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 $\longrightarrow_s$-steps is replaced by a sequence of steps using the parallel reduction $\xrightarrow[1]{}\!\!\!\!\!\!\rightarrow$. This follows from the above mentioned property that the parallel reduction relation is an extension of the s-reduction. Then we iteratively transform the parallel reduction sequence into an SR-sequence. The basis for this step is developed in Lemma Appendix.5.

(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 an SR-sequence from M to N. Otherwise, there is at least one computation step and we must have the following situation:

$$L_1 \longrightarrow\!\!\!\!\!\rightarrow_s L_m \triangleright L_{m+1} \triangleright_s^* L_n.$$

Again by (i), we can form an SR-sequence $K_1, \ldots, K_l$ for the reduction $L_1 \longrightarrow\!\!\!\!\!\rightarrow_s L_m$ 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 SR-sequence 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

$$L_1 \longmapsto_{ss}^* K_j \longrightarrow\!\!\!\!\!\rightarrow_s L_m \equiv K_l \triangleright L_{m+1} \triangleright_s^* L_n$$

and it is not the case that $K_j \longmapsto_{ss} K_{j+1}$. At this point, we can employ the proof of the Consistency Theorem, part (ii) and interchange the computation step with the reduction sequence:

$$L_1 \longmapsto_{ss}^* K_j \triangleright P \longrightarrow\!\!\!\!\!\rightarrow_s L_{m+1} \triangleright_s^* L_n,$$

for some term P. But this can be recast as

$$L_1 \overset{\triangleright}{\longmapsto}_{ss}^{+} P \triangleright_s^* L_n$$

and, because there is one less computation step in the sequence from P to L_n, an application of the inductive hypothesis produces an SC-sequence for this second half of the reduction. Together with the first half, this yields the desired SC-sequence from M to N. □

For completeness, we include a precise formulation of the prefix-property for SR-sequences.

Lemma Appendix.4 *If $N_1, \ldots, N_k$ is an SR-sequence where N_k is a computational redex, then there exists a j, $1 \leq j < k$ such that for all $i, 1 \leq i < j, N_i \longmapsto_{ss} N_{i+1}$, and for all i, $j \leq i < k, N_i \longrightarrow_s N_{i+1}$ and it is not the case that $N_i \longmapsto_{ss} N_{i+1}$.*

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 Appendix.5 *If $M \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_1$ and $N_1, \ldots, N_j$ is an SR-sequence then there exists an SR-sequence $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N_j$.*

As with the preceding lemmas, the proof of this lemma is a mechanical extension of the corresponding traditional proof.