Yale University
Department of Computer Science

Calculi for Functional Programming Languages with Assignment

Daniel Eli Rabin

Research Report YALEU/DCS/RR-1107
May 1996

ABSTRACT
Calculi for Functional Programming Languages with Assignment

Daniel Eli Rabin
1996

Pure functional programming and imperative programming appear to be contradictory approaches to the design of programming languages. Pure functional programming insists that variables have unchanging bindings and that these bindings may be substituted freely for occurrences of the variables. Imperative programming, however, relies for its computational power on the alteration of variable bindings by the action of the program.

One particular approach to merging the two design principles into the same programming language introduces a notion of assignable variable distinct from that of functionally bound variable. In this approach, functional programming languages are extended with new syntactic constructs denoting sequences of actions, including assignment to and reading from assignable variables. Swarup, Reddy and Ireland have proposed a typed lambda-calculus in this style as a foundation for programming-language design. Launchbury has proposed to extend strongly-typed purely functional programming in this style in such a way that assignments are carried out lazily.

In this dissertation we extend and correct the work of Odersky, Rabin, and Hudak giving an *untyped* lambda-calculus as a formalism for designing functional programming languages having assignable variables. The extension encompasses two untyped calculi sharing a common core of syntactic constructs and rules corresponding to the similarities in all the typed proposals, and differing in ways that isolate the semantic differences between those proposals. We prove the consistency and suitability for implementation of the two calculi. We adapt a technique of Launchbury and Peyton Jones to provide a safe type system for the calculi we study, thus providing a correct replacement for systems, now known to be flawed, originally proposed by Swarup, Reddy, Ireland, Chen, and Odersky.

Calculi for Functional Programming Languages with Assignment

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Daniel Eli Rabin

Dissertation Director: Professor Paul R.Hudak

May 1996

To the memory of Richard Howard Trommer

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgments

I began the research described in this dissertation in early 1992 in collaboration with Martin Odersky and under the supervision of Paul Hudak. Previous states of the work have been reported in the conference paper [Odersky *et al.*, 1993] (also available as [Odersky *et al.*, 1992]) and in more detail in the technical report [Odersky and Rabin, 1993]. I have included material from these previous works as appropriate, correcting technical errors large and small, and revising the notation and exposition to fit the expanded scope of the dissertation.

It is safe to say that, without Martin's enthusiasm for our interactive work and his tutelage in the often-frustrating craft of research, this dissertation could not possibly have come into existence. My supervisor, Paul Hudak, acted as an unfailing cheerleader and collaborator during the intial years, and provided sound advice (which I sometimes ignored, to my sorrow).

Paul, Martin, and my other committee members, Young-il Choo, Mark P. Jones, and Uday S. Reddy, have been patient throughout the long process of finishing this dissertation. I thank them for the improvements that their comments and suggestions have brought about.

Paul Smolensky helped my decide to start graduate school, and John D. Corbett helped me enormously in finishing.

I gratefully acknowledge the financial assistance afforded by an IBM Graduate Fellowship during the academic years 1992–94. My parents, William and Alice Rabin, provided the financial support necessary to finish this work in 1994–96.

Palo Alto, California
April 1996

1
Introduction

The design of certain functional programming languages (Haskell [Hudak *et al.*, 1992] and Miranda [Turner, 1985] among them) is renowned for its rejection of assignment to variables as a basis for specifying computations. This rejection stems both from the desire to free the programmer from low-level implementation concerns and from the urge to provide a simple, powerful theory of program transformation. To its proponents, the avoidance of variable assignment earns for these languages the taxonomic designation of *pure* functional programming languages.

The absence of variable assignment, however, mitigates the usefulness of pure functional programming languages in at least two ways. First, although the freedom from the need to express low-level implementation concerns has great force in speeding the prototyping of programs, this force has a dark side in the inability to control these details for engineering purposes. This concern recapitulates the lingering mistrust of compilers on the part of assembly-language programmers, the difference being that functional-programming compiler technology has not (yet) attained the triumphal level of object-code efficiency that characterizes many of today's conventional-language compilers.

A second factor mitigating the virtue of purity is that many programming problems are *about* changing state. The rise in popularity of object-oriented programming (stemming ultimately from the design of Simula [Birtwistle *et al.*, 1973]) attests to the usefulness of a programming paradigm in which program objects encapsulating local states simulate the behavior of the real-world objects that are the subject of the computer program. It is possible for programs written in pure functional programming languages to simulate stateful computations by means of transition functions operating on an explicitly-represented state. However, it is difficult for such a functional program to represent its use of state as simply as programs in a conventional language in which the state is implicit and ever-present.

In the last few years there have been several proposals for introducing assignment into pure functional programming languages without compromising their essential susceptibility to formal reasoning. Although these proposals differ in detail, an identifiable subset of them revolves around a common set of design principles: we add to the syntax of the pure functional programming language a distinct set of terms for representing *commands*, the command terms are constrained to be meaningful only when chained together in a linear sequence, and observations of the store are expressed as commands, not values. These principles, properly implemented, are sufficient to preserve referential transparency in the extended language.

The published proposals are variously presented as untyped or typed lambda-calculi, or informally presented as extensions to functional programming languages. In this dissertation we bring a unity of approach to several of these proposals. We use the techniques of untyped lambda-calculi to develop an untyped operational semantics for pure functional programming with assignment. We give extended lambda-calculi that model two variants of our target class of programming-language proposals; we prove fundamental properties of these calculi, and we introduce a type system that excludes programs having run-time failures owing to misuse of the store constructs.

The most immediate antecedent of the current work is the work of Swarup, Reddy, and Ireland on the Imperative Lambda Calculus (ILC) [Swarup *et al.*, 1991; Swarup, 1992]. This work is presented formally using both operational and denotational techniques, but it is essentially a typed system. In contrast, we follow our previous work on λ_{var} [Odersky *et al.*, 1993; Odersky and Rabin, 1993] in adopting the operational-semantics techniques of untyped lambda-calculus as our formal basis; this approach allows us to study computational issues and type-safety issues separately. We further exercise our approach by extending it to cover the proposal of Launchbury [Launchbury, 1993] that store transformations should be executed on demand just as functional computations are in non-strict functional programming languages; our extension represents the first fully formal basis we know of for Launchbury's proposal.

In addition to bringing contrasting techniques to bear on these existing proposals, the present dissertation also presents some corrections to previous work. We cite our own mistakes first: the proof of the Church-Rosser and standardization properties in [Odersky and Rabin, 1993] contains a subtle flaw that we correct here in an interesting way. In addition, we modify the calculi presented in our earlier work to allow a more

eager store		*lazy store*	
informal:		*informal:*	L 1993
formal, untyped:	ORaH 1993, **here**	*formal, untyped:*	**here**
formal, typed	SReI 1991, CO 1994	*formal, typed*	**here**

C = K. Chen, I = E. Ireland, L = J. Launchbury, O = M. Odersky, Ra = D. Rabin, Re = U. Reddy, S = V. Swarup. Contributions of the present dissertation are identified by the word **here**. Work corrected here is underlined.

Table 1.1: Contributions of the present dissertation to the design of functional programming languages with assignment.

flexible relationship between locality of names and locality of store-computations. We also have a correction to make to previous type systems proposed for both ILC and λ_{var}[Chen and Odersky, 1994]: these published type systems are now known to violate the essential property that reductions preserve typings. The error is classical, having been noticed by Reynolds almost two decades ago in [Reynolds, 1978]. In this dissertation, we adapt a type system proposed in [Launchbury and Peyton Jones, 1994] to provide a safe type system for both the variants of λ_{var} treated here.

Table 1.1 summarizes the contributions of this dissertation.

We now proceed to introduce the subject of this dissertation in greater detail. We give historical background in two sections, first treating our immediate motivation in the addition of assignment to functional programming languages, then reviewing work on the complementary approach that starts with imperative programming and attempts to analyze programs for functionally pure substructure. We mingle the treatment of our contributions with these two historical sections. We then list the important methodologies we employ, review related work not mentioned earlier, and finally give a chapter-by-chapter overview of the remainder of the dissertation.

1.1 The apparent incompatibility of assignment and substitution semantics

We will first make a naive attempt to combine assignment and functional programming in one language. This attempt will fail, but it will serve to define the problem solved by the class of language designs that forms the subject of this dissertation. This naive approach adds variable assignment expressions (:=) and command sequencing (;) to the underlying functional language; we will show that the resulting language is already *referentially opaque*—substituting variables by their bindings does not necessarily preserve the meaning of a program. To take a simple example, we expect the expression

$$\mathbf{let}\, x = 0\, \mathbf{in}\, x := x+1; x$$

to have the value 1. Likewise, we expect the expression

$$\mathbf{let}\, x = 0\, \mathbf{in}\, x := x+1;\ x := x+1;\ x \tag{1.1}$$

to have the value 2.

Now we ask the classic question: What value should the semantics of such a programming language assign to the following expression, which takes advantage of the abstraction facility of the functional component of the language to abbreviate a repeated command?

$$\mathbf{let}\, x = 0\, \mathbf{in}\, (\mathbf{let}\, y = (x := x+1)\, \mathbf{in}\, y;\ y;\ x) \tag{1.2}$$

If we look at this expression from the imperative-programming point of view, we might reason that the updating of x takes place at the point where the assignment expression is bound to y, and hence that the value of the entire expression is 1. On the other hand, we would like our language to be referentially transparent—that is, it ought to be valid to substitute $x := x+1$ for y to obtain (1.1) again, which has the value 2. We now have a conflict: two perfectly reasonable, but different, ways of evaluating this small program yield different answers. [1] If we had given the semantics of this language via a formal calculus, we would say that the calculus is *non-confluent* or lacks the *Church-Rosser property*. Informally, we can just state that the semantics of the language is inconsistent with the semantics of function application defined by substitution (which is what we generally mean by referential transparency).

This conflict lies at the heart of the divergent evolution of functional and imperative languages. If we ban imperative constructs, we can have both referential transparency and confluence (this is the Church-Rosser theorem of classical lambda-calculus [Barendregt, 1984]), but if we insist on retaining assignment we appear to lose both. It is thus an important question whether this conflict can be resolved

The answer is that it can. Just as imperative constructs supply a suitable villain for the functional point of view, so also the underspecified order of evaluation provides a villain for the imperative point of view. If a formal calculus requires that evaluation of a program takes place in a certain order, consistency can be re-established. This is exactly what a denotational semantics for an imperative language does. It is even possible to carry out this approach for a higher-order language, especially if the language uses the *call-by-value* evaluation order, in which argument expressions are always evaluated before the function applications of which they are part. Scheme [Clinger and Rees, 1991] and Standard ML [Milner *et al.*, 1990] are two examples of languages designed in this fashion. These languages, however, are outside the scope of this dissertation because they are not referentially transparent, even with respect to the call-by-value β-rule used in [Plotkin, 1975].

1.2 Approaches from functional programming

Section 1.1 depicts a dichotomous choice in programming-language design: it seems we can obtain either full referential transparency, or have assignment in the language, but not both. Recent research approaching the issue from the point of view of functional programming, however, has shown this dichotomy to be only an apparent one. We now give an overview of this line of research, whose foundations in the lambda-calculus are the subject of this dissertation. We treat the complementary approach, from the point of view of Algol-like languages, in Section 1.3.

The point of attack for all the proposals we survey here is against the tacit assumption in Section 1.1 that, because a notion of assignable store requires a sequencing of commands, this order of evaluation must somehow be derived from the evaluation order already present in the functional language. If sequences of commands affecting the store were somehow disentangled from expressions denoting values, the example (1.2) given above would become meaningless, since that example contains assignment commands in a position in which a pure functional language would require a value expression.

1.2.1 State-transformer monads

One of the first proposals along these lines was Wadler's proposal [Wadler, 1990a; Wadler, 1992a] to take inspiration from the theoretical work of Moggi [Moggi, 1989; Moggi, 1991] and adopt the category-theoretic notion of *monad* as a program-construction framework within functional programming languages. For the purposes of the present informal exposition, a monad on some underlying domain consists of a type constructor that takes any type to a type of "generalized commands" that yield that type. There is an operator $\triangleright$ for sequencing such generalized commands and an operator $\uparrow$ for embedding values as trivial generalized commands. These operators are required to obey algebraic laws similar (but not identical) to the unit and associative laws of a monoid.

The symbols $\triangleright$ and $\uparrow$ that we have just introduced for the monad operators are the same symbols that we use in the body of the dissertation (starting in Section 2.3) for syntactic constructs with a monad-like interpretation. We adopt these symbols here for consistency. Although we treat them informally as functions in a

[1] A situation like this almost occurs in the programming language Standard ML, which is defined by a precise semantics in [Milner *et al.*, 1990]. That semantics specifies the interpretation of (1.2) yielding the answer 1 by way of specifying the order in which evaluation takes place.

programming language, the syntax we later give them as part of formal calculi differs in some respects.[2]

The monad structure is more general than we require—we are only concerned with monads of *state transformers*. As given by Moggi and Wadler, such a monad has its operator and unit element defined (using informal functional-programming notation in the style of Haskell) by

$$\begin{aligned} \textbf{type}\, ST\,\alpha &= S \to S \times \alpha && (1.3)\\ \triangleright &: ST\,\alpha \to (\alpha \to ST\,\beta) \to ST\,\beta && (1.4)\\ f \triangleright g &= \lambda s.\textbf{let}\,\langle s',x\rangle = f\,s\,\textbf{in}\,g\,x\,s', && (1.5)\\ \uparrow &: \alpha \to ST\,\alpha && (1.6)\\ \uparrow x &= \lambda s.\langle s,x\rangle && (1.7) \end{aligned}$$

where the type S represents the type of the store (which will always be invisible to user programs), and s, s' are variables having this type.

This monad would not be interesting if only these generic operators were available. In order to model the creation, update, and reading of assignable variables, we make use of the additional operators

$$\begin{aligned} new &: \alpha \to ST\,(Ref\,\alpha) && (1.8)\\ update &: Ref\,\alpha \to \alpha \to ST\,() && (1.9)\\ read &: Ref\,\alpha \to ST\,\alpha, && (1.10) \end{aligned}$$

where $Ref\ \alpha$ is the type of references to values of another type α.

The operator *new* accepts a value of type α and returns a state-transformer that creates and returns a new reference to that value in the store. The operator *update* returns a state-transformer that carries out a new assignment to an existence reference to a value of type α, and the operator *read* returns a state-transformer for observing the value referred to by a given reference.

These additional operations can be defined in terms of an implementation of the store, but we do not yet require this level of detail. The main point is that the types associated with the store operators enforces the restriction that the commands they denote only have their imperative meaning when placed in a linear sequence using $\triangleright$. It is the imposition of this linear sequence on the accesses to the store that insulates the order of access to the store from the evaluation order of expressions in the language.

In this dissertation we always notate the operator $\triangleright$ as including the bound variable of the function defining its second argument. We also make $\triangleright$ right-associative: $M \triangleright x \cdot (N \triangleright y \cdot P)$ and $M \triangleright x \cdot N \triangleright y \cdot P$ mean the same thing. On the other hand, $(M \triangleright x \cdot N) \triangleright y \cdot P$ is a syntactically distinct expression which is related to the others by the analogue of the associative law for command sequences.

The approach via monads deals with the example (1.2) by distinguishing between commands as state-transformers and their effects. Thus the meaning of (1.2), as re-expressed in terms of state-transformers, is unambiguously 2: the variable y is bound to the state-transformer itself, and the process of carrying out this binding merely constructs the state-transformer without carrying out its effect. The state-transformation is then carried out twice owing to the occurrences of y in a command-sequence context.

Figure 1.1 gives an example of a program in a pure functional programming language extended with an assignment monad. The example function *makeCounter* accepts an initial value (an integer) and initializes a counter variable, returning an object (implemented as a function) that produces a store-transformer for incrementing the count.

Hudak's proposal for mutable abstract data types [Hudak, 1992] is similar to the use of state-transformer monads but less general in intent and somewhat simpler algebraically. Hudak focuses on the relationship between direct-style and continuation-style functional expressions of the mutation of data structures, and on the algebraic construction of accessors and constructors in the two styles for given data types.

[2]The untyped nature of the calculi introduced in this dissertation requires us to incorporate the functional nature of the second operand of $\triangleright$ into the syntax of the operator itself by means of a binding occurrence of a variable. The operator thus always appears in the form $M \triangleright x \cdot N$ in the formal chapters of this dissertation.

$$\boxed{\mathit{mkCounter} \quad = \quad \lambda \mathit{init}.\mathit{new}\ \mathit{init} \triangleright \mathit{count} \cdot \uparrow (\lambda \mathit{incr}.\mathit{read}\ \mathit{count} \triangleright \mathit{old} \cdot \mathit{update}\ \mathit{count}\ (\mathit{old} + \mathit{incr}) \triangleright \mathit{ignore} \cdot \uparrow \mathit{old})}$$

Figure 1.1: The counter-object in monadic style

1.2.2 The Imperative Lambda Calculus

Just now, we shied away from presenting the meaning of the monadic operators *new*, *update*, and *read* in terms of a proposed implementation. Our reluctance derives from the following observation, which motivates the field of formal programming-language semantics. If we define the meaning of a programming construct in terms of a particular implementation, we usually have in mind that many other implementations might also be in accord with our intended semantics. How, then, do we define this accord? Formal semantics answers the question by using precise mathematical abstractions in language definitions, and by judging implementations by whether they fulfill the abstraction.

Whereas the state-transformer monad was introduced informally by Wadler, the Imperative Lambda Calculus (ILC) presented in [Swarup *et al.*, 1991; Swarup, 1992] gives just such a formal semantics for a programming language having both call-by-name abstractions and assignable variables. ILC is a simply-typed lambda-calculus extended with assignment-related constructs. ILC's type system enforces the linear sequencing of store operations in much the same way as the monadic system surveyed in Section 1.2.1 above, but somewhat simpler.

Aside from the move to a formal calculus, ILC makes two notable contributions. The first of these is the adoption of an alphabet of *store-variables* entirely distinct from the alphabet of lambda-bound variables. This new kind of name (informally speaking) denotes a location in the store in *all* contexts. In the jargon associated with the informal semantics of conventional programming languages, ILC store-variable names have no *r-values*. This fact is the key to the referential transparency of ILC.

The second important contribution of ILC to the thread of research under discussion is the introduction of a semantics for *purification* (also called *effect masking* in the literature [Lucassen and Gifford, 1988; Jouvelot and Gifford, 1989]). The semantics of ILC allows the context of an imperative computation to use that computation's result as a functional value. This capability requires that the calculus be able to ensure that the store-using computation neither depends upon nor influences the functional computatation in which the result is to be used. ILC relies upon its type system to guarantee this safety condition.

Unfortunately, the technique used to make this guarantee in [Swarup *et al.*, 1991; Swarup, 1992] has been observed to have a technical flaw (as is pointed out in [Huang and Reddy, 1995]): ILC permits purification under the condition that free variables of the term to be purified are demonstrably safe, but sets of free variables are subject to change under substitution and so the condition is not an invariant of the operational semantics.[3] Nonetheless, this difficulty (which we remedy in this dissertation) should not obscure the important advance into formal reasoning for functional programming with assignent represented by ILC.

1.2.3 The calculus λ_{var}

ILC relies on its type system in an essential way, but the untyped lambda-calculus provides a reasonable computational formalism for functional programming without any recourse to type structure. It thus makes sense to ask whether the untyped lambda-calculus can be extended with assignment constructs to yield a viable formalism for designing a functional programming language with assignments. The calculus λ_{var} introduced in [Odersky *et al.*, 1993; Odersky and Rabin, 1993] answers this question in the affirmative. Rather than using a type system to exclude programs that attempt to purify results containing residual references to a store computation, the untyped calculus λ_{var} forces such programs to yield a runtime error (modeled in the calculus by reductions that get stuck without yielding one of the set of sanctioned *answer* terms).

With a notion of runtime error in the calculus it becomes possible to study the issue of type systems for guaranteeing safe purifiability as separate entities from the underlying computational formalism. Chen and Odersky [Chen and Odersky, 1994] give a type system for λ_{var} based on the general techniques of ILC, but

[3]We give a more detailed exposition of this problem in Chapter 6.

this type system suffers from the same subtle flaw as ILC; the type system we present in Chapter 6 remedies this flaw.

The results of our previous work on λ_{var} are incorporated into this dissertation, but the calculus itself is renamed $\lambda[\beta\delta!eag]$ in the uniform nomenclature introduced here. We also take the opportunity to correct several flaws in the application of proof techniques of the pure lambda-calculus to proving the fundamental properties of λ_{var}; the details are treated in Chapter 3. We also extend the formalism of λ_{var} to account for the variant language discussed in the next subsection.

1.2.4 Lazy store transformers

All the work on lazy functional programming with assignment that we have cited so far makes a tacit assumption that the command sublanguage behaves like commands in a conventional imperative language: commands are executed in order as soon as they are encountered. Launchbury, however, observes in [Launchbury, 1993] that this semantics for command sequences is not the only one possible. The evaluation sequence of a functional language can be lazy or eager (yielding semantically distinct languages); command sequences are subject to a similar choice. Pure lazy functional programs can be understood as carrying out computation to produce a value only in response to a demand for that value; likewise, Launchbury proposes command sequences that are only executed as far as necessary to produce the results that are demanded. Launchbury claims that the resulting style of imperative programming meshes more easily with an ambient lazy functional programming language, and he gives examples to support this claim.

In order to see the distinction introduced by Launchbury, we consider a very short program written in the notation of the calculi we introduce in the formal chapters of this dissertation. In this notation, the construct **pure** marks off a command sequence whose final result is to be returned as a value to the surrounding functional program; the result term itself is marked by the operator $\uparrow$ applied to the final expression in the command sequence. Our example program is

$$\mathbf{pure}(loop; \uparrow 3) \tag{1.11}$$

where *loop* is some command whose execution never ends (we will see later that such terms exist). In a more conventional functional programming language, this example might be notated

```
newstore  (  loop;
             return 3).
```

The result returned by the imperative computation in (1.11) in fact does not depend on that computation. Nevertheless, in ILC and λ_{var} the execution of the program would result in the non-terminating execution of *loop*. In Launchbury's proposal, however, the program yields the value 3, since no demand is ever made for any component of the introduced store. This extra laziness enters quite naturally if one approaches the design of imperative-functional programming languages directly at the level of extending an existing lazy functional language (as is done for example in [Peyton Jones and Wadler, 1993]). In such an approach, it actually takes some extra work to specify that the state-transformers ought to be strict.

The informality of the original presentation of the lazy store-transformer proposal leads us to inquire whether the λ_{var} formalism can be adapted to a different order of command execution. We have carried out this adaptation, and the resulting change in formalism is so minor that the two resulting calculi ($\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$) can be presented in parallel throughout the dissertation with only occasional remarks where the differences are treated. The most significant departure from this parallel structure is reported in Section 5.2.6: a proof technique based on continuation-passing-style translations is not easily transferred from $\lambda[\beta\delta!eag]$ to $\lambda[\beta\delta!laz]$.

1.2.5 Common features

We have seen in this section that a whole family of related proposals for pure functional programming with assignment has arisen. The common features of these proposals are:

- Substitutable (lambda-bound) variables and store-variables are distinct syntactic entities. Store-variables may be a distinct kind of value called a *reference*.

- Commands form a distinct subcategory of expressions. Commands can only be composed in a linear sequence
- Access to the current value of a store-variable or reference is accomplished by a special *command*, not by the use of an expression having a value.

In addition to these features that are common to all of the proposals presented so far, usefulness for programming requires that there be a way of coercing a command into an expression that denotes a value. This coercion must fail if there are dangling references to the command's store or if the command makes reference to another store. ILC and λ_{var} devote explicit, formal attention to this issue; the informal language-based proposals generally concentrate the concern on an opaque primitive function called *newstore* or *run* that carries out the coercion.

In Chapter 2 we will represent these common features in the design of the two lambda-calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. These two calculi reflect these common features in the bulk of their specifications and differ only in their treatment of execution order within command sequences.

1.3 The Algol 60 heritage

The preceding discussion in Section 1.2 recounts the immediate motivation of the present work in the search for a harmonious introduction of assignment constructs into referentially-transparent functional programming languages. This combination of paradigms, however, may also be approached from the imperative-programming point of view, and indeed it has been, dating from the days of the great progenitor, Algol 60 [Naur *et al.*, 1960; Naur *et al.*, 1963]. It is a peculiarity of Algol 60, little imitated in its successors, that its default parameter-passing mechanism is *call-by-name*: procedure invocation is defined by the *copy rule* stating that the procedure invocation has exactly the same meaning as the defining text of the procedure, but with all occurrences of the formal parameter being replaced by the actual parameter supplied to the procedure (as an expression of the language). Of course, the definition requires the renaming of bound variables to avoid clashes, but this requirement easily becomes second nature—the real semantics is in the idea of textual replacement. The copy rule, which is exactly the β-rule of the lambda calculus [Church, 1951] may reasonably be claimed to capture the notion of *referential transparency* which will be characteristic of the language-design proposals we consider in this dissertation.

Algol 60 with call-by-name parameter passing is, in fact, referentially transparent. The semantic ambiguity demonstrated in (1.2) above cannot occur because commands themselves cannot be bound to variables in Algol 60—they can only be abstracted via the procedure-definition mechanism. It is possible to define a procedure whose *body* is the command whose duplication would cause problems, but the definition itself cannot be mistaken for a context in which the incrementing of x should take place. Only in the context of a command sequence is a command defined. This property of Algol 60 is the germ of the common feature of the approaches surveyed in Section 1.2 that separates commands and expressions syntactically, and gives commands their meaning only in the right context. This orthogonality of functional and imperative features of Algol 60 has been refined into a more modern language design by Reynolds in his design of Forsythe [Reynolds, 1988], and it has been studied formally in [Weeks and Felleisen, 1992; Weeks and Felleisen, 1993].

The importance for us of considering the Algol 60 tradition is that it has given rise to a significant alternative approach to the melding of functional and assignment-based programming. Reynolds [Reynolds, 1978] proposed a way of controlling the interference of assignment-based subcomputations within Algol-like programs by means of a syntactic analysis. This concern is analogous to our concern with purification in our functional languages with assignment. Reynolds's proposal, however, has a more symmetrical structure than our language: whereas we distinguish between the inside and outside of a scope of store usage, Reynolds addresses the possibility of mutual interference between (possibly) parallel subexpressions of a program.

The paper [Reynolds, 1978] is also interesting to us because it contains an early exposition of the problem that spoils the type systems given in [Swarup *et al.*, 1991] and [Chen and Odersky, 1994]. The problem was later rectified in the context of Algol-like languages by Reynolds himself by use of intersection types [Reynolds, 1989] and more recently by O'Hearn, Power, Takeyama, and Tennent [O'Hearn *et al.*, 1995] with a type system (SCIR) loosely based on linear logic [Girard, 1987]. This latter type system has been adapted to a revised language based on ILC by Huang and Reddy [Huang and Reddy, 1995]. The type system we present in Chapter 6 is an alternative that more closely follows the original outlines of ILC, although the SCIR-based approach may prove to be more generally applicable.

1.4 Methodology

We have claimed the formalization of several language proposals as a key contribution of this dissertation. We use two main formalisms. First, and most pervasively, we follow [Plotkin, 1975] and model the basic (untyped) computational meaning of programs in each language in terms of an extended lambda-calculus endowed with a reduction relation that models computation. We also use the apparatus of type theory, especially of polymorphic type systems [Girard, 1990; Reynolds, 1974; Milner, 1978], in proving the desirable properties of the proposed type systems with respect to their underlying untyped calculi.

The lambda-calculus serves both as the operational semantics of computation in our programming languages and also as the basis for formal reasoning about these programs. In order to form the basis for a functional language with assignment, a calculus must have several key properties:

- *Church-Rosser property.* The result of a computation must be independent of the order in which reductions are carried out. This property guarantees that terms have a single meaning: the reduction system provides a semantics for the language.
- *Standard reduction order.* There is a deterministic reduction order that always reduces a term to an answer if the term can be reduced to such a normal form. This property makes plausible the claim that the calculus serves as the basis for designing a programming language: it is possible to write a deterministic program to carry out the evaluation. Aside from its implications for automation, standardization simplifies some proofs (such as that of Lemma 5.1.6 in the present dissertation, for example) by allowing the consideration of only a single reduction sequence.
- *Simulation by store-based machine.* There is a faithful simulation of evaluation in the calculus by a calculus that manipulates an explicit store in a single-threaded fashion. This property validates the claim that the imperative features of a calculus can actually be simulated by the imperative features of a machine.

The lambda-calculus methodology requires some change in the usual vocabulary of discussion concerning purification. It is common, in discussing the problem of masking effects in a functional/imperative language, to phrase the central problem in terms of implementation: will the language prevent access to a store location after its thread dies? Can different evaluation orders give different results? In terms of a lambda-calculus, these questions resolve to one question: is the calculus Church-Rosser? If it is, then certainly evaluation order can make no difference in outcome. Furthermore, if we understand a dangling-location reference as a stuck evaluation in the calculus, we see that the Church-Rosser property guarantees that the problem is in the program, not in the choice of evaluation strategy.

The presentation in Chapter 6 uses the standard apparatus of type theory: type systems are given as inference systems, with type derivations as proof trees in the inference system so defined.

1.5 Other related work

We have already surveyed the most closely related literature in the preceding sections. We now survey work that stands outside of our narrowly-defined topic but is related either by attacking similar problems or by use of similar methodology.

The oldest approach to introducing imperative constructs into functional languages is to express the state as an explicit object that is passed around by the program. This is the approach taken by most denotational semantics for imperative languages (see, for example, [Stoy, 1977] or [Schmidt, 1986]). When applied to functional programming, this approach relies on an analysis carried out by the language processor to achieve efficient execution: it must be determined that the use of the state object is actually *single-threaded* and thus that it is safe to implement state mutations via in-place update of data structures. Schmidt [Schmidt, 1985] and Fradet [Fradet, 1991] give syntactic criteria for proving single-threadedness; Guzmán [Guzmán and Hudak, 1990; Guzmán, 1993] gives a type system in which well-typed programs possessing certain designated types are thereby proved to be single-threaded; another approach based on abstract interpretation is presented in [Odersky, 1991]. Wadler has investigated the use of type systems based on linear logic for this purpose [Wadler, 1990b; Wadler, 1991]; several other researchers have also investigated the use of linear logic as a design principal in functional programming languages [Lafont, 1988; Wakeling and Runciman, 1991; Reddy, 1991; Reddy, 1993].

Riecke, along with Viswanathan, [Riecke, 1993; Riecke and Viswanathan, 1995] has approached the construction of a safe type system for roughly the class of languages inhabited by our calculi, but their methodology makes essential use of properties of denotational models and is inherently based on types. Furthermore, the language treated is call-by-value (whereas our calculi are call-by-name). These contrasts to our work place Riecke and Viswanathan's approach in a distinct thread of research.

Not all related research deals directly with the issue of state in pure functional programming: there is a substantial body of work influenced by the designs of Lisp and Scheme. The work of Felleisen, Friedman, and colleagues into the foundations of Scheme-like languages [Felleisen, 1987; Felleisen and Friedman, 1987a; Felleisen and Friedman, 1987b; Felleisen and Hieb, 1992] uses extended lambda-calculi to provide semantics for such languages. Despite the fact that this work deals exclusively with call-by-value languages and does not insist on preserving the reasoning properties of the pure fragment (such as the validity of the call-by-name β-rule), we have adopted much of its methodology in the present work. In a similar vein, Mason and Talcott have investigated the formal semantics of much the same class of programming languages in works such as [Mason, 1986; Mason and Talcott, 1991; Mason and Talcott, 1992a; Mason and Talcott, 1992b].

A recent paper by Sato [Sato, 1994], which is very much in the vein of the work by Mason and Talcott, presents a Scheme-like calculus in which syntactic criteria for the placment of update commands provide referential transparency. Sato's division of reductions into sequential and parallel reductions is reminiscent of the present work's segregation of expressions and commands, but it is not clear whether there is any formal correspondence.

Graham and Kock [Graham and Kock, 1991] present a design for a functional language with assignment. Their design rests on static syntactic detection of opportunities for violation of referential transparency, and they report a formal proof of the desired properties. The language, however, appears not to have higher-order functions, and their syntactic restrictions appear less natural than those for our typed languages; furthermore, the correctness of purification is only proved for a subset of the language.

Further removed from our work on assignment, but still related, is a stream of research on devising constructs for input/output in pure functional programming languages. Input/output shares with assignment the need to institute a sequencing structure for commands: devices used include continuations and streams [Hudak and Sundaresh, 1988] and monads [Wadler, 1992b; Peyton Jones and Wadler, 1993]. Gordon's thesis on the subject [Gordon, 1994] contains a detailed survey of this research. Input/output differs from assignment, however, in its need to represent an activity that is external to the program and not necessarily under its control. Issues of reactivity and synchronization thus arise that have no counterpart in the study of assignment.

1.6 Overview of the dissertation

The contents of this dissertation are organized as follows.

Chapter 2 defines the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ that are studied in rest of the dissertation. The calculi are introduced feature-by-feature both informally in terms of the intended meaning and as formal systems.

The next three chapters, which develop the theory of the untyped calculi in detail, form the core of the thesis. Chapter 3 gives the proofs of the Church-Rosser and standardization theorems for both calculi. Chapter 4 explores the operational-equivalence theory (the intended principles of program equivalence) of the calculi. In Chapter 5 we look at the relationship of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ to other calculi. In the first part of Chapter 5, we introduce the calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ which have explicit stores in the language syntax. The equivalence in operational semantics between these calculi and their forbears substantiates the claim that the latter really axiomatize the use of astore. In the second part of Chapter 5, we prove the important fact that $\lambda[\beta\delta!eag]$ forms a conservative extension of a calculus representing a purely functional programming language; we conjecture that the result also holds for $\lambda[\beta\delta!laz]$.

Chapter 6 complements the completely untyped treatment of the preceding chapters with a presentation of a safe type system for the calculi $\lambda[\beta\delta!\mathit{eag}]$ and $\lambda[\beta\delta!\mathit{laz}]$. This type system is an adaptation of a type system proposed by Launchbury and Peyton Jones [Launchbury and Peyton Jones, 1994]; we prove its safety and remark on the prospect for use of the unadapted type system.

Chapter 7 summarizes the dissertation and suggests further work in the area.

2
Lambda-calculi with assignment

In this chapter we introduce the extended lambda-calculi that form the subject matter of this dissertation. After a brief section that provides for later reference some of the basic mathematical tools we employ in describing the calculi, we build up a description of our calculi with assignment starting from the pure lambda-calculus and adding features incrementally. We concentrate here on motivating and describing the design of these calculi; we leave the mathematical treatment of their properties for the following chapters.

The two calculi we introduce, which correspond to the columns in Table 1.1 on page 2, have almost identical structures: they differ only in one rule. We will thus be able to describe the two calculi in parallel for the greater part of this dissertation, and will treat their differences as the exception rather than the rule.

Our starting point will be the pure lambda-calculus (Section 2.1), which is a pure calculus of functions. Finding this setting too austere to model even the common practice of modern purely functional programming languages, we add primitive functions and data constructors to form the calculus $\lambda[\beta\delta]$ (Section 2.2). With $\lambda[\beta\delta]$ as a point of departure we first add constructs to model sequences of commands. After introducing those command constructs that make sense irrespective of the domain of commands envisioned (Section 2.3), we specialize our attention to stores and assignments by introducing (and axiomatizing) some primitive commands in Section 2.4.

At this point the heart of the matter may seem to be addressed, but there are two important further refinements to be dealt with. First, we address the axiomatization of locally-defined store-variables in Section 2.5, and second we introduce locally-defined *stores* in Section 2.6.1. In respect of our approach to our subject matter from the functional-programming side, we use the term *purification* to denote the introduction of boundaries beyond which a local store is both ineffective and unobservable. It is only in the rules we devise for the treatment of locally-defined stores that the difference between the notion of eager and lazy store-transformations emerges into the design of these calculi.

2.1 Mathematical preliminaries

Our presentation of our lambda-calculi with assignment rests on a large body of well-developed mathematical formalisms. We make particular use of proof techniques from the pure lambda-calculus, for which our main reference is [Barendregt, 1984]. We engage, however, in significant extensions to the pure lambda-calculus; we collect some of the mathematical terms we use for these extensions in the present section.

Languages of terms and reductions

A formal calculus consists of a set of *terms*, called a language, and rules for manipulating the terms. We use the terminology of lambda-calculus in calling these *reduction* rules and the relation between terms that they define *reduction*. We call a subterm that matches the left-hand side of a rule a *redex* and the result of rewriting it a *reduct*. When a term is rewritten according to a reduction rule, any portion of a designated subterm that survives into the rewritten term is called a *residual* of that subterm.

Languages are defined inductively by rules giving basic terms and productions for forming terms from simpler terms. We often have occasion to define a language based on the inductive description of another language. For example, we describe the incremental addition of features to the calculi presented in this chapter by incorporating all the productions of a previously-described language. A further example of this technique is the formation from a language of a language of *contexts* by adjoining a special term [] (the "hole") to the language as a production for the top-level syntactic category of terms; we usually intend to admit only those terms so formed that have exactly one occurrence of the hole.

The language of lambda-calculus

The language of the pure lambda-calculus itself is defined by the productions in Figure 2.1. The pure lambda-calculus contains only variables, abstractions of terms, and applications of one term to another. Although

$$\begin{array}{rcl} x,y & \in & \mathit{Variables} \\ M,N & \in & \mathit{Terms} \end{array}$$

$$\begin{array}{rcll} M & ::= & x & \mathit{variables} \\ & \mid & \lambda x.M & \mathit{abstractions} \\ & \mid & M \bullet N & \mathit{applications} \end{array}$$

Figure 2.1: Syntax of the pure untyped lambda-calculus $\lambda[\beta\delta]$.

the standard notation for application is simple juxtaposition, we deviate from this standard throughout this dissertation because we will have occasion to annotate the application operator, and it is difficult to annotate a blank. We retain the usual lambda-calculus convention that application ($\bullet$) associates the left.

The sole reduction rule of the pure lambda-calculus, β, makes use of the notion of substitution and is given in Figure 2.3.

Bound variables

The abstraction construct $\lambda x.M$ is said to *bind* the variable x, that is, occurrences of x within M but not within any distinct occurrence of $\lambda x.$ are considered to have a unique identity separate from all other variables. This meaning is underscored by considering terms of the lambda-calculus to be distinct only up to the equivalence of *α-renaming*, which permits the renaming the bound variable of an abstraction as long as all the bound occurrences are renamed accordingly. The α-renaming equivalence allows the very convenient *α-renaming convention*, which is that we always pick an exemplar of an equivalence class of terms under α-renaming such that all bound and free variables have distinct names. This convention saves us considerable effort in writing explicit renamings of bound variables whenever we discuss reductions that move terms into the scope of a bound name.

An occurrence of a variable that is not bound is called *free*; we denote the set of free variables of a term M by $\mathit{fv}\ M$. A term M is called *closed* if $\mathit{fv}\ M$ is the empty set.

In the course of this dissertation, we will have occasion to introduce new variable-binding constructs. Such constructs will be subject to the same conventions whether the issue is mentioned or not.

Extended lambda-calculi

We can extend the pure lambda-calculus by introducing term-constructors and reduction rules. Although we seldom use the following terminology, it is convenient for the current discussion to call attention to three possible kinds of syntactic extension:

- *Names.* An extension may introduce new syntactic categories of names, distinct from the variables of pure lambda-calculus.

- *Binding constructs.* As discussed in Section 2.1, constructs may be introduced to delimit scopes for particular names. Each such construct comes with a notion of α-equivalence.

- *Free constructors.* These are term-constructors that are neither names nor binders of names.

The pure lambda-calculus itself has one term-constructor of each type, application ($\bullet$) being the sole free constructor.

An extended calculus may introduce new binding constructs for an existing syntactic category of names.

Substitution

A fundamental operation on terms is *substitution* of a term for all occurrences of a free variable in another term. Substitution forms the foundation of the operational semantics of the lambda-calculus, but it can actually be defined wherever names are used. The notion of substitution is defined along with the notion of *free variable*, which in turn embodies the basic notion of a name-binding construct: the bound name is distinct from any name present in an outer scope.

Definition 2.1.1 (Substitution) *For terms M, N, and variable x in some extended lambda-calculus, the* substitution *of M for x in N, denoted $[M/x]\,N$, is a term defined inductively by the rules*

$$\begin{aligned}
[M/x]\,x &\equiv M \\
[M/x]\,y &\equiv y \;\; (x \not\equiv y) \\
[M/x]\,(\lambda y.N) &\equiv \lambda y.[M/x]\,N \\
[M/x]\,(N_1 \bullet N_2) &\equiv ([M/x]\,N_1) \bullet ([M/x]\,N_2) \\
[M/x]\,T(N_1,\ldots,N_n) &\equiv T([M/x]\,N_1,\ldots,[M/x]\,N_n)
\end{aligned}$$

where T denotes any free syntactic constructor of the extended language, and n denotes its arity.

Definition 2.1.1 makes essential use of the α-renaming convention in the rule for substituting into an abstraction: the convention permits us to avoid mention of a case in which the variable bound by the abstraction has the same name as the variable for which we are substituting. We merely assume that α-renaming has solved the problem for us before the definition of substitution is invoked.

An important elementary property of sequenced substitutions is the following lemma on interchanging the order of two substitutions. This lemma finds use in establishing the confluence of lambda-calculi.

Lemma 2.1.2 (Substitution) *For terms M_1, M_2, N and variables x,y such that $x \not\equiv y$ and $x \notin fv\, M_2$, we have*

$$[M_2/y]\,[M_1/x]\,N \;\equiv\; [[M_2/y]\,M_1/x]\,[M_2/y]\,N.$$

Proof: By induction on the structure of N.

Base cases:

- $N \equiv x$.

  $[M_2/y]\,[M_1/x]\,x \equiv [M_2/y]\,M_1$; likewise, $[[M_2/y]\,M_1/x]\,[M_2/y]\,x \equiv [[M_2/y]\,M_1/x]\,x \equiv [M_2/y]\,M_1$.

- $N \equiv y$.

  $[M_2/y]\,[M_1/x]\,y \equiv [M_2/y]\,y \equiv M_2$; likewise, $[[M_2/y]\,M_1/x]\,[M_2/y]\,y \equiv [[M_2/y]\,M_1/x]\,M_2 \equiv M_2$, since $x \notin fv\, M_2$.

Induction steps:

- $N \equiv \lambda z.N'$.

  In this case, $[M_2/y]\,[M_1/x]\,(\lambda z.N') \equiv [M_2/y]\,(\lambda z.[M_1/x]\,N') \equiv \lambda z.[M_2/y]\,[M_1/x]\,N'$. By the induction hypothesis, this last expression is equivalent to $\lambda z.[[M_2/y]\,M_1/x]\,[M_2/y]\,N'$. Applying the definition of substitution in reverse establishes the statement of the lemma in this case.

- The statement of the lemma follows for all other constructed terms by essentially the same argument.

This concludes the proof of Lemma 2.1.2. ∎

Normal form

A term in a lambda-calculus to which no reduction rule applies is said to be in *normal form*.

$$\begin{array}{rcl}
x,y & \in & \textit{Variables} \\
M,N & \in & \textit{Terms} \\
c^n & \in & \textit{Constructors of arity } n \\
f & \in & \textit{Primitive functions} \\
A & \in & \textit{Answers}
\end{array}$$

$$\begin{array}{rcll}
M & ::= & x & \textit{variables} \\
 & | & c^n & \textit{constructors} \\
 & | & f & \textit{primitives} \\
 & | & \lambda x.M & \textit{abstractions} \\
 & | & M \bullet N & \textit{applications} \\
 & & & \\
A & ::= & f & \\
 & | & c^n \bullet A_1 \bullet \cdots \bullet A_k, \quad k \leq n & \\
 & & & \\
\textbf{let}\, x = M \,\textbf{in}\, N & \equiv & (\lambda x.N) \bullet M & \textit{convenient abbreviation}
\end{array}$$

Figure 2.2: Syntax of the basic untyped lambda-calculus $\lambda[\beta\delta]$.

$$\begin{array}{rcll}
(\lambda x.M) \bullet N & \rightarrow & [N/x]\,M & (\beta) \\
f \bullet M & \rightarrow & \delta(f,M) & (\delta) \\
 & & & \\
\delta(f,(\lambda x.M)) & = & N_f \bullet (\lambda x.M) & \\
\delta(f,f_1) & = & N_{f,f_1} \bullet f_1 & \\
\delta(f,c^n \bullet M_1 \bullet \cdots \bullet M_n) & = & N_{f,c^n} \bullet M_1 \bullet \cdots \bullet M_n &
\end{array}$$

Figure 2.3: Reduction rules for the basic untyped lambda-calculus $\lambda[\beta\delta]$.

2.2 The basic applied lambda-calculus

Since we are using lambda-calculi to model the extension of pure functional programming, we first give the calculus with which we model pure functional programming itself. Although it is possible to use the pure lambda-calculus for this purpose, we find it more plausible to acknowledge the existence of primitive values and operations in all existing functional languages. Furthermore, the constructions given in Chapter 5 make essential use of this feature. This consideration leads us to extend the pure lambda-calculus with constructors and primitive function symbols (Figure 2.2). Constants are represented in this calculus as constructors of arity zero. Since we will need to distinguish among the several calculi to be discussed in this dissertation, we introduce a systematic naming convention for calculi: the character λ is followed by a bracketed list identifying the features of the calculus. The basic calculus presented in this section is denoted $\lambda[\beta\delta]$.

The reduction rules that enable $\lambda[\beta\delta]$ to model the computations of functional programming are given in Figure 2.3. The rule β is exactly as in the pure lambda-calculus: it models parameter passing by term substitution. Rule δ gives the computational model for primitive functions. The apparent complexity of the rule stems from the need to limit the power of such terms, since it is well known (see [Barendregt, 1984]) that in the presence of arbitrary δ-rules a calculus might fail to be confluent (a property defined in Section 3.2 below). The actual rule given is weak enough to avoid this pitfall. Essentially, it says that a primitive f is defined by lambda-terms giving its meaning when applied to a λ-abstraction and when applied to constructed values. Such definitions cannot examine the deeper structure of the argument to f. We will see in Chapter 3 that this restriction permits $\lambda[\beta\delta]$ to be confluent.

As an example of how primitive functions are represented in $\lambda[\beta\delta]$, consider the operator $+$ on integers. The integers themselves are represented by nullary constructors, one for each integer. The operation $+$ is then described by a countable (and obviously recursive) collection of rules such as $+ \bullet 1 \rightarrow +_1$, where $+_1$ is itself a primitive function with defining rules such as $+_1 \bullet 0 \rightarrow 1$ and $+_1 \bullet 5 \rightarrow 6$.

The terms N_f, N_{f,f_1}, and N_{f,c^n} that appear in the definition of the auxiliary function $\delta(\text{—},\text{—})$ in Figure 2.3 are the degrees of freedom we have in our restricted δ-rules. For each primitive function f defined in a particular instance of $\lambda[\beta\delta]$ we may define N_f, specifying the behavior of f on arguments that are lambda-abstractions, one term N_{f,f_1} specifying how f acts on each primitive function f_1, and one term N_{f,c^n} for each constructor c^n specifying how f acts on values constructed by c^n. We need not give all these terms: it is permitted for a primitive function to be partial. For example, the addition function would not be defined on any constructors that are not numerals. This careful structuring of δ-rule definitions restricts the power of such rules to detection of one layer of syntactic structure of value-terms, branching to one of several lambda-definable computations. As we mentioned above, this restriction is sufficient to ensure that our calculi can be confluent in the presence of δ-rules.

We curry our definition of primitives for two main reasons. First (and less important), we obtain the usual notational advantage of currying, in that we do not need to introduce any special notation for multiple-argument primitives. More importantly, we will be proving theorems about the existence of deterministic standard orders of evaluation for all our calculi of concern. For such a theorem to be true, a calculus must commit to the evaluation order for arguments to a primitive function in order to break the symmetry of syntax that an uncurried representation offers. The curried formulation is a convenient way of allowing the standard evaluation order to arise naturally.[1]

We can also define booleans by designating particular nullary constructors *true* and *false*; we can thus define *if* by $N_{if,true} \equiv \lambda x.\lambda y.x$ and $N_{if,false} \equiv \lambda x.\lambda y.y$. However, in the case of *if* it is just as easy to use the Church-encoded truth values $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ directly.[2]

The rule δ only applies when the argument expression is a constructed value, a primitive function, or an abstraction: this restriction is built into the definition of the auxiliary function $\delta(\text{—},\text{—})$. We can exploit this fact to simulate the behavior of call-by-value languages by defining a primitive function *force*, setting $N_{force} \equiv \lambda x.x$. We can also define a function *strict* as $\lambda f.\lambda x.f \bullet (force \bullet x)$. These definitions work because they require the argument to *force* to be a value before the application expression is reduced.

The usual technique used to simulate call-by-value parameter passing is to modify the rule β. This modification (introduced in [Plotkin, 1975]) involves a division of expressions into *applications* and all other expressions, which are now termed *values*. The modified rule β_V has the same rewriting effect as the original rule β, but only applies when the argument expression is a value. This restriction reflects the notion that only an evaluated expression is substituted into the body of the function in a call-by-value language.

We mention the contrast between call-by-name and call-by-value calculi at this point because we will have occasion later in this chapter to introduce a similar contrast between lazy and eager store-computations. Just as the rules β and β_V differ only in the syntactic form of the argument terms, so also the rules defining purifications of lazy and eager store-computations will differ only in the syntactic form of the contexts from which the purification rule extracts a pure result.

The operational semantics we use throughout our treatment distinguishes a certain subset of terms as *answers* A, the observable results of terminating computations. There may be normal forms that are not answers: such terms are regarded as indicating a run-time error. For a basic example of such a term, consider the term

$$+ \bullet true \bullet 1,$$

where $+$ is a primitive implementing addition on integers (themselves represented as nullary constructors). The definition of the primitive $+$ contains no defining case for the constructor *true*, so this term cannot be reduced. However, it is not an answer because $+$ is a primitive, not a constructor of arity 2 or higher as required by the definition when applied to two terms as is the case here. This particular example of a "stuck" term (an adjective we will continue to use) has an obvious correspondence to a type error in a programming language.

[1]The operator $==$ for store-variable identity introduced in Section 2.5 is subject to the same issues as primitive-function definitions.

[2]Church encoding [Church, 1951] represents various mathematical constructs as lambda-terms by giving those constructs interpretations as higher-order functions. Each natural number, for example, is identified with a term that iterates an unknown function that number of times on an unknown argument. The encoding at hand, that for truth values, is based on the idea of conditionals picking one of two arguments. The encoding for *true* picks the first; that for *false* picks the second.

$$C \in \mathit{Commands}$$

$$\begin{array}{rcll}
M & ::= & \cdots(\text{Figure 2.2})\cdots & \textit{previously defined constructs} \\
 & \mid & C & \textit{commands} \\
C & ::= & C_{prim} & \textit{primitive commands} \\
 & \mid & \uparrow M & \textit{trivial command} \\
 & \mid & M_1 \rhd x \cdot M_2 & \textit{sequenced commands} \\
 & & & \\
M_1; M_2 & \equiv & M_1 \rhd x \cdot M_2 & (x \notin fv\, M_2)\ \textit{convenient abbreviation}
\end{array}$$

Figure 2.4: Syntax of generic command constructs.

$$\begin{array}{rcll}
(M \rhd x \cdot N) \rhd y \cdot P & \rightarrow & M \rhd x \cdot (N \rhd y \cdot P) & (\mathit{assoc}) \\
\uparrow M \rhd x \cdot N & \rightarrow & [M/x]\, N & (\mathit{unit})
\end{array}$$

Figure 2.5: Generic command reductions.

2.3 Axiomatizing commands and assignment

We now start the process of describing the extensions that axiomatize the various styles of assignment in functional programming which form our present subject. In constructing a formal treatment of imperative programming we are concerned with the notion of a sequence of commands. The basic concept of a sequence can be specified algebraically in terms of three components: the designation of an empty sequence, a set of primitive elements, and the concatenation of two sequences to form a longer sequence. For axioms, a simple description of sequences states that the operation of concatenation is associative and that the empty sequence is a two-sided identity for this operation—in other words, a sequence is a monoid.[3] A sequence of *commands*, however, ought to have more structure reflecting the consequences of the execution of the commands affecting the meaning of subsequent commands. Algebraically, this extra structure can be captured very generally by the axioms for a *monad* (already introduced in Section 1.2.1). Figure 2.4 shows how we add a syntactic representation of a monad to the basic lambda-calculus. In this monad-based view, a command has both an *effect* and a *result*. The effect is detected by the interaction between commands in a sequence, but the result is a value passed from one command to the next through the variable bound in the construct $M \rhd x \cdot N$.

This incremental addition to the calculus is parametrized by a collection of primitive commands C_{prim}: by varying the selection of primitive commands we vary the particulars of the monad under consideration. In our calculi for assignment, the primitive commands will be constructs for assigning and reading the values of locations in the store.

Aside from the primitive commands yet to be defined, there are two ways of constructing commands in this extension to $\lambda[\beta\delta]$. The first, notated $\uparrow M$, creates from any term M the command that passes M to the rest of the command sequence; such commands have no other effect. The second, $M_1 \rhd x \cdot M_2$, represents a sequencing of two commands, with the result of M_1 being bound to the variable x within M_2.

The reductions given in Figure 2.5 express two of the three monad laws.[4]

Rule *assoc* expresses the associativity of substructure within a command sequence. This rule often comes into play when part of a command sequence has been abstracted as a variable: when a command is substituted for the variable there may be a need for some rewriting according to *assoc* before reduction rules axiomatizing the effect of the command become applicable. The orientation of rule *assoc* gives a preference to command sequences written associated to the right. In the development of the mathematical properties of our calculi in Chapter 3 rule *assoc* will have a special role.

[3] We are glossing over a lot of universal algebra basics here. Actually it is the notion of a *free* monoid that corresponds to an algebra of sequences.

[4] The right-unit law is omitted from this axiomatization; however, it is possible to prove this law as an operational equivalence (see Proposition 4.2.1, part 6).

$$v, w \quad \in \quad \textit{Store-variables}$$

$$\begin{array}{rcll} C_{prim} & ::= & M := N & \textit{store } N \textit{ at } M \\ & | & M? & \textit{fetch from } M \end{array}$$

Figure 2.6: Syntax of basic assignment primitive commands.

$$\begin{array}{rcllr} v := M;\ v?\triangleright x \cdot P & \rightarrow & v := M;\ [M/x]\,P & & (\textit{fuse}) \\ v := N;\ w?\triangleright x \cdot P & \rightarrow & w?\triangleright x \cdot v := N;\ P & (v \not\equiv w) & (\textit{bubble-assign}) \\ v := M \triangleright x \cdot P & \rightarrow & v := M;\ [(\,)/x]\,P & (x \in fv\,P) & (\textit{assign-result}) \end{array}$$

Figure 2.7: Reduction rules for basic primitive store commands.

Rule *unit* expresses the meaning of the construct $\uparrow M$ given informally just above; its right-hand side displays how the result M is substituted for every occurrence of x in M.

2.4 Axioms for assignment

In Section 2.3 the definition of the procedural calculus was parametrized by a set of primitive procedures. We indulged in this bit of over-abstraction in order to isolate the constructs that are reusable for any monad from the constructs that axiomatize assignment to locations in a store—the latter being the actual subject matter of this dissertation. Figure 2.6 extends the syntax from Figure 2.4 further to supply specific primitive commands for assigning and retrieving values in a store.

The most basic new construct in Figure 2.6 is the *store-variable.* Store-variables correspond to the usual notion of variables in imperative programming languages, but we give them a distinct name because the underlying lambda-calculus already has constructs called variables. Store-variables are chosen from a countable alphabet of symbols distinct from the symbols used for lambda-bound variables. Figure 2.6 also gives the definition of the *assignment* command, which models the setting of a store-variable, and of the *read* command, which models the retrieving of the value associated with a store-variable. The syntax of both commands permits a complex expression to be given where the store-variable should appear; this expression must reduce to an actual store-location before the command in which it appears can have an effect. This feature allows us to model pointer variables, anonymous objects, and other features of modern imperative programming languages.

Figure 2.7 gives the rules by which these constructs axiomatize a store. The basic notion underlying these rules is that the history of assignment commands, represented within the command sequence, is a sufficient representation of a store—there is no explicit syntactic construct for modeling a store. Instead, the reduction rules axiomatize the *interaction* between assignments and subsequent reads that is at the heart of the semantics of a store. Rule *fuse* embodies the requirement that reading a store-variable that has just been assigned the value M, with the value read to be received by the remainder of the compuation via the variable x, corresponds to replacing each bound occurrence of x by M. The name of the rule derives from the two occurrences of the same store-variable 'fusing' to produce the transmission of the value read. Rule *fuse* leaves the assignment command in place, since an assignment remains in effect after its value is read. Rule *bubble-assign* takes care of the case in which the store-location read is different from the one just assigned: the relevant assignment must occur further to the left or not at all. Consequently, rule *bubble-assign* rewrites the command sequence to move the read to the left past an assignment to which it is indifferent. The name of the rule *bubble-assign* comes from visualizing the read-expressions as 'bubbling' to the left until they meet an assignment to the same store-variable that is being read.

The rules *fuse* and *bubble-assign* actually do satisfy our intution about assignments. If more than one assignment to the same store-variable is present in the sequence of commands, the most recent from the point of view of a read-expression is the one which supplies the value actually observed. This placement of assignment- and read-expressions into a common sequence that is independent of the mechanism used to model function-application enables our calculi to be referentially transparent.

	v, w	$\in$	*store-variable*
C_{prim}	$::=$	$\cdots$ (Figure 2.6) $\cdots$	*previously-defined commands*
	$\mid$	$\nu w\, M$	*introduce new store-variable*
	$\mid$	$v == w$	*test store-variable identity*

Figure 2.8: Syntax for allocatable store-variables.

The precision required of a formal system demands that we provide the additional rule *assign-result* to cover the case in which the remainder of a command sequence observes not just the effect of an assignment command, but also its functional result. To see why this is necessary, note that the left-hand side of the rule *fuse* is given in terms of the syntactic abbreviation ';' defined in Figure 2.4. Expanding the abbreviation gives the new left-hand side $v := M \triangleright z \cdot v? \triangleright x \cdot P$, where z is a variable that does not occur free in $v? \triangleright x \cdot P$ (which is to say that it does not occur free in P). Thus *fuse* does not apply in the case in which z *does* occur free in P, and hence we must supply a rule to cover this case if we want the calculus to reflect the intended semantics for assignment. Rule *assign-result* does this job by supplying a neutral value as the functional value of the assignment command wherever this value happens to be observed in the sequel. In the remainder of this dissertation we denote this neutral value, which is an instance of a nullary constructor c^0 (constant) by $()$.[5]

The definition of *assign-result* is somewhat arbitrary, in that the rule is specified to apply only when the store-variable v has become known. It is certainly possible to consider a variant rule

$$N := M \triangleright x \cdot P \rightarrow N := M;\ [()/x]\,P, \quad (x \in fv\,P)$$

which does not force the computation of the store-variable. It is easier, however, to define evaluation contexts (Chapter 3) uniformly if we require computation to seek an actual store-variable as target of an assignment command regardless of the context in which the assignment command appears.

There are some notable differences between the notion of assignable store introduced here and the constructs available in most popular imperative programming languages. First, there is no restriction on the values that may be assigned to a store-variable: any expression in the language is eligible, even other store-variables. Second, the name of a store-variable v does *not* denote its 'current value'—there is no such concept. Instead, the name v merely denotes itself, and there is an associated *command* $v?$ for finding the most-recently-assigned value for v and transmitting it to subsequent commands via term-substitution. Lastly, there is no analogue in our calculi to the distinction between variables and pointers present in several popular conventional languages. In conventional languages both a variable itself and a pointer to the variable denote the variable's storage, but in different ways according to context. As l-values, both a variable and its pointer denote the location, but as r-values, the variable denotes the contents whereas the pointer denotes the location. As just mentioned, the r-value interpretation is absent in our calculi.[6]

2.5 Locally defined store-variables

So far, our formalism has nothing to say about where store-variables come from—we are given all the store-variables that will ever exist at the outset. In programming terms, they are all global variables. This simplification results from a deliberate decision to present the basic formalism for assignment in as much isolation as possible. However, no reasonable account of imperative programming can neglect the notion of store-variable scope, by which the programmer may introduce a store-variable that is known (within the scope of its declaration) to be distinct from all other store-variables in the program. We now rectify this omission.

We do so by augmenting the basic store formalism given in Section 2.4 with constructs that allow the allocation of new variables in the store. Figure 2.8 introduces two new syntactic constructs. The expression

[5]By 'neutrality' we mean here that $()$ is distinct from any other constant having an intended meaning, such as constants representing numbers.

[6]The class of "conventional languages" here includes Algol 60, C, Pascal and Scheme but excludes Algol 68, Forsythe, and Standard ML.

$$\begin{array}{rcllr}
(\nu v.M) \triangleright x \cdot P & \to & \nu v.(M \triangleright x \cdot P) & & (\textit{extend}) \\
\nu v.w? \triangleright x \cdot P & \to & w? \triangleright x \cdot \nu v.P & (v \not\equiv w) & (\textit{bubble-new}) \\
v == v & \to & \lambda x.\lambda y.x & & (\textit{identical}) \\
v == w & \to & \lambda x.\lambda y.y & (v \not\equiv w) & (\textit{not identical})
\end{array}$$

Figure 2.9: Reduction rules for allocatable store-variables.

$\nu v.M$ defines the store-variable v over the scope M.[7] Like lambda-variables, store-variables so declared may be α-renamed; we make them subject to the Barendregt bound-variable convention as well.[8] The second new construct is a test for identity of store-variables. Many algorithms that employ pointers require the ability to detect equality of pointers. In our calculi pointer equality can be represented by name equality: two location names are equal if and only if they are allocated by the same occurrence of ν. In order to make this definition of equality internal to the calculus, we introduce the operator $==$. Our construct ν corresponds to the ν construct in Odersky's calculus with local names [Odersky, 1993b; Odersky, 1994]; it is very useful in abstracting away from technicalities required by approaches that model scopes as an explicit collection of valid names at each point. The ν construct can be thought of as axiomatizing a particular, but widely useful, monad.

The rules that axiomatize the behavior of the new constructs are given in Figure 2.9.

The rule *extend* specifies the interaction of the scope of a store-variable with the monadic sequencing construct $\triangleright$. Informally, the rule states that the allocation of a new store-variable, like any other effect on the store, remains in effect from the point in the command sequence at which it occurs to any subsequent point. The form in which it is stated, however, relies crucially on the Barendregt α-renaming convention. The expression P, which is moved from outside the scope of the new store-variable to within it is by this convention assumed to have no free ocurrences of the store-variable in question. This can always be achieved by an (implicit) α-renaming of v over M before carrying out the rewriting. Like the rule *assoc*, the rule *extend* plays a special role in the proofs in Chapter 3.

Since the connection between assignment and reading is carried out by 'bubbling' the read to the left, we must specify that reading from a store-variable is indifferent to declarations of other store-variables. Rule *bubble-new* formalizes this requirement.

The remaining two rules *identical* and *not identical* formalize the notion of store-variable identity. The two different lambda-expressions into which an identity test is rewritten correspond to the Church encodings of truth and falsehood mentioned above; we could instead have chosen distinct constants to play this role. In all respects other than the meta-level condition by which it is defined, the construct $==$ behaves like a two-argument primitive function.

The relation axiomatized by $==$ is informally equivalent to *pointer* equality in conventional languages, even though it is stated in terms of store-variable *names*. The calculi we are defining are thus susceptible to the same problems of pointer aliasing as conventional languages.

2.6 Purification: extracting the result of a store computation

So far in this chapter, we have described the construction of what amounts to a generalized imperative programming language with an embedded lambda-calculus for providing procedural abstraction and computation on pure values. The store-modelling reductions given in Figures 2.5, 2.7, and 2.9 rewrite command-sequences to new command-sequences. As a simple example, consider the command-sequence

$$\nu v.v := 1;\ v? \triangleright x \cdot \uparrow x.$$

[7]The author has found it difficult to remedy the typographic similarity of the symbol ν (the Greek letter nu) with the letter v without incurring some other notational disadvantage. The reader may wish to note that the former symbol is upright, whereas the latter is slanted.

[8]See page 12.

It should be obvious that this little program returns the result 1; indeed we have the sequence of reductions

$$\begin{aligned} \nu v.v := 1;\ v?\triangleright x\cdot\uparrow x &\ \rightarrow\ \nu v.v := 1;\ [1/x]\,(\uparrow x) \quad (\text{by } \mathit{fuse}) \\ &\ \equiv\ \nu v.v := 1;\ \uparrow 1. \end{aligned}$$

This is a normal form; no further reductions are possible. However, this term is *not* the same as the answer-term 1, no matter how obvious it may be that this is its meaning. We have no rules for *extracting* the answer 1 from the residue of the store-computation.

It might be tempting to supply a rule that strips away the context $\nu v.v := 1;\ []$; however, there is danger here. Suppose that we had started with the only-slightly-different initial term

$$\nu w.\nu v.v := w;\ v?\triangleright x\cdot\uparrow x$$

yielding the only-slightly-different sequence of reductions

$$\begin{aligned} \nu w.\nu v.v := w;\ v?\triangleright x\cdot\uparrow x &\ \rightarrow\ \nu w.\nu v.v := w;\ [w/x]\,(\uparrow x) \quad (\text{by } \mathit{fuse}) \\ &\ \equiv\ \nu w.\nu v.v := w;\ \uparrow w. \end{aligned}$$

What happens if we just strip away the command-sequence context in the same way as before? Unfortunately, we would obtain the free store-variable w, which was bound before. In informal implementational terms, this would correspond to allowing the store-variable w to outlive its scope and become a dangling pointer; in formal terms we cannot hope to have a consistent calculus if bound names can escape their bonds.

We refer to the problem raised by these examples as the need for a *purification* construct in designs for functional languages with assignment. The escape of bound store-variables must clearly be prohibited by any reasonable proposal for a purification construct, but name-escape is not the only issue involved: we will also have to consider the effect of purification constructs on the informal interpretation of store-variables as storage locations. As will further emerge, purification constructs are the place where we distinguish between eager and lazy uses of stores.

2.6.1 The general structure of purification rules

The problem of purification divides into two independent questions.

- When in a store-computation do result values become available, and
- What result values are legal?

These two questions are reflected in the general structure of purification rules that we present in this subsection. Purification rules divide the store-computation to be purified into a *store context* and a *result expression*. The question of *when* results become available arises because the result of a store-computation may not depend on the entire computation. For example, suppose we are given the initial term

$$\nu v.v := 1;\ v?\triangleright x\cdot\uparrow 2.$$

The result 2 does not depend on the assignment $v := 1$. It is thus reasonable to try to write purification rules that allow this result to be extracted from the store-computation context even though that context contains a latent assignment and a dependent read. The issue becomes more dramatic if we replace the innocuous assignment $v := 1$ with some term Ω having no normal form to give the fragment

$$\nu v.\Omega;\ v?\triangleright x\cdot\uparrow 2.$$

An eager-store calculus with assignment will compute forever trying to find what command is present, whereas a lazy-store calculus with assignment can return 2 immediately.

The question of *what* terms are considered legal to purify is considerably more complicated, and admits a spectrum of answers. We have already seen two extreme points on this spectrum: a locally-allocated store-variable should never be extracted from its surrounding store-computation, whereas it is always safe to do so with a constant (nullary constructor). With more complex result expressions, which may mix constants with unevaluated expressions, the question of purification must be deferred until it is possible to see which of the simple cases applies. The process of purification must therefore consider the following cases for result expressions:

1. Some subexpressions (such as store-variables) can be recognized to spoil purity merely by their syntactic form;
2. some (such as constants) can be recognized to be purifiable merely by their syntactic form;
3. some (such as variables or applications) must have any judgment as to their purifiability deferred until further computation has taken place; and finally,
4. the purity of constructed expressions depends on the purity of their subexpressions.

In terms of our informal understanding of the problem this can only be correct if the result expression M is free of any remaining reference to the state represented by the store context $S[]$, since such a reference will be nonsensical once the representation of that state is erased from the program. Furthermore, it must be impossible for an expression delimited by **pure** to affect any store-based computation elsewhere in the program, for then the absence of the newly-erased expressions could be detected by their failure to have the expected effect.

The application of this insight results in a sort of reasoning process that is interspersed with computation: it is possible that an impure subterm may be discovered only after considerable computation. In Chapter 6 we investigate the use of a type system to carry out this reasoning in a stage that entirely precedes reduction.

Up to this point, we have been discussing the requirements for purification rules in terms of an informal programmer's understanding of their role. In terms of a formal reduction system, however, the actual validation of a set of purification rules is that the calculus using it must be Church-Rosser. We can connect the two levels of understanding by observing that the Church-Rosser property embodies the notion that the semantics given by convertibility yields a consistent semantics: interconvertible terms having normal forms have the *same* normal form. Our concern about exposing a store-variable outside its native command sequence is essentially a concern about the possibility of giving several different interpretations to the same expression.

2.6.2 Eager versus lazy store-computations

We now introduce the two calculi that will form the subject of the rest of the dissertation; these calculi correspond to the two possible answers for *when* a store-computation result becomes available. The first calculus, $\lambda[\beta\delta!eag]$, is a slight modification of the calculus originally introduced in [Odersky *et al.*, 1993; Odersky and Rabin, 1993] (where it is called λ_{var}). In this calculus, use of the store always proceeds as far as possible before purification. The second calculus, $\lambda[\beta\delta!laz]$, takes into account Launchbury's proposal [Launchbury, 1993] for a *lazy* use of the store: sequences of store actions only proceed as far as necessary to produce results that are actually demanded. We present the two calculi in parallel throughout the rest of this dissertation in order to make their similarities and differences more apparent—the similarities, however, predominate.

There are actually *two* notions of computation present in these calculi: the familiar evaluation of lambda-expressions, and the execution of commands on the store. Each of these can independently be by-name or by-value. We restrict our attention in this dissertation to calculi with call-by-name semantics for applicative computations because, but we expect the treatment of call-by-value languages to be entirely analogous.

Both these calculi are based on the general command calculus of Section 2.3 with the basic store commands of Figure 2.6 as primitive commands. The two calculi differ only in their purification rules; these rules, furthermore, differ only in the syntactic form of the command prefixes they manipulate. The situation resembles the distinction between the β-rules for Plotkin's call-by-name and call-by-value calculi [Plotkin, 1975]: the difference is in the side condition, not in the action of the rule.

In the calculus $\lambda[\beta\delta!eag]$, no value can be purified until the state computation finishes. This condition is enforced by the form of the prefixes $S^{eag}[]$ defined in Figure 2.12. The notable features of this definition are: (1) all assignments are to known (fully computed) store-variables, and (2) there are no pending reads whatsoever.

$$\begin{array}{rcl} M & ::= & \ldots(\text{Figure } 2.8)\ldots \\ & | & \mathbf{pure}\, M \end{array}$$

Figure 2.10: Syntax of purification construct

$$\begin{array}{rcl} S^{eag}[] & ::= & [] \\ & | & \nu v\, S^{eag}[] \\ & | & v := M;\ S^{eag}[] \\ \\ S^{laz}[] & ::= & [] \\ & | & \nu v\, S^{laz}[] \\ & | & M \rhd x \cdot S^{laz}[] \end{array}$$

Figure 2.11: Syntax of purification contexts

$$\begin{array}{rcll} \mathbf{pure}\, S^{eag}[\uparrow c^n \bullet M_1 \bullet \cdots \bullet M_k] & \rightarrow & c^n \bullet (\mathbf{pure}\, S^{eag}[\uparrow M_1]) \bullet \cdots \bullet (\mathbf{pure}\, S^{eag}[\uparrow M_k]) \quad (k \leq n) & (\mathit{pure}\text{-}c^n) \\ \mathbf{pure}\, S^{eag}[\uparrow f] & \rightarrow & f & (\mathit{pure}\text{-}f) \\ \mathbf{pure}\, S^{eag}[\uparrow \lambda x.M] & \rightarrow & \lambda x.\mathbf{pure}\, S^{eag}[\uparrow M] & (\mathit{pure}\text{-}\lambda) \end{array}$$

Figure 2.12: Purification rules for $\lambda[\beta\delta!\mathit{eag}]$.

$$\begin{array}{rcll} \mathbf{pure}\, S^{laz}[\uparrow c^n \bullet M_1 \bullet \cdots \bullet M_k] & \rightarrow & c^n \bullet (\mathbf{pure}\, S^{laz}[\uparrow M_1]) \bullet \cdots \bullet (\mathbf{pure}\, S^{laz}[\uparrow M_k]) \quad (k \leq n) & (\mathit{pure}\text{-}c^n\text{-}\mathit{lazy}) \\ \mathbf{pure}\, S^{laz}[\uparrow f] & \rightarrow & f & (\mathit{pure}\text{-}f\text{-}\mathit{lazy}) \\ \mathbf{pure}\, S^{laz}[\uparrow \lambda x.M] & \rightarrow & \lambda x.\mathbf{pure}\, S^{laz}[\uparrow M] & (\mathit{pure}\text{-}\lambda\text{-}\mathit{lazy}) \end{array}$$

Figure 2.13: Purification rules for $\lambda[\beta\delta!\mathit{laz}]$.

The calculus $\lambda[\beta\delta!\mathit{laz}]$ is more lenient about such matters. The contexts $S^{laz}[]$ defined in Figure 2.11 are only required to be well-formed sequences of store operations.

As an example of how the difference between the two calculi is inherent in the specification of purification contexts, we reintroduce the example

$$\mathbf{pure}(\mathit{loop}; \uparrow 3) \tag{2.1}$$

that we used to introduce the subject in Section 1.2.4. We use notation *loop* to stand for some non-terminating expression such as the well-known example $\Omega \equiv (\lambda x.x \bullet x) \bullet (\lambda x.x \bullet x)$.

In (2.1) the context *loop*; [] surrounding the result expression 3 matches the definition of $S^{laz}[]$ in Figure 2.11 but not that of $S^{eag}[]$. If we regard (2.1) as belonging to $\lambda[\beta\delta!\mathit{laz}]$, therefore, we can reduce the expression immediately to the answer 3 by rule *pure-*c^n*-lazy*. In $\lambda[\beta\delta!\mathit{eag}]$, however, the pure-reduction is not available, and we are doomed to carry out the (non-terminating) reduction within *loop* forever. These outcomes are semantically distinct.

2.6.3 Locally defined stores

We have already discussed (in Section 2.5) how the ν construct gives us locally defined store-variables. In this section we treat the **pure** construct as a definer of local stores, and we explore the interaction between the two store-related notions of locality.

Let us return for the moment to the world of Section 2.4 before we introduced locally defined store-variables. In this universe there was a set of global store-variables that referred to the same location everywhere within a program. Now introduce **pure** into this universe without going through the intermediate stage of introducing ν. The effect obtained is that each **pure** scope defines an entirely new set of bindings for all the globally-defined names, since we do not allow assignment constructs to interact across **pure** boundaries. The interpretation of store-variables as locations in some store is violated: a store-variable potentially denotes many bindings, one for each **pure**-scope in which it is used.

As an example of this situation, consider the term

$$\mathbf{pure}\, w := 1;\ (\mathbf{pure}\, w := 2;\ \uparrow());\ w? \triangleright x \cdot \uparrow x.$$

Even though the assignment to w within the inner **pure** is an assignment to the exact same store-variable as the assignment to w in the outer **pure**-countour, the result of the whole expression is 1, not 2. Commands within the inner **pure** affect a distinct store.

This issue is entirely separate from the local scoping of store-variable names usually associated with the term *block structure* in the programming-languages literature. In this customary understanding, a local name *shadows* a declaration in an outer scope because it is really a new name. The awkwardness with global store-variables, on the other hand, occurs even though the store-variable really is the same name everywhere—it is the **pure** boundary that forces the change in interpretation on us.

In fact, the situation is not remedied by the introduction of ν: we are compelled to accept that store-variables cannot be interpreted as locations if we wish to have a workable set of **pure**-rules. The calculus λ_{var}, which was the predecessor of $\lambda[\beta\delta!eag]$, had a restriction on the form of the contexts $S^{eag}[]$ that attempted to enforce the intuitive notion that all store-variables used locally to the store introduced by a **pure**-expression were also declared locally to that **pure**. In defining the rules for the lazy-store calculus, it proved impossible to retain this condition: there is no way to even see what names are introduced in a lazy-store context without forcing computations that should not be forced. We thus dropped the attempt to force $\lambda[\beta\delta!laz]$ into this mold, and dropped it as well for $\lambda[\beta\delta!eag]$ in order to emphasize the symmetry between the calculi.

All is not lost, however. First of all, we take the major arbiter of whether a calculus embodies a reasonable notion of computation to be whether it has the Church-Rosser property allowing terms to be given consistent interpretations. It turns out that both $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ have this property even without trying to force stores to use only local names (as we will prove in Chapter 3). Second, relaxing the constraint between locality of names and locality of stores turns out to make the encodings of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ into more basic lambda-calculi more attractive (Chapter 5). Finally, a relatively simple type system can enforce the restriction we desire, as we show in Chapter 6. With all these considerations in mind, we choose to view this difficulty with enforcing the locality of store-variables as a pleasant revelation of the independence of two concepts previously thought to be inextricable. This is a good outcome for research.

2.7 Chapter summary

We have now introduced the entire syntax and reduction rules of the calculi with assignment that form the subject of study in the remainder of this dissertation. The formal structure of these calculi is motivated reasonably directly by the structure of the programming constructs we wish to model: stores and commands. The introduction of the **pure** construct serves to delimit the use of a local store from a store-less ambient functional computation, to delimit the use of a local store from a surrounding store-based computation, and to define whether use of the results of a store-based computation is driven by demand for the result or by the execution of the constituent commands.

We summarize the syntax and semantics of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ in Figures 2.14, 2.15, 2.17, and 2.18.

$$
\begin{array}{rcll}
M & ::= & x & \textit{variables} \\
 & \mid & c^n & \textit{constructors} \\
 & \mid & f & \textit{primitives} \\
 & \mid & \lambda x.M & \textit{abstractions} \\
 & \mid & M \bullet N & \textit{applications} \\
 & \mid & C & \textit{commands} \\
 & \mid & \mathbf{pure}\, M & \textit{purified command} \\
C & ::= & C_{prim} & \textit{primitive commands} \\
 & \mid & \uparrow M & \textit{trivial command} \\
 & \mid & M \triangleright x \cdot M & \textit{sequenced commands} \\
M_1; M_2 & \equiv & M_1 \triangleright x \cdot M_2 & (x \notin fv\, P)\ \textit{convenient abbreviation} \\
C_{prim} & ::= & M := N & \textit{store N at M} \\
 & \mid & M? & \textit{fetch from M} \\
 & \mid & \nu v\, M & \textit{introduce new store-variable} \\
 & \mid & v == w & \textit{test store-variable identity} \\
 & & & \\
\mathbf{let}\, x = M\, \mathbf{in}\, N & \equiv & (\lambda x.N) \bullet M & \textit{convenient abbreviation} \\
 & & & \\
A & ::= & f & \\
 & \mid & c^n \bullet A_1 \bullet \cdots \bullet A_k, \quad k \leq n &
\end{array}
$$

Figure 2.14: Syntax of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$

$$
\begin{array}{rcll}
(\lambda x.M) \bullet N & \rightarrow & [N/x]\,M & (\beta) \\
f \bullet M & \rightarrow & \delta(f, M) & (\delta) \\
 & & & \\
(M \triangleright x \cdot N) \triangleright y \cdot P & \rightarrow & M \triangleright x \cdot (N \triangleright y \cdot P) & (\textit{assoc}) \\
\uparrow M \triangleright x \cdot N & \rightarrow & [M/x]\,N & (\textit{unit}) \\
v := M;\ v? \triangleright x \cdot P & \rightarrow & v := M;\ [M/x]\,P & (\textit{fuse}) \\
v := N;\ w? \triangleright x \cdot P & \rightarrow & w? \triangleright x \cdot v := N;\ P \quad (v \not\equiv w) & (\textit{bubble-assign}) \\
v := M \triangleright x \cdot P & \rightarrow & v := M;\ [()/x]\,P \quad (x \in fv\, P) & (\textit{assign-result}) \\
(\nu v.M) \triangleright x \cdot P & \rightarrow & \nu v.(M \triangleright x \cdot P) & (\textit{extend}) \\
\nu v.w? \triangleright x \cdot P & \rightarrow & w? \triangleright x \cdot \nu v.P \quad (v \not\equiv w) & (\textit{bubble-new}) \\
v == v & \rightarrow & \lambda x.\lambda y.x & (\textit{identical}) \\
v == w & \rightarrow & \lambda x.\lambda y.y \quad (v \not\equiv w) & (\textit{not identical}) \\
 & & & \\
\delta(f, (\lambda x.M)) & = & N_f \bullet (\lambda x.M) & \\
\delta(f, f_1) & = & N_{f,f_1} \bullet f_1 & \\
\delta(f, c^n \bullet M_1 \bullet \cdots \bullet M_n) & = & N_{f,c^n} \bullet M_1 \bullet \cdots \bullet M_n &
\end{array}
$$

Figure 2.15: Reduction rules common to both $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$

$$
\begin{array}{rcl}
S^{eag}[] & ::= & [] \\
 & \mid & \nu v.S^{eag}[] \\
 & \mid & v := M;\ S^{eag}[] \\
 & & \\
S^{laz}[] & ::= & [] \\
 & \mid & \nu v.S^{laz}[] \\
 & \mid & M \triangleright x \cdot S^{laz}[]
\end{array}
$$

Figure 2.16: Syntax of purification contexts

$$\begin{array}{rcll}
\mathbf{pure}\,S^{eag}[\uparrow c^n \bullet M_1 \bullet \cdots \bullet M_k] & \rightarrow & c^n \bullet (\mathbf{pure}\,S^{eag}[\uparrow M_1]) \bullet \cdots \bullet (\mathbf{pure}\,S^{eag}[\uparrow M_k]) \quad (k \leq n) & (\mathit{pure}\text{-}c^n) \\
\mathbf{pure}\,S^{eag}[\uparrow f] & \rightarrow & f & (\mathit{pure}\text{-}f) \\
\mathbf{pure}\,S^{eag}[\uparrow \lambda x.M] & \rightarrow & \lambda x.\mathbf{pure}\,S^{eag}[\uparrow M] & (\mathit{pure}\text{-}\lambda)
\end{array}$$

Figure 2.17: Additional reduction rules for $\lambda[\beta\delta!eag]$

$$\begin{array}{rcll}
\mathbf{pure}\,S^{laz}[\uparrow c^n \bullet M_1 \bullet \cdots \bullet M_k] & \rightarrow & c^n \bullet (\mathbf{pure}\,S^{laz}[\uparrow M_1]) \bullet \cdots \bullet (\mathbf{pure}\,S^{laz}[\uparrow M_k]) \quad (k \leq n) & (\mathit{pure}\text{-}c^n\text{-}\mathit{lazy}) \\
\mathbf{pure}\,S^{laz}[\uparrow f] & \rightarrow & f & (\mathit{pure}\text{-}f\text{-}\mathit{lazy}) \\
\mathbf{pure}\,S^{laz}[\uparrow \lambda x.M] & \rightarrow & \lambda x.\mathbf{pure}\,S^{laz}[\uparrow M] & (\mathit{pure}\text{-}\lambda\text{-}\mathit{lazy})
\end{array}$$

Figure 2.18: Additional reduction rules for $\lambda[\beta\delta!laz]$

3 Proofs of the fundamental properties

The Church-Rosser and standardization properties are the minimum necessary to establish a calculus as a reasonable basis for computation. In this chapter we prove these fundamental properties for the calculi presented in Chapter 2. In fact, we prove these properties for calculi with a modified reduction relation that captures the computational intent of the calculi while being more amenable to the desired proof techniques. We then derive the desired properties of the original reduction relation from those of the modified reduction. Section 3.1 outlines the proofs to be undertaken in this chapter; Section 3.2 sets forth some of mathematical framework for the proofs, and Section 3.3 explains and justifies the modified reduction relations for which the main theorems will be established. Section 3.4 introduces marked reduction (our main proof technique), and Sections 3.5, 3.6, and 3.7 carry out the proofs themselves. Section 3.8 deduces the Church-Rosser and standardization properties of the original reduction relation from the results for the modified reduction.

3.1 A roadmap to the proofs in this chapter

The two properties we set out to prove in this chapter both have to do with the possible forms reductions can take. The first, the *Church-Rosser* property, states that different reductions from the same starting term can always be reconciled by finding a common reduct; the second, the *standardization theorem*, states that a simple rule for choosing the next redex to reduce suffices to find any possible reduct of a term—it is not necessary to consider every possible reduction sequence in order to characterize the computational behavior of terms. The proofs of these properties require bookkeeping to keep track of redexes (see Section 3.4), and much of the detailed work presented in this chapter has to do with this bookkeeping.

The proof techniques given in [Barendregt, 1984], Chapter 11, allow us to base proofs of both the Church-Rosser property and the standardization property for each calculus on a property called *finiteness of developments* (FD). This property acts as a 'local' version of strong normalization (a property which does not itself hold for the calculi with assignment): if we mark a collection of redexes in a term, and then carry out a sequence of reductions involving only marked redexes and their descendants, then that reduction sequence must terminate. The *strong* version of this property (FD!) states that all such reduction sequences starting with the same marked term must end in the same term—a sort of Church-Rosser property for marked reductions. Furthermore, FD! can be proved by establishing both the weak Church-Rosser property and strong normalization for this reduction relation. The property FD! in turn can be used to prove the Church-Rosser and standardization properties.

The only problem with this line of attack is that FD! is not actually true for the calculi with assignment: the interactions of the reduction rules *assoc* and *extend* among themselves and with other rules fail to obey the necessary restrictions. We have found it possible, however, to endow these calculi with a modified set of reduction rules in such a way that FD! is restored. Aside from being a way out of a technical difficulty, this modification clarifies the structure of the proposed calculi by separating reduction rules that axiomatize store-computation from those that merely axiomatize the sequential structure of commands. This factoring of the rule-set defining the reductions for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ has some independent intellectual interest. With the modified notion of reduction in place, a proof of FD! along the lines outlined above is the subject of Section 3.5. The modified reduction forms the basis for the proofs given in the next few sections, but later (in Section 3.8) we recover the Church-Rosser and standardization properties for the original reduction relation, which is used as the basis for proofs in the remainder of the dissertation.

Once we have the property FD! in hand for the modified reduction, we prove the Church-Rosser property in Section 3.6 by a technique (due to Tait and Martin-Löf) given in [Barendregt, 1984] in which the originally-defined reduction of the calculus and the reduction defined by complete development from a term have the same transitive closure. The proof of the standard orders of evaluation based on FD! (Section 3.7) requires the definition of notions of head and internal redex for each calculus we treat. The role played by the property FD! here is to justify interchanging the order of head and internal reductions to give a standard-order reduction having the same result as an arbitrary prescribed reduction.

This chapter makes a rather complex use of some very standard and simple proof techniques from the pure lambda-calculus. The complexity derives largely from the much greater number and diverse character of the reduction rules in our calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. We pause before tackling this complexity to review some of the more basic techniques used.

3.2 Basic concepts and techniques of reduction semantics

The proofs in this chapter make use of a number of basic concepts and techniques used in reduction semantics. We collect these basic notions in this section both for future reference and to give a taste of some of the proof techniques to be employed in the remainder of this chapter.

First, we note that reduction relations can be regarded more generally in the setting of set-theoretic relations (sets of ordered pairs); hence, it makes sense to talk of the reflexive, symmetric, and transitive closures of reduction relations. We will also talk freely of one relation being a (proper) subset of another, or of the union of two relations. We denote the reflexive-transitive closure of a relation R by the Kleene star R^*. Especially in diagrams we also denote the reflexive-transitive closure of a reduction relation by a double-headed arrow.

Next, we introduce some definitions that are more typical of reduction relations.

Definition 3.2.1 (Diamond property) *A relation $\rightarrow$ is said to have the* diamond property *if $M \rightarrow M_1$ and $M \rightarrow M_2$ implies that there exists M' such that $M_1 \rightarrow M'$ and $M_2 \rightarrow M'$.*

The diamond property gets its name from its usual diagrammatic representation:

$$\begin{tikzcd}
& M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\
& M' & .
\end{tikzcd}$$

Definition 3.2.2 (Church-Rosser property) *A relation $\rightarrow$ is said to have the* Church-Rosser *property if its reflexive-transitive closure $\rightarrow^*$ has the diamond property.*

The Church-Rosser property is also called *confluence.*

Definition 3.2.3 (Strong normalization) *A reduction relation $\rightarrow$ is* strongly normalizing *if for every term M, all reduction sequences starting at M consist of a finite number of steps.*

Definition 3.2.4 (Weak Church-Rosser property) *A reduction relation $\rightarrow$ has the* weak *Church-Rosser property if for any term M, if $M \rightarrow M_1$ and $M \rightarrow M_2$, then there exists a term M' such that $M_1 \rightarrow^* M'$ and $M_2 \rightarrow^* M'$.*

The "weakness" with respect to the full Church-Rosser property arises from the use of only *single* reduction steps in the top legs of the diagram

$$\begin{tikzcd}
& M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dotted, two heads] & & M_2 \arrow[dl, dotted, two heads] \\
& M' & .
\end{tikzcd}$$

The weak Church-Rosser property is a convenient stepping-stone to proving the full Church-Rosser property for some calculi because of the following standard result:

Proposition 3.2.5 (Newman) *If a reduction relation $\rightarrow$ is both weakly Church-Rosser and strongly normalizing, then it is Church-Rosser.*

Proof: Standard: see [Barendregt, 1984], Proposition 3.1.25. ∎

The proof of the weak Church-Rosser property for a calculus can usually be structured as two nested case analyses. First and outermost, we classify cases by the relative position of the two redexes—whether Δ_1 and Δ_2 are identical, disjoint, or one is a subterm of the other. Second, we form a case for each pair of reduction rules by which Δ_1 and Δ_2 are redexes.

The standard argument for the outer case analysis is as follows:

Case (1) Δ_1 and Δ_2 are disjoint. In this case, each redex is trivially preserved when the other is reduced; reducing the preserved redex completes the diamond diagram.

Case (2) Δ_1 and Δ_2 are identical. In this case the diamond property can be satisfied by carrying out *no* reductions on the lower legs of the diamond diagram.

Case (3) Δ_1 contains Δ_2 as a proper subterm. This case requires consideration of every pair of reduction rules, potentially yielding a number of subcases that is the square of the number of reduction rules. However, we can usually avoid considering this many cases by using the notion of *critical overlap* or *critical pair* from term-rewriting theory (see, for instance, [Klop, 1992]; the crucial lemma is from [Knuth and Bendix, 1970; Huet, 1980]). The necessary observation is that a redex contained inside a meta-variable of a reduction rule will just be carried along intact when rewriting according to the rule. The intact redex is then still available for immediate reduction, so it is trivial to satisfy the diamond diagram. Only if the contained redex overlaps with the containing redex in a non-trivial way (by sharing some term-constructors) do we need to create a special case to establish that the reductions by the redexes Δ_1 and Δ_2 can be reconciled. It should be noted both that a rule may have a critical overlap with itself and that two rules may have a critical overlap in more than one distinct way. One consequence of the definition of critical overlap is that rules which act on redexes defined in terms of disjoint sets of term-constructors cannot form critical pairs.

Some caution is required when applying critical-pair reasoning in the context of lambda-calculi. In lambda-calculi, unlike in elementary term-rewriting systems, substitutable variables are part of the actual syntax of terms, not just meta-expressions for talking about terms. This means in particular that it is no longer necessarily true that the term denoted by a meta-variable is carried along intact, because one or more of its free variables may be substituted via a β-reduction or similar rule. It is often necessary to establish that relations are substitutive in order to use critical-pair reasoning when this occurs.

Case (4) Δ_2 contains Δ_1 as a proper subterm. This case can be argued by symmetry with Case 3 and so does not require separate consideration.

We close this section of basic techniques with a basic method for proving that a stepwise transformation of terms must terminate in a finite number of steps. The technique is simply to define a function that maps terms to nonnegative integers in such a way that the allowed transformations always strictly decrease the value of the function. Such a process must bump into zero after a finite number of iterations.

3.3 Factoring the notion of reduction

Among the reduction rules defining the reduction relation $\rightarrow$ for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, the rules *assoc* and *extend* are unusual in that they do not directly contribute to the semantic substance of the calculi. The rules β, δ, *fuse*, and *assign-result* directly model the intended notions of computation—substitution, primitive operations, and observation of assigned values, respectively. The rules *bubble-assign* and *bubble-new* assist the modeling of assignment by the rule *fuse*. The rules *assoc* and *extend*, however, only rearrange the syntax of command-terms so that the other rules may apply. In fact, the way in which these rearrangements interact with each other and with the other rules disrupts the effectiveness of the proof techniques to be used in Sections 3.4 and 3.5; we will explain the problem when we come to introduce those techniques. Section 3.3.1 considers the factoring-out of these rules as the basis for an equivalence relation on $\lambda[\beta\delta!eag]$- and $\lambda[\beta\delta!laz]$-terms; we then turn our attention in Section 3.3.2 from the reduction relations formed by all the rules of the calculi to a modified relation that operates on the equivalence classes induced by *assoc* and *extend*. It should be noted that this factoring of the notion of reduction is local to the present chapter, for we show in Section 3.8 that we can recover the Church-Rosser and standardization properties for the original reduction relation from those for the factored notion.

3.3.1 The reduction relation $\rightarrow_{\triangleright}$

Certain crucial technical properties of the calculi introduced in Chapter 2 are more easily proved if we somehow take account of the fact that the rules *assoc* and *extend* act to reorder subterms of a command sequence but don't actually have any computational significance of their own. This subsection introduces the reduction relation $\rightarrow_{\triangleright}$ (pronounced "association") based on these rules alone and studies some of its important properties. The conversion relation based on this reduction relation identifies all command sequences having the same commands in the same order, regardless of the way in which this sequence is built up from its subsequences using the operator $\triangleright$. For example, the terms $(m_1 \triangleright x_2 \cdot m_2) \triangleright x_3 \cdot (m_3 \triangleright x_4 \cdot m_4)$ and $m_1 \triangleright x_2 \cdot (m_2 \triangleright x_3 \cdot m_3) \triangleright x_4 \cdot m_4$ belong to the same equivalence class under this conversion relation. We will show, in fact, that the most easily notated member of this equivalence class, $m_1 \triangleright x_2 \cdot m_2 \triangleright x_3 \cdot m_3 \triangleright x_4 \cdot m_4$, which is the $\rightarrow_{\triangleright}$-normal form of the two previously-mentioned terms, serves as a useful canonical representative of the entire equivalence class.

Note. For the remainder of this chapter, the unadorned reduction arrow $\rightarrow$ will refer to the original reduction relations defined in Chapter 2 for the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

Lemma 3.3.1 *The reduction relation $\rightarrow_{\triangleright}$ is weakly Church-Rosser.*

Proof: It is sufficient to consider the cases in which the individual reduction rules *assoc* and *extend* have critical overlaps with themselves or each other. The following is an list of such cases (determined by inspection to be exhaustive), along with the diagrammatic evidence required for each case. There is a critical overlap between *assoc* and itself, and one between *assoc* and *extend*. There are no rewrite rules involving substitution in the definition of $\rightarrow_{\triangleright}$, so critical-pair reasoning can be applied without modification.

Notation. The diagrams we use to convey this proof use a good deal of notation to keep track of redexes, an issue that will become more important in later proofs that use the same notation. In this notation, large horizontal braces indicate the extent of the next redex to be reduced along some path in the diagram. A brace on top refers to the left-hand reduction path; one on the bottom, to the right-hand path. The redexes themselves are marked by overlining or underlining the term-constructors that match the left-hand side of a reduction rule. The marks remain on the term-constructors until the corresponding rule is applied; the location of the braces and of the lines are not intended to correspond. The arrow denoting the application of a rule that reduces a marked redex is marked the same way as the redex being reduced. Solid arrows denote reductions which are given in the hypotheses of a proposition; dotted arrows denote reductions which must be constructed to meet a proof obligation.

Case (1) Δ_1: *assoc*; Δ_2: *assoc*

The one possible overlap between the rule *assoc* and itself is the following:

$$\begin{tikzcd}[column sep=small]
& \overbrace{\underbrace{((N_1 \rhd \overline{x_2} \cdot N_2) \rhd \underline{\overline{x_3}} \cdot N_3)} \rhd \underline{x_4} \cdot N_4} \arrow[dl, "\underline{\mathit{assoc}}"'] \arrow[dr, "\overline{\mathit{assoc}}"] & \\
\overbrace{(N_1 \rhd \overline{x_2} \cdot N_2) \rhd \overline{x_3} \cdot (N_3 \rhd x_4 \cdot N_4)} \arrow[ddr, dotted, "\overline{\mathit{assoc}}"'] & & \underbrace{(N_1 \rhd x_2 \cdot (N_2 \rhd \underline{x_3} \cdot N_3)) \rhd \underline{x_4} \cdot N_4} \arrow[d, dotted, "\mathit{assoc}"] \\
& & N_1 \rhd x_2 \cdot \underbrace{((N_2 \rhd \underline{x_3} \cdot N_3) \rhd \underline{x_4} \cdot N_4)} \arrow[dl, dotted, "\underline{\mathit{assoc}}"] \\
& N_1 \rhd x_2 \cdot (N_2 \rhd x_3 \cdot (N_3 \rhd x_4 \cdot N_4)) &
\end{tikzcd}$$

.

Case (2) Δ_1: *assoc*; Δ_2: *extend*

Rule *assoc* overlaps rule *extend* as follows:

$$\begin{tikzcd}[column sep=small]
& \overbrace{\underbrace{((\overline{\nu v}.N_1) \rhd \underline{\overline{x_2}} \cdot N_2)} \rhd \underline{x_3} \cdot N_3} \arrow[dl, "\underline{\mathit{assoc}}"'] \arrow[dr, "\overline{\mathit{extend}}"] & \\
\overbrace{(\overline{\nu v}.N_1) \rhd \overline{x_2} \cdot (N_2 \rhd x_3 \cdot N_3)} \arrow[ddr, dotted, "\overline{\mathit{extend}}"'] & & \underbrace{(\nu v.(N_1 \rhd \underline{x_2} \cdot N_2)) \rhd \underline{x_3} \cdot N_3} \arrow[d, dotted, "\mathit{extend}"] \\
& & \nu v.\underbrace{((N_1 \rhd \underline{x_2} \cdot N_2) \rhd \underline{x_3} \cdot N_3)} \arrow[dl, dotted, "\underline{\mathit{assoc}}"] \\
& \nu v.(N_1 \rhd x_2 \cdot (N_2 \rhd x_3 \cdot N_3)) &
\end{tikzcd}$$

.

Since all critical overlaps between redexes can be reconciled, the weak Church-Rosser property follows by the usual argument involving non-overlapping redexes given in Section 3.2. ∎

Although we must defer a complete discussion to Section 3.4, the diagrams used in the proof of Lemma 3.3.1 contain the justification for the factoring of the reduction relation that we are now carrying out. The detail to note is that the right-hand path in each diagram requires the reduction of a redex that was not present in the original term in order to re-establish one of the original redexes.

Lemma 3.3.2 *The reduction relation* $\rightarrow_{\rhd}$ *is strongly normalizing.*

Proof: This is almost obvious, but we give a formal proof anyhow. We define a positive integral measure on terms that is strictly decreased by each *assoc*- or *extend*-reduction. This implies that every such reduction sequence must terminate.

We define the measure $\mathcal{M}[\![\,]\!]$ via the equations

$$\begin{array}{rclrcl} \mathcal{M}[\![x]\!] & = & 1 & \mathcal{M}[\![f]\!] & = & 1 \\ \mathcal{M}[\![v]\!] & = & 1 & \mathcal{M}[\![M \triangleright x \cdot N]\!] & = & 2\mathcal{M}[\![M]\!] + \mathcal{M}[\![N]\!] \\ \mathcal{M}[\![c^n]\!] & = & 1 & \mathcal{M}[\![\nu v\, M]\!] & = & 1 + \mathcal{M}[\![M]\!], \end{array}$$

where it is understood that $\mathcal{M}[\![\,]\!]$ is additive on all other compound terms. Note that $\mathcal{M}[\![M]\!]$ is at least 1 for every term M.

We now show that $\mathcal{M}[\![\,]\!]$ decreases under $\twoheadrightarrow_{\triangleright}$; that is, that if $M \twoheadrightarrow_{\triangleright} N$, then $\mathcal{M}[\![M]\!] > \mathcal{M}[\![N]\!]$. We have one case to consider for each of the two reduction rules defining $\twoheadrightarrow_{\triangleright}$.

Case (1) *assoc*: $(M_1 \triangleright x_2 \cdot M_2) \triangleright x_3 \cdot M_3 \twoheadrightarrow_{\triangleright} M_1 \triangleright x_2 \cdot M_2 \triangleright x_3 \cdot M_3$.

In this case, we calculate

$$\begin{array}{rcl} \mathcal{M}[\![(M_1 \triangleright x_2 \cdot M_2) \triangleright x_3 \cdot M_3]\!] & = & 2\mathcal{M}[\![M_1 \triangleright x_2 \cdot M_2]\!] + \mathcal{M}[\![M_3]\!] \\ & = & 2(2\mathcal{M}[\![M_1]\!] + \mathcal{M}[\![M_2]\!]) + \mathcal{M}[\![M_3]\!] \\ & = & 4\mathcal{M}[\![M_1]\!] + 2\mathcal{M}[\![M_2]\!] + \mathcal{M}[\![M_3]\!] \\ & > & 2\mathcal{M}[\![M_1]\!] + 2\mathcal{M}[\![M_2]\!] + \mathcal{M}[\![M_3]\!] \\ & = & \mathcal{M}[\![M_1 \triangleright x_2 \cdot M_2 \triangleright x_3 \cdot M_3]\!]. \end{array}$$

Case (2) *extend*: $(\nu v.M_1) \triangleright x_2 \cdot M_2 \twoheadrightarrow_{\triangleright} \nu v.M_1 \triangleright x_2 \cdot M_2$.

In this case, we calculate

$$\begin{array}{rcl} \mathcal{M}[\![(\nu v\, M_1) \triangleright x_2 \cdot M_2]\!] & = & 2\mathcal{M}[\![\nu v\, M_1]\!] + \mathcal{M}[\![M_2]\!] \\ & = & 2\mathcal{M}[\![M_1]\!] + \mathcal{M}[\![M_2]\!] + 2 \\ & > & 2\mathcal{M}[\![M_1]\!] + \mathcal{M}[\![M_2]\!] + 1 \\ & = & \mathcal{M}[\![\nu v\, M_1 \triangleright x_2 \cdot M_2]\!]. \end{array}$$

In the first case, the inequality depends on knowing that $\mathcal{M}[\![M]\!]$ is always greater than zero. The decrease in $\mathcal{M}[\![M]\!]$, which we have demonstrated for redexes only, extends to entire terms by noting that the measure of subterms always contributes positively to the measure of a superterm. Since $\mathcal{M}[\![\,]\!]$ decreases at every step yet remains bounded below by 0, the number of steps must be finite. ∎

Theorem 3.3.3 *The reduction relation $\twoheadrightarrow_{\triangleright}$ is Church-Rosser.*

Proof: This follows by Newman's Lemma (Proposition 3.2.5) from Lemmas 3.3.1 and 3.3.2. ∎

Notation 3.3.4 *We will denote the $\twoheadrightarrow_{\triangleright}$-normal form of a term M by $\downarrow_{\triangleright} [M]$.*

Definition 3.3.5 *The equivalence relation $\overset{\triangleright}{=}$,* association equivalence, *is the equivalence closure of $\twoheadrightarrow_{\triangleright}$.*

Proposition 3.3.6 *The relation $\overset{\triangleright}{=}$ is decidable.*

Proof: The confluence of $\twoheadrightarrow_{\triangleright}$ implies that terms related by $\overset{\triangleright}{=}$ share a common, unique $\twoheadrightarrow_{\triangleright}$-normal form. Since all $\twoheadrightarrow_{\triangleright}$-reduction sequences terminate, it is sufficient to reduce two candidate terms to $\twoheadrightarrow_{\triangleright}$-normal form and check for syntactic equivalence. ∎

The next proposition establishes that the reduction relation $\twoheadrightarrow_{\triangleright}$ possesses a standard reduction order for reduction to normal form.

Definition 3.3.7 *A* deterministic reduction strategy *is an effective procedure that, given a term, returns a redex within that term if there is one and otherwise indicates that the term is in normal form.*

A normalizing *reduction strategy is one that, when iterated starting with any term, always leads to a normal form (if one exists).*

Proposition 3.3.8 (Standardization) *There exists a normalizing deterministic reduction strategy for $\twoheadrightarrow_{\triangleright}$.*

Proof: The reduction relation $\twoheadrightarrow_{\triangleright}$ is both confluent and terminating, so any method will work as long as it identifies a next redex when one exists. To construct a deterministic such method, define an ordering of subterms for each syntactic construct, use this ordering to define a preorder traversal of subterms, and stipulate that the first redex encountered in this traversal is the next to be reduced (Definition 3.7.1 gives an example of such an ordering). ∎

Unlike the standardization theorem for the pure lambda-calculus (see [Barendregt, 1984], Theorem 11.4.7) Proposition 3.3.8 does not promise a standard reduction sequence from M to any reduct of M whatsoever. However, this weaker theorem suffices for our purposes.

Having now introduced the important properties of the relation $\twoheadrightarrow_{\triangleright}$, we are now ready to discuss its intended application.

3.3.2 The modified computational reduction relation $\twoheadrightarrow_{!}$

The definition of $\twoheadrightarrow_{\triangleright}$ in Section 3.3.1 allows us to define a new reduction relation capturing only the essential computational meaning of the original reduction relation for the calculi of concern. We define the new reduction $\twoheadrightarrow_{!}$ (which we will call "computational reduction" or "reduction modulo association") by a simple construction on the quotient set of terms under the equivalence relation $\overset{\triangleright}{=}$.

The following definitions overload the notation $\twoheadrightarrow_{!}$, but each context of use will make clear which meaning is intended.

Definition 3.3.9 ($\twoheadrightarrow_{!}$ on terms) *The reduction relation $\twoheadrightarrow_{!}$ on terms for* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *is defined by all the rules in Figures 2.15, 2.17, and 2.18 except assoc and extend.*

Notation 3.3.10 *We use $[M]_{\overset{\triangleright}{=}}$ to denote the $\overset{\triangleright}{=}$-equivalence-class containing a representative term M.*

Definition 3.3.11 ($\twoheadrightarrow_{!}$ on classes) *The reduction relation $\twoheadrightarrow_{!}$ holds between $\overset{\triangleright}{=}$-equivalence-classes of terms whenever $\twoheadrightarrow_{!}$ holds between some pair of representatives, that is, $[M]_{\overset{\triangleright}{=}} \twoheadrightarrow_{!} [N]_{\overset{\triangleright}{=}}$ if and only if there exist terms M' and N' such that $M \overset{\triangleright}{=} M'$, $N \overset{\triangleright}{=} N'$, and $M' \twoheadrightarrow_{!} N'$.*

Definition 3.3.11 appears to make it very difficult to compute $\twoheadrightarrow_{!}$ on classes—surely we cannot be expected to see where every $\twoheadrightarrow_{!}$ relation from every element of $[M]_{\overset{\triangleright}{=}}$ leads just to see if one happens to lead to an element of $[N]_{\overset{\triangleright}{=}}$! We will see below, however, that Proposition 3.4.1 and Corollary 3.4.2 give us an easier way: any $\twoheadrightarrow_{!}$-redex present in any term in $[M]_{\overset{\triangleright}{=}}$ is also present in $\downarrow_{\triangleright}[M]$. Therefore one reduction step under this new reduction relation can be computed by one reduction step via any rule except *assoc* or *extend*, followed by a reduction to $\twoheadrightarrow_{\triangleright}$-normal form. Lemma 3.3.2 guarantees that this normal form exists; Theorem 3.3.3 guarantees that the new relation is uniquely defined.

The process just described uses the common, unique $\twoheadrightarrow_{\triangleright}$-normal form shared by all elements of such an equivalence class as the canonical representative of the class. Finding such a representative is computable (Proposition 3.3.6), so the possibility of nontermination resides entirely in the new reduction relation $\twoheadrightarrow_{!}$. In the sequel we will often use this conception of alternating $\twoheadrightarrow_{\triangleright}$- and $\twoheadrightarrow_{!}$-reductions to blur the formal definition of $\twoheadrightarrow_{!}$ and work directly with the underlying reductions on the original term language.

For an example showing how $\twoheadrightarrow_{!}$ differs from $\twoheadrightarrow$, consider the reduction of the term

$$(\lambda x.x \triangleright y \cdot r) \bullet (p \triangleright z \cdot q).$$

Under $\twoheadrightarrow$, this term reduces via β to

$$(p \triangleright z \cdot q) \triangleright y \cdot r,$$

which is itself an *assoc*-redex; reducing this redex is counted as a second reduction to the final result

$$p \triangleright z \cdot q \triangleright y \cdot r.$$

In contrast, under $\rightarrow_!$ we are working with $\overset{\triangleright}{=}$-equivalence classes of terms. The intial term is in $\overset{\triangleright}{=}$-normal form; its reduct, however, contains the $\rightarrow_\triangleright$-redex just pointed out. The $\rightarrow$-reductions induce the one-step $\rightarrow_!$-reduction

$$[(\lambda x.x \triangleright y \cdot r) \bullet (p \triangleright z \cdot q)]_{\overset{\triangleright}{=}} \rightarrow_! [p \triangleright z \cdot q \triangleright y \cdot r]_{\overset{\triangleright}{=}}.$$

Some precedents. The singling out of $\rightarrow_\triangleright$ as a non-computational subset of the overall reduction relation is somewhat reminiscent of the "structural congruences" employed in the exposition of the polyadic π-calculus in [Milner, 1991]. There (as here) the axioms in question are essential for *defining* computations, but do not themselves model the progress of a computation.

There is also an analogy between the definition of $\rightarrow_!$ and the α-renaming convention adopted in [Barendregt, 1984]. Under the α-renaming convention we agree to give all bound and free variables different names before each reduction step. This convention sweeps the complex process of renaming variables under the rug, but allows us to avoid painfully endless invocations of the phrase "up to α-equivalence" as well as numerous side-conditions to the effect that a particular variable does not occur free in a certain subterm. Terms are thus regarded as representatives of their α-equivalence class, the classes being the true computational objects.

Like the α-renaming convention, the definition of $\rightarrow_!$ requires us to think in terms of equivalence classes of terms. This definition folds this (slight) complexity into the basic notion of reduction, but allows us to avoid complexity elsewhere. When using $\rightarrow_!$, we consider terms as representatives of their $\rightarrow_\triangleright$-equivalence classes, the classes being the true computational objects. Corollary 3.4.2 provides the foundation for this view of $\rightarrow_!$-reduction.

3.4 Keeping track of redexes

The proofs of the Church-Rosser and standardization properties depend ultimately on the details of the ways in which different choices of reductions can be reconciled. Proving the Church-Rosser property requires that such reconciliations always exist; proving standardization requires that interchanging the order of reductions so as to fit the definition of standard order is always possible. In this section we present the basic definitions for keeping track of reductions in support of these proofs. In Section 3.5 we apply these techniques to gather information about the reduction relations in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

3.4.1 Marked reductions

The basic tool for keeping track of redexes, already introduced in the diagrams for the proof of Lemma 3.3.1, is to label the term-constructors of redexes with distinctive marks.[1] The marks for a particular redex are erased when that particular redex is reduced but not when other redexes are reduced. For example, the following term has two β-redexes, marked with overlines and underlines respectively:

$$(\underline{\lambda x}.(\overline{\lambda y}.y)\,\overline{\bullet}\,x)\,\underline{\bullet}\,z.$$

A β-redex is specified in terms of two syntactic term constructors: one for the application term-constructor, one for the λ-abstraction term-constructor. Both are marked, and both disappear when the redex is reduced. When the underlined redex above is reduced, the resulting marked term is

$$(\overline{\lambda y}.y)\,\overline{\bullet}\,z.$$

The formal specification of the concept of marked redexes and marked reductions is best left as a sketch—a full development yields no great advantage in precision. The addition of marks creates a new language of

[1]The marking technique presented here is adapted from [Barendregt, 1984], Chapter 11.

terms, with a lifting function that embeds terms from the original language into the marked language as unmarked terms, and an erasing function that provides a mapping in the reverse direction by forgetting the marks. Reduction rules from the unmarked language are lifted to the marked language as already discussed; the addition of these rules turns the marked language into a calculus. Every reduction sequence in the marked calculus is reflected in the unmarked calculus by erasing the marks.

The converse, that all unmarked reduction sequences can be lifted into the marked calculus, is not true; this fact underlies the technical usefulness of the marking technique. Of course, it takes a theorem to show that this converse does not hold, but the basic idea is easy: reductions destroy marks but cannot create them, so sequences consisting only of marked reductions are finite. On the other hand, untyped lambda-calculi admit infinite reduction sequences.

The syntactic form of our store-computation redexes introduces a complication not present in the basic lambda-calculus $\lambda[\beta\delta]$. In $\lambda[\beta\delta]$, β- and δ-redexes cannot overlap in the sense of sharing a syntactic term-constructor. For example, any β- or δ-redex in the term $(\lambda x.M) \bullet N$ over and above the redex that is the root of the term must lie entirely within one of the subterms denoted by the metavariables M and N. It is not so in either of our lambda-calculi with assignment. An example is the term $(\overline{\uparrow} N_1 \underline{\overline{\rhd x_2}} \cdot N_2) \underline{\rhd x_3} \cdot N_3$, in which we have marked with an underline the constructors of terms defining an *assoc*-redex and have marked with a line above the constructors of terms defining a *unit*-redex. We have given the term constructor $\rhd x_2 \cdot$ both marks. To accommodate overlapping redexes, we must formally require that our terms be marked with *sets* of marks, and that the mark for a particular redex appear in the mark-set for all and only those subterms which characterize the redex. We call terms marked with the empty set *unmarked* even when they are considered as elements of a marked calculus. The uses of the marking apparatus that actually appear in the ensuing proofs actually use no more than two marks.

The marking technique as presented so far is only applicable to a calculus defined directly in terms of reduction rules. In order to use marked terms in the proofs we actually want to carry out, we need to show how marked reductions interact with the definition of $\rightarrow_!$ in terms of $\overset{\rhd}{=}$-equivalence classes. This adaptation is the topic of the next subsection.

3.4.2 Interaction of $\rightarrow_\rhd$ with marked computational reduction

We now state and prove a result that lets us reason with the computational reduction $\rightarrow_!$ in a fairly simple manner. Although the method of defining $\rightarrow_!$ by carrying out a $\rightarrow_\rhd$-normalization after every $\rightarrow_!$-step is conceptually simple, it can lead to complicated proofs. The following result gives us some leeway in the placement of $\rightarrow_\rhd$-steps with respect to $\rightarrow_!$-steps when constructing proofs: it shows that we can always choose to perform some of the $\rightarrow_\rhd$-reductions before a $\rightarrow_!$-reduction without changing the interpretation of the reduction sequence in terms of $\overset{\rhd}{=}$-equivalence classes.

Proposition 3.4.1 (Commutation of association) *Suppose we have $M \rightarrow_! N$ via reduction of a marked redex Δ, and suppose also that $M \rightarrow_\rhd M'$. Then there exist terms M'' and N' such that $M' \rightarrow_\rhd^* M''$, $N \rightarrow_\rhd^* N'$, and $M'' \rightarrow_! N'$ by reducing Δ'', where Δ'' is the residual of Δ in M''.*

$$\begin{tikzcd}
 & M \arrow[dl, "\rightarrow_{!}"', "\Delta"] \arrow[dr, "\rightarrow_{\rhd}"] & \\
N \arrow[ddrr, dotted, "\rightarrow_{\rhd}"'] & & M' \arrow[d, dotted, "\rightarrow_{\rhd}"] \\
 & & M'' \arrow[d, dotted, "\Delta''"', "\rightarrow_{!}"] \\
 & & N'
\end{tikzcd}$$

Proof: The existence of Δ'' is part of what we have to prove. It is possible that a residual of Δ might not be a redex, but the proof will show that it will always be possible to re-establish redexhood via a further $\rightarrow_{\rhd}$-reduction. This fact is what makes Proposition 3.4.1 useful.

Following the general pattern of argument given in Section 3.2, we argue by a case analysis on the relative positions of the $\rightarrow_{\rhd}$- and $\rightarrow_{!}$-redexes, and also consider the particular rules defining the redexes.

The arguments reducing the case analysis to cases of critical overlap are valid in this case because the reduction rules defining $\rightarrow_{\rhd}$ are obviously substitutive. We now proceed to examine the critical overlaps between $\rightarrow_{\rhd}$-rules and $\rightarrow_{!}$-rules. We prove each case by means of a diagram in which we mark the term-constructors of the redex Δ. The diagrams themselves correspond to the diagram that states the theorem.

Case (1)

$$\begin{tikzcd}
\overbrace{(\underbrace{\underline{\uparrow} M_1 \underline{\rhd x_2} \cdot M_2}) \rhd x_3 \cdot M_3} \arrow[dd, "\underline{unit}"'] \arrow[dr, "assoc"] & \\
 & \overbrace{\underline{\uparrow} M_1 \underline{\rhd x_2} \cdot (M_2 \rhd x_3 \cdot M_3)} \arrow[dl, dotted, "\underline{unit}"] \\
{[M_1/x_2](M_2 \rhd x_3 \cdot M_3) \equiv ([M_1/x_2]M_2) \rhd x_3 \cdot M_3} &
\end{tikzcd}$$

The syntactic identity $([M_1/x_2]M_2) \rhd x_3 \cdot M_3 \equiv [M_1/x_2](M_2 \rhd x_3 \cdot M_3)$ holds because there can be no occurrences of x_2 in M_3, which was outside the scope of x_2 in the original term M. The α-renaming convention (Section 2.1) assures that no free occurrences of x_2 have been introduced.

Case (2)

$$\begin{tikzcd}
& \overbrace{\underbrace{(v \underline{:=} M_1 \underline{;}\ (v\underline{?}\, \underline{\triangleright x_2} \cdot M_2))} \triangleright x_3 \cdot M_3} \arrow[dl, "\underline{fuse}"'] \arrow[dr, "assoc"] & \\
\underbrace{(v := M_1;\ [M_1/x_2]M_2) \triangleright x_3 \cdot M_3} \arrow[ddr, dotted, "assoc"'] & & v \underline{:=} M_1 \underline{;}\ \overbrace{((v\underline{?}\,\underline{\triangleright x_2}\cdot M_2)\triangleright x_3 \cdot M_3)} \arrow[d, dotted, "assoc"] \\
& & v\underline{:=}M_1\underline{;}\ \overbrace{(v\underline{?}\,\underline{\triangleright x_2}\cdot(M_2\triangleright x_3\cdot M_3))} \arrow[dl, dotted, "\underline{fuse}"] \\
& v := M_1;\ [M_1/x_2](M_2 \triangleright x_3 \cdot M_3) & \\
& \equiv\ v := M_1;\ ([M_1/x_2]M_2 \triangleright x_3 \cdot M_3) &
\end{tikzcd}$$

The syntactic equivalence of the two forms given for the final term is again justified by the α-renaming convention.

Case (3)

$$\begin{tikzcd}
& \overbrace{\underbrace{(v \underline{:=} M_1 \underline{;}\ (w\underline{?}\, \underline{\triangleright x_2} \cdot M_2))} \triangleright x_3 \cdot M_3} \arrow[dl, "\underline{bubble\text{-}assign}"'] \arrow[dr, "assoc"] & \\
\underbrace{(w? \triangleright x_2 \cdot (v := M_1;\ M_2)) \triangleright x_3 \cdot M_3} \arrow[d, dotted, "assoc"'] & & v \underline{:=} M_1 \underline{;}\ \overbrace{((w\underline{?}\,\underline{\triangleright x_2}\cdot M_2)\triangleright x_3 \cdot M_3)} \arrow[d, dotted, "assoc"] \\
w? \triangleright x_2 \cdot \underbrace{((v := M_1;\ M_2) \triangleright x_3 \cdot M_3)} \arrow[dr, dotted, "assoc"'] & & v\underline{:=}M_1\underline{;}\ \overbrace{(w\underline{?}\,\underline{\triangleright x_2}\cdot(M_2\triangleright x_3\cdot M_3))} \arrow[dl, dotted, "\underline{bubble\text{-}assign}"] \\
& w? \triangleright x_2 \cdot (v := M_1;\ M_2 \triangleright x_3 \cdot M_3) &
\end{tikzcd}$$

Case (4)

$$\begin{tikzcd}[column sep=small]
\overbrace{\underbrace{(v \underline{:=} M_1 \underline{\triangleright x_2} \cdot M_2)} \triangleright x_3 \cdot M_3}, \quad (x_2 \in fv\, M_2) \arrow[dd, "\underline{assign\text{-}result}"'] \arrow[dr, "assoc"] & \\
& \overbrace{v \underline{:=} M_1 \underline{\triangleright x_2} \cdot (M_2 \triangleright x_3 \cdot M_3)}, \ (x_2 \in fv\, M_2) \arrow[dl, dotted, "\underline{assign\text{-}result}"] \\
(v := M_1;\ [()/x_2] M_2) \triangleright x_3 \cdot M_3 \equiv v := M_1;\ [()/x_2](M_2 \triangleright x_3 \cdot M_3) &
\end{tikzcd}$$

The last reduction again depends on the fact that there can be no occurrences of x_2 in M_3.

Case (5)

$$\begin{tikzcd}[column sep=small]
& \overbrace{\underbrace{(\underline{\nu v}.(w\underline{?} \underline{\triangleright x_1} \cdot M_1))} \triangleright x_2 \cdot M_2} \arrow[dl, "\underline{bubble\text{-}new}"'] \arrow[dr, "extend"] & \\
\underbrace{(w? \triangleright x_1 \cdot \nu v\, M_1) \triangleright x_2 \cdot M_2} \arrow[d, dotted, "assoc"'] & & \underline{\nu v}.(\overbrace{(w\underline{?} \underline{\triangleright x_1} \cdot M_1) \triangleright x_2 \cdot M_2}) \arrow[d, dotted, "assoc"] \\
w? \triangleright x_1 \cdot \underbrace{(\nu v\, M_1) \triangleright x_2 \cdot M_2} \arrow[dr, dotted, "extend"'] & & \underline{\nu v}.(\overbrace{w\underline{?} \underline{\triangleright x_1} \cdot (M_1 \triangleright x_2 \cdot M_2)}) \arrow[dl, dotted, "\underline{bubble\text{-}new}"] \\
& w? \triangleright x_1 \cdot (\nu v.(M_1 \triangleright x_2 \cdot M_2)) &
\end{tikzcd}$$

Case (6) We now consider the purification rules. For $\lambda[\beta\delta!eag]$, the form of the prefixes $S^{eag}[]$ precludes any overlaps between *pure-eager*-redexes and $\rightarrow_{\triangleright}$-redexes—the store-prefixes allowed in *pure-eager*-redexes are in $\rightarrow_{\triangleright}$-normal form. We thus have only to consider *pure-lazy*-redexes. We first consider the interaction of *pure-lazy* with *extend.*

$$\begin{tikzcd}[column sep=small]
& \overbrace{\underbrace{\underline{\textbf{pure}}\, S_1^{laz}[(\nu\nu.M) \triangleright x \cdot S_2^{laz}[\uparrow (\underline{\lambda y}.N)]]}} \arrow[dl, "\underline{pure\text{-}lazy}"'] \arrow[dr, "extend"] & \\
\lambda y.\textbf{pure}\, \underbrace{S_1^{laz}[(\nu\nu.M) \triangleright x \cdot S_2^{laz}[\uparrow N]]} \arrow[dr, dotted, "extend"'] & & \overbrace{\underline{\textbf{pure}}\, S_1^{laz}[\nu\nu\, M \triangleright x \cdot S_2^{laz}[\uparrow (\underline{\lambda y}.N)]]} \arrow[dl, dotted, "\underline{pure\text{-}lazy}"] \\
& \lambda y.\textbf{pure}\, S_1^{laz}[\nu\nu\, M \triangleright x \cdot S_2^{laz}[\uparrow N]] &
\end{tikzcd}$$

The cases involving the rules for purifying constructed values and primitives, as well as the corresponding cases involving *assoc*, behave analogously.

The completion of this case analysis concludes the proof of Proposition 3.4.1. ∎

As a corollary to Proposition 3.4.1, we can show that a $\twoheadrightarrow_{\triangleright}$-normal form contains every $\twoheadrightarrow_{!}$-redex present in any member of its $\overset{\triangleright}{=}$-equivalence class. Hence these normal forms make a handy canonical representative of the class when one is needed.

Corollary 3.4.2 (Completeness of $\twoheadrightarrow_{\triangleright}$-normal form) *If Δ is a $\twoheadrightarrow_{!}$-redex within any term in a $\overset{\triangleright}{=}$-equivalence class $[M]_{\overset{\triangleright}{=}}$, then there exists a residual Δ' of Δ within the normal form $\downarrow_{\triangleright}[M]$, and Δ' is a redex. Moreover, if $M \twoheadrightarrow_{!} M'$ by reducing Δ and $\downarrow_{\triangleright}[M] \twoheadrightarrow_{!} M''$ by reducing Δ', then $M' \twoheadrightarrow_{\triangleright}^{*} M''$.*

Proof: In diagram form, the corollary states

$$\begin{tikzcd}
& M \in [M]_{\overset{\triangleright}{=}} \arrow[dl, "\Delta"'] \arrow[dr] & \\
M' \in [M']_{\overset{\triangleright}{=}} \arrow[dr, dotted] & & \downarrow_{\triangleright}[M] \arrow[dl, dotted, "\Delta'"'] \\
& M'' \in [M'']_{\overset{\triangleright}{=}} &
\end{tikzcd}$$

We need only stitch together, from upper left to lower right, occurrences of the diagram given in the statement of Proposition 3.4.1. At each step, we re-establish a residual of Δ as a redex using $\twoheadrightarrow_{\triangleright}$-reductions only, hence we stay within the equivalence class. Since $\twoheadrightarrow_{\triangleright}$ is strongly normalizing and Church-Rosser, the process terminates at $\downarrow_{\triangleright}[M]$. Starting this process at every element of $[M]_{\overset{\triangleright}{=}}$ shows that every marked $\twoheadrightarrow_{!}$-redex of every term in the equivalence class is represented by a residual redex in the normal form that characterizes the class. ∎

It should be noted that Corollary 3.4.2 does *not* state that $M'' \equiv \downarrow_{\triangleright}[M']$: this isn't true. For an example, consider the term

$$(\lambda y.y \triangleright x_2 \cdot z_2) \bullet z_1 \triangleright x_3 \cdot z_3 .$$

This term is in $\twoheadrightarrow_{\triangleright}$-normal form, and so can play the role of both M and $\downarrow_{\triangleright}[M]$ in the statement of the corollary. However, reducing the β-redex yields (in the role of M')

$$(z_1 \triangleright x_2 \cdot z_2) \triangleright x_3 \cdot z_3,$$

which is not in $\twoheadrightarrow_{\triangleright}$-normal form, being indeed an *assoc*-redex.

A further corollary to Proposition 3.4.1 advances us toward our goal of proving the Church-Rosser property for our calculi. This corollary allows us to reduce the task of establishing diamond-shaped reduction diagrams for $\twoheadrightarrow_{!}$ on equivalence classes to proving the corresponding diagrams on terms. We will show later (in Lemma 3.5.5) that the weak Church-Rosser properties does in fact hold for terms, and hence (by the following result) for equivalence classes.

Corollary 3.4.3 (Diamond-lifting) *If the property implied by the diamond diagram*

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\ & M_3 & \end{tikzcd} \tag{3.1}$$

holds for $\twoheadrightarrow_{!}$ on terms in $\lambda[\beta\delta!eag]$ or $\lambda[\beta\delta!laz]$, then the corresponding diagram

$$\begin{tikzcd} & {[M]_{\overset{\triangleright}{=}}} \arrow[dl] \arrow[dr] & \\ {[M_1]_{\overset{\triangleright}{=}}} \arrow[dr, dotted] & & {[M_2]_{\overset{\triangleright}{=}}} \arrow[dl, dotted] \\ & {[M_3]_{\overset{\triangleright}{=}}} & \end{tikzcd} \tag{3.2}$$

holds for $\twoheadrightarrow_{!}$ on $\overset{\triangleright}{=}$-equivalence classes in the same calculus.

Proof: Suppose we are given the solid arrows in the diagram (3.2). By Definition 3.3.11, this implies the existence of reductions $M' \twoheadrightarrow_{!} M'_1$, $M'' \twoheadrightarrow_{!} M''_2$, where $M', M'' \in [M]_{\overset{\triangleright}{=}}$, $M'_1 \in [M_1]_{\overset{\triangleright}{=}}$, $M''_2 \in [M_2]_{\overset{\triangleright}{=}}$. By Corollary 3.4.2, similarly-marked reductions will be available starting from the single initial term $\downarrow_{\triangleright}[M]$. Since we are assuming that the reduction on terms is weakly Church-Rosser (diagram (3.1)), we are given the two dotted reduction sequences ending in the same term, and hence (using Definition 3.3.11 in the other direction) we obtain the dotted reduction sequences in diagram (3.2). ∎

3.5 Strong finiteness of developments modulo association

We now set out to prove the strong finiteness of developments modulo association via Newman's Lemma (Proposition 3.2.5) for our two calculi. We first define our terms, and then proceed in subsection 3.5.1 to prove that the marked version of $\twoheadrightarrow_{!}$ is weakly Church-Rosser on terms. We address the termination issue in subsection 3.5.2.

This section is concerned with the notion of *developments* of a term in $\lambda[\beta\delta!eag]$ or $\lambda[\beta\delta!laz]$. We give two definitions of this notion: the first is somewhat easier to understand; the second relates to the proof techniques we actually use.

Definition 3.5.1 (Developments) *Given a term M and a set $\mathcal{F}$ of redexes that are subterms of M, a* development *is reduction sequence starting with M in which only members of $\mathcal{F}$ or their residuals are reduced.*

Definition 3.5.2 (Developments) *Given a term M in the marked calculus, a* development *is a sequence of marked reductions starting with M.*

In both cases, we sometimes use the term "development" informally to refer to the final term in the reduction sequence that forms a development.

Corresponding to the two definitions of developments, we give two definitions of the important notion of *complete* development.

Definition 3.5.3 (Complete developments) *A* complete *development of a term M relative to a set of redexes $\mathcal{F}$ is one in which all the members of $\mathcal{F}$ are reduced.*

Definition 3.5.4 (Complete developments) *A* complete *development of a term M is one in which all marks are erased.*

The weak finite-developments property can perhaps best be thought of in terms of its contrapositive statement: any infinite reduction sequence must create new redexes.

3.5.1 The weak Church-Rosser property

In this subsection we establish that marked $\rightarrow_!$-reductions for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ are weakly Church-Rosser on terms; Corollary 3.4.3 will then establish the weak Church-Rosser property for marked $\rightarrow_!$ on equivalence classes.

Lemma 3.5.5 *The weak Church-Rosser property holds for the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ under marked $\rightarrow_!$-reduction on $\stackrel{\triangleright}{=}$-equivalence classes.*

Proof: We consider all the critical overlaps between all the $\rightarrow_!$-rules in both our calculi; we show that in every case the diamond can be completed. In contrast with the proof of Lemma 3.3.1, we are now dealing with rules involving substitution so we cannot rely completely on critical-pair reasoning. Instead, we will have to account carefully for the interaction of substitutions arising from β-, *unit-*, *fuse-*, and *assign-result*-redexes.

Case (1): Δ_1: β; Δ_2: β

There are two ways a β-redex can be embedded within another: in the function subterm or in the argument subterm of the redex's application. We consider the two cases separately.

(i) The first way in which two β-redexes can interact has the inner redex within the function-subterm of the outer redex.

$$\begin{tikzcd}
 & \overbrace{(\underline{\lambda x_1}.C[\underbrace{(\overline{\lambda x_2}.P_2)\,\bar{\bullet}\,Q_2}])\,\underline{\bullet}\,Q_1} \arrow[dl, "\underline{\beta}"'] \arrow[dr, "\bar{\beta}"] & \\
\begin{array}{ll} & [Q_1/x_1]C[\overbrace{(\overline{\lambda x_2}.P_2)\,\bar{\bullet}\,Q_2}] \\ \equiv & ([Q_1/x_1]C)[\overbrace{(\overline{\lambda x_2}.[Q_1/x_1]P_2)\,\bar{\bullet}\,[Q_1/x_1]Q_2}] \end{array} \arrow[dr, dotted, "\bar{\beta}"'] & & \underbrace{(\lambda x_1.C[[Q_2/x_2]P_2])\,\underline{\bullet}\,Q_1} \arrow[dl, dotted, "\underline{\beta}"] \\
 & \begin{array}{ll} & ([Q_1/x_1]C)[[[Q_1/x_1]Q_2/x_2]([Q_1/x_1]P_2)] \\ \equiv & ([Q_1/x_1]C)[[Q_1/x_1][Q_2/x_2]P_2] \\ \equiv & [Q_1/x_1]C[[Q_2/x_2]P_2] \end{array} &
\end{tikzcd}$$

This diagram is rendered nonobvious by the fact that substitution is not a term-constructor but rather a meta-level operator that denotes the term that is the result of the substitution. The second form of the intermediate term on the left-hand leg of the diamond diagram above emphasises this fact, and the equivalence in the final term in the diagram is a consequence of the Substitution Lemma (Lemma 2.1.2). We use the notation $([M/x]\,C)[\,]$ to stand for the context that is derived from the context $C[\;]$ by substituting as indicated.

(ii) The second way in which two β-redexes may interact has the inner redex in the argument term of the outer redex. The following diagram shows how the requirements of the weak Church-Rosser property are established in this case:

$$\begin{tikzcd}[column sep=small]
& \overbrace{(\underline{\lambda x_1}.P_1)\,\underline{\bullet}\,C[\underbrace{(\overline{\lambda x_2}.P_2)\,\bar{\bullet}\,Q_2}]} \arrow[dl, "\underline{\beta}"'] \arrow[dr, "\overline{\beta}"] & \\
{[C[\overbrace{(\overline{\lambda x_2}.P_2)\,\bar{\bullet}\,Q_2}]/x_1]P_1} \arrow[dr, dotted, twoheadrightarrow, "\overline{\beta}"] & & \underbrace{(\underline{\lambda x_1}.P_1)\,\underline{\bullet}\,C[[Q_2/x_2]P_2]} \arrow[dl, dotted, "\underline{\beta}"'] \\
& {[C[[Q_2/x_2]P_2]/x_1]P_1} &
\end{tikzcd}$$

Here the lower-left arrow in the diamond has a double head because there may be multiple occurrences of x_1 in P_1, leading to multiple occurrences of the second β-redex to reduce.

All the other interactions of rule β with other reduction rules follow one or the other of these prototypes, so we do not work out all the diagrams for those cases explicitly.

Case (2): Δ_1: β; Δ_2: δ We give only the case in which the β-redex encloses the δ-redex. The case in which the enclosure relationship is reversed presents no difficulties due to substitution. We also give only the form of δ-rule dealing with an abstraction as an argument, since the rules for constructed arguments have the same form.

$$\begin{tikzcd}[column sep=small]
& \overbrace{(\underline{\lambda x_1}.C[\underbrace{f\,\bar{\bullet}\,(\lambda y.M)}])\,\underline{\bullet}\,Q} \arrow[dl, "\underline{\beta}"'] \arrow[dr, "\overline{\delta}"] & \\
\begin{array}{rl} & [Q/x_1]\,C[\overbrace{f\,\bar{\bullet}\,(\lambda y.M)}] \\ \equiv & ([Q/x_1]\,C)[\overbrace{f\,\bar{\bullet}\,(\lambda y.[Q/x_1]M)}] \end{array} \arrow[dr, dotted, "\overline{\delta}"'] & & \underbrace{(\underline{\lambda x_1}.C[N_f \bullet (\lambda y.M)])\,\underline{\bullet}\,Q} \arrow[dl, dotted, "\underline{\beta}"] \\
& \begin{array}{rl} & [Q/x_1]\,C[N_f \bullet (\lambda y.M)] \\ \equiv & ([Q/x_1]\,C)[N_f \bullet (\lambda y.[Q/x_1]M)] \end{array} &
\end{tikzcd}$$

Case (3): Δ_1: *unit*; Δ_2: *any*

The rule *unit* behaves just like β, so this case is proved in the same manner as Case 1.

Case (4): Δ_1: *fuse*; Δ_2: *any*

Case (5): Δ_1: *bubble-assign*; Δ_2: *any*

Case (6): Δ_1: *assign-result*; Δ_2: *any*

Case (7): Δ_1: *bubble-new*; Δ_2: *any*

There are no critical overlaps when any of the just-mentioned rules define the outer redex, so completing the diamond to show the weak Church-Rosser property is trivial in all these cases.

Case (8): Δ_1: *pure-eager* or *pure-lazy*; Δ_2: *any*

The purification rules clearly have no overlaps with the rules β or δ, since the left-hand-sides are based on different syntactic constructors. Also, since the purification rules contain no nested instance of **pure**, there is no overlap between purification rules and themselves. We are thus reduced to considering interactions between command-related rules on the one hand and purification rules on the other.

In order to do this, we must consider the particular form of the store-contexts $S[]$ for each calculus. For the eager calculus, we have

$$\begin{array}{rcl} S^{eag}[] & ::= & [] \\ & | & \nu v . S^{eag}[] \\ & | & v := M;\ S^{eag}[], \end{array}$$

It is clear from this definition that eager store-contexts do not overlap with rule *unit*, since these contexts can only have assignments to the left of a $\triangleright$ operator. Nor can eager store-contexts overlap with the left-hand-sides of the rules *fuse*, *bubble-assign*, *identical*, or *not identical*—by design, all the reduction potential of an eager store-context has been exhausted. Since there are no overlaps involving *pure-eager*, there is no need to consider this rule in detail to establish the weak Church-Rosser property.

For $\lambda[\beta\delta!laz]$ with its purification rules *pure-lazy*, the situation is different. Equally by design, the lazy store-contexts $S^{laz}[]$ can contain redexes that overlap with the context's command sequence. We now consider each such possible overlap.

We simplify our consideration of these cases in two ways. First, we write these cases with only the minimal amount of store-context necessary to show the overlap; it is not hard to see that the constructions hold when the overlaps occur in the midst of longer store-contexts. Second, we consider only the purification rule applying to abstractions: this saves us the slight complication arising from the duplication of the pure-context that is prescribed by the rule *pure-lazy* when the result value is an application of a constructor of more than one argument. This duplication requires that the lower-left leg of the corresponding diamond be a multistep reduction consisting of one independent application of the non-pure rule for each occurrence of the pure-context. The essential reasoning, however, is conveyed by the following simple cases.

(i) Δ_1: *pure-lazy*; Δ_2: *unit*

$$\begin{tikzcd}
 & \underbrace{\underline{\mathbf{pure}}\,\overbrace{(\overline{\uparrow} M_1)\,\overline{\triangleright x}\cdot\underline{\uparrow}\,(\lambda y.M_2)}} \arrow[dl, "\underline{\mathit{pure\text{-}lazy}}"'] \arrow[dr, "\overline{\mathit{unit}}"] & \\
\lambda y.\mathbf{pure}\,\underbrace{(\overline{\uparrow} M_1)\,\overline{\triangleright x}\cdot\uparrow M_2} \arrow[dr, dotted, "\overline{\mathit{unit}}"'] & & \underline{\mathbf{pure}}\,\overbrace{[M_1/x]\,\underline{\uparrow}\,(\lambda y.M_2)} \equiv \underline{\mathbf{pure}}\,\overbrace{\underline{\uparrow}\,(\lambda y.[M_1/x]\,M_2)} \arrow[dl, dotted, "\underline{\mathit{pure\text{-}lazy}}"] \\
 & \lambda y.\mathbf{pure}\uparrow[M_1/x]\,M_2 &
\end{tikzcd}$$

(ii) Δ_1: *pure-lazy*; Δ_2: *fuse*

$$\begin{tikzcd}
& \underbrace{\underline{\mathbf{pure}}\,\overbrace{v\,\overline{:\equiv}\,N;\ v\overline{?}\,\overline{\rhd x}\cdot\underline{\uparrow}\,(\lambda y.M)}} \arrow[dl, "\underline{\textit{pure-lazy}}"'] \arrow[dr, "\overline{\textit{fuse}}"] & \\
\lambda y.\mathbf{pure}\,\overbrace{v\,\overline{:\equiv}\,N;\ v\overline{?}\,\overline{\rhd x}\cdot\uparrow M} \arrow[dr, dotted, "\overline{\textit{fuse}}"'] & & \begin{array}{l} \overbrace{\underline{\mathbf{pure}}\,v:=N;\ [N/x]\,(\underline{\uparrow}\,(\lambda y.M))} \\ \equiv\ \overbrace{\underline{\mathbf{pure}}\,v:=N;\ \underline{\uparrow}(\lambda y.[N/x]\,M)} \end{array} \arrow[dl, dotted, "\underline{\textit{pure-lazy}}"] \\
& \lambda y.\mathbf{pure}\,v:=N;\ \uparrow([N/x]\,M) &
\end{tikzcd}$$

(iii) Δ_1: *pure-lazy*; Δ_2: *bubble-assign*

$$\begin{tikzcd}
& \underbrace{\underline{\mathbf{pure}}\,\overbrace{v\,\overline{:\equiv}\,N;\ w\overline{?}\,\overline{\rhd x}\cdot\underline{\uparrow}\,(\lambda y.M)}} \arrow[dl, "\underline{\textit{pure-lazy}}"'] \arrow[dr, "\overline{\textit{bubble-assign}}"] & \\
\lambda y.\mathbf{pure}\,\underbrace{v\,\overline{:\equiv}\,N;\ w\overline{?}\,\overline{\rhd x}\cdot\underline{\uparrow}\,M} \arrow[dr, dotted, "\overline{\textit{bubble-assign}}"'] & & \underline{\mathbf{pure}}\,\overbrace{w?\rhd x\cdot v:=N;\ \underline{\uparrow}\,(\lambda y.M)} \arrow[dl, dotted, "\underline{\textit{pure-lazy}}"] \\
& \lambda y.\mathbf{pure}\,w?\rhd x\cdot v:=N;\ \uparrow M &
\end{tikzcd}$$

(iv) Δ_1: *pure-lazy*; Δ_2: *assign-result*

$$\begin{tikzcd}
& \underbrace{\underline{\mathbf{pure}}\,\overbrace{v\,\overline{:\equiv}\,N\,\overline{\rhd x}\cdot\underline{\uparrow}\,(\lambda y.M)}} \arrow[dl, "\underline{\textit{pure-lazy}}"'] \arrow[dr, "\overline{\textit{assign-result}}"] & \\
\lambda y.\mathbf{pure}\,\underbrace{v\,\overline{:\equiv}\,N\,\overline{\rhd x}\cdot\uparrow M} \arrow[dr, dotted, "\overline{\textit{assign-result}}"'] & & \begin{array}{l} \overbrace{\underline{\mathbf{pure}}\,v:=N;\ [()/x]\,(\underline{\uparrow}\,(\lambda y.M))} \\ \equiv\ \overbrace{\underline{\mathbf{pure}}\,v:=N;\ \underline{\uparrow}(\lambda y.[()/x]\,M)} \end{array} \arrow[dl, dotted, "\underline{\textit{pure-lazy}}"] \\
& \lambda y.\mathbf{pure}\,v:=N;\ \uparrow[()/x]\,M &
\end{tikzcd}$$

(v) Δ_1: *pure-lazy*; Δ_2: *bubble-new*

$$\begin{tikzcd}
 & \underbrace{\underline{\mathbf{pure}}\overbrace{\overline{\nu v}.w\overline{?}\,\overline{\triangleright x}\cdot\underline{\uparrow}\,(\lambda y.M)}^{}}_{} \arrow[dl, "\underline{\textit{pure-lazy}}"'] \arrow[dr, "\overline{\textit{bubble-new}}"] & \\
\lambda y.\mathbf{pure}\,\overbrace{\overline{\nu v}.w\overline{?}\,\overline{\triangleright x}\cdot\uparrow M}^{} \arrow[dr, dotted, "\overline{\textit{bubble-new}}"'] & & \underbrace{\underline{\mathbf{pure}}\,w?\triangleright x\cdot\nu v.\underline{\uparrow}\,(\lambda y.M)}_{} \arrow[dl, dotted, "\underline{\textit{pure-lazy}}"] \\
 & \lambda y.\mathbf{pure}\,w?\triangleright x\cdot\nu v.\uparrow M &
\end{tikzcd}$$

This concludes our consideration of the overlaps involving purification rules.

We have now shown that the diamond can be completed for all overlapping occurrences of rules for $\rightarrow_!$ in both $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, and hence that the weak Church-Rosser property holds for both calculi.
∎

Now that we have used marked reductions in a proof we can complete our remarks on the necessity of factoring the reduction relation. As we noted after the proof of Lemma 3.3.1, the diagrams involved in proving that lemma required some unmarked reductions. In the proof we have just completed for Lemma 3.5.5, however, all the diagrams involve only marked reductions, as they must in order to establish a theorem about marked reductions. Had we not factored the reduction relation, the diagrams from Lemma 3.3.1 would have formed a part of a (fallacious) proof of Lemma 3.5.5. This state of affairs was allowed to stand in [Odersky and Rabin, 1993]; we offer the complexities of the present chapter as a correction. Of course, the error is only one of proof, not result (as we now show), but the structural insight provided by factoring the reduction was felt to be interesting enough to justify pursuing this course.[2]

3.5.2 Finiteness of developments

With the weak Church-Rosser property in hand for the marked calculi modulo association, we now turn to proving the termination of developments modulo association.

It is convenient for the purposes of this section to note that the reduction rules of our calculi fall into several well-defined groups according to the kind of rewriting they specify:

- The rules β, *fuse*, *unit*, and *assign-result* substitutive rules, since they all replace occurrences of a bound variable by a given expression to model value-passing in a programming language. These rules may cause the duplication or delection of the term representing the passed value.
- The rules *bubble-assign* and *bubble-new*, act alike to axiomatize the non-interference of operations involving different locations. These rules re-order subterms, but do not duplicate or delete them.
- All the *purification* rules form a category on their own. These rules can duplicate or delete the subterms forming the store context.

To show that every development (modulo association) is finite, we adapt the weighting technique of [Barendregt, 1984], Section 11.4. The technique as used in the cited source only deals with the β-rule, but we use it without change for all the substitutive rules. This is the only hard part, since the possibility of duplication

[2]For an example of work in which an alternative tack was chosen in the face of the identical problem with a modified lambda-calculus, see [Ariola *et al.*, 1995].

$$\begin{aligned}
\mathcal{W}[\![x^n]\!] &= n\\
\mathcal{W}[\![c^n]\!] &= 1\\
\mathcal{W}[\![f^n]\!] &= n\\
\mathcal{W}[\![\lambda x.M]\!] &= \mathcal{W}[\![M]\!]\\[1ex]
\mathcal{W}[\![M_1 \bullet M_2]\!] &= \mathcal{W}[\![M_1]\!] + \mathcal{W}[\![M_2]\!]\\[1ex]
\mathcal{W}[\![\underline{\nu v}.w? \underline{\triangleright x} \cdot M]\!] &= 1 + \mathcal{W}[\![w? \triangleright x \cdot \nu v.M]\!]\\
\mathcal{W}[\![v \underline{==} w]\!] &= 1 + \mathcal{W}[\![\lambda x.\lambda y.y]\!]\\
\mathcal{W}[\![v \underline{==} v]\!] &= 1 + \mathcal{W}[\![\lambda x.\lambda y.x]\!]\\
\mathcal{W}[\![v \underline{:=} M_1 \underline{;}\ w\underline{?}\underline{\triangleright x_2} \cdot M_2]\!] &= 1 + \mathcal{W}[\![w? \triangleright x_2 \cdot v := M_1;\ M_2]\!]\\[1ex]
\mathcal{W}[\![\underline{\textbf{pure}}\, S^{eag}[\underline{\uparrow}\,(c^n \underline{\bullet} M_1 \underline{\bullet} \cdots \underline{\bullet} M_k)]]\!] &= 1 + \mathcal{W}[\![c^n \bullet (\textbf{pure}\, S^{eag}[\uparrow M_1]) \bullet \cdots \bullet (\textbf{pure}\, S^{eag}[\uparrow M_n])]\!]\\
\mathcal{W}[\![\underline{\textbf{pure}}\, S^{laz}[\underline{\uparrow}\,(c^n \underline{\bullet} M_1 \underline{\bullet} \cdots \underline{\bullet} M_k)]]\!] &= 1 + \mathcal{W}[\![c^n \bullet (\textbf{pure}\, S^{laz}[\uparrow M_1]) \bullet \cdots \bullet (\textbf{pure}\, S^{laz}[\uparrow M_n])]\!]\\
& \quad \text{(and so forth for the other } \textbf{pure} \text{ rules)}
\end{aligned}$$

$\mathcal{W}[\![\,]\!]$ is additive on the subterms of all other terms.

Figure 3.1: Definition of the weight function.

of subterms in these reductions leads to some concern that the duplication of marked redex subterms might outrun their elimination through marked reductions. That this cannot actually happen is not obvious and is the point of the present proof.

For all the other rules, we merely count the number of marked redexes in the start term of the development in the weight of a term.

The weight function $\mathcal{W}[\![\,]\!]$ on marked terms is defined in Figure 3.1. The function $\mathcal{W}[\![\,]\!]$ is actually defined on a slightly modified calculus in which variables and primitive function names bear weights: the notion of erasing weights to recover the original calculus is straightforward from an informal description. Note that $\mathcal{W}[\![M]\!]$ is always at least 1. The definition of $\mathcal{W}[\![\,]\!]$ is designed to decrease on each reduction of a marked redex. For substitutive redexes, this means that it should be possible to arrange to have the weighted occurrence of the substitution variable have a greater weight than the substituted expression; this is the reason for having weighted variables. For δ-redexes, we want to be able to weight the primitive function name more heavily than anything that might replace it. For the remaining kinds of redex, we just define the weight to be 1 greater than the reduced form.[3]

We will be concerned only with terms whose weightings have a special property contrived to make reductions involving substitution and primitive function definitions decrease the weight of a term. The following definition requires that substitution reductions must reduce the weight of a term. There are five cases because the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ have five kinds of substitutive redex.

Definition 3.5.6 *A term M has a* decreasing weighting *if the following conditions on marked-redex subterms all hold:*

1. *for all marked β-redex subterms $(\lambda x.M_1) \bullet M_2$, for all x^n in M_1, we have $n > \mathcal{W}[\![M_2]\!]$,*
2. *for all marked δ-redex subterms $f^n \bullet V$ for which $\delta(f,V)$ is defined, we have $n + \mathcal{W}[\![V]\!] > \mathcal{W}[\![\delta(f,V)]\!]$,*
3. *for all marked unit-redex subterms $\uparrow M_1 \triangleright x_2 \cdot M_2$, we have, for all $x_2{}^n$ in M_2, $n > \mathcal{W}[\![M_1]\!]$,*
4. *for all marked fuse-redex subterms $v := M_1;\ v? \triangleright x_2 \cdot M_2$, for every free occurrence of $x_2{}^n$ in M_2, we have $n > \mathcal{W}[\![M_1]\!]$*

[3] In [Odersky and Rabin, 1993] the authors expend a great deal of effort in designing a weight function that decreases under even statically unknown *pure-eager*-reductions. This is unnecessary—all the relevant redexes must be known in advance in order to be marked.

5. *for all marked assign-result-redex subterms* $(v := M_1 \triangleright x_2 \cdot M_2)_*$, $x_2 \in fv\, M_2$, *for each* ${x_2}^n$ *in* M_2, *we have* $n > \mathcal{W}[\![()]\!]$.

Since the calculi for which we want to prove finiteness of developments are actually defined on equivalence classes of terms, we must establish that the definitions of weighting in Figure 3.1 and of decreasing weighting in Definition 3.5.6 are independent of the chosen representative of the $\overset{\triangleright}{=}$-equivalence class. To do this, we merely insist that the definitions apply to a particular representative, the $\rightarrow_{\triangleright}$-normal form. Since the definitions are largely concerned with marked redexes, and Corollary 3.4.2 shows that all such redexes are present in the $\rightarrow_{\triangleright}$-normal form of a term, the definition is justified.

With the definitions of the weight function and of decreasing weighting in hand, the proof that every development terminates now splits into three parts: decreasing weightings exist, weights decrease under reduction, and reductions preserve decreasing weightings. The first two parts (Lemmas 3.5.7 and 3.5.8) are nearly trivial; the third (Lemma 3.5.9) takes some work.

Lemma 3.5.7 *Every marked term can be given a decreasing weighting.*

Proof: By working from the fringe of the syntax tree toward the root, it is clearly possible to assign to every bound-variable occurrence and to every primitive-function occurrence in a redex a weight sufficiently high to meet the conditions in 3.5.6: the term to be substituted is visible and weighted before each variable is weighted, and we just assign the variable occurrences higher weights than the term to be substituted. ∎

Lemma 3.5.8 *If a term* M_1 *has a decreasing weighting, and* $M_1 \rightarrow M_2$, *then* $\mathcal{W}[\![M_1]\!] > \mathcal{W}[\![M_2]\!]$.

Proof: This holds by the construction of the function $\mathcal{W}[\![\,]\!]$ and by the definition of decreasing weighting. We have defined $\mathcal{W}[\![\,]\!]$ such that the substitutive reductions decrease the total weight owing to the definition of decreasing weighting, and such that the non-substitutive reductions decrease the weight by 1. ∎

Lemma 3.5.9 *If a term* M_1 *has a decreasing weighting, and* $M_1 \rightarrow M_2$, *then the weighting of* M_2 *is decreasing.*

Proof: Since the substitutive rules are all similar, we can capture the argument by the general reasoning suggested by the proof on pp. 288–290 of [Barendregt, 1984]. Since the other reduction rules do not alter substitutive redexes, the decreasing property of those redexes is left intact. ∎

Taken together, Lemmas 3.5.8 and 3.5.9 imply that developments are finite, and hence we have the following result.

Theorem 3.5.10 *For each of the calculi* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$, *all developments modulo association terminate.*

3.5.3 Strong finiteness of developments

We now wrap up this section by stating and proving its main result, of which Theorem 3.5.10 forms a part.

Theorem 3.5.11 *Computational reduction on* $\overset{\triangleright}{=}$*-equivalence classes in* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *is Church-Rosser.*

Proof: By Lemma 3.5.5, the reduction relations in question are weakly Church-Rosser. By Theorem 3.5.10, the same relations are strongly normalizing. Hence by Newman's Lemma (Proposition 3.2.5) they are Church-Rosser. ∎

The results we require from all this work can now be stated:

Theorem 3.5.12 *Let* M *be a term, and let* $\mathcal{F} \subseteq M$ *be a set of redexes that are subterms of* M. *Then*

(i) All developments of $\mathcal{F} \subseteq M$ *modulo association are finite.*

(ii) Any developments of $\mathcal{F} \subseteq M$ *can be extended to a complete development modulo association.*

$$E[] \;::=\; [] \mid E[] \bullet M \mid f \bullet E[]$$

Figure 3.2: Evaluation contexts for λ[βδ].

$$\begin{array}{lcl} E[] & ::= & \cdots(\text{Figure 3.2})\cdots \\ & \mid & E[] \triangleright x \cdot M \mid M \triangleright x \cdot E[] \mid E[] := M \mid E[]? \\ & \mid & \nu v . E[] \mid E[] == M \mid v == E[] \mid \mathbf{pure}\, E[] \end{array}$$

Figure 3.3: Evaluation contexts for both λ[βδ!*eag*] and λ[βδ!*laz*]

$$\begin{array}{lcl} E[] & ::= & \cdots \;(\text{Figure 3.3})\; \cdots \\ & \mid & \mathbf{pure}\, S^{eag}[\uparrow E[]] \end{array}$$

Figure 3.4: Evaluation contexts for λ[βδ!*eag*]

(iii) All complete developments of $\mathcal{F} \subseteq M$ modulo association end in the same term.

Proof: (i) This is Theorem 3.5.10.

(ii) By the same reasoning as in [Barendregt, 1984], Corollary 11.2.22.

(iii) The marked reduction modulo association has been shown to be Church-Rosser.
∎

3.6 The Church-Rosser property

The results of previous sections allow us to use the method of Tait and Martin-Löf, as presented in [Barendregt, 1984], to establish the Church-Rosser property for reduction modulo association.

Theorem 3.6.1 (Church-Rosser) *The calculi* λ[βδ!*eag*] *and* λ[βδ!*laz*] *have the Church-Rosser property under the reduction relations* $\twoheadrightarrow_!$.

Proof: We sketch the proof from [Barendregt, 1984]. A special reduction relation $\vec{\scriptstyle 1}$ is defined by $M \vec{\scriptstyle 1} N$ if the complete development of M by some set of redexes is N. It is shown that the usual reduction ($\twoheadrightarrow_!$ in our case) has the same transitive closure as $\vec{\scriptstyle 1}$, and that $\vec{\scriptstyle 1}$ (and therefore its transitive closure) has the diamond property; FD! plays its role in this part of the proof. But the transitive closure of $\vec{\scriptstyle 1}$ is the same as the transitive closure of $\twoheadrightarrow_!$, so the transitive closure of $\twoheadrightarrow_!$ has the diamond property, which is the definition of the Church-Rosser property. ∎

3.7 Standard evaluation order

We now turn to establishing that λ[βδ!*eag*] and λ[βδ!*laz*] have standard evaluation orders.

Standard evaluation orders are defined by partitioning the set of redex-subterms into two kinds, *head* and *interior* redexes, according to their position in a term. There is at most one head redex; all other redexes are interior redexes. Internal reductions neither create nor destroy nor duplicate head redexes, and residuals of internal redexes by internal reductions are still internal. If this classification of redexes can be carried out, then a standard reduction order can be established as the reduction of head redexes first, then internal redexes.

We will first define the appropriate concepts on terms, and then adapt the concepts to the world of $\stackrel{\triangleright}{=}$-equivalence classes in terms of which our results must actually be proved.

Using a technique from [Felleisen and Friedman, 1986], we specify standard evaluation orders by defining a syntactic category of *evaluation contexts* $E[]$ that serves to identify the location within a term of the redex subterm to be reduced next in the standard order. In order for the definition to define a deterministic evaluation

$$\begin{array}{rcl} E[\,] & ::= & \cdots \text{ (Figure 3.3) } \cdots \\ & | & \mathbf{pure}\, S^{laz}[\uparrow E[\,]] \end{array}$$

Figure 3.5: Evaluation contexts for $\lambda[\beta\delta!laz]$

$$\begin{array}{lclcl}
M & \equiv & M_1 \bullet M_2 & \Rightarrow & M \prec M_1 \prec M_2 \\
M & \equiv & \lambda x.M_1 & \Rightarrow & M \prec M_1 \\
M & \equiv & \uparrow M_1 & \Rightarrow & M \prec M_1 \\
M & \equiv & M_1 \rhd x \cdot M_2 & \Rightarrow & M \prec M_1 \prec M_2 \\
M & \equiv & \nu v.M_1 & \Rightarrow & M \prec M_1 \\
M & \equiv & M_1 ? & \Rightarrow & M \prec M_1 \\
M & \equiv & M_1 := M_2 & \Rightarrow & M \prec M_1 \prec M_2 \\
M & \equiv & \mathbf{pure} M_1 & \Rightarrow & M \prec M_1 \\
M & \equiv & \mathbf{pure} S^{laz}[\uparrow M_1] & \Rightarrow & M \prec M_1 \prec S^{laz}[\,]
\end{array}$$

Figure 3.6: Subterm ordering for defining the leftmost redex

order, the evaluation contexts for a calculus must have the property that every term has a unique leftmost redex in an evaluation context.

Figures 3.2, 3.3, 3.4, and 3.5 give the definition of evaluation contexts for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. As might be expected, the only difference between the definition for $\lambda[\beta\delta!eag]$ in Figure 3.4 and that for $\lambda[\beta\delta!laz]$ in Figure 3.5 is in the form of the store-context for the evaluation contexts with **pure** at the root.

The informal intent of these definitions for evaluation contexts is that store-computation proceeds from left to right; terms that must be store-variables in order to be useful are forced, but the reduction of others is deferred. The special forms of evaluation context involving explicit store-contexts are necessary because the composition of the other rules will never produce an evaluation context that descends inside a $\uparrow$-expression, hence evaluation specified without the special rules would never force the result of a store-computation and would thus fail to be a standard order.

The definition in 3.3 requires the proviso that the pattern $E[\,] \rhd x \cdot M$ is given priority over $M \rhd x \cdot E[\,]$ in determining the head redex. This ordering is formalized as the notion of *leftmost* evaluation redex. Figure 3.6 defines an ordering $\prec$ (read "is to the left of") of the subterms within a given term. Terms precede their subterms in this ordering, the main point being to pin down the relative ordering between subterms for those syntactic constructs having more than one. The rule for $\mathbf{pure} S^{laz}[\uparrow M_1]$ in Figure 3.6 takes precedence over that for $\mathbf{pure} M_1$: it is necessary force possible results before allowing store-computation to resume.

We use the ordering $\prec$ to define:

Definition 3.7.1 (Leftmost redex) *The* leftmost *redex subterm of a term M is the least redex subterm of M according to the ordering $\prec$.*

The preorder traversal specified in Definition 3.7.1 subsumes the usual definition of 'leftmost-outermost".

Definition 3.7.2 (Evaluation, head, internal redexes) *(i) A redex Δ is an* evaluation *redex of a term M if $M \equiv E[\Delta]$ for some evaluation context $E[\,]$.*

(ii) The head *redex Δ_h of a term M is the leftmost evaluation redex of M.*

(iii) Any redex subterm of a term M other than a head redex is called an internal *redex.*

We denote head and internal reductions by affixing the letters h and i respectively to arrows denoting reductions.

Lemma 3.7.3 *If a term M contains an evaluation redex, it contains a head redex.*

Proof: Definition 3.7.2 (i) and (ii) imply this directly. ∎

That fact that the definitions of head and internal redex are context-dependent requires some extra care in constructing proofs involving these concepts. Whether a redex is a head redex depends on the entire term in which it appears, not just on its immediate context. For example, the term $(\lambda x.M) \bullet N$ is head redex in itself, but is internal in the context of the larger term $(\lambda y.M') \bullet ((\lambda x.M) \bullet N)$.

We have now defined what it means for a term to have a head redex; we must now adapt our definition to $\overset{\triangleright}{=}$-equivalence classes. To motivate this definition, we make three basic observations about the interaction of $\to_{\triangleright}$-reduction with $\to_{!}$-reduction. First, $\to_{\triangleright}$-reduction does not permute the order of the subterms designated by meta-variables in the reduction rules, so the definition of "leftmost" among those subterms remains constant under $\to_{\triangleright}$-reduction. Second, any redex overlapping with a sequence of commands subject to $\to_{\triangleright}$-reduction remains present in the $\to_{\triangleright}$-normal form of the term (by Corollary 3.4.2), and occupies the same left-to-right position. Third, a redex in one element of a $\overset{\triangleright}{=}$-equivalence class may not correspond to a redex in all terms in the class (it may not even correspond to a single subterm in all term in the class); however, by Corollary 3.4.2 $\to_{\triangleright}$-normal forms contain all redexes that exist in any member of their class.

These considerations lead us to the following definition:

Definition 3.7.4 (Head redex for $\overset{\triangleright}{=}$-equivalence classes) *Given a term M, the leftmost evaluation redex of $\downarrow_{\triangleright}[M]$ is the* head redex *of $[M]_{\overset{\triangleright}{=}}$ (if such a redex exists).*

The main thrust of these definitions given so far in this section is to give a framework in which to prove the following lemma, which guarantees that any reduction relation deduced via any reduction sequence whatsoever can also be justified by a reduction sequence consisting of head reductions only followed by internal reductions only:

Lemma 3.7.5 (Global Interchange Lemma) *If $M \to^{*} N$, then there exists a term M' and head and internal reductions such that*

$$\begin{tikzcd}
M \arrow[d, dotted, two heads, "h"'] \arrow[dr] & \\
M' \arrow[r, dotted, two heads, "i"'] & N
\end{tikzcd}$$

The rest of this section is devoted to the work of proving this lemma, from which the standardization theorem follows straightforwardly. Following Barendregt's Section 11.4 as our guide, we approach the proof of Lemma 3.7.5 by way of auxiliary results that characterize the way the distinction between head and internal redexes behaves under internal reductions. The following lemmas state that internal reductions do not create head redexes (Lemma 3.7.6), that they preserve an existing head redex (Lemma 3.7.7), and that they preserve internal redexes (Lemma 3.7.8). These facts, taken together, are enough to establish Lemma 3.7.5.

Although Barendregt takes only a few lines to establish these lemmas for the pure lambda-calculus, our proofs will occupy several pages. The added complexity derives in part from the larger syntax and rule sets of our calculi and in part from our need to consider $\overset{\triangleright}{=}$-equivalence classes of terms instead of terms pure and simple.

Lemma 3.7.6 (Heads don't sprout) *If $M \to_{\mathrm{i}} N$ and $[N]_{\overset{\triangleright}{=}}$ has a head redex Δ_{h}, then $[M]_{\overset{\triangleright}{=}}$ has a head redex.*

Proof: Assume that $M \xrightarrow{\Delta}_{\mathrm{i}} N$, and that L is the result of reducing Δ. By Proposition 3.4.1 and Corollary 3.4.2, the normal form $\downarrow_{\triangleright}[M]$ contains a $\to_{!}$-redex Δ' that is a residual of Δ such that the following diagram holds for some term N':

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta"'] \arrow[dr, two heads, "\triangleright"] & \\
N \arrow[dr, two heads, "\triangleright"'] & & \downarrow_{\triangleright}[M] \arrow[dl, "\Delta'"] \\
 & N' \arrow[dr, two heads, "\triangleright"'] & \\
 & & \downarrow_{\triangleright}[N]
\end{tikzcd}$$

We will refer to the subterm of N' that is the result of reducing Δ' as L'. The diagram permits us to consider Δ' and L' instead of Δ and L, since these two terms reflect essentially the same reduction relating the two $\twoheadrightarrow_{\triangleright}$-equivalence classes.

By Definition 3.7.4, Δ_{h} is in head position within $\downarrow_{\triangleright}[N]$. The defining reduction rules of $\twoheadrightarrow_{\triangleright}$, *assoc* and *extend*, neither create nor destroy syntactic structure but only rearrange it. It is thus reasonable to trace the pieces that make up Δ_{h} backward to find the collection of subterms of N' that are rearranged into Δ_{h} by the $\twoheadrightarrow_{\triangleright}$-normalization process. Depending on the form of the redex Δ_{h}, this collection (we will denote it by Δ'_{h} even though it is not necessarily a single term) may take several forms. If Δ_{h} is a *fuse-*, *bubble-assign-*, *assign-result-*, *unit-*, or *bubble-new*-redex, the top-level syntactic constructor of Δ_{h} is $\triangleright$: this case allows Δ'_{h} to consist of separate subterms. If Δ_{h} is a β- or δ-redex Δ'_{h} will consist of a single subterm. If Δ_{h} is a *pure-eager-* or *pure-lazy-*redex then Δ'_{h} will likewise consist of a single subterm.

We decompose the proof into three cases depending on the relative positions of L' and Δ'_{h}. In the first case, L' and Δ'_{h} are disjoint. In this case, the positions of the subterms will be enough to establish that $\downarrow_{\triangleright}[M]$ has an evaluation redex and hence a head redex. In the second case, all the fragments of Δ'_{h} lie within L'. In this case also we will be able to reason from the position of the subterms rather than their composition. In the third case, however, in which L' overlaps Δ'_{h}, we will have to conduct a case analysis on the form of the redex Δ_{h}.

Before we begin the positional analysis, it is important to elaborate on the possible configurations of Δ'_{h}. Consider the maximal command sequence immediately containing Δ_{h}, that is, the largest chain of ν- and $\triangleright$-constructs such that Δ_{h} either overlaps part of the chain or is an element in the chain. Since Δ_{h} is a head redex, there are no evaluation redexes to the left of Δ_{h} in this maximal chain, a fact that is reflected in the collection of subterms Δ'_{h} because $\twoheadrightarrow_{\triangleright}$-reduction preserves the order of subterms. In more formal terms, $\downarrow_{\triangleright}[N]$ must have the form $E_1[S^{laz}[E_2[\Delta_{\mathrm{h}}]]]$, where there is no evaluation redex in $S^{laz}[]$, and the (lazy) store-context $S^{laz}[]$ is the longest such sequence immediately surrounding Δ_{h}.

Case 1: L' and Δ'_{h} are disjoint. In this case, no part of the result of reducing Δ is assembled into Δ_{h}, yet Δ_{h} is head redex in $\downarrow_{\triangleright}[N]$. We argue that the antecedent of Δ_{h} must then exist and be the head redex in $\downarrow_{\triangleright}[M]$ (and this must be, by Definition 3.7.4, the head redex of $[M]_{\underline{\triangleright}}$).

The existence of the antecedent of Δ'_{h} as a redex within $\downarrow_{\triangleright}[M]$ follows from two observations. First, since L' is disjoint from Δ'_{h}, Δ'_{h} is an unaltered copy of a subterm of $\downarrow_{\triangleright}[M]$ disjoint from Δ'. Second, since $\downarrow_{\triangleright}[M]$ is in $\twoheadrightarrow_{\triangleright}$-normal form, so are all its subterms, including Δ'_{h}. Hence Δ'_{h} is really the same as the redex Δ_{h}—it is not possible in this case for Δ'_{h} to be a set of fragments to be assembled into Δ_{h} by $\twoheadrightarrow_{\triangleright}$.

We next note that L' cannot be in an evaluation context to the left of Δ'_{h}, because this would make Δ' a head redex, whereas it is assumed in the statement of the lemma to be internal. Hence the context in which Δ' occurs must be either a non-evaluation context or an evaluation context to the right of the antecedent of Δ'_{h}. If the latter case applies, the mere existence of an evaluation redex establishes the existence of a head redex (Lemma 3.7.3). We are thus left to consider the cases in which Δ' occupies a non-evaluation context.

Since Δ' must be disjoint from the antecedent of Δ'_{h} (which is Δ_{h}, as we showed just above), there must be some multiple-subterm syntax constructor that forms the least common ancestor of the two terms. Furthermore, since Δ_{h} lies within an evaluation context in the whole term (since the only change occurs within Δ), this common ancestor must itself be in an evaluation context and Δ_{h} must be in a nested evaluation context starting at the common ancestor.

The following list enumerates all the ways in which this situation can arise.

(1) $E_1[E_2[\Delta_{\mathrm{h}}] \bullet C[\Delta']]$,

(2) $E_1[E_2[\Delta_{\mathrm{h}}] := C[\Delta']]$,

(3) $E_1[E_2[\Delta_{\mathrm{h}}] \rhd x \cdot C[\Delta']]$,

(4) $E_1[C[\Delta'] \rhd x \cdot E_2[\Delta_{\mathrm{h}}]]$, where $C[\,]$ is not an evaluation context,

(5) $E_1[\mathbf{pure}\, S^{eag}[\uparrow E_2[\Delta_{\mathrm{h}}]]]$, where $\Delta' \subseteq S^{eag}[\,]$. (This case applies to $\lambda[\beta\delta!eag]$),

(6) $E_1[\mathbf{pure}\, S^{laz}[\uparrow E_2[\Delta_{\mathrm{h}}]]]$, where $\Delta' \subseteq S^{laz}[\,]$. (This case applies to $\lambda[\beta\delta!laz]$).

In each of these cases, Δ_{h} is an evaluation redex of $\downarrow_{\rhd}[M]$; hence $\downarrow_{\rhd}[M]$ must have a head redex.

Case 2: $\Delta'_{\mathrm{h}} \subseteq L'$.

Since Δ'_{h} may not be a single subterm, the assertion characterizing this case should be understood to mean that all the components of Δ'_{h} are subterms of L', which in turn implies that all the components of Δ'_{h} arise from the reduction of Δ'. More formally, if Δ' appears in a general term context $C_1[\,]$, i.e. $\downarrow_{\rhd}[M] \equiv C_1[\Delta']$, then $N' \equiv C_1[L']$ and $N' \twoheadrightarrow_{\rhd} C_1[C_2[\Delta_{\mathrm{h}}]]$ for some general term context $C_2[\,]$. Since Δ_{h} is known to be a head redex, the enclosing context $C_1[C_2[\,]]$ must be an evaluation context. Since evaluation contexts are inductively defined in such a way that any prefix of an evaluation context is also an evaluation context, the context $C_1[\,]$ must be an evaluation context. We have now shown M to have at least one evaluation redex, so by Lemma 3.7.3 it must have a head redex.

Case 3: $L' \subset \Delta'_{\mathrm{h}}$.

In case Δ'_{h} consists of more than one term, we interpret $L' \subset \Delta'_{\mathrm{h}}$ to mean that there is some term Q in the set composing Δ'_{h} for which $L' \subset Q$. For an example of this possibility, suppose that N' has the form $((\uparrow N'_1) \rhd x \cdot N'_2) \rhd y \cdot N'_3$, where L' is the subterm $(\uparrow N'_1) \rhd x \cdot N'_2$. The $\twoheadrightarrow_{\rhd}$-normal form $\downarrow_{\rhd}[N']$ then has the form $(\uparrow N'_1) \rhd x \cdot \downarrow_{\rhd}[N'_2 \rhd y \cdot N'_2]$, where Δ_{h} consists of the entire term $\downarrow_{\rhd}[N']$. The subterm L' is not a subterm of this normal form. In the subcases that follow, we must account for this possibility that L' may be fragmented into more than one different subterm of Δ_{h}.

In this case the reduction must have the general form $M' \equiv E[\widehat{M}] \xrightarrow{\Delta'} E[\Delta'_{\mathrm{h}}]$ for some evaluation context $E[\,]$ and subterm $\widehat{M}$ with $\Delta' \subseteq \widehat{M}$. We now carry out a case analysis according to the form of the redex $\Delta_{\mathrm{h}} \subseteq \downarrow_{\rhd}[N']$, and show that for each form, the antecedent $\widehat{M} \subseteq M'$ of Δ'_{h} has an evaluation redex (and hence, by Lemma 3.7.3, a head redex).

For each case, we list the possible forms of $\widehat{M}$ and the corresponding evaluation redex whose existence we must establish. The notation $C_{+}[\Delta]$ refers to the redex Δ in a non-empty context.

(a) β: $\Delta_{\mathrm{h}} \equiv (\lambda x.N_1) \bullet N_2$.

Since $\twoheadrightarrow_{\rhd}$ cannot shuffle terms whose syntactic constructor is $\bullet$, the antecedent Δ'_{h} of Δ_{h} within N' has the same form, with some $\twoheadrightarrow_{\rhd}$-conversion possible within the subterms. So if we let $\Delta'_{\mathrm{h}} \equiv (\lambda x.N'_1) \bullet N_2{}'$, and if we consider all the ways that this subterm can be produced from an antecedent $\widehat{M}$, we obtain the following table which displays the evaluation redex whose existence is implied:

$\widehat{M}$:	eval. redex:
$\Delta' \bullet N'_2$	Δ'
$(\lambda x.C[\Delta']) \bullet N'_2$	$\widehat{M}$
$(\lambda x.N'_1) \bullet C[\Delta']$	$\widehat{M}$

(b) δ: $\Delta_{\mathrm{h}} \equiv f \bullet V$

This case is subject to the same reasoning concerning the form of Δ'_{h} as the case of β and gives us the following table for establishing the existence of an evaluation redex in $\widehat{M}$:

$\widehat{M}$:	eval. redex:
$\Delta' \bullet V'$	Δ'
$f \bullet \Delta'$	Δ'
$f \bullet (C[\Delta'] \bullet \cdots \bullet \cdots \bullet \cdots)$	Δ'
$f \bullet (c^n \bullet \cdots \bullet C[\Delta'] \bullet \cdots)$	$\widehat{M}$
$f \bullet (\lambda x.C[\Delta'])$	$\widehat{M}$

(c) *unit*: $\Delta_{\mathrm{h}} \equiv \uparrow N_1 \triangleright x \cdot N_2$

For this and other forms of redex whose root syntactic constructor is $\triangleright$ we need to consider carefully the effect of the $\rightarrow_{\triangleright}$-reduction leading from Δ'_{h} to Δ_{h}. If we suppose that Δ_{h} is the result of a *assoc*- or *extend*-reduction involving the occurrence of $\triangleright$ that forms the root of Δ_{h}, we see that the (possibly multiple) terms making up Δ'_{h} must include exactly one of the form $\uparrow N'_1 \triangleright x \cdot N'_2$, where N'_2 might be a shorter command sequence (in terms of number of occurrences of $\triangleright$ on its spine) than N_2. The other terms in Δ'_{h} are those tacked onto N'_2 by *assoc*- or *extend*-reductions. We call the former kind *essential* and the latter *trivial*.

If L' is a subterm of the essential element of Δ'_{h}, we can deduce the following table giving the required evaluation redexes:

$\widehat{M}$:	eval. redex:
$\Delta' \triangleright x \cdot N'_2$	Δ'
$\uparrow C[\Delta'] \triangleright x \cdot N'_2$	$\widehat{M}$
$\uparrow N'_1 \triangleright x \cdot C[\Delta']$	$\widehat{M}$

If, on the other hand, L' is a subterm of a trivial element of Δ'_{h}, then the entire antecedent $\widehat{M}$ is an evaluation redex.

(d) *fuse*: $\Delta_{\mathrm{h}} \equiv v := N_1\,;\, v? \triangleright x \cdot N_2$

The considerations and terminology given under the case for *unit* apply here as well. If L' is a subterm of the essential element of Δ'_{h}, then we construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\Delta';\, v? \triangleright x \cdot N'_2$	Δ'
$v := C[\Delta'];\, v? \triangleright x \cdot N'_2$	$\widehat{M}$
$\Delta' := N'_1\,;\, v? \triangleright x \cdot N'_2$	Δ'
$v := N'_1\,;\, \Delta'$	Δ'
$v := N'_1\,;\, \Delta' \triangleright x \cdot N'_2$	Δ'
$v := N'_1\,;\, \Delta'? \triangleright x \cdot N'_2$	Δ'
$v := N'_1\,;\, v? \triangleright x \cdot C[\Delta']$	$\widehat{M}$

If, on the other hand, L' is a subterm of a trivial element of Δ'_{h}, then $\widehat{M}$ itself must be an evaluation redex.

(e) *bubble-assign*: $\Delta_{\mathrm{h}} \equiv v := N_1\,;\, w? \triangleright x \cdot N_2$

We invoke the same reasoning and terminology as in the previous two cases. If L' is a subterm of an essential element of Δ'_{h}, then we construct the table

$\widehat{M}$:	eval. redex:
$\Delta';\, w? \triangleright x \cdot N'_2$	Δ'
$v := C[\Delta'];\, w? \triangleright x \cdot N'_2$	$\widehat{M}$
$\Delta' := N'_1\,;\, w? \triangleright x \cdot N'_2$	Δ'
$v := N'_1\,;\, \Delta'$	Δ'
$v := N'_1\,;\, \Delta' \triangleright x \cdot N'_2$	Δ'
$v := N'_1\,;\, \Delta'? \triangleright x \cdot N'_2$	Δ'
$v := N;\, \Delta \triangleright x \cdot C[\Delta']$	$\widehat{M}$

If, on the other hand, L' is a subterm of a trivial element of Δ'_h, then $\widehat{M}$ itself is the evaluation redex we seek.

(f) *assign-result*: $\Delta_h \equiv v := N_1 \triangleright x \cdot N_2, x \in fv\, N_2$

The usual reasoning for redexes formed by $\triangleright$ applies once more. If L' is a subterm of an essential element of Δ'_h we can deduce the information in the following table.

$\widehat{M}$:	eval. redex:
$\Delta' \triangleright x \cdot N'_2$	Δ'
$\Delta' := N'_1 \triangleright x \cdot N'_2$	Δ'
$v := C[\Delta'] \triangleright x \cdot N'_2$	$\widehat{M}$
$v := N'_1 \triangleright x \cdot C[\Delta']$	$\widehat{M}$

If, on the other hand, L' is a subterm of a trivial element of Δ'_h, then $\widehat{M}$ itself is the evaluation redex we seek.

(g) *bubble-new*: $\Delta_h \equiv \nu v.w? \triangleright x \cdot N_1$

The usual reasoning for redexes formed by $\triangleright$ applies once more. If L' is a subterm of an essential element of Δ'_h we can deduce the information in the following table.

$\widehat{M}$:	eval. redex:
$\nu v.\Delta'$	Δ'
$\nu v.\Delta' \triangleright x \cdot N'_1$	Δ'
$\nu v.\Delta'? \triangleright x \cdot N'_1$	Δ'
$\nu v.w? \triangleright x \cdot C[\Delta']$	$\widehat{M}$

If, on the other hand, L' is a subterm of a trivial element of Δ'_h, then the entire antecedent $\widehat{M}$ is the evaluation redex we seek.

(h) *pure-eager*: $\Delta_h \equiv \mathbf{pure}\, S^{eag}[\uparrow V]$

This case applies in the calculus $\lambda[\beta\delta!eag]$.

Since the root syntactic constructor, **pure**, of this redex is not subject to $\twoheadrightarrow_{\triangleright}$-reductions, Δ'_h must also be a **pure** construct, perhaps with some associative rearrangement of the store-context. We subdivide this case according the the position of L' as a subterm of Δ'_h. We denote the antecedent of V in Δ'_h by V'.

(1) L' disjoint from V'.

In this case, $\widehat{M} \equiv \mathbf{pure}\, S^{eag}_1[C_1[\Delta'] \triangleright x \cdot S^{eag}_2[\uparrow V]]$ for some store contexts $S^{eag}_1[]$ and $S^{eag}_2[]$ and general term context $C_1[]$. Since the whole body of the **pure** expression must associate into a **pure**-redex, the subterm $C_1[L']$ must actually (associate to) a fragment of an eager store-context, that is, it must consist of a sequence of ν and assignment commands. We now subdivide the possible locations into cases yet again:

(a) L' encompasses one or more command.

In this case the redex $\Delta' \subseteq \downarrow_{\triangleright}[M]$ that reduces to L' occupies the prefix of a command sequence. This is an evaluation context.

(b) L' lies completely within a command.

The following table gives all the ways this can happen, along with the evaluation redex whose existence is required. We note that it is not possible in this case for further $\twoheadrightarrow_{\triangleright}$-reduction of the main command sequence to be required, since Δ' occurs in a term that is already in $\twoheadrightarrow_{\triangleright}$-normal form.

$\widehat{M}$:	eval. redex:
$\mathbf{pure}\, S^{eag}_1[\Delta' := N'_2 \triangleright x \cdot S^{eag}_2[\uparrow V]]$	Δ'
$\mathbf{pure}\, S^{eag}_1[N'_1 := \Delta' \triangleright x \cdot S^{eag}_2[\uparrow V]]$	$\widehat{M}$

(2) $L' \subseteq V'$.

Recalling that $\mathbf{pure}\, S^{eag}[\uparrow []]$ is an evaluation context, we construct the following table of possibilities:

$\widehat{M}$:	eval. redex:
$\mathbf{pure}\, S^{eag}[\uparrow \Delta']$	Δ'
$\mathbf{pure}\, S^{eag}[\uparrow (c^n \bullet \cdots \bullet C[\Delta'] \bullet \cdots)]$	$\widehat{M}$
$\mathbf{pure}\, S^{eag}[\uparrow (\lambda x.C[\Delta'])]$	$\widehat{M}$

(3) $V' \subset L'$.
In this case, the reduction of Δ' produces a tail of the main command sequence of Δ'_h containing the result-producing occurrence of $\uparrow$. The antecedent $\widehat{M}$ thus takes the form $\mathbf{pure}\, S^{eag}[\Delta']$, in which Δ' occurs in an evaluation context.

This concludes the proof in the case of a *pure-eager*-redex.

(i) *pure-lazy*: $\Delta_h \equiv \mathbf{pure}\, S^{laz}[\uparrow V]$
The case occurs in dealing with the calculus $\lambda[\beta\delta!laz]$. We can reason here as we did for the case of an *pure-eager*-redex, with the exception that the case in which L' is disjoint from V' is easier. The form of a lazy store-context is not as constrained as that of an eager store-context, so whenever L' is disjoint from V' the entire antecedent $\widehat{M}$ of Δ'_h is an evaluation redex because the rule *pure-lazy* is indifferent to substructure in the prefix of a store-context.

Gathering the results for all the relative positions of L' and Δ'_h, we see that we have now completed the proof of Lemma 3.7.6. ∎

Lemma 3.7.7 (Heads are preserved) *Let Δ_h be a head redex and Δ_i be an internal redex in $[M]_{\underline{\triangleright}}$. If $[M]_{\underline{\triangleright}} \xrightarrow{\Delta_i} [N]_{\underline{\triangleright}}$, then the residual of Δ_h in $\downarrow_{\triangleright}[N]$ consists of a single redex that is head redex in $[N]_{\underline{\triangleright}}$.*

Proof: Since a head redex cannot be a subterm of another redex, we have only two cases to consider concerning the relative positions of Δ_h and Δ_i.

Case 1 Δ_i and Δ_h are disjoint.

The ways in which this case can arise have already been enumerated in the case analysis for Lemma 3.7.6, Case (1):

(1) $E_1[E_2[\Delta_h] \bullet C[\Delta_i]]$,
(2) $E_1[E_2[\Delta_h] := C[\Delta_i]]$,
(3) $E_1[E_2[\Delta_h] \triangleright x \cdot C[\Delta_i]]$,
(4) $E_1[C[\Delta_i] \triangleright x \cdot E_2[\Delta_h]]$, where $C[]$ is not an evaluation context, or
(5) $E_1[\mathbf{pure}\, S^{eag}[\uparrow E_2[\Delta_h]]]$, where $\Delta_i \subseteq S^{eag}[]$. (This case applies to $\lambda[\beta\delta!eag]$).
(6) $E_1[\mathbf{pure}\, S^{laz}[\uparrow E_2[\Delta_h]]]$, where $\Delta_i \subseteq S^{laz}[]$. (This case applies to $\lambda[\beta\delta!laz]$).

In each of these cases the reduction of Δ_i leaves Δ_h as an evaluation redex; it is leftmost because it was leftmost already in $\downarrow_{\triangleright}[M]$.

Case 2 $\Delta_i \subset \Delta_h$.

In this case the residual of Δ_h (call it Δ'_h) is a redex, as can be verified by inspecting the interactions between reductions analyzed in the proof of Proposition 3.4.1 and Lemma 3.5.5. Furthermore, since $\downarrow_{\triangleright}[M] \equiv E[\Delta_h] \to E[\Delta'_h] \equiv N$ for some evaluation context $E[]$, Δ'_h is an evaluation redex of N. It remains to show that it is in fact a head redex, that is, that it is the leftmost evaluation redex.

Assume to the contrary that Δ'_h is not the leftmost evaluation redex. Then Δ'_h is either properly contained in another evaluation redex Δ, or else there is another evaluation redex Δ disjoint from, and to the left of, Δ'_h.

In the first case there are evaluation contexts $E_1[]$ and $E_2[]$ with $E_2[] \not\equiv []$ such that

$$N \equiv E_1[\Delta] \equiv E_1[E_2[\Delta'_h]].$$

Since $\Delta_i \subset \Delta_h$, there is a nonempty general term context $C[\,]$ such that

$$\downarrow_{\triangleright}[M] \equiv E_1[E_2[\Delta_h]] \equiv E_1[E_2[C[\Delta_i]]],$$

and thus (carrying out the reduction)

$$N \equiv E_1[E_2[C[\Delta_i']]],$$

where Δ_i' is the residual of Δ_i. Now the question is, how did Δ arise? The reduction of Δ_i as a subterm of Δ_h could not have created a redex outside of Δ_h, as an inspection of the set of reduction rules will show. Therefore, the antecedent of Δ existed as an evaluation redex in $\downarrow_{\triangleright}[M]$, contradicting the assumption that its proper subterm Δ_h was the head redex.

In the second case, we can find evaluation contexts $E_1[\,]$, $E_2[\,]$, $E_3[\,]$ such that N is one of the following:

(1) $E_1[E_2[\Delta] \bullet E_3[\Delta_h']]$

(2) $E_1[E_3[\Delta_h'] := E_2[\Delta]]$

(3) $E_1[v := E_2[\Delta] \triangleright x \cdot E_3[\Delta_h']]$.

Hence, $\downarrow_{\triangleright}[M]$ is one of the following:

(1) $E_1[E_2[\Delta] \bullet E_3[\Delta_h]]$

(2) $E_1[E_3[\Delta_h] := E_2[\Delta]]$

(3) $E_1[v := E_2[\Delta] \triangleright x \cdot E_3[\Delta_h]]$.

In each case, Δ is an evaluation redex to the left of Δ_h, contradicting the assumption that Δ_h is the head redex.

∎

Lemma 3.7.8 (Internals are preserved) *For any term M in $\rightarrow_{\triangleright}$-normal form and internal redex $\Delta_i \subset M$, for any internal reduction $M \rightarrow_i N$, all residuals of Δ_i are internal redexes of $\downarrow_{\triangleright}[N]$.*

Proof: Assume, to the contrary, that there is a residual Δ of Δ_i that is a head redex in N. By Lemma 3.7.6, M has a head redex Δ_h. By Lemma 3.7.7, Δ is the residual of Δ_h. This contradicts the assumption that Δ was a residual of the *internal* redex Δ_i. ∎

We have now established the properties of our calculi needed to prove the standardization theorem by the techniques (attributed to Mitschke) of [Barendregt, 1984], Section 11.4. The main thrust of the cited work is to prove the result we have stated in Lemma 3.7.5. Knowing that head and internal reductions can be interchange lets us single out the standard reduction sequences as those that proceed in an orderly fashion from left to right.

Definition 3.7.9 (Standard reduction sequence) *A sequence of reductions $M_0 \xrightarrow{\Delta_0} M_1 \xrightarrow{\Delta_1} \cdots \xrightarrow{\Delta_{n-1}} M_n$ is a* standard reduction sequence *if, for all $i < j$, Δ_j is not a residual of a redex to the left of Δ_i.*

Theorem 3.7.10 (Standardization) *If $M \rightarrow^* N$, then there exists a standard reduction sequence from M to N.*

The following related result on the computation of answer terms is used in the proof of Theorem 5.1.7. The theorem says that answer terms are built in a top-down order, outermost constructors first, then components.

Definition 3.7.11 (Top-down reduction to an answer) *A reduction sequence ending in an answer A is* top-down *if either it consists entirely of head reductions, or consists of a head reduction sequence ending in a term $c^n \bullet M_1 \bullet \cdots \bullet M_k, k \leq n$, followed by a top-down reduction sequence for M_1, and so on through M_k.*

In Definition 3.7.11 the head-ness of the subsequences is interpreted with respect to the subterm in question.

Theorem 3.7.12 (Standard reduction to an answer) *If $M \rightarrow^* A$, where A is an answer, then the standard reduction sequence from M to A is top-down.*

Proof: The proof is by induction on the struction of the answer term A. The base cases are $A \equiv c^0$ and $A \equiv f$, both of which are terms with no internal structure. In these cases the last reduction of the sequence must have produced the answer term by reducing the entire term. This is a head reduction, so there is no segment of the reduction sequence using internal reductions. The reduction sequence is thus top-down.

Now assume that $A \equiv c^n \bullet A_1 \bullet \cdots \bullet A_k, 0 < k \leq n$,. Again consider the last reduction that produced A: this reduction must have either involved the entire term as redex, producing all of A, or else it resulted in one of the subterms A_i. In the former case the last reduction is a head reduction, and hence all its predecessors are also; thus the reduction sequence is top-down. In the latter case the induction hypothesis applies to show that the reduction sequence producing A_i is top-down. Furthermore, since the reduction is standard, the subterms to the left of A_i must have been produced earlier in the reduction sequence, and the subterms to the right of A_i must already be answers. We can thus apply the induction hypothesis to the term $c^n \bullet A_1 \bullet \cdots \bullet A_{i-1}$. Putting all the top-down reduction sequences together in the order we have deduced yields a top-down reduction sequence for A. ∎

3.8 Relating properties of $\rightarrow_!$ to properties of $\rightarrow$

We have now proved the Church-Rosser theorem and standardization theorem for $\rightarrow_{\triangleright}$ on $\overset{\triangleright}{=}$-equivalence classes. In this section we translate these results back into $\rightarrow$, which is the reduction relation we actually intend to use.

Theorem 3.8.1 (Church-Rosser) *The reduction relation $\rightarrow$ on $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ is Church-Rosser.*

Proof: Suppose we are given the top half of a Church-Rosser diagram under reduction $\rightarrow$:

$$\begin{tikzcd}
& M \arrow[dl, "\rightarrow"'] \arrow[dr, "\rightarrow"] & \\
M_1 & & M_2 .
\end{tikzcd} \tag{3.3}$$

Each of the given reduction sequences consists of some interleaving of $\rightarrow_{\triangleright}$-reductions and $\rightarrow_!$-reductions on terms. Passing to $\overset{\triangleright}{=}$-equivalence classes, the $\rightarrow_{\triangleright}$-steps collapse (since they stay within the same equivalence class), and the $\rightarrow_!$-steps on terms become $\rightarrow_!$-steps on equivalence classes:

$$\begin{tikzcd}
& {[M]_{\overset{\triangleright}{=}}} \arrow[dl, "\rightarrow_!"'] \arrow[dr, "\rightarrow_!"] & \\
{[M_1]_{\overset{\triangleright}{=}}} & & {[M_2]_{\overset{\triangleright}{=}}} .
\end{tikzcd}$$

Since $\rightarrow_{!}$ is Church-Rosser on equivalence classes (Theorem 3.6.1), we can complete the diamond to obtain a term N' such that the diagram

$$\begin{tikzcd}
 & {[M]_{\overset{\triangleright}{=}}} \arrow[dl, "\rightarrow_{!}"'] \arrow[dr, "\rightarrow_{!}"] & \\
{[M_1]_{\overset{\triangleright}{=}}} \arrow[dr, dotted, "\rightarrow_{!}"'] & & {[M_2]_{\overset{\triangleright}{=}}} \arrow[dl, dotted, "\rightarrow_{!}"] \\
 & N' &
\end{tikzcd} \tag{3.4}$$

commutes. We would now like to use this diagram for $\rightarrow_{!}$ on equivalence classes to help us complete the diagram 3.3. The first step in doing this is to note that each of the terms M_1 and M_2 can be normalized with respect to $\rightarrow_{\triangleright}$ while staying within the same $\overset{\triangleright}{=}$-equivalence class, giving us (so far), the diagram

$$\begin{tikzcd}
 & M \arrow[dl, "\rightarrow"'] \arrow[dr, "\rightarrow"] & \\
M_1 \arrow[d, "\rightarrow_{\triangleright}"'] & & M_2 \arrow[d, "\rightarrow_{\triangleright}"] \\
\downarrow_{\triangleright}[M_1] & & \downarrow_{\triangleright}[M_2]
\end{tikzcd}$$

The next step is to use the information given us by diagram 3.4. This diagram assures us that, somewhere in each of the equivalence classes $[M_1]_{\overset{\triangleright}{=}}$ and $[M_2]_{\overset{\triangleright}{=}}$ there is a term that reduces via $\rightarrow_{!}$ on terms to some term in the common equivalence class N'. However, if such reductions exist, then by Corollary 3.4.2 we can be assured that they exist starting at the normal forms $\downarrow_{\triangleright}[M_1]$ and $\downarrow_{\triangleright}[M_2]$. Furthermore, since these reductions land in the same equivalence class N', the terms in which the end must be convertible via $\rightarrow_{\triangleright}$. Thus we have

$$\begin{tikzcd}
 & M \arrow[dl, "\rightarrow"'] \arrow[dr, "\rightarrow"] & \\
M_1 \arrow[d, "\rightarrow_{\triangleright}"'] & & M_2 \arrow[d, "\rightarrow_{\triangleright}"] \\
\downarrow_{\triangleright}[M_1] \arrow[d, "\rightarrow_{!}"'] & & \downarrow_{\triangleright}[M_2] \arrow[d, "\rightarrow_{!}"] \\
M_1' & \overset{\triangleright}{=} & M_2'
\end{tikzcd}$$

But now, since $\rightarrow_{\triangleright}$ is Church-Rosser (Theorem 3.3.3), the convertible terms M'_1 and M'_2 must have a common reduct M'. This fact allows us to establish the diagram

$$\begin{tikzcd}
 & M \arrow[dl, "\rightarrow"'] \arrow[dr, "\rightarrow"] & \\
M_1 \arrow[d, "\rightarrow_{\triangleright}"'] & & M_2 , \arrow[d, "\rightarrow_{\triangleright}"] \\
\downarrow_{\triangleright}[M_1] \arrow[d, "\rightarrow_{!}"'] & & \downarrow_{\triangleright}[M_2] \arrow[d, "\rightarrow_{!}"] \\
M'_1 \arrow[dr, "\rightarrow_{\triangleright}"'] & \overset{\triangleright}{=} & M'_2 \arrow[dl, "\rightarrow_{\triangleright}"] \\
 & M' &
\end{tikzcd}$$

which is a complete diamond for $\rightarrow^*$. We have thus shown that $\rightarrow$ is Church-Rosser. ▮

We now turn to the question of deriving a standardization result for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ from the results we have for the reduction $\rightarrow_!$ on $\overset{\triangleright}{=}$-equivalence classes. We lift sequences of $\rightarrow_!$-reductions to sequences of $\rightarrow$-reductions by reducing to $\rightarrow_{\triangleright}$-normal form before each $\rightarrow_!$-step. We take a standard reduction sequence under $\rightarrow$ to be one thus derived from a standard reduction sequence under $\rightarrow_!$. The question now is whether it has the same nice characterization in terms of evaluation contexts as the original reduction sequence. We note that the reduction order so defined is certainly deterministic: Proposition 3.3.8 and Theorem 3.7.10 combine to show this. Since head redexes are defined in terms of $\rightarrow_{\triangleright}$-normal forms, the derived $\rightarrow$-reduction selects the same redexes as the original $\rightarrow_!$-reduction sequence. Furthermore, by the remark in the proof of Proposition 3.3.8, we can choose the same definition of evaluation contexts to define the reduction order for the $\rightarrow_{\triangleright}$-steps as we do in defining the standard order for $\rightarrow_!$-reductions. Collecting all these remarks, we have established that the evaluation contexts defined for Definition 3.7.1 yield a standard order of evaluation for $\rightarrow$.

3.9 Chapter summary

This chapter has established the credentials of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ as a foundation for programming-language design on the same basis as the pure lambda-calculus. We have introduced the derived reduction relation of reduction modulo association and shown it to be Church-Rosser and to possess a standard evaluation order, and we have used these results to establish the same properties of the original reduction relations. In the remainder of the dissertation we will thus have no need to refer to the factored reduction relation employed as a proof technique in this chapter.

4
Operational equivalence

The reduction semantics studied in Chapter 3 gives us a basic notion of computation for programs in these calculi: a program M evaluates to an answer A if and only if the terms M and A are convertible in the calculus. However, when reasoning about program transformations, we usually want to know whether two program fragments are interchangeable based on whether they induce the same behavior in all programs containing them—this is one of the main motivations for conducting language semantics in the first place. The appropriate notion of equivalence is *operational equivalence.* Since it is possible for two program fragments to be equivalent in this sense without being convertible according to the reduction semantics, operational equivalence requires separate study. In this chapter we expand our treatment to consider the question of operational equivalence for the calculi of concern. We give the standard definition of operational equivalence, and we prove a suite of basic operational equivalences for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

4.1 Definitions and basic results

The definition of operational equivalence formalizes the notion of two terms being indistinguishable under any test.

Definition 4.1.1 *Let λ_* be some extension of the λ-calculus having the term language Λ_*. Two terms N and M are* operationally equivalent *in λ_*, written $\lambda_* \models N \cong M$, if for all contexts C in Λ_* such that $C[M]$ and $C[N]$ are closed, and for all answers A,*

$$\lambda_* \vdash C[M] = A \quad \Leftrightarrow \quad \lambda_* \vdash C[N] = A.$$

The remainder of this short section consists of a series of elementary results about operational equivalence that are used elsewhere in the dissertation.

Lemma 4.1.2 $\lambda_* \vdash M = N$ *implies* $\lambda_* \models M \cong N$.

Proof: Assume $\lambda_* \vdash M = N$, and suppose $\lambda_* \vdash C[M] = A$. Since convertibility is closed under term-formation, $\lambda_* \vdash M = N$ implies $\lambda_* \vdash C[M] = C[N]$, for any context C. The symmetry and transitivity of convertibility then imply that $\lambda_* \vdash C[N] = A$. The symmetric argument proves the converse implication, thus establishing $\lambda_* \vdash C[M] = A \Leftrightarrow \lambda_* \vdash C[N] = A$, which is the definition of $\lambda_* \models M \cong N$. ∎

Lemma 4.1.3 *For any context C, $\lambda_* \models M \cong N$ implies $\lambda_* \models C[M] \cong C[N]$.*

Proof: Assume that $\lambda_* \models M \cong N$, and suppose that $\lambda_* \vdash C'[C[M]] = A$, for some context $C'[\,]$ such that $C'[C[M]]$ and $C'[C[N]]$ are closed. Then the assumed operational equivalence implies that $\lambda_* \vdash C'[C[N]] = A$, and similar reasoning establishes the reverse implication. Thus $\lambda_* \models C[M] \cong C[N]$. ∎

Lemma 4.1.4 *For any variable x, $\lambda_* \models M \cong N \Leftrightarrow \lambda_* \models \lambda x.M \cong \lambda x.N$.*

Proof: The left-to-right direction is a special case of 4.1.3.

To prove the converse, suppose that $\lambda_* \models \lambda x.M \cong \lambda x.N$, and suppose that $\lambda_* \vdash C[M] = A$, where $C[M]$ is closed. If $x \in fv\, M$, then $\lambda_* \vdash (\lambda x.M)x = M$, and $x \in bv(C[\,])$. Let $C'[\,] \equiv C[[\,]x]$. Then $\lambda_* \vdash C'[\lambda x.M] = C[M] = A$. Since $\lambda_* \models \lambda x.M \cong \lambda x.N$ it follows that $\lambda_* \vdash C'[\lambda x.N] = A$, and therefore also $\lambda_* \vdash C[N] = A$. Since C was arbitrary, $\lambda_* \models M \cong N$. If $x \notin fv;M$, choose $C'[\,] = C[[\,]c]$ for any constant c^0. ∎

Lemma 4.1.5 *For any answer term A and arbitrary term M, $\lambda_* \models M \cong A$ if and only if $\lambda_* \vdash M = A$.*

Proof: The "if" direction is a special case of Lemma 4.1.2.

To establish the "only if" implication, suppose that $\lambda_* \models M \cong A$. Then, by Definition 4.1.1, for all contexts $C[\,]$ and answers A', $\lambda_* \vdash C[M] = A'$ if and only if $\lambda_* \vdash C[A] = A'$. In the particular case where $C[\,] \equiv [\,]$ and $A \equiv A'$, this gives us $\lambda_* \vdash M = A$ if and only if $\lambda_* \vdash A = A$. The second component of the logical equivalence is certainly true, so the first component, which is our desired result, is established. ∎

4.2 Some operational equivalences in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$

In this section we prove a small collection of operational equivalences that hold in the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. These equivalences are selected to correspond to frequently-used informal properties of programming with store-variables and assignment.

Proposition 4.2.1 *The following are operational equivalences in* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$*:*

$$\begin{array}{lrcll}
(1) & v?\triangleright x\cdot w?\triangleright y\cdot M & \cong & w?\triangleright y\cdot v?\triangleright x\cdot M & \\
(2) & v:=N;\, w:=N';\, M & \cong & w:=N';\, v:=N;\, M, & (v\neq w) \\
(3) & \nu v.w:=N;\, M & \cong & w:=N;\, \nu v.M, & (v\neq w, v\notin fsv N) \\
(4) & \nu v.\nu w.M & \cong & \nu w.\nu v.M & \\
(5) & v:=N;\, v:=N';\, M & \cong & v:=N';\, M & \\
(6) & P\triangleright x\cdot\uparrow x & \cong & P &
\end{array}$$

Proposition 4.2.2 *The following is an operational equivalence in the calculus* $\lambda[\beta\delta!eag]$ *if there is no assignment in* $S^{eag}[]$ *to any store-variable* $v\in fv\,P$.

$$(7)\quad S^{eag}[P] \;\cong\; P,\quad (fv\, S^{eag}[P] = fv\, P)$$

We will prove these operational equivalances immediately below, but first we describe what they mean in terms of informal programming concepts.

Equivalence (1) says that store-variable lookups commute. Equivalences (2), (3) and (4) say that assignments and store-variable declarations commute with themselves and with each other. Equivalence (5) says that if a variable is written twice in a row, the second assigned value is the one that counts.

Equivalence (6) is the right-identity law for monads. Although our reduction semantics does not axiomatize this law, we recover it in the theory of operational equivalence.

Equivalence (7) represents "garbage collection": it says that a store-context $S^{eag}[]$ of an expression $S^{eag}[P]$ can be dropped if no variable written or in $S^{eag}[]$ is used in P. Note that, using the "bubble" conversion laws and the commutative laws (2), (3) and (4), garbage can always be moved to an eager store-prefix. This operational equivalence is stated only for $\lambda[\beta\delta!eag]$ because there is no way of defining the set of store-variables assigned in a lazy store-context $S^{laz}[]$—not all assignment commands in such a store-context need be apparent statically.

We should note that none of these proposed operational equivalences is provable by conversion. If we consider the meta-variables as ordinary variables there is not a single redex on either side of any of the equivalences, so the two sides are actually distinct normal forms and hence not interconvertible.

Proof technique

The definition of operational equivalence demands that we prove statements that are quantified over all contexts in each calculus; the statements themselves are quantified over all answer-producing reductions. It is not immediately clear how to proceed in the face of these complex requirements, but we have available to us a proof technique developed by Odersky [Odersky, 1993a] to reduce this burden of proof. This technique requires us only to prove certain properties of the interaction between the proposed operational equivalence and the reduction rules of the calculus. We give this technique in outline before we proceed to use it in the proofs below.

Oderksy's technique requires that, for each proof, we construct a collection of symmetric rewrite rules that define a relation, *similarity*, intended to characterize the operational equivalence to be proved. In our present proposition, this collection will always consist of a single rule, which will be the operational equivalence as stated. We must then show that, in all cases in which the similarity rules have a critical overlap (see Section 3.2) with the reduction rules of the calculus, there exists a parallel application of (already-established) operational equivalences to the similar term that yields a term similar to the result of the reduction.

The main proof requirements for each operational equivalence are explained in relation to the following diagram:

$$\begin{tikzcd}
R \arrow[r, "h"'] \arrow[d, dash, "\sim_1"'] & R' \arrow[r, dotted, "h"'] & R'_1 \arrow[d, dotted, dash, "\sim_1"] \\
R'' \arrow[rr, dotted, "\cong"'] & & R''_1
\end{tikzcd}$$

In this diagram $\sim_1$ is the *parallel similarity* derived from $\sim$. In our proofs it will suffice to think of $\sim_1$ as being the union of the relation $\sim$ with syntactic identity $\equiv$. The h labeling the arrow means that the corresponding reduction is a head reduction.

For each proof below, we will give concrete meta-terms and reductions to instantiate this diagram.

Applying the proof technique requires in addition that we establish certain highly technical conditions. First, we must verify that the evaluation contexts of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ are *downward closed*, that is, that $E = C_1 \cdot C_2$ implies that C_2 is itself an evaluation context. This is obvious from inspection of the definition of evaluation contexts in Figures 3.2, 3.3, 3.4, and 3.5. Second, we must establish, for each similarity relation we introduce, that the relation *preserves evaluation contexts* and is *answer-preserving*.

Preservation of evaluation contexts means that substituting instances of a similarity rule for a metavariable in an non-evaluation context cannot transform that context into an evaluation context. Put another way, the requirement is that collapsing similarity-rule instances to a single metavariable cannot destroy an evaluation context. It is easy to see that the definitions of the evaulation contexts for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ given in Figures 3.2, 3.3, 3.4, and 3.5 prevent this possibility. The point to note is that substructure of a rule defining $E[]$ is always a metavariable or the recursive reference to $E[]$. Thus any collpase of a similarity-rule instance must occur either within a metavariable of an $E[]$-rule, because the context hole cannot occur within a similarity rule instance in the technical formulation of [Odersky, 1993a].

An answer-preserving relation satisfies $M \sim A \Rightarrow M \twoheadrightarrow^* A$; this is easily seen to be true for all our proposed similarity relations (most often vacuously, since our similarity patterns do not match answer terms), so we will omit further mention of this issue.

Proof: We now proceed to prove each of the proposed operational equivalences. Although the bottom edge of the diagram can be any operational equivalence, in many cases the operational equivalence is easily established by exhibiting a conversion. We only remark on the exceptions to this observation.

For each of the operational equivalences to be proved, we supply the needed similarity relation in the form of a set of rewrite rules and then give the data needed to instantiate the diagram above. We label the proofs of the parts of Propositions 4.2.1 and 4.2.2 by number.

(1) We adopt the similarity relation implied by $\mathcal{S} = \{v? \triangleright x \cdot w? \triangleright y \cdot M \sim w? \triangleright y \cdot v? \triangleright x \cdot M\}$.

The rules in $\mathcal{S}$ do not overlap with any of the productions defining evaluation contexts in Figures 3.2, 3.3, and 3.4. Hence, $\sim$ preserves evaluation contexts.

We now enumerate the various ways in which $\sim$ can intefere with $\twoheadrightarrow$; for each case, we show how to instantiate the diagram so as to prove that the compatible equivalence closure of $\sim$ is an operational equivalence. In this instance the cases are distinguished by which store-variables are identical in the various terms.

Case 1.1:

$$\begin{array}{ccll}
R & \equiv & u:=N;\ v?\triangleright x\cdot w?\triangleright y\cdot M, & (u\not\equiv v, u\not\equiv w, v\not\equiv w)\\
R'' & \equiv & u:=N;\ w?\triangleright y\cdot v?\triangleright x\cdot M & \\
R' & \equiv & v?\triangleright x\cdot u:=N;\ w?\triangleright y\cdot M & \\
R'_1 & \equiv & v?\triangleright x\cdot w?\triangleright y\cdot u:=N;\ M & \\
R''_1 & \equiv & w?\triangleright y\cdot v?\triangleright x\cdot u:=N;\ M &
\end{array}$$

$$\begin{array}{ccl}
R & \rightarrow_{\mathit{bubble\text{-}assign}} & R'\\
R' & \rightarrow_{\mathit{bubble\text{-}assign}} & R'_1\\
R'' & \rightarrow_{\mathit{bubble\text{-}assign}} & w?\triangleright y\cdot u:=N;\ v?\triangleright x\cdot M\\
 & \rightarrow_{\mathit{bubble\text{-}assign}} & R''_1
\end{array}$$

Case 1.2:

$$\begin{array}{ccll}
R & \equiv & w:=N;\ v?\triangleright x\cdot w?\triangleright y\cdot M, & (u\not\equiv v, v\not\equiv w)\\
R'' & \equiv & w:=N;\ w?\triangleright y\cdot v?\triangleright x\cdot M & \\
R' & \equiv & v?\triangleright x\cdot w:=N;\ w?\triangleright y\cdot M & \\
R'_1\equiv R''_1 & \equiv & v?\triangleright x\cdot w:=N;\ [N/y]\,M &
\end{array}$$

$$\begin{array}{ccl}
R & \rightarrow_{\mathit{bubble\text{-}assign}} & R'\\
R' & \rightarrow_{\mathit{fuse}} & R'_1\\
R'' & \rightarrow_{\mathit{fuse}} & w:=N;\ [N/y]\,(v?\triangleright x\cdot M)\\
 & \equiv & w:=N;\ v?\triangleright x\cdot[N/y]\,M\\
 & \rightarrow_{\mathit{bubble\text{-}assign}} & R''_1
\end{array}$$

Case 1.3:

$$\begin{array}{ccll}
R & \equiv & v:=N;\ v?\triangleright x\cdot w?\triangleright y\cdot M, & (v\not\equiv w)\\
R'' & \equiv & v:=N;\ w?\triangleright y\cdot v?\triangleright x\cdot M & \\
R' & \equiv & v:=N;\ [N/x]\,(w?\triangleright y\cdot M) & \\
 & \equiv & v:=N;\ w?\triangleright y\cdot[N/x]\,M & \\
R'_1\equiv R''_1 & \equiv & R' &
\end{array}$$

$$\begin{array}{ccl}
R & \rightarrow_{\mathit{fuse}} & R'\\
R'' & \rightarrow_{\mathit{bubble\text{-}assign}} & w?\triangleright y\cdot v:=N;\ v?\triangleright x\cdot M\\
 & \rightarrow_{\mathit{fuse}} & R''_1
\end{array}$$

Case 1.4:

$$\begin{array}{ccll}
R & \equiv & u:=N;\ v?\triangleright x\cdot v?\triangleright y\cdot M, & (u\not\equiv v)\\
R' & \equiv & v?\triangleright x\cdot u:=N;\ v?\triangleright y\cdot M & \\
R'' & \equiv & u:=N;\ v?\triangleright y\cdot v?\triangleright x\cdot M & \\
R'_1 & \equiv & v?\triangleright x\cdot v?\triangleright y\cdot u:=N;\ M & \\
R''_1 & \equiv & v?\triangleright y\cdot v?\triangleright x\cdot u:=N;\ M &
\end{array}$$

$$\begin{array}{ccl}
R & \rightarrow_{\mathit{bubble\text{-}assign}} & R'\\
R' & \rightarrow_{\mathit{bubble\text{-}assign}} & R'_1\\
R'' & \rightarrow_{\mathit{bubble\text{-}assign}} & v?\triangleright y\cdot u:=N;\ v?\triangleright x\cdot M\\
 & \rightarrow_{\mathit{bubble\text{-}assign}} & R''_1
\end{array}$$

Case 1.5:

$$\begin{array}{lcl}
R & \equiv & v := N;\ v?\triangleright x \cdot v?\triangleright y \cdot M \\
R' & \equiv & v := N;\ [N/x]\, v?\triangleright y \cdot M \\
 & \equiv & v := N;\ v?\triangleright y \cdot [N/x]\, M \\
R'' & \equiv & R \\
R'_1 & \equiv & R' \\
R''_1 & \equiv & R'_1
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{fuse} & R' \\
R' & \equiv & R'_1 \\
R'' & \rightarrow_{fuse} & R''_1
\end{array}$$

For the remaining operational equivalences, we give only the basic data needed to complete the diagram.

(2) $\mathcal{S} = \{v := N;\ w := N';\ M \sim w := N';\ v := N;\ M\}, (v \not\equiv w)$.

Case 2.1:

$$\begin{array}{lcll}
R & \equiv & v := N;\ w := N';\ w?\triangleright x \cdot M', & (v \not\equiv w) \\
R' & \equiv & v := N;\ w := N';\ [N'/x]\, M & \\
R'_1 & \equiv & R' & \\
R'' & \equiv & w := N';\ v := N;\ w?\triangleright x \cdot M' & \\
R''_1 & \equiv & w := N';\ v := N;\ [N'/x]\, M' &
\end{array}$$

$$\begin{array}{lcll}
R & \rightarrow_{fuse} & R' & \\
R'' & \rightarrow_{bubble\text{-}assign} & w := N';\ w?\triangleright x \cdot v := N;\ M', & (\text{since } x \notin fv\, N) \\
 & \rightarrow_{fuse} & w := N';\ [N'/x]\,(v := N;\ M') & \\
 & \equiv & R''_1 &
\end{array}$$

Case 2.2:

$$\begin{array}{lcll}
R & \equiv & v := N;\ w := N';\ v?\triangleright x \cdot M, & (v \not\equiv w) \\
R' & \equiv & v := N;\ v?\triangleright x \cdot w := N';\ M, & (x \notin fv\, N') \\
R'' & \equiv & w := N';\ v := N;\ v?\triangleright x \cdot M & \\
R'_1 & \equiv & v := N;\ w := N';\ [N/x]\, M & \\
R''_1 & \equiv & w := N';\ v := N;\ [N/x]\, M &
\end{array}$$

$$\begin{array}{lcll}
R & \rightarrow_{bubble\text{-}assign} & R' & \\
R' & \rightarrow_{fuse} & v := N;\ [N/x]\,(w := N';\ M) & \\
 & \equiv & R'_1, & (\text{since } x \notin fv\, N') \\
R'' & \rightarrow_{fuse} & R''_1 &
\end{array}$$

Case 2.3:

$$\begin{array}{lcll}
R & \equiv & v := N;\ w := N';\ u?\triangleright x \cdot M, & (u \not\equiv w, u \not\equiv v) \\
R' & \equiv & v := N;\ u?\triangleright x \cdot w := N';\ M & \\
R'' & \equiv & w := N';\ v := N;\ u?\triangleright x \cdot M & \\
R'_1 & \equiv & u?\triangleright x \cdot v := N;\ w := N';\ M & \\
R''_1 & \equiv & u?\triangleright x \cdot w := N';\ v := N;\ M &
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{bubble\text{-}assign} & R' \\
R' & \rightarrow_{bubble\text{-}assign} & R'_1 \\
R'' & \rightarrow_{bubble\text{-}assign} & w := N';\ u?\triangleright x \cdot v := N;\ M \\
 & \rightarrow_{bubble\text{-}assign} & R''_1
\end{array}$$

(3) $\mathcal{S} = \{\nu v . w := N;\ M \sim w := N;\ \nu v . M \mid v \not\equiv w, v \notin fv\, N\}$

Case 3.1:

$$\begin{array}{lcl}
R & \equiv & w:=N;\ \nu v.u?\triangleright x\cdot M, \quad (u\not\equiv w, u\not\equiv v)\\
R' & \equiv & w:=N;\ u?\triangleright x\cdot \nu v\, M\\
R'' & \equiv & \nu v.w:=N;\ u?\triangleright x\cdot M\\
R'_1 & \equiv & u?\triangleright x\cdot w:=N;\ \nu v\, M\\
R''_1 & \equiv & u?\triangleright x\cdot \nu v.w:=N;\ M
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{\mathit{bubble\text{-}new}} & R'\\
R' & \rightarrow_{\mathit{bubble\text{-}assign}} & R'_1\\
R'' & \leftarrow_{\mathit{bubble\text{-}assign}} & \nu v.u?\triangleright x\cdot w:=N;\ M\\
 & \rightarrow_{\mathit{bubble\text{-}new}} & R''_1
\end{array}$$

The last lines of the preceding derivation are the first case in which we establish the operational equivalence on the bottom edge of the diagram via a general conversion rather than a left-to-right sequence of reductions.

Case 3.2:

$$\begin{array}{lcl}
R & \equiv & w:=N;\ \nu v.w?\triangleright x\cdot M, \quad (u\not\equiv v)\\
R' & \equiv & w:=N;\ w?\triangleright x\cdot \nu v\, M\\
R'' & \equiv & \nu v.w:=N;\ w?\triangleright x\cdot M\\
R'_1 & \equiv & w:=N;\ [N/x]\,(\nu v\, M)\\
 & \equiv & w:=N;\ \nu v.[N/x]\,M\\
R''_1 & \equiv & \nu v.w:=N;\ [N/x]\,M
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{\mathit{bubble\text{-}new}} & R'\\
R' & \rightarrow_{\mathit{fuse}} & R'_1\\
R'' & \rightarrow_{\mathit{fuse}} & R''_1
\end{array}$$

Case 3.3:

$$\begin{array}{lcl}
R & \equiv & \nu v.w:=N;\ u?\triangleright x\cdot M, \quad (u\not\equiv w, u\not\equiv v)\\
R' & \equiv & \nu v.u?\triangleright x\cdot w:=N;\ M\\
R'' & \equiv & w:=N;\ \nu v.u?\triangleright x\cdot M\\
R'_1 & \equiv & u?\triangleright x\cdot \nu v.w:=N;\ M\\
R''_1 & \equiv & u?\triangleright x\cdot w:=N;\ \nu v\, M
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{\mathit{bubble\text{-}assign}} & R'\\
R' & \rightarrow_{\mathit{bubble\text{-}new}} & R'_1\\
R'' & \rightarrow_{\mathit{bubble\text{-}new}} & w:=N;\ u?\triangleright x\cdot \nu v\, M\\
 & \rightarrow_{\mathit{bubble\text{-}assign}} & R''_1
\end{array}$$

Case 3.4:

$$\begin{array}{lcl}
R & \equiv & \nu v.w:=N;\ w?\triangleright x\cdot M\\
R' & \equiv & \nu v.w:=N;\ [N/x]\,M\\
R'' & \equiv & w:=N;\ \nu v.w?\triangleright x\cdot M\\
R'_1 & \equiv & R'\\
R''_1 & \equiv & w:=N;\ \nu v.[N/x]\,M
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{\mathit{fuse}} & R'\\
R'' & \rightarrow_{\mathit{bubble\text{-}new}} & w:=N;\ w?\triangleright x\cdot \nu v\, M\\
 & \rightarrow_{\mathit{fuse}} & w:=N;\ [N/x]\,(\nu v\, M)\\
 & \equiv & R''_1
\end{array}$$

Case 3.5:

$$\begin{array}{lcl} R & \equiv & (\nu v.w := N;\ M_1) \rhd x \cdot M_2 \\ R' & \equiv & \nu v.w := N;\ M_1 \rhd x \cdot M_2 \\ R'' & \equiv & (w := N;\ \nu v.M_1) \rhd x \cdot M_2 \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & w := N;\ \nu v.M_1 \rhd x \cdot M_2 \end{array}$$

$$\begin{array}{lcl} R & \rightharpoonup_{extend} & R' \\ R'' & \rightharpoonup_{assoc} & R''_1 \end{array}$$

(4) $\mathcal{S} = \{\nu v.\nu w.M \sim \nu w.\nu v.M\}$

Some of the critical pairs we encounter in the course of proving this operational equivalence require more refined reasoning than we have yet encountered in these proofs.

Case 4.1:

$$\begin{array}{lcl} R & \equiv & (\nu v.\nu w.M) \rhd x \cdot N \\ R' & \equiv & \nu v.(\nu w.M) \rhd x \cdot N \\ R'' & \equiv & (\nu w.\nu v.M) \rhd x \cdot N \\ R'_1 & \equiv & \nu v.\nu w.(M \rhd x \cdot N) \\ R''_1 & \equiv & \nu w.\nu v.(M \rhd x \cdot N) \end{array}$$

$$\begin{array}{lcl} R & \rightharpoonup_{extend} & R' \\ R' & \rightharpoonup_{extend} & R'_1 \\ R'' & \rightharpoonup_{extend} & \nu w.(\nu v.M) \rhd x \cdot N \\ & \rightharpoonup_{extend} & R''_1 \end{array}$$

Case 4.2:

$$\begin{array}{lcll} R & \equiv & \nu v.\nu w.u? \rhd x \cdot M & (u \not\equiv v, u \not\equiv w) \\ R' & \equiv & \nu v.u? \rhd x \cdot \nu w.M & \\ R'' & \equiv & \nu w.\nu v.u? \rhd x \cdot M & \\ R'_1 & \equiv & u? \rhd x \cdot \nu v.\nu w.M & \\ R''_1 & \equiv & u? \rhd x \cdot \nu w.\nu v.M & \end{array}$$

$$\begin{array}{lcl} R & \rightharpoonup_{bubble\text{-}new} & R' \\ R' & \rightharpoonup_{bubble\text{-}new} & R'_1 \\ R'' & \rightharpoonup_{bubble\text{-}new} & \nu w.u? \rhd x \cdot \nu v.M \\ & \rightharpoonup_{bubble\text{-}new} & R''_1 \end{array}$$

Case 4.3:

$$\begin{array}{lcll} R & \equiv & \nu v.\nu w.v? \rhd x \cdot M & (v \not\equiv w) \\ R' & \equiv & \nu v.v? \rhd x \cdot \nu w.M & \\ R'' & \equiv & \nu w.\nu v.v? \rhd x \cdot M & \\ R'_1 & \equiv & R' & \\ R''_1 & \equiv & R'_1 & \end{array}$$

$$\begin{array}{lcll} R & \rightharpoonup_{bubble\text{-}new} & R' & \\ R'' & \cong & R' & \text{(by the following argument)} \end{array}$$

Informally, both R'' and R' are stuck terms: the only way a program containing either one as a subterm can reduce to an answer is to throw the term away[1], since no rule (including the δ-rule), can examine the substructure of terms constructed with ν (or any other term except a value or a fully-applied constructor). Thus one would expect that putting both terms in the same context would yield a computation that either gets stuck or yields the same result regardless of the terms in question.

Formally, we apply the critical-pair method once again. We define an auxiliary similarity relation $\mathcal{S} =$

[1] Actually, there could be reductions inside M, but there is still no way for either term as a *whole* to reduce to an answer.

$\{\nu v.v?\triangleright x\cdot P \sim \nu w.\nu v.v?\triangleright x\cdot P | v \not\equiv w\}$. This relation interferes with no reduction rules whatsoever, and so it induces an operational equivalence, as required. This rather vacuous observation is the formal justification for the informal expectation expressed just above.

(5) $\mathcal{S} = \{v := N;\ v := N';\ M \sim v := N';\ M\}$

Case 5.1:

$$\begin{array}{lcl} R & \equiv & v := N';\ v?\triangleright x\cdot M \\ R' & \equiv & v := N';\ [N'/x]\,M \\ R'' & \equiv & v := N;\ v := N';\ v?\triangleright x\cdot M \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & v := N;\ v := N';\ [N'/x]\,M \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{fuse} & R' \\ R'' & \rightarrow_{fuse} & R''_1 \end{array}$$

Case 5.2:

$$\begin{array}{lcl} R & \equiv & v := N';\ w?\triangleright x\cdot M \\ R' & \equiv & w?\triangleright x\cdot v := N';\ M \\ R'' & \equiv & v := N;\ v := N';\ w?\triangleright x\cdot M \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & w?\triangleright x\cdot v := N;\ v := N';\ M \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{bubble\text{-}assign} & R' \\ R'' & \rightarrow_{bubble\text{-}assign} & v := N;\ w?\triangleright x\cdot v := N';\ M \\ & \rightarrow_{bubble\text{-}assign} & R''_1 \end{array}$$

Case 5.3:

$$\begin{array}{lcl} R & \equiv & v := N;\ v := N';\ v?\triangleright x\cdot M \\ R' & \equiv & v := N;\ v := N';\ [N'/x]\,M \\ R'' & \equiv & v := N';\ v?\triangleright x\cdot M \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & v := N';\ [N'/x]\,M \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{fuse} & R' \\ R'' & \rightarrow_{fuse} & R''_1 \end{array}$$

Case 5.4:

$$\begin{array}{lcl} R & \equiv & v := N;\ v := N';\ u? \triangleright x \cdot M \quad (u \not\equiv v) \\ R' & \equiv & v := N;\ u? \triangleright x \cdot v := N';\ M \\ R'' & \equiv & v := N';\ u? \triangleright x \cdot M \\ R'_1 & \equiv & u? \triangleright x \cdot v := N;\ v := N';\ M \\ R''_1 & \equiv & u? \triangleright x \cdot v := N';\ M \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{\textit{bubble-assign}} & R' \\ R' & \rightarrow_{\textit{bubble-assign}} & R'_1 \\ R'' & \rightarrow_{\textit{bubble-assign}} & R''_1 \end{array}$$

(6) For this equivalence, we adopt the similarity relation defined by $\mathcal{S} = \{P \triangleright x \cdot \uparrow x \sim P\}$, where the metavariable P ranges over procedures only. This similarity relation only interferes with the rule *(assoc)*.

$$\begin{array}{lcl} R & \equiv & (P \triangleright x \cdot \uparrow x) \triangleright y \cdot Q \\ R' & \equiv & P \triangleright x \cdot (\uparrow x \triangleright y \cdot Q) \\ R'' & \equiv & P \triangleright y \cdot Q \\ R'_1 & \equiv & P \triangleright x \cdot [x/y]\, Q \\ R''_1 & \equiv & R'_1 \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{\textit{assoc}} & R' \\ R' & \rightarrow_{\textit{unit}} & R'_1 \\ R'' & \equiv & R''_1 \end{array}$$

The last equivalence mentioned is just the α-renaming of R'_1.

This concludes the proof of Proposition 4.2.1. ∎

Proof: (7) $\mathcal{S} = \{S^{eag}[P] \sim P \mid S^{eag}[\,]$ an eager store-context, $\mathit{fv}\ S^{eag}[P] = \mathit{fv}\ P$, and no free store-variable of P is assigned in $S^{eag}[\,]\}$

This similarity relation can interfere both with the purification rules and with the assignment rules. All the critical pairs having to do with the purification laws yield essentially the same diagram verification task, so we give only a single example:

$$\begin{array}{lcl} R & \equiv & \mathbf{pure}(S^{eag}[\uparrow (\lambda x.M)]) \\ R' & \equiv & \lambda x.\mathbf{pure}\, S^{eag}[\uparrow M] \\ R'' & \equiv & \mathbf{pure}(\uparrow (\lambda x.M)) \\ R'_1 & \equiv & R' \\ R''_1 & \equiv & \lambda x.\mathbf{pure}(\uparrow M) \end{array}$$

$$\begin{array}{lcl} R & \rightarrow_{\textit{pure-eager}} & R' \\ R'' & \rightarrow_{\textit{pure-eager}} & R''_1 \end{array}$$

The uses of $\sim$ in the converse direction are equally simple to verify.

The verification of the proof conditions in the cases of interference with assignment rules requires that we invoke our knowledge concerning the structure of $S^{eag}[\,]$. By the definition of a store-context (Figure 2.11), all assignments in $S^{eag}[\,]$ must be to tags defined in $S^{eag}[\,]$. Also, we have assumed that $S^{eag}[\,]$ is nonempty, so the immediate context of the metavariable M in the definition of $\mathcal{S}$ is either of the form $\nu v.S^{eag}[\,]$ or the form $v := N;\ S^{eag}[\,]$. We work out one of the three simple cases of interference with an assignment rule:

$$\begin{array}{lcl}
R & \equiv & S^{eag}[v := N;\ w?\triangleright x \cdot M] \\
R' & \equiv & S^{eag}[w?\triangleright x \cdot v := N;\ M] \\
R'' & \equiv & v := N;\ w?\triangleright x \cdot M \\
R'_1 & \equiv & R' \\
R''_1 & \equiv & w?\triangleright x \cdot v := N;\ M
\end{array}$$

$$\begin{array}{lcl}
R & \rightarrow_{bubble\text{-}assign} & R' \\
R'' & \rightarrow_{bubble\text{-}assign} & R''_1
\end{array}$$

This concludes the proof of Proposition 4.2.2. ∎

4.3 Chapter summary

We have introduced the general definitions of operational equivalence and proved some simple operational equivalences concerning assignment and stores for our calculi. We have chosen the particular operational equivalences proved in this chapter both because they are representative of the informal reasoning that programmers (and compilers) make about programs with assignment and because they are simple; the selection is in no sense complete. Although one could hope for a more systematic collection of operational equivalence results covering more of the usual territory of program transformation, the overall theory must remain undecidable, like all theories of program equivalence in powerful languages containing nonterminating programs.

5
Simulation and conservation

In previous chapters we have presented calculi for reasoning about certain programming languages having assignable variables. We have already shown the calculi to be reasonable (Church-Rosser) and deterministically evaluable (standardization). We now turn our attention to showing that our formalisms do indeed have an interpretation mapping procedures to state transformers on a suitable abstract machine.

We use this formal interpretation in two ways. First, it justifies the calculi as a basis for design and implementation of a programming language, since one can implement the abstract machine. Second, by describing the abstract machine in terms of a lambda-calculus with constants, we are able to establish that the eager-store calculus is a *conservative extension* of the lambda-calculus so employed. This means that the added constructs do not modify the operational semantics of the embedded functional calculus. This property supports the assertion that the implied programming language not only implements assignment, but also remains functional.

Section 5.1 introduces the calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ which provide a view of assignment by modeling a store explicitly in the calculus. We prove the equivalence of these calculi with $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, respectively.

In Section 5.2 we use the newly-introduced calculus $\lambda[\beta\delta\sigma eag]$ as part of a chain of reasoning that establishes that the original (implicit-store) calculus with assignment is a conservative extension of a basic functional lambda-calculus. We discuss the difficulties behind our failure to establish a parallel result for $\lambda[\beta\delta\sigma laz]$ in Section 5.2.6.

5.1 Simulation of calculi with assignment by calculi with explicit stores

The connection between lambda-calculi and machines for evaluating them is well-established, as is the connection between assignable stores and machines. In the most basic formulation from the theory of computation, a *state machine* or *automaton* is a set S of *internal states*, along with a *state transition function* mapping S to S. Landin's SECD machine [Landin, 1964] elaborates this basic notion in order to define an abstract machine suitable for describing the evaluation mechanism of the expressions making up a programming language. The implementation of an assignable store via a state machine is usually taken as obvious.

The calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, however, do not model stores explicitly. Rather, they provide an axiomatic basis for reasoning about store-computations by dictating how each assignment affects subsequent reads. Our goal in the first part of this chapter is to translate the calculi of concern into intermediate calculi having explicit stores. We then invoke the argument of the preceding paragraph to justify stopping with this intermediate form without proceeding to construct a detailed abstract machine for each calculus.

As we mentioned in Section 1.5, functional programming models that represent stores explicitly must usually account for the fact that use of a store may not be single-threaded, and thus might not be realizable by destructive update of an actual machine's store. The calculi we introduce in this chapter do not directly address single-threadedness; rather, this property emerges from the correspondence we prove in Section 5.1.2 between the implicit-store and explicit-store calculi. Because we are only interested in explicit-store terms that correspond to implicit-store terms, and because the correspondence relation imposes single-threadedness, this issue is subsumed in the results presented in this section.

			$\sigma \in$ *Stores*
M	$::=$	$\cdots$ Figure 2.8 $\cdots$	*previously defined constructs*
	$\mid$	$\nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle$	*term M in store σ with local store-variables $\vec{v}$*
σ	$::=$	$\{\}$	*the empty store*
	$\mid$	$\sigma \oplus \{v : M\}$	*store augmented with a new binding*

Figure 5.1: Additional syntax for explicit-store calculi

$dom\,\sigma$	The set of store-variables in the domain of σ
$\sigma\!\downarrow v$	The binding of store-variable v in store σ, if one exists
$\sigma\,!\,v : M$	The store having binding M for v, and all other bindings as in σ

Figure 5.2: Notation for explicit stores

$$
\begin{array}{rcll}
\nu_\sigma \vec{v}.\sigma \cdot \langle \nu v.N \rangle & \rightarrow & \nu_\sigma \{\vec{v}, v\}.\sigma \cdot \langle N \rangle & & (\sigma\nu) \\
\nu_\sigma \vec{v}.\sigma \cdot \langle v := M; N \rangle & \rightarrow & \nu_\sigma \vec{v}.(\sigma\,!\,v : M :\,) \cdot \langle N \rangle & & (\sigma{:=}) \\
\nu_\sigma \vec{v}.\sigma \cdot \langle v? \rhd x \cdot N \rangle & \rightarrow & \nu_\sigma \vec{v}.\sigma \cdot \langle [\sigma\!\downarrow v/x]\, N \rangle & (v \in dom\,\sigma) & (\sigma?) \\
\mathbf{pure}\, M & \rightarrow & \nu_\sigma \{\,\}.\{\,\} \cdot \langle M \rangle & & (\sigma_{block}) \\
\\
\{\vec{v}, v\} & \equiv & \vec{v} \cup \{v\} & & (abbreviation)
\end{array}
$$

Figure 5.3: Common reduction rules for explicit-store calculi

$$
\begin{array}{rcll}
\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow (c^n \bullet M_1 \bullet \cdots \bullet M_k) \rangle & \rightarrow & c^n \bullet (\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow M_1 \rangle) \bullet \cdots \bullet (\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow M_k \rangle) & \sigma_{peag} \\
\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow f \rangle & \rightarrow & f & \sigma_{peag} \\
\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow \lambda x.M \rangle & \rightarrow & \lambda x.\nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle & \sigma_{peag}
\end{array}
$$

Figure 5.4: Reduction rules for eager explicit-store calculus $\lambda[\beta\delta\sigma eag]$

$$
\begin{array}{rcll}
\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[\uparrow (c^n \bullet M_1 \bullet \cdots \bullet M_k)] \rangle & \rightarrow & c^n \bullet (\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[\uparrow M_1] \rangle) \bullet \cdots \bullet (\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[\uparrow M_k] \rangle) & \sigma_{plaz} \\
\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[\uparrow f] \rangle & \rightarrow & f & \sigma_{plaz} \\
\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[\uparrow \lambda x.M] \rangle & \rightarrow & \lambda x.\nu_\sigma \vec{v}.\sigma \cdot \langle S^{laz}[M] \rangle & \sigma_{plaz}
\end{array}
$$

Figure 5.5: Reduction rules for lazy explicit-store calculus $\lambda[\beta\delta\sigma laz]$

5.1.1 The calculi with explicit stores

The calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ are defined in Figures 5.1, 5.3, 5.4, and 5.5. The basic idea of these calculi is to replace rules *fuse*, *bubble-assign*, and *bubble-new* of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ by rules for the explicit manipulation of a store σ. Instead of acting as a boundary for observations, the **pure** construct initializes a store computation with an empty store. Instead of observations of the store bubbling from right to left, allocation of locations, updates, and reads are performed from left to right. The computation within a **pure** can thus be regarded as a machine execution embedded within the reductions of an expression-oriented calculus. Stores are unordered sets of bindings in which no store-variable is bound more than once; this is the same notion as that of finite functions or records.

It is notable that the syntax for stores in Figure 5.1 admits arbitrary expressions for the value bound to a store-variable. This definition conforms to the semantics implied in the definitions of the implicit-store calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$: the value stored is subject to the evaluation order of *application*, which is always by-name for our calculi. The distinction between $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ lies in the evaluation order that pertains to the relationship between a store-variable and the expression stored, which is an entirely orthogonal issue. It would be quite surprising if merely assigning an expression to a store-variable forced it to be reduced to weak head normal form!

Figure 5.2 gives the meta-notations that we use in referring to explicit stores in this chapter. Like the notation $[M/x]\,N$ that we use for substitution, these notations are not themselves part of the formal syntax of the calculi, whereas the notation $\sigma \oplus v : M$ *is* part of the formal syntax.

As Figure 5.1 shows, the explicit-store calculi extend $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ with an extra kind of term denoting a term in the context of an explicit store. This syntactic construct is rather complex because the scoping of store-variable names is independent of the locality of stores in our calculi. The term $\nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle$ declares a set of store-variables $\vec{v} \equiv \{v_1, \ldots, v_n\}, n \geq 0$ to be local to the notated store σ and to the term M

(we use $\vec{v}$ as a compact notation for such a set of variables). The store σ may contain bindings of non-local store-variables, and the terms bound to store-variables may contain non-local store-variables.

The syntax $\nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle$ is intended to be used when M is a command; however, other terms are not prohibited (although we do not assign them an informal meaning—they will all become "stuck"). It is not possible to prohibit nested occurrences of $\nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle$ because this syntax models **pure**-expressions, and **pure**-expressions can be nested. We tolerate the proliferation of meaningless terms based on this syntax because we intend to use the explicit-store calculi primarily as a target of translation from $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$; the terms that do not actually occur as translations are thus filtered out naturally.

The reduction rules specified in Figure 5.3 make use of the notations for explicit stores that are tabulated in Figure 5.2. These rules are straightforward transcriptions of the meaning of the various assignment constructs into the notation of explicit stores. Since our notation for explicit stores separates the declaration of the existence of a store-variable from its binding in the store, an undefined store-variable is represented by its absence from the store in which it is sought—this is the side condition on rule $\sigma ?$. Rule $\sigma\nu$ merely transfers the local scope of the newly-declared store-variable from the command sequence in which it occurs to the explicit store in which the command was encountered. As always in this dissertation, we observe Barendregt's convention that bound and free names are distinct; there is thus no danger of semantic gibberish resulting from capture of occurrences of v in σ because the bound v is really α-renamed behind the reduction arrow where we can't see it. We also note that rule $\sigma{:=}$ is indifferent to the prior existence of a binding for v in the store σ and that all of the rules are indifferent to the locality of declaration of a store-variable.

Rule σ_{block} initializes a store; it corresponds to a **pure** boundary in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

The eager- and lazy-store versions of the explicit-store calculus differ, as before, in the rules for propagating the store into the subterms of a returned value. In the eager version (Figure 5.4), all store actions must have been reduced away before the store is pushed past a constructor of the result; this corresponds to the structure of store contexts $S^{eag}[]$, in which there may be no outstanding reads. In the lazy version (Figure 5.5), the store may be pushed past the constructor as soon as the fact that a result is returned can be seen from the structure of the argument to **pure**; this corresponds to the more lenient lazy store contexts $S^{laz}[]$. The presence of the store-context in the rules in Figure 5.5 reflects the possibility of pushing the store-computation into the structure of a result term before having executed all the commands in the context.

The calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ have both the Church-Rosser property and standard orders of reduction. Because the proof techniques involved are basically a repetition of the corresponding proofs for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ in Chapter 3, we have placed the proofs of these properties for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ in Appendix A.

In the following discussion of explicit-store calculi reductions in an explicit-store calculus are denoted $\rightarrow_\sigma$ where the distinction is not obvious from the context. Figure 5.2 gives several other notatio used in the following exposition.

5.1.2 Equivalence of bubble/fuse rules and store-transition rules

In Section 5.1.1, we introduced calculi that represent a store explicitly. In this section, we consider programs in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ under the reduction rules of $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$; since every term of each implicit-store calculus is also a legal term of the corresponding explicit-store calculus this endeavor makes sense. We show that such programs are assigned the same semantics by both the explicit-store and the implicit-store calculi.

The central step in establishing this equivalence is to establish the correspondence of $\lambda[\beta\delta!eag]$ or $\lambda[\beta\delta!laz]$ terms with $\lambda[\beta\delta\sigma eag]$ or $\lambda[\beta\delta\sigma laz]$ terms such that the correspondence relates terms in different calculi that have the same informal meaning as store operations. We first define the correspondence relation and then show (in a sense to be defined) that the correspondence is preserved under reductions.

In the following definition, we refer to eager store-contexts $S^{eag}[]$ despite the fact that the definition is applicable to lazy-store calculi as well: we are referring only to the syntactic form of such contexts, not to their role in defining reductions for eager-store calculi.

The correspondence relation, which we denote by the symbol $\hat{=}$, relates terms of an implicit-store calculi and terms of explicit-store calculi. The relation links a term of $\lambda[\beta\delta!eag]$ or $\lambda[\beta\delta!laz]$ (on the left) and $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ (on the right). We define $\hat{=}$ by means of two mutually inductive sets of inference

$$\frac{M \equiv M'}{M \mathrel{\hat{=}} M'} \qquad \frac{M \mathrel{\hat{=}} M'}{M? \mathrel{\hat{=}} M'?}$$

$$\frac{S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle \quad M \mathrel{\hat{=}} M'}{\mathbf{pure}\, S^{eag}[M] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle M' \rangle} \qquad \frac{M \mathrel{\hat{=}} M' \quad N \mathrel{\hat{=}} N'}{(M := N) \mathrel{\hat{=}} (M' := N')}$$

$$\frac{M \mathrel{\hat{=}} M' \quad N \mathrel{\hat{=}} N'}{M \bullet N \mathrel{\hat{=}} M' \bullet N'} \qquad \frac{M \mathrel{\hat{=}} M' \quad N \mathrel{\hat{=}} N'}{(M \triangleright x \cdot N) \mathrel{\hat{=}} (M' \triangleright x \cdot N')}$$

$$\frac{M \mathrel{\hat{=}} M'}{\lambda x.M \mathrel{\hat{=}} \lambda x.M'} \qquad \frac{M \mathrel{\hat{=}} M'}{\uparrow M \mathrel{\hat{=}} \uparrow M'}$$

$$\frac{M \mathrel{\hat{=}} M'}{\nu v.M \mathrel{\hat{=}} \nu v.M'} \qquad \frac{M \mathrel{\hat{=}} M'}{\mathbf{pure}\, M \mathrel{\hat{=}} \mathbf{pure}\, M'}$$

Figure 5.6: The correspondence relation for terms

$$\frac{}{[\,] \mathrel{\hat{=}} \nu_\sigma\{\}.\{\} \cdot \langle [\,] \rangle} \qquad \frac{S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle \quad M \mathrel{\hat{=}} M'}{v := M;\ S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \,!\, v : M' \cdot \langle [\,] \rangle}\ (v \notin dom\,\sigma)$$

$$\frac{S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle}{\nu S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma\{\vec{v}, v\}.\sigma \cdot \langle [\,] \rangle} \qquad \frac{S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle}{v := M;\ S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle}\ (v \in dom\,\sigma)$$

Figure 5.7: The correspondence relation for store-contexts and stores

rules given in Figure 5.6. The rule whose hypothesis is $M \equiv M'$ acts as the axiom for the inference system; all but one of the other rules are those we would expect if we were defining $\hat{=}$ to be a congruence. The exception is the rule involving **pure**, which acts as the interface to the definition of correspondence between store-contexts (on the left) and explicit stores (on the right) given in Figure 5.7.

Among the inference rules given in Figure 5.7 there is an obvious axiom relating empty store-contexts and empty stores. The rule for store-variable declaration straightforwardly relates the declaration mechanisms in the two pairs of calculi. The remaining rules implement the informal notion that the store corresponding to an implicit store-context records only the most recent (innermost) assignment to a store-variable; the presence of an overriding binding is detected by the presence of the store-variable in the domain of the store given by a nested application of the inference rules. The interface back into the rules for terms is the hypothesis $M \mathrel{\hat{=}} M'$ of the rule for introducing the actually effective assignment into both contexts.

This definition of correspondence implies that every term or store in an explicit-store calculus has a corresponding term or store context in an implicit-store calculus:

Lemma 5.1.1 *(i) For any term M in an explicit-store calculus, there exists a term implicit(M) in the corresponding implicit-store calculus such that implicit(M)* $\mathrel{\hat{=}} M$.

(ii) Likewise, for any explicit store context $\nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle$ *there exists an implicit store-context* $S^{eag}[\,] = implicit(\sigma)$ *such that* $S^{eag}[\,] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [\,] \rangle$.

Proof: The proof consists of the obvious structural induction. The base cases for terms are the terminal syntactic classes; the induction for terms is by the rules in Figure 5.6, and that for contexts is by the rules in Figure 5.7. ▮

The next lemma assures that reductions in the explicit-store calculi act as an inverse to *implicit*, in that they actually build, step by step, a store corresponding to a store context.

Lemma 5.1.2 *For any term M and store context $S^{eag}[]$ in an explicit-store calculus there is an explicit store context $\nu_\sigma \vec{v}.\sigma \cdot \langle [] \rangle$ such that $S^{eag}[] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [] \rangle$ and such that*

$$\mathbf{pure}\, S^{eag}[M] \twoheadrightarrow^{*} \nu_\sigma \vec{v}.\sigma \cdot \langle M \rangle.$$

Proof: We can immediately apply a σ_{block}-reduction to the initial term, giving $\mathbf{pure}\, S^{eag}[M] \twoheadrightarrow \{\} \cdot \langle S^{eag}[M] \rangle$. A straightforward induction on the structure of $S^{eag}[]$, carrying out $\sigma{:=}$- and $\sigma\nu$-reductions as required, builds a store that corresponds to $S^{eag}[]$. ∎

Now we come to the central lemma in our proof that explicit-store computations simulate implicit-store reductions.

Lemma 5.1.3 *Given terms M' and N' in an explicit-store calculus, and term $M \mathrel{\hat{=}} M'$ in the corresponding implicit-store calculus, if $M' \twoheadrightarrow_\sigma N'$, then there is a term N such that $N \mathrel{\hat{=}} N'$ and $M \twoheadrightarrow N$.*

In a picture, there exists a term N that makes the following diagram commute:

$$\begin{tikzcd}
M' \arrow[r] \arrow[d, dash, "\hat{=}"'] & N' \arrow[d, dotted, dash, "\hat{=}"] \\
M \arrow[r, dotted] & N
\end{tikzcd}$$

Proof: We decompose the problem into one case for each possible reduction rule that might give $M' \twoheadrightarrow N'$. Since the relation $\hat{=}$ distributes through all term-forming operators in the explicit-store calculus, we can assume without loss of generality that the redex in $M' \twoheadrightarrow N'$ is the whole term M'.

If M' is a β-, δ-, *assoc-*, *extend-*, *unit-*, or *assign-result*-redex, the result follows immediately, since these rules are also reduction rules in the corresponding implicit-store calculus: we just take $M \equiv M'$, $N \equiv N'$.

The remaining cases are as follows. We use Lemma 5.1.1 freely without mention.

Case (1) $M' \equiv \nu_\sigma \vec{v}.\sigma \cdot \langle \nu v.P' \rangle$.

In this case, $M \equiv \mathbf{pure}\, S^{eag}[\nu v.P]$ for some eager store context $S^{eag}[]$ and term P such that $S^{eag}[] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma \cdot \langle [] \rangle$ and $P \mathrel{\hat{=}} P'$. The result of the lemma then follows from the diagram

$$\begin{tikzcd}
\nu_\sigma \vec{v}.\sigma \cdot \langle \nu v.P' \rangle \arrow[r] \arrow[d, dash, "\hat{=}"'] & \nu_\sigma \{\vec{v}, v\}.\sigma \cdot \langle P' \rangle \arrow[dl, dash, "\hat{=}"] \\
\mathbf{pure}\, S^{eag}[\nu v.P]. &
\end{tikzcd}$$

Case (2) $M' \equiv \nu_\sigma \vec{v}.\sigma \cdot \langle v := M_2'; P' \rangle$

In this case $M \equiv \mathbf{pure}\, S^{eag}[v := M_2;\ P]$ for some store context $S^{eag}[]$ and terms M_2 and P such that

$S^{eag}[] \hat{=} \nu_\sigma \vec{v}.\sigma \cdot \langle [] \rangle$, $M_2 \hat{=} M_2'$, and $P \hat{=} P'$. The result of the lemma then follows from the diagram

$$\begin{tikzcd}
\nu_\sigma \vec{v}.\sigma \cdot \langle v := M_2';\, P' \rangle \arrow[rr] \arrow[d, dash, "\hat{=}"'] & & \nu_\sigma \vec{v}.\sigma \,!\, v : M_2' \cdot \langle P' \rangle \arrow[dll, dash, "\hat{=}"] \\
\mathbf{pure}\, S^{eag}[v := M_2;\, P]. & &
\end{tikzcd}$$

Proving that the correspondence relation is re-established in this case follows from an induction on the structure of $S^{eag}[]$ in which the assignment of M_2 to v is the only one that affects the store. In the "before" case, the assignment is not treated as part of the store-context; in the "after" case it is treated as part of the store-context.

Case (3) $M' \equiv \nu_\sigma \vec{v}.\sigma \cdot \langle v? \triangleright x \cdot P' \rangle$

In this case $M \equiv \mathbf{pure}\, S^{eag}[v? \triangleright x \cdot P]$, where $S^{eag}[] \hat{=} \nu_\sigma \vec{v}.\sigma \cdot \langle [] \rangle$ and $P \hat{=} P'$. Since the reduction is assumed to take place, we have $v \in dom\,\sigma$. By the defining rules for $\hat{=}$ in Figure 5.7, this could only be the case if there exists an assignment to v in $S^{eag}[]$. Suppose the innermost such assignment assigns the term M_1 to v: the corresponding store must then bind M_1' to v, where $M_1 \hat{=} M_1'$.

The implicit-store term in question thus has the form

$$M \equiv \mathbf{pure}\, S_1^{eag}[v := M_1;\, S_2^{eag}[v? \triangleright x \cdot P]],$$

where there is no assignment to v in $S_2^{eag}[]$. An easy induction on the structure of $S_2^{eag}[]$ then shows that

$$\begin{array}{ll}
 & M \\
\to^* & \mathbf{pure}\, S_1^{eag}[v := M_1;\, v? \triangleright x \cdot S_2^{eag}[P]] \\
\to & \mathbf{pure}\, S_1^{eag}[v := M_1;\, [M_1/x]\, S_2^{eag}[P]] \\
\equiv & \mathbf{pure}\, S_1^{eag}[v := M_1;\, S_2^{eag}[[M_1/x]\, P]] \\
\equiv & \mathbf{pure}\, S^{eag}[[M_1/x]\, P],
\end{array}$$

where the transposition of the substitution is justified by the fact that all of $S_2^{eag}[]$ was originally outside the scope of the substitution variable x. The result of the lemma then follows from the diagram

$$\begin{tikzcd}
\nu_\sigma \vec{v}.\sigma \cdot \langle v? \triangleright x \cdot P' \rangle \arrow[r] \arrow[d, dash, "\hat{=}"'] & \nu_\sigma \vec{v}.\sigma \cdot \langle [M_1'/x]\, P' \rangle \arrow[d, dash, "\hat{=}"] \\
\mathbf{pure}\, S^{eag}[v? \triangleright x \cdot P]. \arrow[r, two heads] & \mathbf{pure}\, S^{eag}[[M_1/x]\, P]
\end{tikzcd}$$

We take it as obvious that substitution preserves correspondence: almost every rule in Figures 5.6 and 5.7 hypothesizes correspondence for subterms of the corresponding terms. The only exception is the rule for overridden assignments in store-contexts, but here a substituted term for the assigned value does not contribute to the corresponding store, so correspondence still holds.

Case (4) $M' \equiv \mathbf{pure}\, M_1'$

In this case $M \equiv \mathbf{pure} M_1$ for some $M_1 \mathrel{\hat{=}} M_1'$. The result of the lemma follows from the diagram

$$\begin{tikzcd}
\mathbf{pure} M_1' \arrow[r] \arrow[d, dash, "\hat{=}"'] & \nu_\sigma\{\}.\{\}\cdot\langle M_1'\rangle \arrow[dl, dash, "\hat{=}"] \\
\mathbf{pure} M_1 .
\end{tikzcd}$$

Case (5) ($\lambda[\beta\delta\sigma eag]$ only) $M' \equiv \nu_\sigma \vec{v}.\sigma\cdot\langle\uparrow(\lambda x.M_1')\rangle$

We take this case as representative of the purification reductions; the proofs in the other cases are similar. In this case $M \equiv \mathbf{pure} S^{eag}[\uparrow(\lambda x.M_1)]$, where $S^{eag}[] \mathrel{\hat{=}} \vec{v}\cdot\langle\sigma\rangle[]$ and $M_1 \mathrel{\hat{=}} M_1'$. The result of the lemma then follows from the diagram

$$\begin{tikzcd}
\nu_\sigma \vec{v}.\sigma\cdot\langle\uparrow(\lambda x.M_1')\rangle \arrow[r] \arrow[d, dash, "\hat{=}"'] & \lambda x.\nu_\sigma \vec{v}.\sigma\cdot\langle\uparrow M_1'\rangle \arrow[d, dash, "\hat{=}"] \\
\mathbf{pure} S^{eag}[\uparrow(\lambda x.M_1)] \arrow[r] & \lambda x.\mathbf{pure}\, S^{eag}[\uparrow M_1].
\end{tikzcd}$$

Case (6) ($\lambda[\beta\delta\sigma laz]$ only) $M' \equiv \nu_\sigma \vec{v}.\sigma\cdot\langle S^{laz}[\uparrow(\lambda x.M_1')]\rangle$

Again, we take this case as representative of the purification reductions. The proofs in the other cases are similar. In this case $M \equiv \mathbf{pure} S^{eag}[S_1^{laz}[\uparrow(\lambda x.M_1)]]$, where $S^{eag}[] \mathrel{\hat{=}} \nu_\sigma \vec{v}.\sigma\cdot\langle[]\rangle$, and

$$S_1^{laz}[\uparrow(\lambda x.M_1)] \mathrel{\hat{=}} S_1^{laz}[\uparrow(\lambda x.M_1')].$$

Noting that every eager store-context is also a lazy store-context, and that the nesting of two lazy stor- contexts is again a lazy store context, we have the diagram

$$\begin{tikzcd}
\nu_\sigma \vec{v}.\sigma\cdot\langle S^{laz}[\uparrow(\lambda x.M_1')]\rangle \arrow[r] \arrow[d, dash, "\hat{=}"'] & \lambda x.(\nu_\sigma \vec{v}.\sigma\cdot\langle S^{laz}[\uparrow M_1']\rangle) \arrow[d, dash, "\hat{=}"] \\
\mathbf{pure} S^{eag}[S_1^{laz}[\uparrow(\lambda x.M_1)]] \arrow[r] & \lambda x.(\mathbf{pure}\, S^{eag}[S_1^{laz}[\uparrow M_1]]).
\end{tikzcd}$$

This concludes the proof of Lemma 5.1.3. ∎

The maintenance of correspondence under reduction established by Lemma 5.1.3 forms the core of the main simulation theorem (Theorem 5.1.7), but the full theorem requires some additional lemmas concerning the action of imperative reductions in the implicit-store calculi on the form of store contexts. We now begin a series of results intended to characterize the behavior of reductions in command sequences in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

Our first lemma notes that much of the structure of command sequences is preserved "up to bubbling" by reductions in the implicit-store calculi.

Definition 5.1.4 *A* bubble context *is a context defined by the productions*

$$B[] ::= \;\; [] \;\;|\;\; v?\triangleright x{\cdot}B[].$$

Lemma 5.1.5 (Persistence of store-context) *Assume that* $\lambda[\beta\delta!eag] \vdash M \twoheadrightarrow^* M'$ *or* $\lambda[\beta\delta!laz] \vdash M \twoheadrightarrow^* M'$. *Then:*

(i) If $M \equiv \nu v\, N$ *then there is a term* N' *and a bubble context* $B[]$ *such that* $M' \equiv B[\nu v\, N']$.

(ii) If $M \equiv v := N;\, P$ *then there are terms* N' *and* P' *and a bubble context* $B[]$ *such that* $M' \equiv B[v := N';\, P']$.

(iii) If $M \equiv \uparrow N$ *then there is a term* N' *such that* $M' \equiv \uparrow N'$.

(iv) If $M \equiv v?\triangleright x{\cdot}P$ *then there is a term* P' *such that* $M' \equiv v?\triangleright x{\cdot}P'$.

Proof: By inspection of the reduction rules for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, it is clear that only the rules *pure-eager* and *pure-lazy* can remove a ν or := from a term. Furthermore, there is no rule that can reduce a term prefixed with one of these constructs to one that has it only within the context of a **pure**. The only remaining reduction rules that explicitly involve terms of the form in (i) and (ii) are the rules *bubble-assign* and *bubble-new*, which clearly act to embed the former top-level constructs within a bubble context. Similar reasoning establishes (iii) and (iv). ∎

The next result shows that, if reduction of a command-redex is essential in a computation that produces an answer, then that redex must occur in the context of a **pure**-expression. The informal notion of "essential" is represented by being a head redex. The proof of Lemma 5.1.6 argues in several places that a stuck reduction in an evaluation context prevents a term from reducing to an answer. In this form of reasoning, getting stuck means never reducing either to part of an answer or to a subterm that makes some containing term a redex. This clearly prevents the standard reduction from progressing to an answer, and hence the whole term could not have reduced to an answer in the first place.

Lemma 5.1.6 (Only pure purifies) *Let* M *be a term in* $\lambda[\beta\delta!eag]$ *or* $\lambda[\beta\delta!laz]$, *and assume that* $M \equiv E[\Delta_h] \twoheadrightarrow^* A$, *where* Δ_h *is a fuse-, bubble-assign-, or bubble-new-redex that is head redex in* M.

Then there is an evaluation context $E_1[]$ *and an eager store context* $S^{eag}[]$ *such that* $M \equiv E_1[$**pure** $S^{eag}[\Delta_h]]$.

Proof: Assume the contrary, that is, that M is not of the required form. We consider each remaining possible structure of the term M (as determined by the structure of the evaluation context $E[]$) and show, as a contradiction, that these structures cannot reduce to an answer.

Case (1) If $E[] \equiv S^{eag}[]$ for some store-context $S^{eag}[]$, then Lemma 5.1.5 applies to show that $M \not\twoheadrightarrow^* A$, and we have an immediate contradiction.

Case (2) The alternative is that the structure of $E[]$ is interrupted by some nested evaluation context that is not also an eager store-context. We introduce names for the contexts implied by this statment by asserting that $E[] \equiv E_1[E_2[S^{eag}[]]]$, where $E_1[]$ is an evaluation context, $E_2[]$ is a non-empty evaluation context that is not an unrestricted store context, and $S^{eag}[]$ is an eager store-context. Without loss of generality, we can assume that $E_1[] \equiv []$ and that $E_2[]$ is minimal, that is, that $E_2[]$ can be obtained by exactly one application of a production in Figures 3.2, 3.3, and 3.4. We subdivide this case according to the form of $E_2[]$.

Case (2a) $E_2[\,] \equiv [\,] \bullet N$.
Assume $S^{eag}[\Delta_h] \twoheadrightarrow^* A$. By Theorem 3.7.10, we can choose the reduction sequence to reduce Δ_h first. By Lemma 5.1.5, the reduction can only yield a term of the form $B[S_1^{eag}[N_1]]$, where N_1 cannot be an abstraction. Such a residual term can only interact with its context $S^{eag}[\,]$ by means of one of the rules *bubble-assign* or *bubble-new*; hence, there is no way to reduce $S^{eag}[\Delta_h]$ to an abstraction or to a primitive-function name. There can thus be no way to reduce the application to an answer; this is the desired contradiction.

Case (2b) $E_2[\,] \equiv f \bullet [\,]$.
The argument for this case is similar to that for Case (2a), noting instead that $S^{eag}[\Delta_h]$ can reduce neither to an abstraction nor to a constructed value.

Case (2c) $E_2[\,] \equiv [\,]?$, $E_2[\,] \equiv [\,] := N$, $E_2[\,] \equiv \textbf{pure}\, S^{eag}[\uparrow [\,]]$.
These cases are all similar to Case (2a).

Case (2d) $E_2[\,] \equiv [\,] \rhd x \cdot N$.
In this case $E_2[S^{eag}[\Delta_h]]$ is a redex, contradicting the assumption that Δ_h is the head redex (outermost evaluation redex).

Case (2e) $E_2[\,] \equiv N_1 \rhd x \cdot [\,]$.
For this case to apply, N_1 cannot have the form $E_3[\Delta]$ for any evaluation context $E_3[\,]$ and redex Δ, because this would make Δ, not Δ_h, the head redex; in other words, N_1 must be in head normal form. Also, $N_1 \rhd x \cdot \Delta_h$ itself cannot be a redex for the same reason. Thus N_1 cannot have any of the forms $N_2 \rhd y \cdot N_3$, $\uparrow N_2$, or $\nu v. N_2$. Furthermore, if N_1 has the form $N_2 := N_3$ then N_2 is not a store-variable, because then $E_2[\,]$ would be an eager store-context, which it was assumed not to be. Now by Lemma 5.1.5, $\Delta_h \twoheadrightarrow^* B[S^{eag}[N_1']]$, so a head-reduction sequence will bubble the store-variable-reads that are outermost in $B[\,]$ to the left, where they cannot interact with any of the remaining possible forms for N_1. Hence the entire term cannot reduce to an answer.

Case (2f) $E_2[\,] \equiv \nu v.[\,]$, $E_2[\,] \equiv v := N \rhd x \cdot [\,]$, $E_2[\,] \equiv \uparrow [\,]$.
These cases all contradict the assumption that $E_2[\,]$ is not an eager store context.

Case (2g) (in $\lambda[\beta\delta!eag]$) $E_2[\,] \equiv \textbf{pure}\, S_1^{eag}[\uparrow\ ]$.
In this case $M \equiv \textbf{pure}\, S_1^{eag}[S^{eag}[\uparrow \Delta_h]]$.
By Lemma 5.1.5 or by inspection of the set of reduction rules the residuals of Δ_h are syntactically all command-sequences, having $\rhd$ as their top-level syntactic constructor. There is thus no way for these residuals to interact with their containing $\uparrow$ construct, and hence it is impossible for M to reduce to an answer, a contradiction.

Case (2h) (in $\lambda[\beta\delta!laz]$) $E_2[\,] \equiv \textbf{pure}\, S_1^{laz}[\,]$.
In this case $M \equiv \textbf{pure}\, S_1^{laz}[S^{eag}[\Delta_h]]$, and we proceed by the same reasoning as that used for the corresponding case in $\lambda[\beta\delta!eag]$.

The only remaining case is $E_2[\,] \equiv \textbf{pure}[\,]$, hence we must have

$$M \equiv E_1[E_2[S^{eag}[\Delta_h]]] \equiv E_1[\textbf{pure}\, S^{eag}[\Delta_h]].$$

But this proves Lemma 5.1.6. ∎

We are now prepared to state and prove the main theorem on the mutual simulation between $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ on the one hand and $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ on the other. The notion of operational semantics we use in this proof is of reduction to a constant c^0. It is possible to extend this semantics to compound answers at the meta-level by testing several programs, each of which is the application of a composition of selection primitives to the original program. We restrict the form of answer considered because standard reduction sequences to constants have a particularly simple form: they are always head reduction sequences (by Theorem 3.7.12).

Theorem 5.1.7 *For every closed term M and constant c^0,*

(i) $\lambda[\beta\delta!eag] \vdash M = c^0$ *if and only if* $\lambda[\beta\delta\sigma eag] \vdash M = c^0$.

(ii) $\lambda[\beta\delta!laz] \vdash M = c^0$ *if and only if* $\lambda[\beta\delta\sigma laz] \vdash M = c^0$.

Proof: We first prove the "only if" direction of the implications.

Assume that $\lambda[\beta\delta!eag] \vdash M = c^0$. By Theorem 3.7.12 there exists a head reduction sequence reducing M to c^0. We now carry out an induction on the length of this head reduction sequence.

The base case of the induction is a reduction sequence of length zero, in which case $M \equiv c^0$ and there is nothing to show.

For the induction step, assume that the statement of the theorem holds for head reduction sequences of length n or less, and consider M such that $M \twoheadrightarrow^* c^0$ by head reductions in $n+1$ steps (in the implicit-store calculus). Then the head reduction sequence from M has the form

$$M \equiv E[\Delta] \to E[L] \equiv M' \twoheadrightarrow^* c^0$$

for some evaluation context $E[]$, head redex Δ, and terms L and M'. We now show that $\lambda[\beta\delta\sigma eag] \vdash M = M'$ by a case analysis on the form of Δ.

Case 1: If Δ is a β-, δ-, *assoc-*, *extend-*, *assign-result-*, or *unit*-redex, the result is immediate because these reduction rules apply in $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ as well.

Case 2: $\Delta \equiv v := N;\ v? \triangleright x \cdot P$

By Lemma 5.1.6 there is an evaluation context $E[]$ and an eager store context $S^{eag}[]$ such that

$$M \equiv E[\mathbf{pure}\, S^{eag}[v := N;\ v? \triangleright x \cdot P]].$$

By Lemma 5.1.2, $M \twoheadrightarrow^* M_\sigma$ in the explicit-store calculus, where

$$M_\sigma \equiv E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle v := N;\ v? \triangleright x \cdot P\rangle],$$

where $\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle [] \rangle$ is the explicit store context whose existence is guaranteed by Lemma 5.1.1. Then we have the following derivation that $\lambda[\beta\delta\sigma eag] \vdash M_\sigma = M'$, which in turn implies $\lambda[\beta\delta\sigma eag] \vdash M = c^0$ via the transitivity of conversion.

$$\begin{array}{rcll}
M_\sigma & \equiv & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle v := N;\ v? \triangleright x \cdot P\rangle] & \\
& \to & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]}\, !\, v : N \cdot \langle v? \triangleright x \cdot P\rangle] & (\sigma{:=}) \\
& \to & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]}\, !\, v : N \cdot \langle [N/x]\, P\rangle] & (\sigma?) \\
& \leftarrow^* & E[\mathbf{pure}\, S^{eag}[v := N;\ [N/x]\, P]] & (\textit{Lemma 5.1.2}, \sigma{:=}) \\
& \equiv & M' &
\end{array}$$

This derivation works equally well for $\lambda[\beta\delta!laz]$ being simulated by $\lambda[\beta\delta\sigma laz]$.

Case 3: $\Delta \equiv v := N;\ w? \triangleright x \cdot P$

Reasoning as in the previous case, we have

$$M \equiv E[\mathbf{pure}\, S^{eag}[v := N;\ w? \triangleright x \cdot P]] \twoheadrightarrow M_\sigma,$$

where

$$M_\sigma \equiv E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle v := N;\ w? \triangleright x \cdot P\rangle].$$

We now have the following derivation showing that $\lambda[\beta\delta\sigma eag] \vdash M_\sigma = M'$. We are assured that $S^{eag}[]$ contains an assignment to w because we know that the store-context eventually reduces away; whereas a read of an unassigned variable w bubbles leftward to meet either the **pure** or a declaration of w. It must do so via head reductions because we know that the spine of a store-context with a **pure** within an evaluation context is itself within an evaluation context.

$$\begin{array}{lll}
M_\sigma & \equiv & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle v{:=}N;\ w?\triangleright x\cdot P\rangle] & \\
& \rightarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]}\ !\,v : N \cdot \langle w?\triangleright x\cdot P\rangle] & (\sigma{:=}) \\
& \rightarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]}\ !\,v : N \cdot \langle [\sigma' \downarrow w/x]\,P\rangle] & (\sigma?) \\
& \leftarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle v{:=}N;\ [\sigma' \downarrow w/x]\,P\rangle] & (\sigma{:=}) \\
& \equiv & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle [\sigma' \downarrow w/x]\,(v{:=}N;\ P)\rangle] & (\textit{Note: } x \notin fv\,N) \\
& \leftarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle w?\triangleright x\cdot v{:=}N;\ P\rangle] & (\sigma?) \\
& \leftarrow^* & E[\mathbf{pure}\,S^{eag}[w?\triangleright x\cdot v{:=}N;\ P]] & (\textit{Lemma 5.1.2}) \\
& \equiv & M' &
\end{array}$$

Case 4: $\Delta \equiv \nu v.w?\triangleright x\cdot P, v \neq w$

Once again we invoke Lemma 5.1.6 to place our redex Δ in context as

$$M \equiv E[\mathbf{pure}\,S^{eag}[\nu v.w?\triangleright x\cdot P]].$$

Lemma 5.1.2 allows us to begin the following derivation of the desired result. Our assurance that w is bound in the store rests here on the same ground as in the previous case.

$$\begin{array}{lll}
M & \rightarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle \nu v.w?\triangleright x\cdot P\rangle] & (\textit{Lemma 5.1.2}) \\
& \rightarrow & E[\nu_\sigma \{\vec{v},v\}.\sigma_{S^{eag}[]} \cdot \langle w?\triangleright x\cdot P\rangle] & (\sigma\nu) \\
& \rightarrow & E[\nu_\sigma \{\vec{v},v\}.\sigma_{S^{eag}[]} \cdot \langle [\sigma_{S^{eag}[]} \downarrow w/x]\,P\rangle] & (\sigma?) \\
& \leftarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle \nu v.[\sigma_{S^{eag}[]} \downarrow w/x]\,P\rangle] & (\sigma\nu) \\
& \equiv & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle [\sigma_{S^{eag}[]} \downarrow w/x]\,\nu v.P\rangle] & \\
& \leftarrow & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle w?\triangleright x\cdot \nu v.P\rangle] & (\sigma?) \\
& \leftarrow^* & E[\mathbf{pure}\,S^{eag}[w?\triangleright x\cdot \nu v.P]] & \textit{Lemma 5.1.2} \\
& \equiv & M'. &
\end{array}$$

Case 5: ($\lambda[\beta\delta!eag]/\lambda[\beta\delta\sigma eag]$ only) $\Delta \equiv \mathbf{pure} S^{eag}[\uparrow(\lambda x.M)]$.

Again using Lemma 5.1.2, we have:

$$\begin{array}{lll}
M & \equiv & E[\mathbf{pure}\,S^{eag}[\uparrow(\lambda x.M)]] & \\
& \rightarrow^* & E[\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle \uparrow(\lambda x.M)\rangle] & (\textit{Lemma 5.1.2}) \\
& \rightarrow & E[\lambda x.\nu_\sigma \vec{v}.\sigma_{S^{eag}[]} \cdot \langle \uparrow M\rangle] & (\sigma_{block}) \\
& \leftarrow^* & E[\lambda x.\mathbf{pure}\,S^{eag}[\uparrow M]] & (\textit{Lemma 5.1.2}) \\
& \equiv & M'. &
\end{array}$$

The cases of the other *pure-eager*-reduction rules behave similarly.

Case 6: ($\lambda[\beta\delta!laz]/\lambda[\beta\delta\sigma laz]/$ only) $\Delta \equiv \mathbf{pure} S^{laz}[\uparrow(\lambda x.M)]$.

By the same general form of reasoning as in the other cases, we have

$$\begin{array}{lll}
M & \equiv & E[\mathbf{pure}\,S^{laz}[\uparrow(\lambda x.M)]] & \\
& \rightarrow & E[\nu_\sigma\{\}.\{\} \cdot \langle S^{laz}[\uparrow(\lambda x.M)]\rangle] & (\sigma_{block}) \\
E[\lambda x.\nu_\sigma\{\}.\{\} \cdot \langle S^{laz}[\uparrow M]\rangle] & (\sigma_{plaz}) & & \\
& \leftarrow & E[\lambda x.\mathbf{pure}\,S^{laz}[\uparrow M]] & (\sigma_{block}) \\
& \equiv & M'. &
\end{array}$$

This concludes the proof that $\lambda[\beta\delta!eag] \vdash M = c^0$ implies $\lambda[\beta\delta\sigma eag] \vdash M = c^0$.

We establish the converse by applying Lemma 5.1.3. Assume that $\lambda[\beta\delta!eag'] \vdash M = c^0$. By the definition of $(\hat{=})$, $M \hat{=} M$; furthermore, $c^0 \hat{=} c'^0$ implies $c^0 \equiv c'^0$. The theorem can then be established by pasting together diagrams of the form provided by Lemma 5.1.3 as follows:

$$\begin{tikzcd}
M \arrow[r, "\sigma"'] \arrow[d, dash, "\hat{=}"'] & M'_1 \cdots\cdots \arrow[r, "\sigma"'] \arrow[d, dash, "\hat{=}"] & \cdots\cdots c^0 \arrow[d, dash, "\hat{=}"] \\
M \arrow[r] & M_1 \cdots\cdots \arrow[r] & \cdots\cdots c^0
\end{tikzcd}$$

This concludes the proof of Theorem 5.1.7. ∎

5.2 Calculi with assignments as conservative extensions of lambda-calculi

Every term and reduction in the calculus $\lambda[\beta\delta]$ is also a term or reduction of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. Hence any convertibility relation that holds in $\lambda[\beta\delta]$ must also hold in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. We recall from Chapter 4, however, that the operational-equivalence theory of a lambda-calculus is really the interesting theory for reasoning about programs. The definition of operational equivalence involves a universal quantification over *all* contexts in a calculus. When we embed a term from $\lambda[\beta\delta]$ into a calculus with assignments, some of these contexts will contain constructs that are not present in the original $\lambda[\beta\delta]$. We could thus conceive of a situation in which the operational-equivalence theory of the imperative calculus either equates or distinguishes $\lambda[\beta\delta]$-terms that are not equated or distinguished by the operational-equivalence theory of $\lambda[\beta\delta]$.

In this section, we show that the calculi $\lambda[\beta\delta!eag]$ is, in fact, a *conservative* extension of $\lambda[\beta\delta]$: it preserves the operational-equivalence theory of $\lambda[\beta\delta]$ exactly. The strategy we employ is to give meaning-preserving translations in both directions between the language of $\lambda[\beta\delta\sigma eag]$ and the language of $\lambda[\beta\delta]$ with specially-designed primitives that carry out a simulation of the imperative constructs. The preservation of meaning, along with the definability of the constructs in $\lambda[\beta\delta]$, is sufficient to establish conservative extension.

Before we prove this result we must be more precise about what we actually prove. Because we have left the constructors and primitive functions unspecified, every one of the "calculi" proposed in this dissertation is actually an entire family of calculi. Up to this point, we have not belabored this distinction because all our results have held uniformly for all calculi in each family. We must now break this pattern in order to be precise about our conservative-extension results, since it is not the case that *every* calculus with assignment is an extension of *every* member of the family $\lambda[\beta\delta]$. Instead, for each calculus with assignment we will show that there *exists* a member of the family $\lambda[\beta\delta]$ for which the original calculus is a conservative extension. The proof of conservative extension will use this freedom to construct a basic lambda-calculus in an essential way: the constructs implementing the translation from the calculi with assignments will be new primitives within a basic lambda-calculus, and we will construct inverse translations that recognize these primitives as such.

We actually carry out the proof of conservative extension using the explicit-store calculus $\lambda[\beta\delta\sigma eag]$; we then use the results of Section 5.1 to conclude that $\lambda[\beta\delta!eag]$ itself is a conservative extensions of $\lambda[\beta\delta]$ in the sense in which we use the term here. The proof that $\lambda[\beta\delta\sigma eag]$ is a conservative extension of $\lambda[\beta\delta]$ is itself mediated by a translation into a calculus $\lambda\nu$ devised by Odersky [Odersky, 1993b; Odersky, 1994]; Odersky has shown $\lambda\nu$ to be a conservative extension of $\lambda[\beta\delta]$. Section 5.2.1 introduces the technical notion of syntactic embedding which forms the backbone of the proof. Section 5.2.2 introduces the relevant features of the calculus $\lambda\nu$, Section 5.2.3 details the translation from $\lambda[\beta\delta\sigma eag]$ into $\lambda\nu$, Section 5.2.4 gives an inverse translation that is needed to show the existence of a syntactic embedding, and Section 5.2.5 joins the pieces together to give a proof of conservative extension. Section 5.2.5 comments on the difficulty of carrying out this process for $\lambda[\beta\delta\sigma laz]$.

5.2.1 Syntactic embedding

The precise statement and proof of our conservative-extension results requires some preliminary definitions. We devote this subsection to these definitions and to an outline of the general proof method to be employed. The focal point for the proofs in this section is the property of *syntactic embedding*, which implies conservative extension but is specialized to be more easily proved with the methods we have at hand.

Definition 5.2.1 (Syntactic embedding) *Let λ_* and λ_0 be extensions of $\lambda[\beta\delta]$ with the same set of answer terms. Suppose that $\Lambda(\lambda_*) \supseteq \Lambda(\lambda_0)$. Let $\mathcal{E}$ be a syntactic mapping from λ_*-terms to λ_0-terms. Then $\mathcal{E}$ is a* syntactic embedding *of λ_* into λ_0 if it satisfies the following two properties:*

(i) $\mathcal{E}$ preserves λ-closed λ_0-subterms up to convertibility, that is, for all λ_*-contexts $C[\,]$ and λ-closed λ_0-terms M,

$$\lambda_0 \vdash \mathcal{E}[\![C[M]]\!] = \mathcal{E}[\![C]\!][M].$$

(ii) $\mathcal{E}$ preserves the conversion semantics, that is, for all closed λ_*-terms M and answers A,

$$\lambda_* \vdash M = A \textit{ if and only if } \lambda_0 \vdash \mathcal{E}[\![M]\!] = A.$$

The notation $\mathcal{E}[\![C]\!][M]$ used in Definition 5.2.1 is the obvious homomorphic extension of the mapping to terms to a mapping on contexts by specifying that $\mathcal{E}[\![[\,]]\!] = [\,]$.

Syntactic embeddings generalize the syntactic mappings introduced in [Felleisen, 1991] in a discussion of the expressive power of programming languages. Felleisen's mappings are essentially macro-expansions; the mappings allowed by Definition 5.2.1 are somewhat more general. The importance of Definition 5.2.1 for our purposes here is that syntactic embeddings can be used to prove conservative extension:

Theorem 5.2.2 *Let λ_* and λ_0 be two extensions of $\lambda[\beta\delta]$ having the same sets of answer terms, and suppose that $\Lambda(\lambda_*) \supseteq \Lambda(\lambda_0)$. If there exists a syntactic embedding of λ_* into λ_0 then for any two terms M, N in $\Lambda(\lambda_0)$,*

$$\lambda_0 \models M \cong N \textit{ implies } \lambda_* \models M \cong N.$$

Proof: Note: in the statement of the theorem, the syntactic embedding is not notated because every λ_0-term is also a λ_*-term.

Assume that $\lambda_0 \models M \cong N$. Then, for all answers A and λ_0-contexts $C_0[\,]$ such that $C_0[M]$ and $C_0[N]$ are closed,

$$\lambda_0 \vdash C_0[M] = A \text{ if and only if } \lambda_0 \vdash C_0[N] = A.$$

Assume first that both M and N are λ-closed. Let $\mathcal{E}$ be a syntactic embedding of λ_* into λ_0. Let $C[\,]$ be an arbitrary λ_*-context such that $C[M]$ and $C[N]$ are closed, and let A be an answer such that $\lambda_* \vdash C[M] = A$. Then we have the following chain of logical equivalences:

$$\begin{array}{lll}
 & \lambda_* \vdash C[M] = A & \\
\Leftrightarrow & \lambda_0 \vdash \mathcal{E}[\![C[M]]\!] = A & (\mathcal{E} \text{ preserves semantics}) \\
\Leftrightarrow & \lambda_0 \vdash \mathcal{E}[\![C]\!][M] = A & (\mathcal{E} \text{ preserves } \lambda\text{-closed } \lambda_0\text{-subterms, } = \text{ is transitive}) \\
\Leftrightarrow & \lambda_0 \vdash \mathcal{E}[\![C]\!][N] = A & (\text{by the premise that } \lambda_0 \models M \cong N) \\
\Leftrightarrow & \lambda_* \vdash C[N] = A & (\text{reversing the argument with } N \text{ replacing } M).
\end{array}$$

Since A and C were arbitrary, we can conclude that $\lambda_* \models M \cong N$.

We now extend the reasoning from closed terms to arbitrary terms by reasoning about their closures. Let M and N be arbitrary λ_0-terms, with $fv\, M \cup fv\, N = \{x_1, \ldots, x_n\}$. Then we have the following chain of reasoning:

$$\begin{array}{lll}
 & \lambda_0 \models M \cong N & \\
\Leftrightarrow & \lambda_0 \models \lambda x_1 \ldots . \lambda x_n.M \cong \lambda x_1 \ldots . \lambda x_n.N & (\text{by Lemma 4.1.4}) \\
\Rightarrow & \lambda_* \models \lambda x_1 \ldots . \lambda x_n.M \cong \lambda x_1 \ldots . \lambda x_n.N & (\text{by the first part of the proof}) \\
\Leftrightarrow & \lambda_* \models M \cong N & (\text{by Lemma 4.1.4}).
\end{array}$$

This concludes the proof of Theorem 5.2.2 ∎

$$v \in \textit{Local names}$$

M	$::=$	$\cdots$ (Figure 2.2) $\cdots$	
	$\mid$	$\nu v.M$	*local name v over scope M*
	$\mid$	$M_1 == M_2$	*test of name equality*

Figure 5.8: Syntax of the local-name calculus $\lambda\nu$

		$\cdots$ (Figure 2.3) $\cdots$		
$\nu n.c^m \bullet M_1 \bullet \cdots \bullet M_k$	$\rightarrow$	$c^m \bullet (\nu n.M_1) \bullet \cdots \bullet (\nu n.M_k).$		$\lambda\nu c^n$
$\nu n\,\lambda x.M$	$\rightarrow$	$\lambda x.\nu n.M.$		$\lambda\nu\lambda$
$v == v$	$\rightarrow$	*true*		$\lambda\nu ==$
$v == w$	$\rightarrow$	*false*	$(v \not\equiv w)$	$\lambda\nu ==$

Figure 5.9: Reduction rules for the local-name calculus $\lambda\nu$

5.2.2 The calculus $\lambda\nu$

Before we prove the conservative-extension theorem for the calculi of concern, we quickly state why finding a syntactic embedding of those calculi into $\lambda[\beta\delta]$ is nontrivial. A natural implementation of a store calculus in terms of $\lambda[\beta\delta]$ maps assignments and reads to read and write operations on an actual store. This suggests using the explicit-store calculi defined earlier in this chapter as an intermediate step in the implementation. Because all operational equivalences of $\lambda[\beta\delta!\mathit{eag}]$ and $\lambda[\beta\delta!\mathit{laz}]$ are preserved in the corresponding explicit-store calculi, proving our conservative-extension results for the explicit-store calculi will immediately establish the results for the implicit-store calculi as well.

With this scheme in mind, a store from an explicit-store calculus can be represented in $\lambda[\beta\delta]$ as a mapping from store-variables to terms. Unfortunately, store-variables have no obvious corresponding entity in $\lambda[\beta\delta]$: they are pure names with scope and a test for equality, but they are not locuses of substitution as are the normal λ-variables. We can try to encode the allocation of names by some technique from pure functional programming, such as passing around a supply of names, but then we cannot fulfill the part of the definition of syntactic embedding that demands that closed programs in the target calculus as found in the extended calculus map to themselves—name-supply passing is a pervasive transformation (as is continuation-passing).

The papers [Odersky, 1993b; Odersky, 1994] define a calculus $\lambda\nu$ which specifically solves this problem. Rather than attempting any implementation of locations as some more primitive entity, Odersky's calculus provides a direct axiomatization of the important properties of distinct names defined over scopes. The calculus $\lambda\nu$ is itself a conservative extension of $\lambda[\beta\delta]$, so it provides another suitable intermediate calculus for the overall proof that the calculi of concern conservatively extend $\lambda[\beta\delta]$. Odersky's proof that $\lambda\nu$ forms a conservative extension of $\lambda[\beta\delta]$ constructs a syntactic embedding from $\lambda\nu$ to $\lambda[\beta\delta]$ that encodes a de Bruijn-like level-numbering scheme for the name bindings into the target lambda calculus. The technique of translating $\lambda\nu$-terms into a name-supply-passing style is ruled out by the requirement that a syntactic embedding must preserve those $\lambda[\beta\delta]$-terms that are present in $\lambda\nu$.

Basically, $\lambda\nu$ adds only one feature to $\lambda[\beta\delta]$: the name-introduction construct $\nu n.M$, along with a primitive $==$ for detecting the equality of names. To the reader of this dissertation, therefore, $\lambda\nu$ may just seem like a stripped-down version of our calculi with assignment. The reduction rules of $\lambda\nu$ are all the rules of $\lambda[\beta\delta]$ (given in Figure 2.3) augmented as given in Figure 5.9. The calculus $\lambda\nu$ does not require any construct analogous to **pure** to demarcate subterms over which local names are in use—the entire term acts as the unit of 'purification' when we are concerned to get rid of local names.

Rule $\lambda\nu c^n$ allows the scope of a local name to be pushed down through a constructed value; the splitting of the scope is rationalized in much the same fashion as the splitting of the store-context in the purification rules of $\lambda[\beta\delta!\mathit{eag}]$ and $\lambda[\beta\delta!\mathit{laz}]$. Rule $\lambda\nu\lambda$ allows the scope of a local name to be pushed down through a lambda-abstraction. This allows the lambda-abstraction to be exposed, perhaps to participate in a β-reduction that places the argument of the lambda inside of the name's scope: this is a characteristic mode of computation in

$\lambda\nu$. As usual, the bound names ν are α-renamable, and we observe Barendregt's convention for the avoidance of name-capture issues.

Rule $\lambda\nu$ $==$ behaves similarly to its counterpart in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

5.2.3 Translating $\lambda[\beta\delta\sigma eag]$ into $\lambda\nu$

Throughout the preceding parts of this dissertation we have been able to treat calculi for eager and lazy stores in parallel, since it has turned out that the differences between them are rather minor. We now come to the parting of the ways: the translations we are about to define work very differently in the two cases. The situation is analogous to the treatment of the call-by-name and call-by-value lambda-calculi: the calculi only differ in a syntactic condition in their β-rules, but when we come to translate them into continuation-passing style (as in [Fischer, 1972; Plotkin, 1975; Fischer, 1993]) the translations have significantly different structure.

We now give an implementation of $\lambda[\beta\delta\sigma eag]$ in terms of a particular member of the $\lambda\nu$ family. We require the target calculus to have certain constructors and primitive functions that make our implementation straightforward. Essentially, these constructs allow us to imitate sum-of-products data types by testing for unique tags which we know not to be used for any other purpose in the target calculus.

The main thrust of the implementation translation given in Figures 5.10 and 5.11 is to define a syntactic embedding $\mathcal{F}$ from $\lambda[\beta\delta\sigma eag]$ to $\lambda\nu$. The first requirement to be satisfied is to preserve the $\lambda\nu$ terms that happen to be present in the larger language $\lambda[\beta\delta\sigma eag]$. It is easy to see that $\mathcal{F}$ satisfies this requirement. The nontrivial portion of the translation is the implementation of the store constructs by selected new constructs in the target calculus. The essence of this translation is to give a concrete functional implementation along the outlines of the state-transformers delineated in [Wadler, 1992a] (among other places).

The mapping $\mathcal{F}$ defined in Figure 5.10 is defined in terms of the auxiliary definitions given in Figure 5.11. The auxiliary functions themselves are defined as primitive functions in $\lambda\nu$. In these definitions we use a pattern-matching notation (based on the syntax of functional programming languages) in the interest of conciseness, but the expanded definitions should be clear enough. These auxiliary definitions are essentially the same as the definitions of state-transformers one would write in a functional programming language. It is in these definitions that the ability to freely coin constructors and primitives in the target calculus comes into play.

State-transformers are represented in continuation-passing, store-passing style. A state-transformer is a function accepting a continuation and a store and producing a result; a continuation is a function accepting an intermediate result and a store and producing a final result. This basic structure of the definitions given in Figure 5.11 reflects this simple conception; however, there is additional administrative structure dictated by considerations which we will treat shortly.

Stores themselves are represented in classic denotational-semantics style as functions from names (store-variables) to their bindings. The $\lambda\nu$ primitive *upd* serves to construct new stores on this principle. We abuse notation to give the freshly-devised constructor for representing the empty store in $\lambda\nu$ the same notation $\{\}$ that we use for the empty store in $\lambda[\beta\delta\sigma eag]$.

Variables and store-variables in $\lambda[\beta\delta\sigma eag]$ are mapped to variables and names in $\lambda\nu$ having the same name.[1]

Some of the complexity of the translation arises from the need to maintain a semantic distinction between results of store computations and other values in the target calculus $\lambda\nu$. Such results are wrapped in a distinct constructor *Res*; the constructor *Unit* represents the result of an assignment command. Without this wrapping, a term such as **pure**M, where M is a functional term in $\lambda\nu$ that happens to behave like a state-transformer, would be translated to $pure \bullet M$, which might well reduce to a result. However, the original term **pure**M will always get stuck in reduction, since M is not a command. Hence, the naive translation scheme would fail to preserve operational semantics, and would thus fail to be a syntactic embedding. In Figure 5.11, the function *strip* is introduced to coerce wrapped results back to ordinary terms for passing to other functions. This entire complication essentially arises as an attempt to encode an abstract data type in an untyped calculus.

[1] In [Odersky and Rabin, 1993] the specification of the translation keeps track explicitly of the mapping between corresponding names in the two calculi. Although this level of attention is required for the strictest accuracy, it encumbers the technical development, and so is suppressed here.

$$
\begin{array}{rcl}
\mathcal{F}[\![f]\!] & = & f \\
\mathcal{F}[\![c^n]\!] & = & c^n \\
\mathcal{F}[\![x]\!] & = & x \\
\mathcal{F}[\![v]\!] & = & v \\
\mathcal{F}[\![\lambda x.M]\!] & = & \lambda x.\,\mathcal{F}[\![M]\!] \\
\mathcal{F}[\![M \bullet N]\!] & = & \mathcal{F}[\![M]\!] \bullet \mathcal{F}[\![N]\!] \\
\mathcal{F}[\![\uparrow M] & = & \mathit{return} \bullet \mathcal{F}[\![M]\!] \\
\mathcal{F}[\![M \rhd x \cdot N]\!] & = & \mathit{bind} \bullet \mathcal{F}[\![M]\!] \bullet \mathcal{F}[\![\lambda x.N]\!] \\
\mathcal{F}[\![\nu v.M]\!] & = & \nu v.\mathcal{F}[\![M]\!] \\
\mathcal{F}[\![M?]\!] & = & \mathit{deref} \bullet \mathcal{F}[\![M]\!] \\
\mathcal{F}[\![M := N]\!] & = & \mathit{assign} \bullet \mathcal{F}[\![M]\!] \bullet \mathcal{F}[\![N]\!] \\
\mathcal{F}[\![\mathbf{pure}\, M] & = & \mathit{pure} \bullet \mathcal{F}[\![M]\!] \\
\mathcal{F}[\![\nu_\sigma \{v_1,\ldots,v_n\}.\sigma \cdot \langle M \rangle]\!] & = & \mathit{unwrap} \bullet (\nu v_1 \ldots \nu v_n.\mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
\mathcal{F}[\![v_1 : M_1,\ldots,v_n : M_n]\!] & = & \mathit{upd} \bullet v_n \bullet \mathcal{F}[\![M_n]\!] \bullet (\ldots(\mathit{upd} \bullet v_1 \bullet \mathcal{F}[\![M_1] \bullet \{\,\}))
\end{array}
$$

Figure 5.10: The implementation translation $\mathcal{F}$ for $\lambda[\beta\delta\sigma eag]$

$$
\begin{array}{rcl}
\mathit{pair} & \equiv & \lambda x.\lambda s.\langle x,s \rangle \\
\mathit{unwrap} \bullet \langle a,s \rangle & = & \textbf{case } a \textbf{ of} \\
 & & \quad \mathit{Res} \bullet (c^n \bullet M_1 \bullet \cdots \bullet M_n) \\
 & & \qquad \rightarrow c^n \bullet (\mathit{unwrap} \bullet \langle \mathit{Res} \bullet M_1,s \rangle) \bullet \cdots \bullet (\mathit{unwrap} \bullet \langle \mathit{Res} \bullet M_n,s \rangle) \\
 & & \quad \mathit{Res} \bullet f \rightarrow \lambda x.\mathit{unwrap} \bullet \langle f \bullet x,s \rangle \\
\mathit{pure} \bullet p & = & \mathit{unwrap} \bullet (p \bullet \mathit{pair} \bullet \{\,\}) \\
\mathit{return} \bullet a \bullet k \bullet s & = & k \bullet (\mathit{Res} \bullet a) \bullet s \\
\mathit{bind} \bullet p \bullet q \bullet k \bullet s & = & p \bullet (\lambda x.q \bullet (\mathit{strip} \bullet x) \bullet k) \bullet s \\
\mathit{strip} \bullet x & = & \textbf{case } x \textbf{ of} \\
 & & \; \mathit{Res} \bullet x' \rightarrow x' \\
 & & \; \mathit{Unit} \rightarrow () \\
\mathit{deref} \bullet t \bullet k \bullet s & = & \textbf{case } s \bullet t \textbf{ of } \mathit{Def} \bullet a \rightarrow k \bullet (\mathit{Res} \bullet a) \bullet s \\
\mathit{assign} \bullet t \bullet a \bullet k \bullet s & = & k \bullet \mathit{Unit} \bullet (\mathit{upd} \bullet t \bullet a \bullet s) \\
\{\,\} \bullet t & = & \mathit{Undef} \\
\mathit{upd} \bullet t \bullet a \bullet s & = & \lambda u.\textbf{if } t == u \textbf{ then } \mathit{Def} \bullet a \textbf{ else } s \bullet u
\end{array}
$$

Figure 5.11: Auxiliary function definitions for the translation $\mathcal{F}$

In implementing the store it is necessary to distinguish an undefined binding from all possible defined values. This distinction is enforced by introducing the constructors *Undef* and *Def*. The translation of $\nu v.M$ intializes the newly-introduced variable (via the auxiliary function *newref*) to *Undef*; the translation of $M?$ (via *deref*) performs an error check for *Undef*, but the translation of $M := N$ (via *assign*) is indifferent to the definedness of the store-variable.

Before turning to the proofs involving the translation $\mathcal{F}$, it may be helpful to discuss its definition on a line-by-line basis. The main definition in Figure 5.10 divides into two parts. First, all the syntactic ingredients of $\lambda[\beta\delta\sigma eag]$ are mapped homomorphically to their cognates, if any, in λv. Second, the store-related constructs of $\lambda[\beta\delta\sigma eag]$ are each translated into an application of the appropriate auxiliary primitive from Figure 5.11.

These auxiliary primitives themselves can be organized into several groups. In the first group we have the functions *bind* and *return* which provide the basic glue for translations of compound commands.. The members of the second group (*newref*, *assign*, and *deref*) implement the store-commands as state-transformers, taking a continuation k and a store s as their last two arguments. The function *pure* stands in a group by itself; the remaining functions provide administrative services. With this grouping in mind, we now give an individual account of the function definitions.

The functions *bind* and *return* are straightforward encodings of the corresponding commands..

We now begin the proof that $\mathcal{F}$ is indeed a syntactic embedding. The requirement in Definition 5.2.1 (i) is already satisfied by construction, since $\mathcal{F}$ is homomorphic on the fragment of $\lambda\nu$ that is contained within $\lambda[\beta\delta\sigma eag]$. We thus move on to the proof that $\mathcal{F}$ preserves semantics. This proof requires establishing that the convertibility relation is preserved in both directions across the translation. We prove one direction here; the reverse implication will be proved in Section 5.2.4.

Lemma 5.2.3 *$\mathcal{F}$ is stable under reduction, that is, for all terms M, N in $\lambda[\beta\delta\sigma eag]$,*

$$\lambda[\beta\delta\sigma eag] \vdash M \rightarrow N \ \textit{ implies } \ \lambda\nu \models \mathcal{F}[\![M]\!] \cong \mathcal{F}[\![N]\!].$$

Proof: We carry out a case analysis according to the reduction rule by which $M \rightarrow N$. The derivations for the store-related cases all follow the same general pattern: carry out the translation into $\lambda\nu$ and reduce away all the administrative paraphernalia of the translated definition, perform the crucial steps that carry out the semantics, and then reverse direction to add back administrative material and reverse the translation.

We give several representative cases in full detail; omitted cases are each similar to some case that is given.

Case (1) β. In this case the reduction in $\lambda[\beta\delta\sigma eag]$ is

$$(\lambda x.M) \bullet N \rightarrow [N/x]\,M.$$

The translation $\mathcal{F}[\![\,]\!]$ is both compositional and linear, that is, the translation of a term contains exactly one copy of the translation of each of its subterms. Moreover, $\mathcal{F}[\![\,]\!]$ maps λ-variables to themselves. These facts are sufficient to establish that substitution commutes with translation, and we have in the translation the following chain of conversions:

$$\begin{array}{cl} & \mathcal{F}[\![(\lambda x.M) \bullet N]\!] \\ \equiv & (\lambda x.\, \mathcal{F}[\![M]\!]) \bullet \mathcal{F}[\![N]\!] \\ = & [\mathcal{F}[\![N]\!]/x]\ \mathcal{F}[\![M]\!] \\ \equiv & \mathcal{F}[\![[N/x]\,M]\!] \end{array}$$

Case (2) δ.

In this case the reduction in $\lambda[\beta\delta\sigma eag]$ is

$$f \bullet V \rightarrow \delta(f, V),$$

where $\delta(f, V)$ is as in Figure 2.3. This case clearly goes through under the assumption that the defining terms N_f, N_{f_1}, N_{f,c^n} of the δ-reduction are subjected to the translation to give their counterparts in the target calculus $\lambda\nu$.

Case (3) *assoc*. In this case the reduction in $\lambda[\beta\delta\sigma eag]$ is

$$(M \triangleright x \cdot N) \triangleright y \cdot P \rightarrow M \triangleright x \cdot (N \triangleright y \cdot P),$$

and we have in the λv translation the following chain of conversions:

$$\begin{array}{cl}
 & \mathcal{F}[\![(M \rhd x\cdot N) \rhd y\cdot P]\!] \\
\equiv & bind \bullet (bind \bullet \mathcal{F}[\![M]\!] \bullet (\lambda x.\, \mathcal{F}[\![N]\!])) \bullet (\lambda y.\, \mathcal{F}[\![P]\!]) \\
 & \text{(by definition of } \mathcal{F}\text{)} \\
= & \lambda k.(bind \bullet \mathcal{F}[\![M]\!] \bullet (\lambda x.\, \mathcal{F}[\![N]\!])) \bullet (\lambda x'.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x') \bullet k) \\
 & \text{(by definition of } bind\text{)} \\
= & \lambda k.(\lambda k'.\, \mathcal{F}[\![M]\!] \bullet (\lambda x''.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet (strip \bullet x'') \bullet k')) \bullet (\lambda x'.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x') \bullet k) \\
 & \text{(by definition of } bind\text{)} \\
= & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x''.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet (strip \bullet x'') \bullet (\lambda x'.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x') \bullet k)) \\
 & \text{(by } \beta\text{)} \\
= & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x''.[(strip \bullet x'')/x]\; \mathcal{F}[\![N]\!] \bullet (\lambda x'.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x') \bullet k)) \\
 & \text{(by } \beta\text{)}
\end{array}$$

Starting with the translation of the reduct, we have the derivation

$$\begin{array}{cl}
 & \mathcal{F}[\![M \rhd x\cdot(N \rhd y\cdot P)]\!] \\
= & bind \bullet \mathcal{F}[\![M]\!] \bullet (\lambda x.bind \bullet \mathcal{F}[\![N]\!] \bullet (\lambda y.\, \mathcal{F}[\![P]\!])) \\
 & \text{(by the definition of } \mathcal{F}\text{)} \\
= & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x'.(\lambda x.bind \bullet \mathcal{F}[\![N]\!] \bullet (\lambda y.\, \mathcal{F}[\![P]\!])) \bullet (strip \bullet x') \bullet k) \\
 & \text{(by the definition of } bind\text{)} \\
= & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x'.(\lambda x.\lambda k.\, \mathcal{F}[\![N]\!] \bullet (\lambda x''.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x'') \bullet k)) \bullet (strip \bullet x') \bullet k) \\
 & \text{(by the definition of } bind\text{)} \\
\rightarrow & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x'.([(strip \bullet x')/x]\; (\lambda k.\, \mathcal{F}[\![N]\!] \bullet (\lambda x''.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x'') \bullet k))) \bullet k) \\
 & \text{(by } \beta\text{)} \\
\equiv & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x'.(\lambda k.[(strip \bullet x')/x]\; \mathcal{F}[\![N]\!] \bullet (\lambda x''.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x'') \bullet k)) \bullet k) \\
 & \text{(because the original scope of } x \text{ is } N \text{ only)} \\
\equiv & \lambda k.\, \mathcal{F}[\![M]\!] \bullet (\lambda x'.[(strip \bullet x')/x]\; \mathcal{F}[\![N]\!] \bullet (\lambda x''.(\lambda y.\, \mathcal{F}[\![P]\!]) \bullet (strip \bullet x'') \bullet k)) \\
 & \text{(by } \beta\text{).}
\end{array}$$

Since the two derivations reach the same term (up to α-renaming), the redex and its reduct are convertible (and hence operationally equivalent).

Case (4) *unit.*

In this case the reduction in $\lambda[\beta\delta\sigma eag]$ is

$$(\uparrow M) \rhd x\cdot N \;\rightarrow\; [M/x]\, N.$$

Starting with the translation of the redex, we obtain the derivation

$$\begin{array}{cl}
 & \mathcal{F}[\![(\uparrow M) \rhd x\cdot N]\!] \\
\equiv & bind \bullet (return \bullet \mathcal{F}[\![M]\!]) \bullet (\lambda x.\, \mathcal{F}[\![N]\!]) \\
 & \text{(by definition of } \mathcal{F}\text{)} \\
\rightarrow & \lambda ks.(\lambda ks.k \bullet (Res \bullet \mathcal{F}[\![M]\!]) \bullet s) \bullet (\lambda x'.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet (strip \bullet x') \bullet k) \bullet s \\
 & \text{(by the definitions of } bind \text{ and } return\text{)} \\
\rightarrow & \lambda ks.(\lambda x'.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet (strip \bullet x') \bullet k) \bullet (Res \bullet \mathcal{F}[\![M]\!]) \bullet s \\
 & \text{(by } \beta\text{)} \\
\rightarrow & \lambda ks.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet (strip \bullet (Res \bullet \mathcal{F}[\![M]\!])) \bullet k \bullet s \\
 & \text{(by } \beta\text{)} \\
\rightarrow & \lambda ks.(\lambda x.\, \mathcal{F}[\![N]\!]) \bullet \mathcal{F}[\![M]\!] \bullet k \bullet s \\
 & \text{(by definition of } strip\text{)} \\
\rightarrow & \lambda ks.[\mathcal{F}[\![M]\!]/x]\; \mathcal{F}[\![N]\!] \bullet x \bullet s.
\end{array}$$

Starting with the translation of the reduct, we obtain

$$\begin{array}{ll} & \mathcal{F}[\![[M/x]\,N]\!] \\ \equiv & [\mathcal{F}[\![M]\!]/x]\ \mathcal{F}[\![N]\!] \\ & \text{(by the same argument used in the case of } \beta\text{).} \end{array}$$

Since we assume that η-reduction is a rule of λν, the two translations are convertible (and hence operationally equivalent).

Case (5) *extend.*

In this case the $\lambda[\beta\delta\sigma eag]$ reduction takes the form

$$(\nu v\, M) \triangleright x \cdot N \;\rightarrow\; \nu M \triangleright x \cdot N..$$

Starting with the translation of the redex, we obtain

$$\begin{array}{ll} & \mathcal{F}[\![(\nu v\, M) \triangleright x \cdot N]\!] \\ \equiv & \mathit{bind} \bullet (\nu v\,.\mathcal{F}[\![M]\!]) \bullet (\lambda x\,.\,\mathcal{F}[\![N]\!]) \\ & \text{(by the definition of } \mathcal{F}) \\ \rightarrow & \lambda ks\,.(\nu v\,.\mathcal{F}[\![N]\!]) \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet \mathit{strip} \bullet x' \bullet k) \bullet s \\ & \text{(by the definitions of } \mathit{bind}) \\ \leftarrow & \lambda ks\,.(\nu v\,.\lambda ks\,.\,\mathcal{F}[\![N]\!] \bullet k \bullet s) \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet \mathit{strip} \bullet x' \bullet k) \bullet s \\ & \text{(by } \eta) \\ \rightarrow & \lambda ks\,.(\lambda ks\,.\nu v\,.\mathcal{F}[\![N]\!] \bullet k \bullet s) \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet \mathit{strip} \bullet x' \bullet k) \bullet s \\ & \text{(by } \lambda\nu\lambda) \\ \rightarrow & \lambda ks\,.\nu v\,.\mathcal{F}[\![N]\!] \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet \mathit{strip} \bullet x' \bullet k) \bullet s \\ & \text{(by } \beta). \end{array}$$

Starting with the translation of the reduct, we get

$$\begin{array}{ll} & \mathcal{F}[\![\nu M \triangleright x \cdot N.]\!] \\ \equiv & \nu v\, \mathit{bind} \bullet \mathcal{F}[\![M]\!] \bullet (\lambda x\,.\,\mathcal{F}[\![N]\!]) \\ & \text{(by the definition of } \mathcal{F}) \\ \rightarrow & \nu v\, \lambda ks\,.\,\mathcal{F}[\![M]\!] \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet (\mathit{strip} \bullet x') \bullet k) \bullet s \\ & \text{by the definitions of } \mathit{bind} \\ \rightarrow & \lambda ks\,.\nu v\,.\mathcal{F}[\![M]\!] \bullet (\lambda x'\,.(\lambda x\,.\,\mathcal{F}[\![N]\!]) \bullet (\mathit{strip} \bullet x') \bullet k) \bullet s \\ & \text{(by } \lambda\nu\lambda). \end{array}$$

Since both derivations meet a the same term, the translations of the redex and reduct are convertible and hence operationally equivalent.

Case (6) σν.

In this case a reduction takes the form

$$\nu_\sigma \vec{v}.\sigma \cdot \langle \nu v\, M \rangle \;\rightarrow\; \nu_\sigma \{\vec{v}, v\}.\sigma \cdot \langle M \rangle.$$

Starting with the translation of the redex, we obtain the derivation

$$\begin{array}{ll}
 & \mathcal{F}[\![\nu_\sigma \vec{v}.\sigma \cdot \langle \nu v\, M\rangle]\!] \\
\equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\nu v . \mathcal{F}[\![M]\!]) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathcal{F}) \\
\leftarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\nu v\, \lambda ks . \mathcal{F}[\![M]\!] \bullet k \bullet s) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by } \eta) \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\lambda ks . \nu v . \mathcal{F}[\![M]\!] \bullet k \bullet s) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by } \lambda\nu\lambda) \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \nu v . \mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by } \beta).
\end{array}$$

The translation of the reduct is given by

$$\begin{array}{ll}
 & \mathcal{F}[\![\nu_\sigma \{\vec{v}, v\} . \sigma \cdot \langle M\rangle]\!] \\
\equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \nu v . \mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathcal{F}).
\end{array}$$

We have thus shown that the translations of the redex and reduct are convertible in λν.

Case (7) $\sigma ?$.

In this case a reduction takes the form

$$\nu_\sigma \vec{v}.\sigma \cdot \langle v? \triangleright x \cdot M\rangle \rightarrow \nu_\sigma \vec{v}.\sigma \cdot \langle [\sigma \downarrow v / x]\, M\rangle .$$

The fact that the reduction takes place assures us that $v \in \mathit{dom}\,\sigma$. Starting with the translation of the redex, we obtain the derivation

$$\begin{array}{ll}
 & \mathcal{F}[\![\nu_\sigma \vec{v}.\sigma \cdot \langle v? \triangleright x \cdot M\rangle]\!] \\
\equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\mathit{bind} \bullet (\mathit{deref} \bullet v) \bullet (\lambda x . \mathcal{F}[\![M]\!])) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathcal{F}) \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\lambda ks . (\mathit{deref} \bullet v) \bullet (\lambda x' . (\lambda x . \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet k) \bullet s) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathit{bind}) \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\mathit{deref} \bullet v) \bullet (\lambda x' . (\lambda x . \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet \mathit{pair}) \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by } \beta). \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \mathbf{case}\ \ \mathcal{F}[\![\sigma]\!] \bullet v\ \mathbf{of}\ \mathit{Def} \bullet a \rightarrow K) \\
 & \text{where } \ K \equiv (\lambda x' . (\lambda x . \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet \mathit{pair}) \bullet (\mathit{Res} \bullet a) \bullet \mathcal{F}[\![\sigma]\!] \\
 & \text{(by the definition of } \mathit{deref} \text{ and } \beta). \\
 & \ldots
\end{array}$$

It should clear from the implementation of stores in λν that the presence of a binding for v in σ implies that $\mathcal{F}[\![\sigma]\!] \bullet v \rightarrow \mathit{Def} \bullet \mathcal{F}[\![\mathcal{F}[\![\sigma]\!] \downarrow v]\!]$. Using this observation, we continue our derivation:

$$\begin{array}{ll}
 & \ldots \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\lambda x' . (\lambda x . \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet \mathit{pair}) \bullet (\mathit{Res} \bullet \mathcal{F}[\![\sigma \downarrow v]\!]) \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the discussion above and } \beta) \\
\rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\lambda x . \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet (\mathit{Res} \bullet \mathcal{F}[\![\sigma \downarrow v]\!])) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by } \beta)
\end{array}$$

Starting with the translation of the reduct, we obtain

$$\begin{array}{ll} & \mathcal{F}[\![\nu_\sigma \vec{v}.\sigma \cdot \langle [\sigma \downarrow v/x]\, M \rangle]\!] \\ \equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \mathcal{F}[\![[\sigma \downarrow v/x]\, M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\ & \text{(by the definition of } \mathcal{F}) \\ \equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . [\mathcal{F}[\![\sigma \downarrow v]\!]/x]\, \mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\ & \text{(by the substitutivity of } \mathcal{F}) \end{array}$$

Case (8) $\sigma{:=}$. In this case the reduction in $\lambda[\beta\delta\sigma eag]$ is

$$\nu_\sigma \vec{v}.\sigma \cdot \langle v := N;\, M \rangle \rightarrow \nu_\sigma \vec{v}.(\sigma \,!\, v : N) \cdot \langle M \rangle.$$

We show that this conversion translates into an operational equivalence via the following chain of reasoning.

$$\begin{array}{ll} & \mathcal{F}[\![\nu_\sigma \vec{v}.\sigma \cdot \langle v := N;\, M \rangle]\!] \\ \equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . (\mathit{bind} \bullet (\mathit{assign} \bullet v \bullet \mathcal{F}[\![N]\!]) \bullet (\lambda z.\, \mathcal{F}[\![M]\!])) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\ & \text{(by the definition of } \mathcal{F};\ z \text{ has no bound occurrences)} \\ \rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . (\mathit{assign} \bullet v \bullet \mathcal{F}[\![N]\!]) \bullet (\lambda x'.(\lambda z.\, \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet \mathit{pair}) \bullet \mathcal{F}[\![\sigma]\!]) \\ & \text{(by the definition of } \mathit{bind} \text{ and } \beta) \\ \rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . (\lambda x'.(\lambda z.\, \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet x') \bullet \mathit{pair}) \bullet \mathit{Unit} \bullet (\mathit{upd} \bullet v \bullet (\mathit{Def} \bullet a) \bullet \mathcal{F}[\![\sigma]\!])) \\ & \text{(by the definition of } \mathit{assign} \text{ and } \beta) \\ \rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . ((\lambda z.\, \mathcal{F}[\![M]\!]) \bullet (\mathit{strip} \bullet \mathit{Unit}) \bullet \mathit{pair}) \bullet (\mathit{upd} \bullet v \bullet (\mathit{Def} \bullet a) \bullet \mathcal{F}[\![\sigma]\!])) \\ & \text{(by } \beta) \\ \rightarrow & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet (\mathit{upd} \bullet v \bullet (\mathit{Def} \bullet a) \bullet \mathcal{F}[\![\sigma]\!])) \\ & \text{(by } \beta\text{, noting that } z \text{ has no bound occurrences)} \end{array}$$

Starting with the translation of the reduct, we obtain

$$\begin{array}{ll} & \mathcal{F}[\![\nu_\sigma \vec{v}.(\sigma \,!\, v : N) \cdot \langle M \rangle]\!] \\ \equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma \,!\, v : N]\!]) \\ & \text{(by the definition of } \mathcal{F}) \end{array}$$

We have the desired result if we can show that

$$\mathit{upd} \bullet v \bullet (\mathit{Def} \bullet a) \bullet \mathcal{F}[\![\sigma]\!] \cong \mathcal{F}[\![\sigma \,!\, v : N]\!]$$

in $\lambda\nu$.

Case (9) σ_{block}.

In this case the reduction takes the form

$$\mathbf{pure}\, M \rightarrow \nu_\sigma \{\}.\{\} \cdot \langle M \rangle.$$

Starting with the translation of the redex, we obtain the derivation

$$\begin{array}{ll} & \mathcal{F}[\![\mathbf{pure}\, M]\!] \\ \equiv & \mathit{pure} \bullet \mathcal{F}[\![M]\!] \\ & \text{(by the definition of } \mathcal{F}) \\ \rightarrow & \mathit{unwrap} \bullet (\mathcal{F}[\![M]\!] \bullet \mathit{pair} \bullet \{\}), \end{array}$$

which is exactly the translation of the reduct $\nu_\sigma \{\}.\{\} \cdot \langle M \rangle$.

Case (10) σ_{peag}. Using the purification of an abstraction as a representative case, the reduction in $\lambda[\beta\delta\sigma eag]$ takes the form

$$\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow \lambda x.M \rangle \to \lambda x.\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow M \rangle.$$

We establish the semantic faithfulness of $\mathcal{F}$ in this case via the following chain of reasoning:

$$\begin{array}{ll}
 & \mathcal{F}[\![\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow \lambda x.M \rangle]\!] \\
\equiv & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\mathit{return} \bullet (\lambda x.\, \mathcal{F}[\![M]\!])) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathcal{F}\text{)} \\
\to & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \mathit{pair} \bullet \mathit{Res} \bullet (\lambda x.\, \mathcal{F}[\![M]\!]) \bullet \mathcal{F}[\![\sigma]\!]) \\
 & \text{(by the definition of } \mathit{return} \text{ and } \beta\text{)} \\
\to & \mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n . \langle \mathit{Res} \bullet (\lambda x.\, \mathcal{F}[\![M]\!]), \mathcal{F}[\![\sigma]\!] \rangle) \\
 & \text{(by the definition of } \mathit{pair}\text{)} \\
\to & \mathit{unwrap} \bullet \langle \nu v_1 \ldots . \nu v_n . \mathit{Res} \bullet (\lambda x.\, \mathcal{F}[\![M]\!]), \nu v_1 \ldots . \nu v_n . \mathcal{F}[\![\sigma]\!] \rangle \\
 & \text{(by } \lambda\nu c^n\text{)} \\
\to & \lambda x.\mathit{unwrap} \bullet \langle \nu v_1 \ldots . \nu v_n . \mathit{Res} \bullet \mathcal{F}[\![M]\!], \nu v_1 \ldots . \nu v_n . \mathcal{F}[\![\sigma]\!] \rangle \\
 & \text{(by the definition of } \mathit{unwrap}, \beta, \text{ and } \delta\text{).}
\end{array}$$

The translation of the reduct is given by

$$\begin{array}{ll}
 & \mathcal{F}[\![\lambda x.\nu_\sigma \vec{v}.\sigma \cdot \langle \uparrow M \rangle]\!] \\
\equiv & \lambda x.\mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n .(\mathit{return} \bullet \mathcal{F}[\![M]\!]) \bullet \mathit{pair} \bullet \mathcal{F}[\![\sigma]\!]),
\end{array}$$

from which a similar derivation leads to the last term in the derivation from the redex. This establishes the convertibility of the translations.

The proofs involving the other forms of σ_{peag}-rule are similar.

This concludes the proof of Lemma 5.2.3. ∎

5.2.4 Reversing the translation

Lemma 5.2.3 shows that convertible terms in $\lambda[\beta\delta\sigma eag]$ are mapped by $\mathcal{F}$ to operationally-equivalent terms in $\lambda\nu$. This result goes part of the way toward showing that $\mathcal{F}$ preserves semantics, since it shows that $\mathcal{F}$ preserves semantic equalities. However, we are also obligated to show that semantic *distinctions* are preserved as well. The present section is dedicated to proving the implication in this direction.

To do so, we define a left inverse $\mathcal{F}^{-1}$ for $\mathcal{F}$, and show that $\mathcal{F}^{-1}$ also preserves semantics in the same sense that $\mathcal{F}$ does. The definition of $\mathcal{F}^{-1}$ is given in Figure 5.12. Since $\mathcal{F}^{-1}$ only needs to apply to $\lambda\nu$-terms that are in the range of $\mathcal{F}$, not all possible $\lambda\nu$-terms are represented in the left-hand-sides of the defining equations for $\mathcal{F}^{-1}$. One particular nuance of the definition is that terms of the form $\nu v.M$ in $\lambda\nu$ arise in two different ways as a result of $\mathcal{F}$: as a translation of a $\lambda[\beta\delta\sigma eag]$ term of the form $\nu v.M'$, or as the translation of a $\lambda[\beta\delta\sigma eag]$ term of the form $\sigma \cdot \langle M' \rangle$. The definition of $\mathcal{F}^{-1}$ takes advantage of the detectable syntactic difference between these two cases to distinguish them for the purpose of mapping them back to their original forms.

Lemma 5.2.4 (Left inverse) *For all terms M in $\lambda[\beta\delta\sigma eag]$,*

$$\mathcal{F}^{-1}[\![\mathcal{F}[\![M]\!]]\!] \equiv M.$$

Proof: This proof of Lemma 5.2.4 is a straightforward induction on the structure of M; the definition in Figure 5.12 does all the work. ∎

Lemma 5.2.5 ($\mathcal{F}^{-1}$ is substitutive) *If M and N are terms in $\lambda\nu$ such that $\mathcal{F}^{-1}[\![M]\!]$ and $\mathcal{F}^{-1}[\![N]\!]$ are defined, then*

$$\mathcal{F}^{-1}[\![[N/x]\,M]\!] \equiv [\mathcal{F}^{-1}[\![N]\!]/x]\; \mathcal{F}^{-1}[\![M]\!].$$

$$
\begin{aligned}
\mathcal{F}^{-1}[\![f]\!] &= f \\
\mathcal{F}^{-1}[\![c^n]\!] &= c^n \\
\mathcal{F}^{-1}[\![x]\!] &= x \\
\mathcal{F}^{-1}[\![v]\!] &= v \\
\mathcal{F}^{-1}[\![\lambda x.M]\!] &= \lambda x.\,\mathcal{F}^{-1}[\![M]\!] \\
\mathcal{F}^{-1}[\![M \bullet N]\!] &= \mathcal{F}^{-1}[\![M]\!] \bullet \mathcal{F}^{-1}[\![N]\!] \\
\mathcal{F}^{-1}[\![\nu v.M]\!] &= \nu v.\mathcal{F}^{-1}[\![M]\!] \\
\mathcal{F}^{-1}[\![\mathit{deref} \bullet M]\!] &= \mathcal{F}^{-1}[\![M]\!]? \\
\mathcal{F}^{-1}[\![\mathit{assign} \bullet M \bullet N]\!] &= \mathcal{F}^{-1}[\![M]\!] := \mathcal{F}^{-1}[\![N]\!] \\
\mathcal{F}^{-1}[\![\mathit{return} \bullet M]\!] &= \uparrow \mathcal{F}^{-1}[\![M]\!] \\
\mathcal{F}^{-1}[\![\mathit{bind} \bullet M \bullet (\lambda x.N)]\!] &= \mathcal{F}^{-1}[\![M]\!] \triangleright x \cdot \mathcal{F}^{-1}[\![N]\!] \\
\mathcal{F}^{-1}[\![\mathit{pure} \bullet M]\!] &= \mathbf{pure}\, \mathcal{F}^{-1}[\![M]\!] \\
\mathcal{F}^{-1}[\![\mathit{unwrap} \bullet (\nu v_1 \ldots . \nu v_n.(p \bullet s))]\!] &= \nu_\sigma \{v_1, \ldots, v_n\}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle \mathcal{P}^{-1}[\![p]\!] \rangle \\[1em]
\mathcal{S}^{-1}[\![\{\}]\!] &= \{\} \\
\mathcal{S}^{-1}[\![\mathit{upd} \bullet (\mathit{Def} \bullet a) \bullet v \bullet s]\!] &= \mathcal{S}^{-1}[\![s]\!] \,!\, v : a \\[1em]
\mathcal{P}^{-1}[\![p \bullet \mathit{pair}]\!] &= \mathcal{F}^{-1}[\![p]\!] \\
\mathcal{P}^{-1}[\![p \bullet (\lambda x.q \bullet (\mathit{strip} \bullet x) \bullet k)]\!] &= \mathcal{F}^{-1}[\![p]\!] \triangleright x \cdot \mathcal{P}^{-1}[\![q \bullet k]\!]
\end{aligned}
$$

Figure 5.12: The inverse translation $\mathcal{F}^{-1}$ from $\lambda\nu$ to $\lambda[\beta\delta\sigma eag]$

The form in which we state the main lemma of this subsection requires some justification. We are in the process of attempting to show that $\mathcal{F}$ preserves operational distinctions by showing that its left inverse preserves operational similarities. The basic datum of operational semantics is a term (in a context) reducing to an answer. If it helps our proof in some way, we are free to select a particular reduction sequence that witnesses this fact. In the present situation, such an approach is desirable, since only certain reductions keep the reduct within the image of the map $\mathcal{F}$. However, $\mathcal{F}$ was *designed* so that reductions of its translations of terms denoting commands in $\lambda[\beta\delta\sigma eag]$ would have an easy interpretation as a deterministic machine. In fact, the (deterministic) standard reduction ordering in $\lambda\nu$ corresponds to the execution order of this notional state machine. Hence the following lemma refers to standard reductions only.

Lemma 5.2.6 *The translation $\mathcal{F}^{-1}$ is stable under standard reduction. That is, for all terms M and N in $\lambda\nu$ such that*

$$\lambda\nu \vdash M \to_s N \to^* A,$$

if $\mathcal{F}^{-1}[\![M]\!]$ is defined, then so is $\mathcal{F}^{-1}[\![N]\!]$, and

$$\lambda[\beta\delta\sigma eag] \models \mathcal{F}^{-1}[\![M]\!] \cong \mathcal{F}^{-1}[\![N]\!].$$

Proof: The proof is a case analysis on the form of the redex Δ by which $M \to N$ in the hypothesis of the lemma. We give only one representative case here, that for the translation of the store-manipulating rule $\sigma{:=}$. The other cases yield similar derivations.

Case (1) $\Delta \equiv \mathit{assign} \bullet a \bullet n \bullet k \bullet s$.

Since the reduction appears in a standard reduction sequence to an answer A, Δ must occur in a superterm of the form $\mathit{unwrap} \bullet \Delta$, k must be a continuation of the form $\lambda x.q \bullet (\mathit{strip} \bullet x) \bullet k'$, and s must be a store: otherwise, reduction would get stuck before an answer could be reached.

In $\lambda\nu$ the reduct R of Δ is then

$$[\mathit{Unit}/x]\,(q \bullet k' \bullet (\mathit{upd} \bullet n \bullet (\mathit{Def} \bullet x) \bullet s)),$$

and we have to show that

$$\lambda[\beta\delta\sigma eag] \models \mathcal{F}^{-1}[\![unwrap \bullet \mathrm{v}\vec{v}.\Delta]\!] \cong \mathcal{F}^{-1}[\![unwrap \bullet \mathrm{v}\vec{v}.R]\!].$$

The left-hand side of this operational equivalence expands to

$$\mathrm{v}_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle \mathcal{P}^{-1}[\![assign \bullet n \bullet a \bullet k]\!] \rangle. \tag{5.1}$$

We turn first to the store portion of this term. Since the entire term is a product of translation via $\mathcal{F}$, we can rely on s having the form

$$upd \bullet n_m \bullet a_m \bullet (\ldots (upd \bullet n_1 \bullet a_1 \bullet \{\})),$$

for some names n_i and terms a_i. Since we know (by hypothesis) that the reduction of this term will not get stuck, we can now assert that the particular name n is in the domain of s and hence is equal to at least one of the n_i. We can thus safely assume the $n \in dom\, \mathcal{S}^{-1}[\![s]\!]$.

We now turn to the command part of the inverse translation 5.1. Since $\lambda[\beta\delta\sigma eag]$ has the special rule *assign-result* which applies when the functional result of an assignment command is actually observed, we need to consider two cases according to whether the variable x occurs free in the portion q of the continuation expression $k = \lambda x.q \bullet (strip \bullet x) \bullet k'$. It is easy to see that x cannot occur free in k'.

We then have the following chain of reasoning in $\lambda[\beta\delta\sigma eag]$:

$$\begin{array}{ll}
 & \mathrm{v}_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle \mathcal{P}^{-1}[\![assign \bullet n \bullet a \bullet (\lambda x.q \bullet (strip \bullet x) \bullet k')]\!] \rangle \\
\equiv & \mathrm{v}_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle n := \mathcal{F}^{-1}[\![a]\!] \triangleright x \cdot \mathcal{P}^{-1}[\![q \bullet k']\!] \rangle \\
 & \text{(by the definition of } \mathcal{P}^{-1}) \\
= & \mathrm{v}_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle n := \mathcal{F}^{-1}[\![a]\!];\ [(\,)/x]\ (\mathcal{P}^{-1}[\![q \bullet k']\!]) \rangle \\
 & \text{(by } \textit{assign-result} \text{ and } \beta) \\
\equiv & \mathrm{v}_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!] \cdot \langle n := \mathcal{F}^{-1}[\![a]\!];\ (\mathcal{P}^{-1}[\![[Unit/x]\, q \bullet k']\!]) \rangle \\
 & \text{(by Lemma 5.2.5)} \\
= & \mathrm{v}_\sigma \vec{v}.(\mathcal{S}^{-1}[\![s]\!]\ !\ n : \mathcal{F}^{-1}[\![a]\!]) \cdot \langle \mathcal{P}^{-1}[\![[Unit/x]\, q \bullet k']\!] \rangle \\
 & \text{(by } \sigma{:=}) \\
\equiv & \mathrm{v}_\sigma \vec{v}.(\mathcal{S}^{-1}[\![s]\!]\ !\ n : \mathcal{F}^{-1}[\![a]\!]) \cdot \langle \mathcal{P}^{-1}[\![([Unit/x]\, q) \bullet k']\!] \rangle \\
 & \text{(since } x \text{ is not free in } k') \\
\equiv & \cdots
\end{array}$$

From this point in the derivation, we give two alternate sequels, one for each of the two possible forms of the continuation expression k'.

If k' has the form $\lambda y.q' \bullet (strip \bullet y) \bullet k''$, the derivation continues as follows:

$$\begin{array}{ll}
 & \cdots \\
\equiv & \mathrm{v}_\sigma \vec{v}.(\mathcal{S}^{-1}[\![s]\!]\ !\ n : \mathcal{F}^{-1}[\![a]\!]) \cdot \langle \mathcal{P}^{-1}[\![([Unit/x]\, q) \bullet \lambda y.q' \bullet (strip \bullet y) \bullet k'']\!] \rangle \\
\equiv & \mathrm{v}_\sigma \vec{v}.(\mathcal{S}^{-1}[\![s]\!]\ !\ n : \mathcal{F}^{-1}[\![a]\!]) \cdot \langle \mathcal{F}^{-1}[\![([Unit/x]\, q)]\!] \triangleright y \cdot \mathcal{P}^{-1}[\![q' \bullet k'']\!] \rangle \\
 & \text{(by the definition of } \mathcal{P}^{-1}) \\
\equiv & \mathcal{F}^{-1}[\![unwrap \bullet ([Unit/x]\, q) \bullet (\lambda y.q' \bullet (strip \bullet y) \bullet k''(upd \bullet n \bullet (Def \bullet z) \bullet s))]\!] \\
 & \text{(by the definitions of } \mathcal{F}^{-1}, \mathcal{S}^{-1}, \text{ and } \mathcal{P}^{-1}) \\
\equiv & \mathcal{F}^{-1}[\![R]\!]
\end{array}$$

If, on the other hand, the continuation expression k' is of the form $\lambda x.pair$, then the derivation can be continued as follows:

$$\begin{array}{ll}
 & \ldots \\
\equiv & \nu_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!]\,!\,n : \mathcal{F}^{-1}[\![a]\!] \cdot \langle \mathcal{P}^{-1}[\![([Unit/x]\,q) \bullet pair]\!]\rangle \\
\equiv & \nu_\sigma \vec{v}.\mathcal{S}^{-1}[\![s]\!]\,!\,n : \mathcal{F}^{-1}[\![a]\!] \cdot \langle \mathcal{P}^{-1}[\![[Unit/x]\,q]\!]\rangle \\
 & \text{(by the definition of } \mathcal{P}^{-1}\text{)} \\
\equiv & \mathcal{F}^{-1}[\![unwrap \bullet ([Unit/x]\,q) \bullet pair \bullet (upd \bullet n \bullet (Def \bullet a) \bullet s]\!] \\
 & \text{(by the definitions of } \mathcal{F}^{-1} \text{ and } \mathcal{S}^{-1}\text{)} \\
\equiv & \mathcal{F}^{-1}[\![R]\!]
\end{array}$$

This concludes the proof for the case in which x occurs free in q. When x does not occur free in q, the steps in the derivation are the same except that the step involving *assign-result* is omitted, and there is no need to carry along the substition of *Unit* for x.

This concludes the proof of Lemma 5.2.6. ∎

5.2.5 Establishing conservative extension

We now have all the ingredients necessary to establish our desired conservative extension result. We first observe that we have in hand a syntactic embedding of $\lambda[\beta\delta\sigma eag]$ into $\lambda[\beta\delta]$.

Proposition 5.2.7 *The translation $\mathcal{F}$ is a syntactic embedding of $\lambda[\beta\delta\sigma eag]$ into $\lambda[\beta\delta]$.*

Proof: We first show that $\mathcal{F}$ preserves λ-programs. Let M be a closed $\lambda[\beta\delta]$-term. Then by an easy induction on the structure of M we can establish that $\mathcal{F}[\![M]\!] \equiv M$. Since there is a syntactic embedding $\mathcal{E}_v$ of λv into $\lambda[\beta\delta]$, $\mathcal{E}_v[\![M]\!] \equiv M$ (syntactic embeddings preserve programs). Hence the composed mapping $\mathcal{E}_\sigma = \mathcal{E}_v \circ \mathcal{F}$ preserves programs.

We now show that $\mathcal{E}_\sigma$ preserves semantics. Since $\mathcal{E}_v$ is a syntactic embedding this follows from the condition

$$\lambda[\beta\delta\sigma eag] \vdash M = A \quad \text{if and only if} \quad \lambda v \vdash \mathcal{F}[\![M]\!] = A$$

for all terms M and answers A in $\lambda[\beta\delta\sigma eag]$.

We show each direction of the "if and only if" separately. First, we tackle "only if". Assume that $M = A$. Then by an induction on the length of the reduction sequence from M to A (using Lemma 5.2.3 at each step), we obtain $\mathcal{F}[\![M]\!] \cong \mathcal{F}[\![A]\!]$. But the latter term equals A by the preservation of programs.

Now we deal with "if". Assume $\mathcal{F}[\![M]\!] = A$. Then by an induction on the length of a standard reduction from $\mathcal{F}[\![M]\!]$ to A, we obtain $\mathcal{F}^{-1}[\![\mathcal{F}[\![M]\!]]\!] \cong \mathcal{F}^{-1}[\![A]\!]$. The right-hand side of this operational equivalence equals A by the preservation of programs, and the left-hand side equals M by Lemma 5.2.4. Thus we obtain $M \cong A$, which is what must be proved. ∎

Theorem 5.2.8 (Conservative extension) $\lambda[\beta\delta\sigma eag]$ *is a conservative operational extension of* $\lambda[\beta\delta]$*: for any two* $\lambda[\beta\delta]$*-terms* M, N,

$$\lambda[\beta\delta] \models M \cong N \quad \textit{if and only if} \quad \lambda[\beta\delta\sigma eag] \models M \cong N.$$

Proof: By Proposition 5.2.7, there is a syntactic embedding of $\lambda[\beta\delta\sigma eag]$ into $\lambda[\beta\delta]$. By Theorem 5.2.2, this implies that $\lambda[\beta\delta\sigma eag]$ is an operational extension of $\lambda[\beta\delta]$. It remains to be shown that the extension is conservative.

To prove this, let M and N be arbitrary terms in the common language of $\lambda[\beta\delta]$ and $\lambda[\beta\delta\sigma eag]$, and assume that $\lambda[\beta\delta\sigma eag] \models M \cong N$. By the definition of operational equivalence, this says that, for all contexts $C[\,]$ such that $C[M]$ and $C[N]$ are closed, we have

$$\lambda[\beta\delta\sigma eag] \vdash C[M] = A \quad \text{if and only if} \quad \lambda[\beta\delta\sigma eag] \vdash C[N] = A.$$

Since this statement is true for all $\lambda[\beta\delta\sigma eag]$-contexts, it is true for that subset which are also $\lambda[\beta\delta]$-contexts. Restricting the statement to those contexts, and observing that M and N are $\lambda[\beta\delta]$-terms and hence only have

$\lambda[\beta\delta]$-redexes, we establish the statement

$$\lambda[\beta\delta] \vdash C[M] = A \text{ if and only if } \lambda[\beta\delta] \vdash C[N] = A,$$

which asserts that $\lambda[\beta\delta] \models M \cong N$, which was to be proved. ∎

Corollary 5.2.9 *$\lambda[\beta\delta!eag]$ is a conservative operational extension of $\lambda[\beta\delta]$: for any two $\lambda[\beta\delta]$-terms M, N,*

$$\lambda[\beta\delta] \models M \cong N \textit{ if and only if } \lambda[\beta\delta!eag] \models M \cong N.$$

Proof: This follows immediately from Theorems 5.2.8 and 5.1.7. ∎

5.2.6 Conservative extension for $\lambda[\beta\delta!laz]$

Constructing a translation from $\lambda[\beta\delta\sigma laz]$ into $\lambda[\beta\delta]$ has so far proved a severe challenge for the techniques used in this section. The problem is not in the evaluation order itself—the resources of continuation-passing style are sufficient for modeling the evaluation order we require. The problem is that we have other constraints. The nesting of ν-constructs requires that later commands in a chain be lexically contained within the translations of earlier commands, but the natural functional implementation of a lazy store places later commands at *outer* levels so that the result appears outermost and causes the execution of earlier commands by propagation of demand. We have experimented with a couple of approaches but without success so far.

5.3 Chapter summary

This chapter has established the relationship between the operational semantics of the calculi of concern and the operational semantics derived from consideration of store-machines. Furthermore, the results of this chapter show that the eager-store assignment calculus $\lambda[\beta\delta!eag]$ can be viewed as a conservative extension of a basic lambda-calculus. We conjecture that the same is true of the lazy-store calculi but we have so far had only partial success in constructing a proof of this fact along the lines of the proof offerred for $\lambda[\beta\delta!eag]$.

6
Typed lambda-calculi with assignment

Up to this point, our presentation of lambda-calculi with assignment has been entirely in terms of untyped calculi. We have shown that these calculi offer a suitable basis for a theory of functional programming with assignment. Nevertheless, we have several motivations for considering the construction of type systems for these calculi:

- The current standard practice in functional programming language design is to devise *typed* languages.
- A significant precursor of our work, the Imperative Lambda Calculus of Swarup, Reddy, and Ireland [Swarup *et al.*, 1991; Swarup, 1992] is a typed system, and the relationship to the current work has never been stated precisely.
- The calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ have a couple of rough edges that can be smoothed by the introduction of types:
  - It is possible to have runtime errors (stuck terms) resulting from attempts to read unitialized store-variables or store-variables introduced in outer **pure**-scopes.
  - It is possible for purification to fail owing to an attempt to return an inappropriate value as the result of a store computation.
  - As discussed in Section 2.6.3, the interpretation of store-variables as locations is inhibited by the inability of the untyped calculi to guarantee that store-variable names are used only locally.

For these reasons we now turn our attention to the construction of typed versions of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. We make two contributions in so doing. First (and most distressingly), the type system of Imperative Lambda Calculus, as well as the type system for $\lambda[\beta\delta!eag]$ previously proposed by Chen and Odersky [Chen and Odersky, 1993; Chen and Odersky, 1994], is flawed, as has been pointed out in [Huang and Reddy, 1995]. We propose a new type system to correct this flaw and establish its important properties. Second, we establish that this type system also suffices for $\lambda[\beta\delta!laz]$.

The guiding slogan of these type systems, as for all modern polymorphic type systems, comes from Milner's seminal paper on the ML type system [Milner, 1978]: *well-typed programs do not go wrong.* In our context, this will mean that, in addition to assuring that all function applications match the type of the actual argument to the input type of the function, and that values stored in a store-variable have the same type over time, these type systems assure that well-typed **pure** expressions never get stuck.

Section 6.1 introduces the core Hindley/Milner polymorphic type system, which serves as background to the systems considered in this chapter. Section 6.2 discusses the additional requirements imposed on a type system by lambda-calculi with assignments. In Section 6.3 we present the Chen/Odersky type system, both to exhibit its known flaw and to serve as concrete introduction to the more successful type system LPJ^- to be presented in Section 6.4 along with the proof of its safety for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. Section 6.5 examines the possibility of devising an untyped analogue of the full Launchbury/Peyton Jones typed language.

Our consideration of type systems in this chapter is confined to their safety properties only; we do not consider the issue of implementing type-checking or type-reconstruction algorithms, or even the question of whether such an algorithm exists. Although it is unrealistic to proceed with a practical language design without knowing whether type-checking can be implemented, we exclude these subjects in order to limit the scope of the present investigation. There is, in fact, little reason to doubt that the type system we propose in Section 6.4 possesses reasonable typing algorithms.

6.1 The Hindley/Milner polymorphic type system for functional languages

The untyped calculi we consider have a common functional core formed by the calculus $\lambda[\beta\delta]$. In this section we present the Hindley/Milner polymorphic type system [Hindley, 1969; Milner, 1978; Damas and Milner,

$$\begin{array}{rcl} M & ::= & (\cdots \text{Figure } 2.2 \cdots) \\ & | & \mathbf{let}\, x = M_1 \,\mathbf{in}\, M_2 \end{array}$$

Figure 6.1: Syntax of the **let** expression

$$\mathbf{let}\, x = M_1 \,\mathbf{in}\, M_2 \quad \rightarrow \quad [M_1/x]\, M_2$$

Figure 6.2: Reduction rule for **let**

$$\begin{array}{rcl} \theta & \in & \mathit{Types} \\ \alpha & \in & \mathit{Type\ variables} \\ \kappa^n & \in & \mathit{Type\ constructors} \\ \sigma & \in & \mathit{Type\ schemes} \\ \Gamma & \in & \mathit{Type\ environments} \end{array}$$

$$\begin{array}{rcll} \theta & ::= & \alpha & \\ & | & \theta \rightarrow \theta' & \textit{function types} \\ & | & \kappa^n\, \theta_1 \,\ldots\, \theta_n & \textit{constructed types} \\ \sigma & ::= & \theta \mid \forall\alpha.\sigma & \\ \Gamma & ::= & \{\} \mid \Gamma, x: \theta & \end{array}$$

Figure 6.3: Syntax of types for the Hindley/Milner polymorphic type system

$$(\mathit{Const})\, \frac{}{\Gamma \vdash c^n : \theta_1 \rightarrow \cdots \rightarrow \theta_n \rightarrow \theta}\, (c^n : \theta_1 \rightarrow \cdots \rightarrow \theta_n \rightarrow \theta) \qquad (\mathit{App})\, \frac{\Gamma \vdash M: \theta' \rightarrow \theta \quad \Gamma \vdash N: \theta'}{\Gamma \vdash M \bullet N: \theta}$$

$$(\mathit{Prim})\, \frac{}{\Gamma \vdash f: \theta_1 \rightarrow \theta_2}\, (f: \theta_1 \rightarrow \theta_2) \qquad (\mathit{Let})\, \frac{\Gamma \vdash N: \sigma \quad \Gamma, x: \sigma \vdash M: \theta}{\Gamma \vdash \mathbf{let}\, y = N \,\mathbf{in}\, M: \theta}$$

$$(\mathit{Var})\, \frac{}{\Gamma \vdash x: \theta}\, (x: \theta \in \Gamma) \qquad (\mathit{Gen})\, \frac{\Gamma \vdash M: \sigma}{\Gamma \vdash M: \forall\alpha.\sigma}\, (\alpha \notin \mathit{fv}\, \Gamma)$$

$$(\mathit{Abs})\, \frac{\Gamma, x: \theta' \vdash M: \theta}{\Gamma \vdash \lambda x.M: \theta' \rightarrow \theta} \qquad (\mathit{Spec})\, \frac{\Gamma \vdash M: \forall\alpha.\sigma}{\Gamma \vdash M: [\theta/\alpha]\, \sigma}$$

Figure 6.4: Typing rules for the Hindley/Milner polymorphic type system

1982] for this common core language in the form in which we intend to build upon it in later sections. Figures 6.3 and 6.4 give our version of the typing rules for the Hindley/Milner type system.

The main features of this type system are *type variables*, which may be introduced on the same basis as explicit types, and *type schemes*, which quantify types over free variables. Type polymorphism is restricted in the Hindley/Milner system by requiring that all such quantifiers must be at the outermost level in a type expression; this restriction is enforced by the syntax of types and type schemes given in Figure 6.3.

The **let** construct is given an added meaning in these calculi in order to provide a syntactic notation for the introduction of polymorphism. Figure 6.1 gives the syntax of the basic functional language including the **let** construct, and Figure 6.2 gives the reduction rule for the **let** construct. The special typing behavior of **let** is given by the rule *Let* in Figure 6.4: this type rule is the only one that applies when a variable is assumed to have a type scheme rather than a type. It is because of this special typing behavior that we cannot merely stipulate that **let** $x = M_1$ **in** M_2 reduces to $(\lambda x.M_2) \bullet M_1$, for the latter term can only reflect a single monomorphic instantiation of the type of x, which would violate the usual requirement that reduction preserves typability.

Since none of the syntax-specific rules except for *Let* allow for type schemes, the Hindley/Milner type

$$(Weaken)\,\frac{\Gamma \vdash M : \theta}{\Gamma, x : \theta' \vdash M : \theta} \qquad (Permute)\,\frac{\ldots, x_1 : \sigma_1, \ldots, x_2 : \sigma_2, \ldots \vdash M : \theta}{\ldots, x_2 : \sigma_2, \ldots, x_1 : \sigma_1, \ldots \vdash M : \theta}$$

Figure 6.5: Structural rules for type derivations

system provides a way of introducing and eliminating type schemes by using the rules *Gen* and *Spec*. The rule *Gen* allows the introduction of a schematic variable α provided that no type assumed in the type environment Γ has a free occurrence of α. Informally, this side condition means that the type judgment $\Gamma \vdash M : \sigma$ is indifferent as to what actual type that may be represented by the type variable α. The rule *Spec* expresses the inverse process of instantiating a type scheme with any particular type. Informally, since the type scheme could only have been established by indifference as to the actual type to take the place of α, this specialization ought to be safe. The following type derivation of a polymorphic type for a use of the identity function illustrates the use of these rules:

$$(App)\,\frac{(Spec)\,\dfrac{(Gen)\,\dfrac{(Abs)\,\dfrac{(Var)\,\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha}}{\vdash \lambda x.x : \mathrm{Int} \to \mathrm{Int}} \qquad (Const)\,\dfrac{}{\vdash 1 : \mathrm{Int}}}{\vdash (\lambda x.x) \bullet 1 : \mathrm{Int}} \quad .$$

One feature of our presentation of the Hindley/Milner type system is not present in Milner's paper, but is adopted following present practice (including [Chen and Odersky, 1994]): we assume the existence of *type constructors* κ^n of arity n. These type constructors are useful in typing the value constructors of the basic calculus. For example, if our lambda-calculus contains a constructor *pair* of arity 2, we may wish to posit a constructed type *Pair* such that *pair* has type $\forall\alpha.\forall\beta.\alpha \to \beta \to Pair\,\alpha\,\beta$. Since value-constructors in our basic lambda-calculi subsume constants, the introduction of type-constructors also serves to define types of constants, such as the type of the integers, as nullary type constructors.

In addition to constructed types, the basic type system in Figure 6.4 provides a rule that assigns a predetermined type to primitive function names. In order for the type system to prevent well-typed terms from getting stuck, we must assume that these primitive functions are defined on all arguments of their argument type, where f being defined on an argument V means that $f \bullet V$ is a δ-redex. In terms of types, we assume that for every primitive function f with type $\tau_1 \to \tau_2$ and every value-constructor c^n such that $c^n \bullet M_1 \bullet \cdots \bullet M_n$ has type τ_1, $f \bullet (c^n \bullet M_1 \bullet \cdots \bullet M_n)$ is defined and has type τ_2. We make a similar assumption for primitives that take functional arguments: if $\vdash f : (\tau_1 \to \tau_2) \to \tau_3$ and $\vdash \lambda x.M : \tau_1 \to \tau_2$, then $f \bullet (\lambda x.M)$ is a redex; likewise for $\vdash f' : \tau_1 \to \tau_2$ and $f \bullet f'$. In addition to forming the base case of type safety for our functional programming languages, these restrictions eliminate the possibility for such pathological primitive function types as $\forall\alpha.\forall\beta.\alpha \to \beta$ because there is no way to generalize to such polymorphic types from the concrete defining application terms that we are required to provide for each primitive function. The typing of constructors is specified by type axioms of the same form as those for primitive functions.

Our reasoning with type derivations in this chapter will make use of additional rules, called *structural* rules, given in Figure 6.5. The rule *Weaken* allows us to add irrelevant type assignments to an assumption without disturbing the conclusion; the rule *Permute* lets us reorder the type assignments within a type environment at will. These rules are actually theorems of the given type systems; properly stated, they give us the existence of a derivation for the conclusion from a derivation of the premise. These theorems are established by induction over the structure of type derivations. We omit the details, but we note for example that for *Weaken* the axiom *Var* forms the interesting base case, and that the induction steps typically work because weakened type assumptions on the subterms are just carried through from the premises to the conclusions of the rules.

The safety of the Hindley/Milner type system rests on the assumption that the primitive functions do not go wrong on arguments of the correct type: this means that when they terminate, they really do produce as results terms having the types implied by their declarations. In our calculus, which has multistep δ-rules, we actually need the stronger assumption that every δ-rewriting preserves types; this is a condition on the the constant terms N_f and so forth that appear in the definitions of primitive functions. The proof of soundness is essentially an induction on reductions showing that well-typed terms not only contain no applications of primitive functions to expressions of inappropriate type but also can never produce such an application by reduction. The key component of this proof is the proof of *subject reduction*, the property that any type derivable for a term is also derivable for any of its reducts. The contrapositive statement of this property is that the antecedent of an ill-typed term must also have been ill-typed; there is therefore no way to arrive at an ill-typed primitive application from a well-typed starting term.

A key part of a proof of subject reduction, in turn, is a *substitutivity* lemma establishing that the β-rule preserves well-typing. In the basic functional programming language with only the β- and δ-rules this is almost the whole of subject reduction, but in the treatment of our extended calculi to follow we have many additional cases to consider.

One further aspect of the Hindley/Milner type system is noteworthy for the developments in this chapter: it works equally well for call-by-name and call-by-value versions of the underlying lambda-calculus. Informally, this property may be understood as deriving from the fact that the type system infers the possible consequences of an application based on approximate information that does not capture the *time* of reduction. Formally, the soundness result for the call-by-name calculus is a consequence of soundess for call-by-value by the following reasoning: Every call-by-value β-redex is also a call-by-name β-redex, therefore a stuck term in the call-by-name calculus is also stuck in the call-by-value calculus. The type system is sound for call-by-value, therefore the term in question must be ill-typed, a property that is independent of the evaluation order of the calculus. Thus a stuck call-by-name term is ill-typed. This establishes soundness for the call-by-name calculus. This form of reasoning depends on establishing the subject-reduction property for both calculi.

6.2 What should a type system for a lambda-calculus with assignments do?

The assignment constructs that we have added to the basic lambda-calculus present several new challenges for a type system. The most important of these, and the most complex to address, is to assure that the purification of the result of a store-computation never becomes stuck. Encountering a stuck purification in the calculus corresponds informally to an attempt to export observation of a store outside its lifetime; only terms that can be interpreted independently of a particular store can be purified in the calculus.[1]

Besides assuring that purification of well-typed terms always proceeds to completion, a type system for lambda-calculi with assignment can also ease a couple of less important nuisances present in the untyped calculi. As discussed in Chapter 2, the untyped calculi cannot enforce a requirement that store-variables involved in a particular store-computation are local to that computation. This shortcoming prevents us from interpreting the name of a store-variable as a particular location in which a value is stored because it only becomes meaningful to speak of a location with respect to a particular store. If non-local names can be used to access a store, the interpretation of such a name must be associated with the point of use, not the point of definition.

A type system typically performs a static analysis that approximates a dynamic behavior of a program. In the basic Hindley-Milner type system, this analysis tracks which types of values might be supplied to primitives through all the twists and turns of lambda-abstractions and applications; a type system for a lambda-calculus with assignment ought correspondingly to track locality of declaration for uses of store-variables. We will see below that this analysis can be carried out in a straightforward fashion.

With a type system to enforce that only locally-declared store-variables affect a store-computation, it is possible to restore our originally-intended semantics identifying store-variables with locations: now there is only one store relevant for the interpretation of such names.

6.3 The Chen/Odersky type system for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$

The type system of Chen and Odersky (given both in [Chen and Odersky, 1993] and in [Chen and Odersky, 1994]) was proposed to provide a sound typing mechanism for λ_{var}, the predecessor of $\lambda[\beta\delta!eag]$. Many of the

[1]But not nearly all such independently-meaningful terms can be purified in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. The language-feature design proposed by Launchbury and Peyton Jones [Launchbury and Peyton Jones, 1994] allows far more such terms than our present calculi; we discuss the difficulties of modeling their proposal as a lambda-calculus with assignment in Section 6.5.

$$
\begin{array}{lll}
\tau & \in & \textit{Applicative types} \\
\theta & \in & \textit{Types} \\
\sigma & \in & \textit{Type schemes} \\
\alpha & \in & \textit{Type variables} \\
\underline{\alpha} & \in & \textit{Applicative type variables} \\
 & & \\
\tau & ::= & \underline{\alpha} \mid \tau_1 \rightarrow \tau_2 \mid \kappa^n\, \tau_1 \ldots \tau_n \\
\theta & ::= & \tau \mid \alpha \mid \theta_1 \rightarrow \theta_2 \mid \kappa^n\, \theta_1 \ldots \theta_n \mid \mathbf{Ref}\,\theta \mid \mathbf{Cmd}\,\theta \\
\sigma & ::= & \theta \mid \forall\alpha.\sigma \mid \forall\underline{\alpha}.\sigma
\end{array}
$$

Figure 6.6: Syntax of types for Chen/Odersky type system for $\lambda[\beta\delta!eag]$

$(\cdots$ Figure 6.4 $\cdots)$

$$(\mathit{Seq})\ \frac{\Gamma \vdash M_1 : \mathbf{Cmd}\,\theta' \quad \Gamma, x : \theta' \vdash M_2 : \mathbf{Cmd}\,\theta}{\Gamma \vdash M_1 \rhd x \cdot M_2 : \mathbf{Cmd}\,\theta}$$

$$(\mathit{Unit})\ \frac{\Gamma \vdash M : \theta}{\Gamma \vdash \uparrow M : \mathbf{Cmd}\,\theta}$$

$$(\mathit{Ref})\ \frac{}{\Gamma \vdash v : \mathbf{Ref}\,\theta}\ (v : \mathbf{Ref}\,\theta \in \Gamma)$$

$$(\mathit{New})\ \frac{\Gamma, v : \mathbf{Ref}\,\theta' \vdash M : \mathbf{Cmd}\,\theta}{\Gamma \vdash \nu v\, M : \mathbf{Cmd}\,\theta}$$

$$(\mathit{Assign})\ \frac{\Gamma \vdash M_1 : \mathbf{Ref}\,\theta \quad \Gamma \vdash M_2 : \theta}{\Gamma \vdash M_1 := M_2 : \mathbf{Cmd}\,()}$$

$$(\mathit{Read})\ \frac{\Gamma \vdash M : \mathbf{Ref}\,\theta}{\Gamma \vdash M? : \mathbf{Cmd}\,\theta}$$

$$(\mathit{Pure})\ \frac{\Gamma_\tau \vdash M : \mathbf{Cmd}\,\tau}{\Gamma_\tau \vdash \mathbf{pure}\,M : \tau}\ (M \equiv S^{laz}[\uparrow N])$$

$$(\mathit{Gen'})\ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\underline{\alpha}.\sigma}\ (\underline{\alpha} \notin \mathit{fv}\,\Gamma)$$

$$(\mathit{Spec'})\ \frac{\Gamma \vdash M : \forall\underline{\alpha}.\sigma}{\Gamma \vdash M : [\tau/\underline{\alpha}]\sigma}\ (\underline{\alpha} \notin \mathit{fv}\,\tau)$$

Figure 6.7: Chen/Odersky type system for $\lambda[\beta\delta!eag]$

features of the Chen/Odersky system derive from the work of Swarup, Reddy, and Ireland on the Imperative Lambda Calculus [Swarup *et al.*, 1991]. The Chen/Odersky type system, however, shares with the Imperative Lambda Calculus a fundamental flaw in its proof of type safety. We introduce the Chen/Odersky system here both in order to point out its flaw before fixing it and also because some of its structure will be used in the repaired type system given in Section 6.4.

The Hindley/Milner type system guarantees that well-typed functional programs do not get stuck because of mismatches between an expression's type and the type required by its context of use. A type system for a lambda-calculus with assignment should have a similar property with respect to the added features: commands, store-variables, and purification. There are two issues at work here. First, since the sequencing construct $\rhd$ of $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$ passes a command result to the next command, the type system must duplicate the services it already provides for function application in the context of command sequencing. Second (and more difficult), the type system must ensure the safety of **pure**-expressions. In order to accomplish this, the Chen/Odersky type system introduces new type constructors **Ref** and **Cmd** to distinguish reference types and command types, respectively, from all other types.

The type system attempts to guarantee the safety of purification by requiring that a **pure**-expression can only be type-correct if the type environment assumes only functional (non-**Ref** , non-**Cmd**) types for free variables in the purified expression. This requirement is expressed by stratifying the collection of types so that purely applicative types can be distinguished syntactically from types involving imperative constructs. The type system makes use of this syntactic distinction by limiting the type of imperative computation results to the applicative layer.[2] Although the particular use of this stratification by the Chen/Odersky type system

[2]The application of the stratification technique to lambda-calculi with assignment derives from [Swarup *et al.*, 1991].

is insufficient to achieve its goal of assuring the safety of attempted purifications, the basic stratified structure works and is retained in the type system LPJ^- which we will introduce in Section 6.4.

The syntax of type expressions in the Chen/Odersky system is given in Figure 6.6, and the additional typing rules are given in Figure 6.7.

Several features of this type system require special comment. First, it is assumed that there are no constants of reference or command type, nor are there primitive functions involving such types (except perhaps as projections from constructed types). Second, the types in Figure 6.6 incorporate a special class of type variables that are only capable of being instantiated by types in the applicative layer. Corresponding to this class of type variables are special versions of the typing rules *Gen* and *Spec*. Third, the rule *Pure* requires that the types assumed for every variable in the type environment be restricted to the applicative layer. This restriction is notated by annotating the meta-symbol for type environments with a subscript τ. Rule *Pure* also requires that the result type of the command be an applicative type. The side condition $M \equiv S^{laz}[\uparrow N]$ on rule *Pure* is intended by Chen and Odersky to assure that the **pure**-expression returns *some* value.[3] It should be noted that the restriction in question is very easily satisfied by writing $\mathbf{pure}\,M \rhd x \cdot \uparrow x$ instead of $\mathbf{pure}\,M$ if M does not have the required form; it is not clear that the restriction is really required, but it is certainly harmless.

It is in fact the formulation of the rule *Pure* that trips up the Chen/Odersky type system. As we discussed in Section 6.1, the property of subject reduction is a crucial step in establishing the soundness of a type system according to standard proof techniques; it is subject reduction that fails for the Chen/Odersky system. The culprit in this failure is the restriction in the rule *Pure*, which requires a type environment that assumes only τ types. Unfortunately, the set of free variables that must be assigned in such an environment is subject to change upon substitution during a β-reduction. This issue can be traced as far back as the work of Reynolds on the syntactic control of interference [Reynolds, 1978], as pointed out in the introductory sections of [O'Hearn *et al.*, 1995].

The following counterexample to the Chen/Odersky is similar to those given in the literature on syntactic control of interference; we have adapted it to the syntax of $\lambda[\beta\delta!eag]$. We consider the term

$$\mathbf{pure}\,\nu w.w := 2; \uparrow((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)),$$

which can be given a type derivation as follows. We present first a subtree of the type derivation in consideration of our limited page width. This subtree is presented for completeness only; the interesting part of the derivation follows afterwards.

$$\dfrac{\dfrac{\dfrac{}{\vdash fst : \forall\alpha.\forall\beta.\langle\alpha,\beta\rangle \to \alpha}}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash fst : \langle \mathrm{Int}, \mathbf{Ref}\,\mathrm{Int}\rangle \to \mathrm{Int}} \quad \dfrac{}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash \langle 1,w\rangle : \langle \mathrm{Int}, \mathbf{Ref}\,\mathrm{Int}\rangle}}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash fst \bullet \langle 1,w\rangle : \mathrm{Int}} \qquad (6.1)$$

The interesting part of the type derivation is the following:

$$\dfrac{\dfrac{\dfrac{}{w : \mathbf{Ref}\,\mathrm{Int} \vdash w := 2 : \mathbf{Cmd}\,()} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathrm{Int} \vdash x : \mathrm{Int}}}{x : \mathrm{Int} \vdash \uparrow x : \mathbf{Cmd}\,\mathrm{Int}}}{x : \mathrm{Int} \vdash \mathbf{pure}\uparrow x : \mathrm{Int}}}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash \lambda x.\mathbf{pure}\uparrow x : \mathrm{Int} \to \mathrm{Int}} \quad (6.1)}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash ((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)) : \mathrm{Int}}}{w : \mathbf{Ref}\,\mathrm{Int}, z : () \vdash \uparrow((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)) : \mathbf{Cmd}\,\mathrm{Int}}}{w : \mathbf{Ref}\,\mathrm{Int} \vdash w := 2; \uparrow((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)) : \mathbf{Cmd}\,\mathrm{Int}}}{\dfrac{\vdash \nu w.w := 2; \uparrow((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)) : \mathbf{Cmd}\,\mathrm{Int}}{\vdash \mathbf{pure}\,\nu w.w := 2; \uparrow((\lambda x.\mathbf{pure}\uparrow x) \bullet (fst \bullet \langle 1,w\rangle)) : \mathrm{Int}}} \quad .$$

[3]The definition of $S^{laz}[]$ is given in Figure 2.11. It is used here as a handy syntactic abbreviation—its role in the statement of the rule has nothing to do with laziness.

Note that the inner **pure**-expression is judged typable in a type environment containing a type assignment only for the free lambda-bound variable x. The assumption is that x has type Int, which is an applicative type, so the side condition is satisfied.

Now let us carry out the β-reduction, obtaining the term

$$\mathbf{pure}\, \nu w.w := 2;\ \uparrow \mathbf{pure}\uparrow (\mathit{fst} \bullet \langle 1,w \rangle).$$

If we try to construct a type derivation for this term, we run into trouble: the inner **pure**-expression now contains a free occurrence of the store-variable w, which must be accounted for in the type environment present in the application of rule *Pure*. The side condition will thus be violated, since the type of w is constrained by its origin in the outer **pure** to have type **Ref** Int. The β-reduction step has caused a loss of typability, and hence subject reduction does not hold for the Chen/Odersky type system.

The rather fussy subterm $\mathit{fst} \bullet \langle 1,w \rangle$ appearing in this counterexample may appear puzzling, but this construction is necessary in order to pin the blame on the side condition rather than on the condition that the *result* type of the **pure**-expression belongs to the applicative layer. Putting the offending store-variable into the ignored slot of a projection from a product type runs afoul of the side condition (because *all* free variables must be assigned types in the type environment, and hence must be assigned *applicative* types) but satisfies the result-type condition (because the type of w is, in fact, ignored).

This counterexample to subject reduction actually only invalidates a proof technique for type soundness; its existence does not necessarily disprove type soundness itself. In fact, the counterexample given reduces successfully to the answer 1. When a proof technique fails without suggesting that the theorem itself is false, we are faced with a choice between finding another proof technique for establishing the same theorem or modifying the theorem. In our repair of the situation in Section 6.4 we will modify the theorem by considering a different type system that is not sensitive to the changes in free-variable sets that come with substitution. We rationalize that subject reduction is more than just a proof technique—it is intimately tied to our informal understanding that a program denotes a value and that that value has an unchanging type.

Uninitialized store-variables

One technical detail of [Chen and Odersky, 1994] that we will treat differently here has to do with the way in which errors due to uninitialized store-variable are dealt with. Chen and Odersky introduce a reduction $\nu v.v? \triangleright x \cdot P \rightarrow \nu v.v? \triangleright x \cdot P$ in order to make such terms diverge rather than get stuck, since the type system does not, in fact, prohibit them. As a matter of taste, we choose instead to stipulate that all store-variables are initialized immediately upon declaration (as in $\nu v.v := M;\ P$), and that the check for this restriction is made independently of the type system (perhaps by modifying the syntax of the language).

This solution to the unitialized-variable problem creates a minor secondary problem: the original reduction rules in Figures 2.7 and 2.9 allow the creation of intermediate forms that do not satisfy our stipulation that all allocated store-variables be immediately initialized. In particular, the rule *bubble-assign* can bubble a store-variable read into the position between the allocation and the initialization, as in the reduction

$$\begin{aligned} \nu v.v := M_1;\ w? \triangleright x_2 \cdot M_2 &\rightarrow \nu v.w? \triangleright x_2 \cdot v := M_1;\ M_2 \\ &\rightarrow w? \triangleright x_2 \cdot \nu v.v := M_1;\ M_2. \end{aligned}$$

The way out of this minor dilemma is to note that, if the first reduction above is permitted by virtue of v and w being distinct store-variables, then so is the second. In fact, a standard reduction will perform these reductions consecutively or not at all, so the condition that the declaration of a store-variable must be followed immediately by an initial assignment to it is already maintained by the reduction semantics.

Another minor issue arising from our insistence that store-variables be immediately initialized is that building certain recursive store structures becomes awkward. For example, it is perfectly reasonable in an untyped calculus to set up two store variables having each other as assigned values, as in

$$\nu v.\nu w.v := w;\ w := v.$$

It should be noted, however, that this term can have no finite type, because the reference type of each of v

$$\begin{array}{rcl} \kappa^n & \in & \textit{Type constructors} \\ \theta & \in & \textit{Type} \\ \sigma & \in & \textit{Type schemes} \\ \alpha & \in & \textit{Type variables} \end{array}$$

$$\begin{array}{rcll} \theta & ::= & \kappa^n\, \theta_1 \ldots \theta_n \mid \alpha & \\ & \mid & \theta \to \theta' & \textit{applicative types} \\ & \mid & \theta\, \mathbf{Ref}\, \theta' & \textit{reference to value of type } \theta' \textit{ in store of type } \theta \\ & \mid & \theta\, \mathbf{Cmd}\, \theta' & \textit{command yielding value of type } \theta' \textit{ operating on store of type } \theta \\ \sigma & ::= & \theta \mid \forall \alpha . \sigma & \end{array}$$

Figure 6.8: Syntax of types for *LPJ*

and w is defined in terms of the other without any foundation to the inference. The type of each of v and w must thus be a solution τ to the equation $\tau = \mathbf{Ref\ Ref}\ \tau$. Although it is possible to devise reasonable type systems which can express and infer such types (as is done, for example, in [Amadio and Cardelli, 1991]), we do not wish to become involved in this additional complexity when our main purpose is to account for **pure**. In the presence of constructed types it is possible to achieve the desired effect of the example just given by interposing constructed values into the circle of references. For example, the type of nodes in the classic data structure of doubly-linked lists (which inherently requires circular memory structures) is naturally represented by a union of two constructed types: one for nil and one for a reference to the nodes fore and aft.

A reasonable way to mitigate this situation is to introduce into the calculus a construct permitting the declaration of several mutually-recursive store-variables at once as in Haskell's **let** or Scheme's **letrec**. The mutual recursion would apply to the initializing expressions, which would be interpreted in an environment in which all the simultaneously-declared store-variables were already defined. We do not pursue this matter any further here because the language loses no essential expressivity by the omission; indeed, ILC also lacks such a construct. This discussion of store-variable initialization applies equally to the type system LPJ^- to be presented in the next section.

6.4 The type system LPJ^- for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$

As we saw in Section 6.3, the approach to typing purification expressions by confining assumed types for free variables of the store computation to the applicative layer has a serious flaw. In this section we propose a new type system for lambda-calculi with assignment that is free of this flaw, and we prove it safe for both $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

Our new type system is a restriction of one proposed by Launchbury and Peyton Jones [Launchbury and Peyton Jones, 1994]; we will refer to their system as *LPJ*. The main feature of *LPJ* is that the imperative types depend on an additional type parameter that identifies the particular thread of store-computation in which a store-variable or command is valid. The typing rules for command sequences force an entire store-computation expression to have an identical value for this type parameter; unique values for the type parameter for each thread are generated at **pure** boundaries by requiring the type of the thread to be generic in the thread type parameter. Our proposed type system, which we will call LPJ^-, restricts the *LPJ* purification rule, but only in the result type rather than in the assumed types which proved so troublesome for the Chen/Odersky system and for ILC.

In order to give an uncluttered view of the design principles involved, we first present the full *LPJ* system, after which we give the modification required to yield LPJ^- followed by the proof of type soundness for LPJ^-. Figures 6.8 and 6.9 present the type system *LPJ*. Figure 6.9 gives only the inference rules peculiar to the command and assignment constructs of the calculi; the type rules for the functional core of the calculus are retained from Figure 6.4.

The central technique of *LPJ* is adapted from the way in which higher-order polymorphic lambda-calculus [Girard, 1990] encodes existentially-quantified types as universally-quantified types (see [Girard *et al.*, 1989]). The Launchbury/Peyton Jones type system uses this technique to require that the argument to **pure** be generic in the type associated with the store. At the technical level in the type system, this requirement has the effect

$$(Seq)\ \frac{\Gamma \vdash M_1 : \theta\,\mathbf{Cmd}\,\theta' \quad \Gamma, x : \theta' \vdash M_2 : \theta\,\mathbf{Cmd}\,\theta''}{\Gamma \vdash M_1 \rhd x \cdot M_2 : \theta\,\mathbf{Cmd}\,\theta''}$$

$$(Read)\ \frac{\Gamma \vdash M : \theta\,\mathbf{Ref}\,\theta'}{\Gamma \vdash M? : \theta\,\mathbf{Cmd}\,\theta'}$$

$$(Unit)\ \frac{\Gamma \vdash M : \theta'}{\Gamma \vdash \uparrow M : \theta\,\mathbf{Cmd}\,\theta'}$$

$$(Assign)\ \frac{\Gamma \vdash M_1 : \theta\,\mathbf{Ref}\,\theta' \quad \Gamma \vdash M_2 : \theta'}{\Gamma \vdash M_1 := M_2 : \theta\,\mathbf{Cmd}\,()}$$

$$(Ref)\ \frac{}{\Gamma \vdash v : \theta\,\mathbf{Ref}\,\theta'}\ (v : \theta\,\mathbf{Ref}\,\theta' \in \Gamma)$$

$$(Pure)\ \frac{\Gamma \vdash M : \forall\alpha.\alpha\,\mathbf{Cmd}\,\theta}{\Gamma \vdash \mathbf{pure}\,M : \theta}\ \left(\begin{array}{l}\alpha \notin fv\,\theta, \\ M \equiv S^{laz}[\uparrow N]\end{array}\right)$$

$$(New)\ \frac{\Gamma, v : \theta\,\mathbf{Ref}\,\theta' \vdash M : \theta\,\mathbf{Cmd}\,\theta''}{\Gamma \vdash \nu v\,M : \theta\,\mathbf{Cmd}\,\theta''}$$

Figure 6.9: Type inference rules for *LPJ*

of requiring that the type variable be absent from the type of the returned value of the **pure**. Since occurrences of the type variable correspond to occurrences of the store, the purified value must be free of references to that particular thread. This device goes beyond the level of polymorphism present in the Hindley/Milner, since it effectively gives the **pure** operator the type scheme $\forall\beta.(\forall\alpha.\alpha\,\mathbf{Cmd}\,\beta) \to \beta$, which does not have all its quantifiers at the outermost level, but Launchbury and Peyton Jones argue that the negative consequences of this minor foray into second-order polymorphism are minimal.

Figure 6.8 gives the syntax of *LPJ* type expressions. There are two major contrasts to the Chen/Odersky system to be noted. First, there is only one stratum and one kind of type variable. Second, the imperative type constructors take an extra parameter, which we write before the keyword as in θ_1 **Ref** θ_2 and θ_1 **Cmd** θ_2.[4] This extra parameter is the unique type of the store thread in which the reference or command is valid; in expressions of applicative type it will always be a type variable.

The type rules of *LPJ* are given in Figure 6.9. These rules all follow the same pattern as the corresponding Chen/Odersky rules, with two exceptions. First, the rules for typing compound commands carry along the new store-thread types and require that they match. Second, the purification rule *Pure* implements the trick alluded to above: a command can only be purified if its type is generic in the store-thread type that must be matched by every component of the command. Note that this does *not* prohibit the presence of a different store-thread type as a component of the result type τ as the Chen/Odersky system would. This leniency is a major contribution of the Launchbury/Peyton Jones type system.

Unfortunately, it is precisely this leniency that prohibits us from using *LPJ* as a substitute for the Chen/Odersky system. Our untyped calculi do not under any circumstances permit a command or store-variable to be the result of a store-computation, so *LPJ* is necessarily unsound for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. We clearly need to restrict the *LPJ* rule *Pure* so that these types of constructs will never be considered for purification.[5]

Our solution is to construct a hybrid of *LPJ* with the stratified type structure characteristic of ILC and the Chen/Odersky type system. Figure 6.10 gives the syntax of types for the hybrid type system *LPJ*$^-$. The applicative types are denoted by τ, the more general types including the store-variable and command types by θ, and a special class of types specific to the purification rule by ψ. The ψ types are exactly the types of the constructs that can be completely purified by the reduction rules *pure-eager* and *pure-lazy*; the modified *Pure* rule given in Figure 6.11 restricts the type inferred for the result of the command sequence to be purified to ψ types. Aside from this restriction, *LPJ*$^-$ is just *LPJ*.

The safety provided by *LPJ*$^-$ is a matter for the proofs to follow, but we first note that the problem raised by the counterexample given in Section 6.3 is not present, because we have no condition on the free variables of the store-computation's result expression, but only on its *type* (which must be a ψ type). We have taken advantage of an informal parametricity property of our type system: the type of an input that actually contributes

[4] We adopt this infix notation (which is ours) in order to reduce the need for parentheses that arise with the use of the usual style of prefix placement of the type constructor. We cite the function-type constructor as a precedent.

[5] It is certainly possible to attempt to allow the *LPJ* type system's attractive leniency to influence the design of an untyped lambda-calculus with assignment with respect to which *LPJ* can be proved sound. We report on difficulties with this attempt in Section 6.5.

$$
\begin{array}{lcl}
\kappa & \in & \textit{Type constructors} \\
\tau & \in & \textit{Applicative types} \\
\theta & \in & \textit{Types} \\
\psi & \in & \textit{OK types} \\
\sigma & \in & \textit{Type schemes} \\
\underline{\alpha} & \in & \textit{Applicative type variables} \\
\alpha & \in & \textit{Type variables}
\end{array}
$$

$$
\begin{array}{lcll}
\tau & ::= & \kappa^n\,\tau_1 \ldots \tau_n \mid \underline{\alpha} & \\
 & \mid & \tau \to \tau' & \textit{applicative types} \\
 & & & \\
\theta & ::= & \alpha \mid \tau \mid \theta \to \theta' & \\
 & \mid & \kappa^n\,\theta_1 \ldots \theta_n & \\
 & \mid & \tau\,\mathbf{Ref}\,\tau' & \textit{reference to value of type } \tau' \textit{ in store of type } \tau \\
 & \mid & \tau\,\mathbf{Cmd}\,\tau' & \textit{command yielding value of type } \tau' \textit{ operating on store of type } \tau \\
\psi & ::= & \tau \mid \theta \to \psi & \\
 & \mid & \kappa^n\,\psi_1 \ldots \psi_n & \\
\sigma & ::= & \tau \mid \forall\alpha.\sigma \mid \forall\underline{\alpha}.\sigma &
\end{array}
$$

Figure 6.10: Syntax of types for *LPJ*$^-$

$$
(\textit{Pure})\,\frac{\Gamma \vdash M : \forall\alpha.\alpha\,\mathbf{Cmd}\,\psi}{\Gamma \vdash \mathbf{pure}\,M : \psi}\left(\begin{array}{l}\alpha \notin \textit{fv}\,\psi, \\ M \equiv S^{laz}[\uparrow N]\end{array}\right)
$$

Figure 6.11: Modified inference rule for *LPJ*$^-$

$$
(\textit{Gen'})\,\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\underline{\alpha}.\sigma}\,(\underline{\alpha} \notin \textit{fv}\,\Gamma) \qquad (\textit{Spec'})\,\frac{\Gamma \vdash M : \forall\underline{\alpha}.\sigma}{\Gamma \vdash M : [\tau/\underline{\alpha}]\sigma}\,(\underline{\alpha} \notin \textit{fv}\,\tau)
$$

Figure 6.12: Additional rules for *LPJ*$^-$

to the output must be reflected in the output type.

We now begin the proof that *LPJ*$^-$ is safe for $\lambda[\beta\delta!\textit{eag}]$ and $\lambda[\beta\delta!\textit{laz}]$. We first prove the substitutivity lemma that supports the β-reduction case in the proof of subject reduction. We formulate the lemma in terms of type schemes both because it makes a better induction hypothesis and because we need to take let-polymorphism into account.

Lemma 6.4.1 (Substitutivity) *In LPJ*$^-$*, if* $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$ *and* $\Gamma \vdash M_2 : \sigma_1$*, then* $\Gamma \vdash [M_2/x]\,M_1 : \sigma_2$.

Proof: We carry out an induction on the structure of the type derivation for the hypothesised judgment $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$.

The only interesting base case is $M_1 \equiv x$, in which case $[M_2/x]\,M_1 \equiv M_2$ and the lemma follows from the hypothesis $\Gamma \vdash M_2 : \sigma_1$. In all the other base cases ($M_1 \equiv v$, $M_1 \equiv y \not\equiv x$, $M_1 \equiv f$, $M_1 \equiv c^0$) the lemma is easily seen to hold: no substitution actually takes place. In these cases there is no occurrence of x in M_1, and the desired conclusion follows from the hypotheses by omitting the irrelevant type assignment for x from the type judgment.

We take as induction hypothesis the statement of the lemma, but with M_1 restricted to terms of lower height than the term under consideration.

Case (1) Rule *App*: $M_1 \equiv N_1 \bullet N_2$.

In this case the assumed typing for M_1 must be supported by an inference tree of the form

$$\frac{\Gamma, x\colon \sigma_1 \vdash N_1 : \theta_3 \to \theta_2 \quad \Gamma, x\colon \sigma_1 \vdash N_2 : \theta_3}{\Gamma, x\colon \sigma_1 \vdash N_1 \bullet N_2 : \theta_2}\;.$$

The assumptions that we are forced to make in this type derivation, along with the induction hypothesis, are sufficient to establish $\Gamma \vdash [M_2/x]\,N_1 : \theta_3 \to \theta_2$ and $\Gamma \vdash [M_2/x]\,N_2 : \theta_2$. We use these type judgments as assumptions in the following type derivation, which establishes the lemma in this case:

$$\frac{\Gamma \vdash [M_2/x]\,N_1 : \theta_3 \to \theta_2 \quad \Gamma \vdash [M_2/x]\,N_2 : \theta_2}{\Gamma \vdash [M_2/x]\,N_1 \bullet [M_2/x]\,N_2 (\equiv [M_2/x]\,N_2 \bullet N_2) : \theta_2}.$$

Case (2) Rule *Abs*: $M_1 \equiv \lambda y.N$.

In this case we must have a type derivation of the form

$$\frac{\Gamma, x\colon \sigma_1, y\colon \theta_3 \vdash N\colon \theta_4}{\Gamma, x\colon \sigma_1 \vdash \lambda y.N\colon (\theta_3 \to \theta_4)(\equiv \theta_2)}.$$

The assumption of this derivation together with the induction hypothesis then gives the assumption of the following type derivation, which establishes the lemma in this case. The only twist is that we have used the fact that the type assignments within type environments may be permuted:

$$\frac{\Gamma, y\colon \theta_3 \vdash [M_2/x]\,N\colon \theta_4}{\Gamma \vdash \lambda y.[M_2/x]\,N (\equiv [M_2/x]\,(\lambda y.N))\colon (\theta_3 \to \theta_4)(\equiv \theta_2)}.$$

This pattern of proof requires only the fact that substitution commutes with term-construction. The rest of the cases based on type rules for syntactic constructs follow this pattern; we treat the case for rule *Pure* in full because of the presence of the condition on the result type.

Case (3) Rule *Gen*:

In this case, we have a type derivation of the form

$$\frac{\Gamma, x\colon \sigma_1 \vdash M_1 : \sigma_2}{\Gamma, x\colon \sigma_1 \vdash M_1 : \forall\alpha.\sigma_2}\,(\alpha \notin fv\,\Gamma, x\colon \sigma_1)\;.$$

The induction hypothesis (recall that the induction is over the structure of type derivations, not that of terms) gives us the existence of a derivation of the judgment $\Gamma \vdash [M_2/x]\,M_1 : \sigma_2$. Now if the type variable α is not free in the type environment $\Gamma, x\colon \sigma_1$, it certainly cannot be free in the strictly smaller type environment Γ. Hence the side condition for rule *Gen* is satisfied, and we can infer $\Gamma \vdash M_1 : \forall\alpha.\sigma_2$ by rule *Gen*.

Case (4) Rule *Spec*:

In this case the root node of our given type derivation takes the form

$$\frac{\Gamma, x\colon \sigma_1 \vdash M\colon \forall\alpha.\sigma_3}{\Gamma, x\colon \sigma_1 \vdash M\colon [\theta_3/\alpha]\,\sigma_3 (\equiv \sigma_2)}\,(\alpha \notin fv\,\theta_3)\;.$$

Using the assumption of this type derivation, the induction hypothesis gives us the judgment $\Gamma \vdash [M_2/x]\,M\colon \forall\alpha.\sigma_3$, from which the desired conclusion $\Gamma \vdash [M_2/x]\,M\colon [\theta_3/\alpha]\,\sigma_3$ follows by type rule *Spec* (the side condition having already been established by the fact that it must already hold for *Spec* to have been applied in the first place).

$$\begin{array}{rcl} bns([\,]) & = & \{\} \\ bns(M \triangleright x{\cdot}S^{laz}[\,]) & = & bns(S^{laz}[\,]) \cup \{x\} \\ bns(\nu v.S^{laz}[\,]) & = & bns(S^{laz}[\,]) \cup \{v\} \end{array}$$

Figure 6.13: Bound variables of a store-context

Case (5) Rules *Gen'* and *Spec'*.

The proofs for these cases are similar to those for their counterparts *Gen* and *Spec*.

Case (6) Rule *Let*: $M_1 \equiv \mathbf{let}\, y = N_1 \,\mathbf{in}\, N_2$.

In this case, we have a type derivation of the form

$$\frac{\Gamma, x\colon \sigma_1 \vdash N_1 : \sigma' \quad \Gamma, x\colon \sigma_1, y\colon \sigma' \vdash N_2 : \theta_2}{\Gamma, x\colon \sigma_1 \vdash \mathbf{let}\, y = N_1 \,\mathbf{in}\, N_2 : \theta_2} \quad .$$

Applying the induction hypothesis to the assumptions of this derivation, we obtain the assumptions of the type derivation

$$\frac{\Gamma \vdash [M_2/x]\,N_1 : \sigma' \quad \Gamma, y\colon \sigma' \vdash [M_2/x]\,N_2 : \theta_2}{\Gamma \vdash \mathbf{let}\, y = [M_2/x]\,N_1 \,\mathbf{in}\, [M_2/x]\,N_2 (\equiv [M_2/x]\,(\mathbf{let}\, y = N_1 \,\mathbf{in}\, N_2)) : \theta_2}.$$

Case (7) Rule *Pure*: $M_1 \equiv \mathbf{pure}\, S^{laz}[\uparrow N]$.

In this case we must have a type derivation of the form

$$\frac{\Gamma, x\colon \sigma_1 \vdash S^{laz}[\uparrow N] : \forall\alpha.\alpha\,\mathbf{Cmd}\,\psi}{\Gamma, x\colon \sigma_1 \vdash \mathbf{pure}\, S^{laz}[\uparrow N] : \psi (\equiv \theta_2)}.$$

Using the induction hypothesis with the assumption of this type derivation gives us the assumption of the following type derivation, which establishes this case:

$$\frac{\Gamma \vdash [M_2/x]\, S^{laz}[\uparrow N] : \forall\alpha.\alpha\,\mathbf{Cmd}\,\psi}{\Gamma \vdash \mathbf{pure}[M_2/x]\, S^{laz}[\uparrow N] (\equiv [M_2/x]\,\mathbf{pure}\, S^{laz}[\uparrow N]) : \psi (\equiv \theta_2)}.$$

Having considered all the typing rules, we have now established the lemma. ∎

In addition to the substitutivity lemma, the proof of subject reduction for *LPJ*$^-$ also requires the following lemma on the typing of command sequences. The lemma uses the inductive definition of $bns(S^{laz}[\,])$ given in Figure 6.13 ('$bns()$' is an abbreviation for 'bound names').

Lemma 6.4.2 (Typing command sequences) *In* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *with the LPJ*$^-$ *type system, the type judgment* $\Gamma \vdash S^{laz}[\uparrow M] : \alpha\mathbf{Cmd}\,\theta$ *holds if and only if the judgment* $\Gamma^+ \vdash M : \theta$ *holds, where* Γ^+ *extends* Γ *with type assignments for the variables and store-variables in* $bns(S^{laz}[\,])$.

Proof: We carry out an induction on the structure of $S^{laz}[\,]$.

The base case, in which $S^{laz}[\,] \equiv [\,]$, is established by the instance of the typing rule *Unit*:

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash \uparrow M : \alpha\mathbf{Cmd}\,\theta.}$$

That this inference also works in the reverse direction is implied by the fact that this is the only typing rule that infers a non-schematic type for a term of the form $\uparrow M$.

We now treat the two inductive cases. If $S^{laz}[\,] \equiv M_1 \rhd x \cdot S_1^{laz}[\,]$, the relevant type inference is an instance of the rule *Seq*:

$$\frac{\Gamma \vdash M_1 : \alpha\,\mathbf{Cmd}\,\theta_1 \quad \Gamma, x\colon \theta_1 \vdash S_1^{laz}[\uparrow M] : \alpha\,\mathbf{Cmd}\,\theta}{\Gamma \vdash M_1 \rhd x \cdot S_1^{laz}[\,] : \alpha\,\mathbf{Cmd}\,\theta} \quad .$$

If we are given the right-hand assumption of this inference by means of the induction hypothesis, the conclusion line follows and establishes the "if" direction of the lemma in this case, since the type environment Γ is augmented by a type assignment for the variable x that is added when going from $bns\,(S_1^{laz}[\,])$ to $bns\,(S^{laz}[\,])$. Conversely, if we are given the conclusion line, the fact that we have only one typing rule for terms of the given form implies the existence of derivations for the assumptions. Applying the induction hypothesis to the right-hand assumption gives $\Gamma_1 \vdash M : \theta$, where Γ_1 extends $\Gamma, x\colon \theta_1$ with type assignments for all the names in $bns\,(S_1^{laz}[\,])$. But this last statement is equivalent to saying that Γ_1 extends Γ with type assignments for all the names in $M_1 \rhd x \cdot S_1^{laz}[\,]$, which is what we have to show.

The induction case in which $S^{laz}[\,] \equiv \nu v.S_1^{laz}[\,]$ follows by similar reasoning modulo use of the inference rule *New* instead of *Seq*. ∎

We now state and prove the subject reduction property for LPJ^-.

Lemma 6.4.3 (Subject reduction) *In* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *under the type system* LPJ^-, *if* $\Gamma \vdash M : \theta$ *and* $M \to N$, *then* $\Gamma \vdash N : \theta$.

Proof: Before we get to the heart of the proof, there are a couple of reasoning principles to note in preparation.

First, it is sufficient to consider only the cases in which M is a redex. We note that (1) every subtree of a valid type derivation is a valid type derivation and (2) the typing rule for each syntactic construct depends on a type judgment for each of that construct's subterms. Thus if we are given a valid type derivation for a term containing a deeply embedded redex, there will exist some subtree of that type derivation that establishes a type for the redex.

Second, we need to account for the interaction of the generalization and specialization rules with our proof technique. For each redex, we start with the hypothesis that there exists a valid type derivation for that redex. We then observe that the syntax-directed typing rule for the root syntactic construct of the redex requires that certain judgments concerning subterms of the redex must be valid; finally, we use these judgments to construct a valid type derivation for the reduct.

It is possible, however, that there may be more than one valid type derivation for the term in question, since the rules *Spec* and *Gen* relate judgments concerning the same term in the premise and conclusion. There may thus be some intervening applications of these polymorphism-related rules between the root of the derivation tree and the first application of the syntactically required typing rule. Nonetheless, it should be clear from inspection of the typing rules that every type derivation for a term must contain a unique instance of a syntax-directed typing that is closest to the root of the derivation. It is to this closest syntax-directed rule application that we are referring in the rest of the proof when we say that a type derivation must have a certain form. Furthermore, since specializations and generalizations only depend on the type environment and adjudged type (and not on the typed term) it is possible to reapply them after proving that a redex and its reduct have the same type. This reasoning lets us complete the round trip from the original type derivation for the redex to the trimmed type derivation with a syntax-directed rule at the root to a trimmed type derivation for the reduct and finally back to a full derivation for the reduct having the original type.

We consider one case for each reduction rule in $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$.

Case (1) Rule β: $(\lambda x.M_1) \bullet M_2 \;\to\; [M_2/x]\,M_1$.

A β-redex must have a type derivation of the form

$$\dfrac{\dfrac{\Gamma, x\colon \theta_1 \vdash M_1 : \theta_2}{\Gamma \vdash \lambda x.M_1 : \theta_1 \rightarrow \theta_2} \quad \Gamma \vdash M_2 : \theta_1}{\Gamma \vdash (\lambda x.M_1) \bullet M_2 : \theta_2} \quad .$$

The assumptions we are forced to make if we assume the existence of this type derivation are exactly the hypotheses of the substitution lemma (Lemma 6.4.1), which allows us to conclude that $\Gamma \vdash [M_2/x]\, M_1 : \theta_2$, which is the required judgment concerning the type of the reduct.

Case (2) Rule δ.

This case is covered by our assumptions concerning the types of primitives defined in the calculus.

Case (3) Rule *assoc*: $(M_1 \triangleright x_2 \cdot M_2) \triangleright x_3 \cdot M_3 \;\rightarrow\; M_1 \triangleright x_2 \cdot M_2 \triangleright x_3 \cdot M_3$.

A type derivation for an *assoc*-redex must have the form

$$\dfrac{\dfrac{\Gamma \vdash M_1 : \alpha\mathbf{Cmd}\,\theta_1 \quad \Gamma, x_2 : \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash M_1 \triangleright x_2 \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \qquad \Gamma, x_3 : \theta_2 \vdash M_3 : \alpha\mathbf{Cmd}\,\theta_3}{\Gamma \vdash M_1 \triangleright x_2 \cdot M_2 \triangleright x_3 \cdot M_3 : \alpha\mathbf{Cmd}\,\theta_3} \quad .$$

The judgment $\Gamma, x_3 : \theta_2 \vdash M_3 : \alpha\mathbf{Cmd}\,\theta_3$ can be weakened to $\Gamma, x_2 : \theta_1, x_3 : \theta_2 \vdash M_3 : \alpha\mathbf{Cmd}\,\theta_3$; the variable x_2 cannot occur free in M_3 because M_3 is outside the scope of x_2 in the redex, and we observe Barendregt's convention that bound and free variables have distinct names. We can thus use the assumptions of the type derivation above to establish the following type derivation for the reduct:

$$\dfrac{\Gamma \vdash M_1 : \alpha\mathbf{Cmd}\,\theta_1 \qquad \dfrac{\Gamma, x_2 : \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2 \quad \Gamma, x_2 : \theta_1, x_3 : \theta_2 \vdash M_3 : \alpha\mathbf{Cmd}\,\theta_3}{\Gamma, x_2 : \theta_1 \vdash M_2 \triangleright x_3 \cdot M_3 : \alpha\mathbf{Cmd}\,\theta_3}}{\Gamma \vdash M_1 \triangleright x_2 \cdot M_2 \triangleright x_3 \cdot M_3 : \alpha\mathbf{Cmd}\,\theta_3} \quad .$$

Case (4) Rule *unit*: $\uparrow M_1 \triangleright x \cdot M_2 \;\rightarrow\; [M_1/x]\, M_2$.

A type derivation for the redex has the form

$$\dfrac{\dfrac{\Gamma \vdash M_1 : \theta_1}{\Gamma \vdash \uparrow M_1 : \alpha\mathbf{Cmd}\,\theta_1}\left(\begin{array}{l}\alpha \notin fv\,\theta_1, \\ \alpha \notin fv\,\Gamma\end{array}\right) \quad \Gamma, x\colon \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash \uparrow M_1 \triangleright x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

The assumptions of this type derivation match the hypotheses of Lemma 6.4.1, so we can infer the judgment $\Gamma \vdash [M_1/x]\, M_2 : \alpha\mathbf{Cmd}\,\theta_2$ for the reduct, which is what is required.

Case (5) Rule *extend*: $(\nu v\, M_1) \triangleright x \cdot M_2 \;\rightarrow\; \nu v\, M_1 \triangleright x \cdot M_2$.

A valid type derivation for an *extend*-redex takes the form

$$\dfrac{\dfrac{\Gamma, v\colon \alpha\mathbf{Ref}\,\theta_0 \vdash M_1 : \alpha\mathbf{Cmd}\,\theta_1}{\Gamma \vdash \nu v\, M_1 : \alpha\mathbf{Cmd}\,\theta_1} \qquad \Gamma, x\colon \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash (\nu v\, M_1) \triangleright x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

The assumptions of this type derivation (modulo some weakening to make them fit the inference rules) give us the assumptions of the following type derivation for the reduct:

$$\dfrac{\dfrac{\Gamma, v\colon \alpha\mathbf{Ref}\,\theta_0 \vdash M_1 : \alpha\mathbf{Cmd}\,\theta_1 \quad \Gamma, v\colon \alpha\mathbf{Ref}\,\theta_0, x\colon \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma, v\colon \alpha\mathbf{Ref}\,\theta_0 \vdash M_1 \triangleright x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2}}{\Gamma \vdash \nu v\, M_1 \triangleright x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Case (6) Rule *assign-result*: $v := M_1 \rhd x \cdot M_2$, $x \in fv\, M_2$.

A valid type derivation for the redex in this case takes the form

$$\frac{\Gamma \vdash v := M_1 : \alpha\mathbf{Cmd}\,() \qquad \Gamma, x : () \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash v := M_1 \rhd x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Taking into account the global type assignment $() : ()$ and the assumption $\Gamma, x : () \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2$, the substitution lemma lets us conclude $\Gamma \vdash [()/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2$. We use this judgment aong with the other assumption to construct the following valid type derivation for the reduct:

$$\frac{\Gamma \vdash v := M_1 : \alpha\mathbf{Cmd}\,() \qquad \Gamma, z : () \vdash [()/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash v := M_1;\ [()/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad ,$$

where z is a variable resulting from the expansion of the abbreviation ";", and we have weakened the judgment given to us by the substitution lemma.

Case (7) Rule *fuse*: $v := M_1;\ v? \rhd x \cdot M_2 \;\rightarrow\; v := M_1;\ [M_1/x]\,M_2$.

A type derivation for the redex takes the form

$$\dfrac{\dfrac{\Gamma \vdash v : \alpha\mathbf{Ref}\,\theta_0 \qquad \Gamma \vdash M_1 : \theta_0}{\Gamma \vdash v := M_1 : \alpha\mathbf{Cmd}\,()} \qquad \dfrac{\dfrac{\Gamma, z : () \vdash v : \alpha\mathbf{Ref}\,\theta_0}{\Gamma, z : () \vdash v? : \alpha\mathbf{Cmd}\,\theta_0} \qquad \Gamma, z : (), x : \theta_0 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma, z : () \vdash v? \rhd x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2}}{\Gamma \vdash v := M_1;\ v? \rhd x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad ,$$

where the variable z arises from the expansion of the syntactic abbreviation ";" and occurs nowhere except its binding occurrence.

With a suitable weakening of the assumptions given by the above type derivation we can establish a type derivation for the reduct in two steps. First, the assumptions $\Gamma, z : () \vdash M_1 : \theta_0$ (note the weakened type judgment) and $\Gamma, z : (), x : \theta_0 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2$ match the hypotheses of Lemma 6.4.1, so we can infer $\Gamma, z : () \vdash [M_1/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2$. Second and finally, we now have enough assumptions to construct the following type deriva for the reduct:

$$\frac{\Gamma \vdash v := M_1 : \alpha\mathbf{Cmd}\,() \qquad \Gamma, z : () \vdash [M_1/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma \vdash v := M_1;\ [M_1/x]\,M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Case (8) Rule *bubble-assign*: $v := M_1;\ w? \rhd x \cdot M_2 \;\rightarrow\; w? \rhd x \cdot v := M_1;\ M_2$

A type derivation for this form of redex takes the form

$$\dfrac{\dfrac{\Gamma \vdash v : \alpha\mathbf{Ref}\,\theta_0 \qquad \Gamma \vdash M_1 : \theta_0}{\Gamma \vdash v := M_1 : \alpha\mathbf{Cmd}\,()} \qquad \dfrac{\dfrac{\Gamma, z : () \vdash w : \alpha\mathbf{Ref}\,\theta_1}{\Gamma, z : () \vdash w? : \alpha\mathbf{Cmd}\,\theta_1} \qquad \Gamma, z : (), x : \theta_1 \vdash M_2 : \alpha\mathbf{Cmd}\,\theta_2}{\Gamma, z : () \vdash w? \rhd x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2}}{\Gamma \vdash v := M_1;\ w? \rhd x \cdot M_2 : \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

The assumptions of the above derivation allow us to establish the following type derivation for the reduct. We can legitimately drop type assignments concerning the variable z from the type environment because it occurs nowhere except its binding occurrence; we also weaken some type judgments to match the type inference rules:

$$\dfrac{\dfrac{\Gamma \vdash w\colon \alpha\mathbf{Ref}\,\theta_1}{\Gamma \vdash w?\colon \alpha\mathbf{Cmd}\,\theta_1} \qquad \dfrac{\dfrac{\Gamma,x\colon \theta_1 \vdash v\colon \alpha\mathbf{Ref}\,\theta_0 \quad \Gamma,x\colon \theta_1 \vdash M_1\colon \theta_0}{\Gamma,x\colon \theta_1 \vdash v{:=}M_1\colon \alpha\mathbf{Cmd}\,()} \qquad \Gamma,z\colon (),x\colon \theta_1 \vdash M_2\colon \alpha\mathbf{Cmd}\,\theta_2}{\Gamma,x\colon \theta_1 \vdash v{:=}M_1;\, M_2\colon \alpha\mathbf{Cmd}\,\theta_2}}{\Gamma \vdash w?\triangleright x\cdot v{:=}\,M_1;\, M_2\colon \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Case (9) Rule *bubble-new*: $\nu v.w?\triangleright x\cdot M \;\dashrightarrow\; w?\triangleright x\cdot \nu v\, M$.

Type derivations for this form of redex take the form

$$\dfrac{\dfrac{\dfrac{\Gamma, v\colon\ \alpha\mathbf{Ref}\ \theta_0 \vdash w\colon \alpha\mathbf{Ref}\ \theta_1}{\Gamma, v\colon\ \alpha\mathbf{Ref}\ \theta_0 \vdash w?\colon \alpha\mathbf{Cmd}\,\theta_1} \quad \Gamma, v\colon\ \alpha\mathbf{Ref}\ \theta_0, x\colon \theta_1 \vdash M\colon \alpha\mathbf{Cmd}\,\theta_2}{\Gamma, v\colon\ \alpha\mathbf{Ref}\ \theta_0 \vdash w?\triangleright x\cdot M\colon \alpha\mathbf{Cmd}\,\theta_2}}{\Gamma \vdash \nu v.w?\triangleright x\cdot M\colon \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Before we use the assumptions of this type derivation to derive the same type for the reduct, we must observe that the judgment $\Gamma, v\colon\ \alpha\mathbf{Ref}\ \theta_0 \vdash w\colon\ \alpha\mathbf{Ref}\ \theta_1$ actually implies the judgment $\Gamma \vdash\ w\colon\ \alpha\mathbf{Ref}\ \theta_1$: since v and w are distinct store-variables when this reduction rule applies, the first judgment could only hold if Γ already holds a type assignment for w. We can now proceed in the usual way:

$$\dfrac{\dfrac{\Gamma \vdash w\colon \alpha\mathbf{Ref}\ \theta_1}{\Gamma \vdash w?\colon \alpha\mathbf{Cmd}\,\theta_1} \quad \dfrac{\Gamma,x\colon\ \theta_1, v\colon\ \alpha\mathbf{Ref}\ \theta_0 \vdash M\colon \alpha\mathbf{Cmd}\,\theta_2}{\Gamma,x\colon\ \theta_1 \vdash \nu v M\colon \alpha\mathbf{Cmd}\ \theta_2}}{\Gamma \vdash w?\triangleright x\cdot \nu v\, M\colon \alpha\mathbf{Cmd}\,\theta_2} \quad .$$

Case (10) Rule *pure-eager*: $\mathbf{pure}\, S^{eag}[\uparrow(\lambda x.M)] \;\dashrightarrow\; \lambda x.\mathbf{pure}\, S^{eag}[\uparrow M]$.

A type derivation for such a redex must have the form

$$\dfrac{\dfrac{\Gamma \vdash S^{eag}[\uparrow(\lambda x.M)]\colon \alpha\mathbf{Cmd}\,(\theta \to \psi)}{\Gamma \vdash S^{eag}[\uparrow(\lambda x.M)]\colon \forall\alpha.\alpha\,\mathbf{Cmd}\,(\theta \to \psi)}\,(\alpha \notin fv\,\Gamma)}{\Gamma \vdash \mathbf{pure}\, S^{eag}[\uparrow(\lambda x.M)]\colon \theta \to \psi} \quad .$$

Using the assumption of this type derivation, Lemma 6.4.2 tells us that the judgment

$$\Gamma^{+} \vdash \lambda x.M\colon \theta \to \psi$$

holds for some type environment Γ^{+} that extends Γ with type assignments for all the names in $bns\,(S^{eag}[\,])$. Lemma 6.4.2 applies because every eager store-context $S^{eag}[\,]$ is also a lazy store-context. This judgment, in turn, must arise from an application of the typing rule *Abs* under the assumption $\Gamma^{+},x\colon\ \theta \vdash M\colon \psi$. With this judgment in hand, we reverse the direction of reasoning (noting that the type environment $\Gamma^{+},x\colon\ \theta$ can also be thought of as being an extension $(\Gamma,x\colon\ \theta)^{+}$ of $\Gamma,x\colon\ \theta$ with type assignments for all the names in $bns\,(S^{eag}[\,])$. In this way we obtain the following type derivation for the reduct:

$$\dfrac{\dfrac{\dfrac{(Lemma\ 6.4.2)\,\dfrac{(\Gamma,x\colon\ \theta)^{+} \vdash M\colon \psi}{\Gamma,x\colon\ \theta \vdash S^{eag}[\uparrow M]\colon \alpha\mathbf{Cmd}\,\psi}}{\Gamma,x\colon\ \theta \vdash S^{eag}[\uparrow M]\colon \forall\alpha.\alpha\,\mathbf{Cmd}\,\psi}\,(\alpha \notin fv\,\Gamma,x\colon\ \theta)}{\Gamma,x\colon\ \theta \vdash \mathbf{pure}\, S^{eag}[\uparrow M]\colon \psi}}{\Gamma \vdash \lambda x.\mathbf{pure}\, S^{eag}[\uparrow M]\colon \theta \to \psi} \quad .$$

The cases of the other two forms of *pure-eager*-redex are proved similarly.

$$\begin{array}{lcl} \Lambda_{ok} & ::= & v \mid f \mid \lambda x.M \mid c^n \bullet M_1 \bullet \cdots \bullet M_k,\ (k \leq n) \mid v?\triangleright x{\cdot}M \mid \Theta_{ok} \\ \Theta_{ok} & ::= & v? \mid v := \Lambda_{ok} \mid \uparrow \Lambda_{ok} \mid \nu v.v := M;\ \Theta_{ok} \mid v := M;\ \Theta_{ok} \end{array}$$

Figure 6.14: The language Λ_{ok} of non-stuck normal forms

Case (11) Rule *pure-lazy*: $\mathbf{pure}\, S^{laz}[\uparrow(\lambda x.M)] \rightarrow \lambda x.\mathbf{pure}\, S^{laz}[\uparrow M]$.

The proof of this case is identical to that for the *pure-eager*.

Case (12) Rule *let*: $\mathbf{let}\, x = M_1\ \mathbf{in}\, M_2 \rightarrow [M_1/x]\, M_2$.

In this case, a type derivation must take the form

$$\frac{\Gamma \vdash M_1 : \sigma \quad \Gamma, x : \sigma \vdash M_2 : \theta}{\Gamma \vdash \mathbf{let}\, x = M_1\ \mathbf{in}\, M_2 : \theta}\ .$$

By Lemma 6.4.1, the assumptions of this derivation suffice to establish the judgment $\Gamma \vdash [M_1/x]\, M_2 : \theta$, which is the typing we desire for the reduct.

Since we have established the property of subject reduction for each of the possible forms of redex in LPJ^-, Lemma 6.4.3 is now proved. ∎

We now face the task of proving that the type system LPJ^- is safe for the calculi $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. In order to state this result precisely we formalize the requirements for type safety stated in Section 6.2. As discussed in that section, the primary responsibility of the type system is to ensure that certain stuck terms (normal forms that are not answer terms) cannot arise among the well-typed terms. If we can show this, then we can deduce that no computation starting from a well-typed term can go wrong: Lemma 6.4.3 assures that subsequent reducts of such a term retain the same type, and thus no stuck term can ever arise.

Our proof is that of [Chen and Odersky, 1994], adapted slightly to make use of LPJ^-. This proof proceeds by giving an inductively-defined language of non-stuck normal forms and proving that all the well-typed normal forms belong to this language. Figure 6.14 gives this language of non-stuck normal forms. The form $\nu v.v := M;\ \Theta_{ok}$ accounts for our convention that all store-variables are initialized immediately after being declared.

Lemma 6.4.4 (Well-typed normal forms) *In* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *with the* LPJ^- *type system, if* $\Gamma \vdash M : \theta$, *where* $\Gamma \equiv \{v_1 : \alpha_1\ \mathbf{Ref}\ \theta_1, \ldots, v_m : \alpha_m\ \mathbf{Ref}\ \theta_m\}$ *and M is irreducible, then* $M \in \Lambda_{ok}$.

Proof: We prove the lemma by induction on the structure the term M in the statement of the lemma. We consider every possible syntactic construction of M and show that each is either reducible (and hence not a subject of the lemma) or is in Λ_{ok}.

Case (1) $M \equiv f$, $M \equiv c^n$, or $M \equiv \lambda x.M_1$.

In all these cases the definition of Λ_{ok} gives $M \in \Lambda_{ok}$ directly.

Case (2) $M \equiv M_1 \bullet M_2$.

We consider several cases based on the syntactic form of M_1. By the induction hypothesis, $M_1 \in \Lambda_{ok}$. Furthermore, since M_1 is well-typed and occurs applied to another term within the well-typed typed M, it must be assigned a functional type. Among the elements of Λ_{ok} only those of the following forms satisfy this requirement:

(a) $M_1 \equiv f$.

Since we are assuming that primitives are typed as functions taking arguments of constructed type or function type, the argument term M_2 must have a constructed type or function type. Furthermore, the induction hypothesis demands that $M_2 \in \Lambda_{ok}$. Thus the only forms M_2 can take in this case are f, $\lambda y.M_2'$, or $c^n \bullet M_{2_1} \bullet \cdots \bullet M_{2_k}$, $(k \leq n)$. All these possibilities make M into a δ-redex (since we assume that primitives are total on their domains of definition). Since this form for M is not irreducible, it is removed from consideration.

(b) $M_1 \equiv c^n \bullet M'_1 \bullet \cdots \bullet M'_k$, $(k \leq n)$.

Since $M \equiv M_1 \bullet M_2$ is well-typed, it remains so when we write it out fully as $c^n \bullet M'_1 \bullet \cdots \bullet M'_k \bullet M_2$. Well-typing guarantees that $k+1 \leq n$, and hence the entire term is in Λ_{ok} as given directly by the definition.

(c) $M_1 \equiv \lambda x.M'_1$.

This case makes M a β-redex, and it is thus removed from consideration.

Case (3) $M \equiv \mathbf{let}\, x = M_1\ \mathbf{in}\, M_2$.

All **let**-expressions are redexes, so this case is removed from consideration.

Case (4) $M \equiv v$.

In this case, $M \in \Lambda_{ok}$ directly from the definition.

Case (5) $M \equiv M_1?$.

Since M is well-typed, M_1 must have a reference type. By the induction hypothesis, $M_1 \in \Lambda_{ok}$. The only form of term in Λ_{ok} that can have a reference type is v, so $M \equiv v?$, which is in Λ_{ok} by direct use of the definition.

Case (6) $M \equiv M_1 := M_2$.

By well-typedness, M_1 must have a reference type; by the induction hypothesis, $M_1 \in \Lambda_{ok}$. Thus $M_1 \equiv v$, and $M \equiv v := M_2$, which is in Λ_{ok} by definition.

Case (7) $M \equiv \uparrow M_1$.

In this case $M \in \Lambda_{ok}$ directly from the definition.

Case (8) $M \equiv \nu v.v := M_2; M_1$.

By well-typedness we have $\Gamma, v\colon \alpha\mathbf{Ref}\,\theta \vdash M_1 : \alpha\mathbf{Ref}\,\theta'$ for some α, θ, and θ'. By the induction hypothesis, we also know that $M_1 \in \Lambda_{ok}$. We now consider all the possible forms for M_1 satisfying both these constraints:

(a) $M_1 \equiv w? \rhd x \cdot M'$. In this case, M is reducible by rule *fuse* (if $v \equiv w$) or *bubble-assign* (if $v \not\equiv w$), and thus this case is eliminated from consideration.

(b) $M_1 \in \Theta_{ok}$. In this case, $M \in \Lambda_{ok}$.

Case (9) $M \equiv M_1 \rhd x \cdot M_2$.

By the usual argument combining well-typedness (M_1 must have command type) with the membership of the subterms in Λ_{ok}, we have only to consider the following cases for M_1:

(a) $M_1 \equiv v? \rhd x \cdot M'$. This makes M reducible by rule *assoc*, so this case is removed from consideration.

(b) $M_1 \equiv \uparrow M_1$. This makes M reducible by rule *unit*, so this case is removed from consideration.

(c) $M_1 \equiv \nu v\, M'_1$. This makes M reducible by rule *extend*, so this case is removed from consideration.

(d) $M_1 \equiv v?$. In this case, $M \in \Lambda_{ok}$.

(e) $M_1 \equiv v := M'_1$. If $x \in fv\, M_2$, then M is reducible by rule *assign-result*, which would remove this case from consideration. Assuming $x \notin fv\, M_2$, we can abbreviate $M \equiv v := M'_1; M_2$. Well-typedness dictates that $\Gamma, x\colon () \vdash M_2 : \alpha\mathbf{Cmd}\,\theta$, for some α and θ. Since $x \notin fv\, M_2$, we can weaken this judgment to $\Gamma \vdash M_2 : \alpha\mathbf{Cmd}\,\theta$, which lets us apply the induction hypothesis to M_2. We now have the following cases to consider:

(i) $M_2 \equiv w? \rhd y \cdot M'_2$. This makes M reducible either by rule *fuse* (if $v \equiv w$) or *bubble-assign* (if $v \not\equiv w$), thus removing this case from consideration.

(ii) $M_2 \in \Theta_{ok}$. In this case, $M \in \Lambda_{ok}$.

Case (10) $M \equiv \textbf{pure}S^{laz}[\uparrow M_1]$.

Since M is well-typed, we must have

$$\Gamma \vdash S^{laz}[\uparrow M_1] : \forall\alpha.\alpha\,\textbf{Cmd}\,\psi,$$

which in turn requires

$$\Gamma \vdash S^{laz}[\uparrow M_1] : \alpha\textbf{Cmd}\,\psi,$$

where $\alpha \notin fv\,\Gamma$. Our being forced to assume this side condition here is of crucial importance. Since the type assignments in Γ are all of the form $v : \alpha'\,\textbf{Ref}\,\theta'$, α must be distinct from any thread parameter assumed for any v_i assigned a type in Γ. This in turn implies via the typing rule *Seq* that no such v_i is dereferenced or assigned to along the spine of the command sequence $S^{laz}[]$.

Now the induction hypothesis tells us that $S^{laz}[\uparrow M_1] \in \Lambda_{ok}$. Since we also know that

$$\Gamma \vdash S^{laz}[\uparrow M_1] : \alpha\textbf{Cmd}\,\psi,$$

we can write down a set of productions for the possible forms of $S^{laz}[\uparrow M_1]$ by selecting from Figure 6.14 those productions that have command type:

$$\begin{array}{rcl} P & ::= & v?\triangleright x\cdot M \mid \Theta_{ok} \\ \Theta_{ok} & ::= & v? \mid v := M \mid \uparrow M \\ & \mid & \nu v.\Theta_{ok} \mid v := M;\,\Theta_{ok}. \end{array} \tag{6.2}$$

The first possible syntax in (6.2) is ruled out by the free occurrence of v, since we have just remarked that there can be no such occurrence in $S^{laz}[\uparrow M_1]$. Nor can the forms $v?$ or $v := M$ appear, both for the reason just cited and because we require that a command sequence that forms the body of a **pure**-expression ends in an occurrence of $\uparrow M$. Inspecting the remaining possibilities, we see that we are left with the exact productions that make up the definition of $S^{eag}[]$ in Figure 2.11. Furthermore, all the free names in M_1 are bound within $S^{laz}[]$; in fact, all such free names must be store-variables, since the form $v := M;\,\Theta_{ok}$ abbreviates a binding for a variable that does not occur free in the inner Θ_{ok}.

We now examine the possible forms for the result expression M_1. We know that it has a type of the form ψ; it can therefore have no free occurrences of store-variables or commands. It is also true by the induction hypothesis that $M_1 \in \Lambda_{ok}$. Surveying the definition of Λ_{ok}once more, we see that the only possible forms for M_1 are f, $\lambda x.M_1'$, and $c^n \bullet M_1' \bullet \cdots \bullet M_k'$, $(k \leq n)$. But each one of these possibilities, when combined with our knowledge from the previous paragraph that $S^{laz}[] \equiv S^{eag}[]$, makes M a *pure-eager*-redex (thus *a fortiori* a *pure-lazy*-redex). Hence we can remove **pure**-expressions from further consideration as possible normal forms absent from Λ_{ok}.

Having now shown that all possible terms M well-typed under the given form of type environment are either reducible or members of Λ_{ok}, we have now established Lemma 6.4.4. ∎

Our main theorem is a relatively simple corollary of Lemma 6.4.4:

Theorem 6.4.5 (Type safety for *LPJ*$^-$**)** *In* $\lambda[\beta\delta!eag]$ *and* $\lambda[\beta\delta!laz]$ *with the LPJ*$^-$ *type system, if* $\vdash M : Ans$*, where Ans is a type of answers, then any reduction sequence starting with M either diverges or terminates in an answer term A with* $\vdash A : Ans$.

Proof: The types of answers are the constructed applicative types. By Lemma 6.4.4, the hypotheses of the current theorem imply $\downarrow(M) \in \Lambda_{ok}$, where $\downarrow(M)$ is the normal form of M (provided one exists). By subject reduction (Lemma 6.4.3) $\downarrow(M)$ has any type that M has. Inspecting the definition of Λ_{ok}, we see that the only possible form for $\downarrow(M)$ is $c^n \bullet M_1 \bullet \cdots \bullet M_k$, $(k \leq n)$. But our remarks apply recursively to the M_i, and any term so constructed is by definition an answer term, which is what was to be proved. ∎

6.5 The prospect for an untyped version of *LPJ*

The success of the central technique of *LPJ* in underpinning the LPJ^{-} type system for λ[βδ!*eag*] and λ[βδ!*laz*] suggests that we investigate whether the unrestricted *LPJ* type system might play a similar role with respect to some untyped lambda-calculus with assignments. As we have already remarked, *LPJ* is unsafe for λ[βδ!*eag*] and λ[βδ!*laz*] themselves. For example, the term

$$\mathbf{pure}\,\nu v.v := 1;\ (\mathbf{pure}\uparrow v)?\triangleright x \cdot\!\uparrow x \tag{6.3}$$

is typable in *LPJ* but gets stuck in λ[βδ!*eag*] owing to the attempt to return a store-variable from the inner **pure**.

As a first attempt, we might consider lifting the restriction on the forms of term that can be returned from **pure** constructs. This, however, leads to immediate disaster, because store-contexts can bind variables that might be present in the returned form. We already know that imposing a condition that no variables be freed by purification leads to the problem that brings down the ILC and Chen/Odersky type systems, so we must classify this approach as unpromising.

The next attempt (one to which this author has devoted considerable energy) is to introduce into the untyped calculus a new kind of name for tagging threads, in the hope that some sort of run-time check for thread identity might cause bad purification attempts to get stuck. Unfortunately, it is not enough that all such attempts get stuck: our desire for confluence demands that all stuck terms resulting from reduction starting at the same term should be identical. This demand is difficult to satisfy: a characteristic failure is that **pure**-contexts become duplicated around a portion of the result on one reduction path but not on another. Some sort of idempotence rule for **pure**-contexts seems in order, but at this point the calculus becomes vastly overbalanced toward technicalities—the hope that an untyped calculus might expose computational intuitions is lost.

These negative experiences lead to the tentative conclusion that in order to obtain the generality of purification afforded by the Launchbury/Peyton Jones type system it may be best just to work in a typed framework from the start.

6.6 Chapter summary

In this chapter, we have investigated the construction of a safe type system for λ[βδ!*eag*] and λ[βδ!*laz*]. We have noted a known flaw in the previous efforts along these lines (Chen/Odersky and ILC), and have constructed and proved safe a hybrid (LPJ^{-}) of the flawed approaches with the type system proposed by Launchbury and Peyton Jones.

7
In conclusion

We now conclude this study of lambda-calculi with assignment. Section 7.1 acknowledges the ways in which the actual work falls short of covering its entire scope; Section 7.2 looks beyond the boundaries we have set for this dissertation to see what further work might take advantage of the foundations we have established here. Section 7.3 provides a final summary of the work we have presented.

7.1 Unfinished business

This dissertation has one major hole: our failed attempt to incorporate the lazy-store calculus $\lambda[\beta\delta!laz]$ into the scheme of the conservative-extension results presented in Section 5.2. As we noted in Section 5.2.6, the prospects for such a proof using techniques analogous to those employed for $\lambda[\beta\delta!eag]$ are rather dim, since the syntactic structure required for a continuation-passing-style encoding of the command execution order for $\lambda[\beta\delta!laz]$ is directly at odds with the nesting structure required for the local-name construct $\nu w.M$. It might be possible to use Odersky's de Bruijn-like encoding scheme from [Odersky, 1993b] directly, but this approach lacks the satisfying aspect of dealing with commands and name-locality in a modular fashion. Progress in filling this gap seems to require new insight which the author of this dissertation has not yet achieved.

7.2 Future Work

The calculi introduced in this dissertation only model store-variables with a single assignable data component, and only describe conventional sequential control constructs that can be modeled by basic blocks or procedure invocations. We now indicate briefly two directions in which our core treatment may be extended with some further effort. In Section 7.2.1, we consider *compound* mutable entities (such as data structures or arrays) that are allocated all at once, but have independently assignable parts. In Section 7.2.2 we consider the extension of the techniques used in this dissertation to model non-local control constructs such as Scheme's **call/cc**.

7.2.1 Mutable arrays and data structures

Our entire treatment of mutable data in this dissertation has been in terms of store-variables which can be associated with a single data value. As previously remarked on page 18, this treatment does not exactly correspond to the notion of data structure present in conventional programming languages such as C. In this section we sketch some possible further work toward bridging this semantic gap.

Object identity versus component identity

The notion of store-variable treated in this dissertation resembles the notion of variable in conventional imperative programming languages, but the two concepts are far from identical. Data structures in C are identified with the storage they occupy. This means, for example, that in a structure with two integer fields the address of the whole structure and of the first field are the same, and that programs can detect this sharing by assigning to the whole structure and the components separately. With store-variables, on the other hand, every allocated entity has a distinct identity. If we want to have a structure containing two mutable integer fields, yet also itself mutable, we are forced to allocate a single store-variable containing an immutable pair of two store-variables containing integers. To satisfy the semantics of the functional language, the implementation would have to be something like a pointer to a structure of two (pointers to) integers. A conventional language implementation, however, would identify the storage allocated to the structure with the storage allocated to the contained references. Although the difference is not semantically detectable in our functional calculi with assignment, the additional storage allocation would surely be visible to performance tools in any practical software development environment.

What is lacking in our calculi is the ability to subdivide the mutability of a store-variable. A reasonable solution would be to introduce some structure on store-variable names so that a cluster of names can itself be given a unique name. The semantics could be defined by translation into the calculus with individual store-variables, taking into account type information to guide the identification of storage for contained references with storage for the container reference.

Arrays and pointer arithmetic

At first glance there seems to be no problem in adding the conventional notion of data arrays to our languages. If we add our assignable variables to a functional programming language already having immutable arrays, we can simulate a conventional mutable array by an immutable array of store-variables. However, in light of the issue raised immediately above, we can see that more is required if the array itself is to be a mutable object. Extending the solution suggested for structures above, we could introduce a notion of store-variable names qualified with indices rather than component names.

7.2.2 Advanced control constructs

It is possible to augment the calculus $\lambda[\beta\delta!eag]$ with a non-local control operator in the style of [Felleisen *et al.*, 1987; Felleisen and Hieb, 1992; Felleisen, 1988]. This operator is defined in a way that works with any syntactic monad; however, we will only consider it in the context of our calculi of assignments.

The cited works by Felleisen and his colleagues investigate the introduction of syntactically-defined control operators into call-by-value lambda-calculi for the purpose of providing an operational semantics for the language Scheme [Clinger and Rees, 1991] including the construct call/cc. Our aim here differs from theirs in several respects. First, our base lambda-calculus is call-by-name rather than call-by-value. Second, we do not incorporate our control construct into the applicative language; rather, we have it operate on a syntactically distinct monadic fragment of our calculus. We could thus conjecture that an extension of functional programming with control constructs based on our monad-inspired approach might actually be a conservative extension of the base language, whereas this is not expected of a model of Scheme. Although we are not in a position to deny that a calculus modeling Scheme with call/cc can be a conservative extension of the call-by-value lambda-calculus, this is a reasonable conjecture because of the ability of a term containing call/cc to grab its evaluation context. Perhaps such a term can discover differences in evaluation patterns between terms that are operationally equivalent in the base calculus. [1] At any rate, we are motivated to study control operators by the methods of this dissertation.

A control operator for the command calculi

The idea behind the control operator C is simple. If a procedure is in $\rightarrow_{\triangleright}$-normal form, every tail of the procedure (portion following a $\triangleright$) represents the future of the computation (the *continuation*) from that point. The control operator should be able to substitute a different continuation while storing the old one.

Formally, the action of the control operator is defined by Figure 7.1, which gives the rule implementing the replacement-with-remembrance of the current continuation. The control operator takes a procedure, called the *escape continuation* as an argument, and produces an *escape procedure*. At the point where the escape procedure is invoked, the then-current continuation is captured and passed as argument to the escape continuation, which is where execution now continues.

The usual control operators **call/cc** and **abort** can be defined in terms of C as follows:

$$\begin{array}{rcl} \textbf{call/cc} & \equiv & \lambda m.\,C\,(\lambda k.m \bullet (\lambda x.\,C\,(\lambda k'.k \bullet x)) \triangleright x \cdot k \bullet x) \\ \textbf{abort} & \equiv & \lambda x.\,C\,(\lambda k.\uparrow x) \end{array}$$

[1] It is also strongly suggestive that [Felleisen, 1991] finds that an idealized Scheme with call/cc is more expressive than pure Scheme in a well-defined sense that is related to operational semantics.

$$\boxed{\textbf{pure}\, S^{laz}[(C\, M) \triangleright x \cdot N] \quad \longrightarrow \quad \textbf{pure}\, S^{laz}[M \bullet (\lambda x.N)]}$$

Figure 7.1: Rule for control operator

Interaction of control and assignment

It is interesting to note that the introduction of the operator C is incompatible with $\lambda[\beta\delta!laz]$. Consider the term

$$\textbf{pure}\, C\,(\lambda y.\uparrow 2) \triangleright x \cdot \uparrow 1.$$

We can reduce this by a *pure-lazy*-reduction to 1, or via a control reduction to

$$\textbf{pure}\,(\lambda y.\uparrow 2) \bullet (\lambda x.\uparrow 1),$$

which reduces to 2 by β followed by *pure-lazy*. The lazy-store calculus with the added control operator is thus not Church-Rosser. This counterexample does not apply in $\lambda[\beta\delta!eag]$ because the purification contexts there do not allow control operators to remain along the spine of the command sequence. This failure of a counterexample is not the same thing as having a proof of Church-Rosser for $\lambda[\beta\delta!eag]$ plus the control operator, so it would be interesting further work to work out the theory of this calculus, compare it with the work of Felleisen and colleagues cited above, and find the deep reason why the simple approach fails in the lazy-store calculus.

This concludes our discussion of possible future work based on the work in this dissertation.

7.3 Summary of technical results

We have introduced and studied formal representations of two potential paradigms of functional programming with assignment statements. We have brought the apparatus of the study of the untyped lambda-calculus to bear on these representations, and have found that they have the Church-Rosser and standardization properties just as do the foundations of pure functional programming. We have explored the operational-equivalence theory implied by this reduction-semantics basis and have found it to correspond to our informal understanding in some simple test cases. The proposed calculi correspond in a precise way with store-computations as modeled by alternative calculi which represent the store explicitly, and both calculi are conservative extensions of appropriately chosen pure functional calculi. Finally, we have shown that a reasonably simple type system guarantees safe usage of the assignment constructs in our untyped calculi.

We hope that the theoretical results presented here will lead to the confident adoption of a safe and reasonable design for a programming language combining the powerful higher-order program-construction facilities of functional programming with the convenient resource-consumption model of programming with assignments.

A

Proofs of the fundamental properties for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$

We give in this appendix the proofs of the Church-Rosser and standardization theorems for the calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ introduced in Chapter 5. We use the structure of the corresponding proofs in Chapter 3 as a guide in structuring the proofs in this appendix. Our presentation here is thus rather sketchy: we give only the features that differ from those in the corresponding proofs and assume that these differing features are to be embedded in a matrix of general argument that is lifted directly from Chapter 3.

In general, the proofs for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ require less work than the corresponding proofs for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$. The shift of the axiomatization of the store from rewriting of command-sequences to the use of an explicit store removes many critical overlaps between rules. It is still necessary, however, to carry out the proofs in terms of $\overset{\triangleright}{=}$-equivalence classes, since the issue that led to their introduction derives entirely from the structure of the rules *assoc* and *extend*, which are retained in the explicit-store calculi.

A.1 The reductions $\rightarrow_{\triangleright}$ and $\rightarrow_{!}$

The reduction $\rightarrow_{\triangleright}$ is the exact same for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ as for $\lambda[\beta\delta!eag]$ and $\lambda[\beta\delta!laz]$, and the definition of $\rightarrow_{!}$ from $\rightarrow$ and $\rightarrow_{\triangleright}$ is derived in the same way as it is in Section 3.3.2.

The first result we establish is the analogue of Proposition 3.4.1:

Proposition A.1.1 (Commutation of association) *Suppose we have $M \rightarrow_{!} N$ via reduction of a marked redex Δ, and suppose also that $M \rightarrow_{\triangleright} M'$. Then there exist terms M'' and N' such that $M' \rightarrow_{\triangleright}^{*} M''$, $N \rightarrow_{\triangleright}^{*} N'$, and $M'' \rightarrow_{!} N'$ by reducing Δ'', where Δ'' is the residual of Δ in M''.*

Proof: The statement of the proposition is illustrated by the following diagram:

$$\begin{tikzcd}
& M \arrow[dl, "\rightarrow_{!}"', "\Delta"] \arrow[dr, "\rightarrow_{\triangleright}"] & & \\
N \arrow[ddrr, dotted, "\rightarrow_{\triangleright}"'] & & M' \arrow[dr, dotted, "\rightarrow_{\triangleright}"] & \\
& & & M'' \arrow[dl, dotted, "\Delta''\ \rightarrow_{!}"] \\
& & N' &
\end{tikzcd}$$

The only critical overlap of reduction rules that survives from the proof of Proposition 3.4.1 is the overlap between rules *unit* and *assoc*; the old proof for this case still works here. All the other overlap cases considered in proving Proposition 3.4.1 involve reduction rules that are no longer present in the same form in $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$.

Of the reduction rules that are introduced in the explicit-store calculi, most cannot overlap with *assoc* or *extend* because they are written in terms of the term-in-store syntax $\sigma \cdot \langle M \rangle$, which is not involved in either of the $\rightarrow_{\triangleright}$-rules. The only newly-introduced rule requiring consideration is the purification rule in $\lambda[\beta\delta\sigma laz]$, which can have $\rightarrow_{\triangleright}$-redexes in the store-context $S^{laz}[]$. For this case, the old proof is easily translated into the context of $\lambda[\beta\delta\sigma laz]$:

$$\begin{tikzcd}[column sep=tiny]
 & \overbrace{\sigma\cdot\underline{\langle} S_1^{laz}\underbrace{[(\nu v\,M)\rhd x\cdot S_2^{laz}[\uparrow(\lambda\underline{y}.N)]]}\underline{\rangle}} \arrow[dl, "\underline{\sigma_{pure}}"'] \arrow[dr, "extend"] & \\
\lambda y.\sigma\cdot\langle S_1^{laz}\underbrace{[(\nu v\,M)\rhd x\cdot S_2^{laz}[\uparrow N]]}\rangle \arrow[dr, dotted, "extend"'] & & \overbrace{\sigma\cdot\underline{\langle} S_1^{laz}[\nu v\,M\rhd x\cdot S_2^{laz}[\uparrow(\lambda\underline{y}.N)]]\underline{\rangle}} \arrow[dl, dotted, "\underline{\sigma_{pure}}"] \\
 & \lambda y.\sigma\cdot\langle S_1^{laz}[\nu v\,M\rhd x\cdot S_2^{laz}[\uparrow N]]\rangle &
\end{tikzcd}$$

With the consideration of these two rule-overlaps, we have now established Proposition A.1.1. ∎

The reasoning by which Corollaries 3.4.2 and 3.4.3 are established is independent of the calculus to which they apply, so we can use this reasoning to obtain these same results for the explicit-store calculi. We have now completed proving the necessary properties of the interaction between $\longrightarrow_{\rhd}$ and $\longrightarrow_{!}$ for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$.

A.2 Finite developments

We next prove that $\longrightarrow_{!}$ for the calculi with explicit stores has the strong finite-developments property. The first step is to show that marked $\longrightarrow_{!}$-reduction is weakly Church-Rosser on $\overset{\rhd}{=}$-equivalence classes (analogous to Lemma 3.5.5):

Lemma A.2.1 *The weak Church-Rosser property holds for the calculi $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ under marked $\longrightarrow_{!}$-reduction on $\overset{\rhd}{=}$-equivalence classes.*

Proof: The cases having to do with β and δ are proved just as for the implicit-store calculi, since this core of the calculus is unchanged. Once again, none of the previously existing overlaps carries over, and only the new rule σ_{pure} has an overlap (with *unit*). The proof for this case is exactly analogous to that for the corresponding case in Lemma 3.5.5. ∎

The next step in proving finiteness of developments is to construct a termination measure for marked $\longrightarrow_{!}$-reduction. The explicit-store calculi require no new reasoning here, only (1) some adjustment in the definition of the weighting function $\mathcal{W}[\![\,]\!]$ to reflect the new reduction rules and similarly (2) modification of Definition 3.5.6 to reflect the new reduction rules. We thus obtain the finiteness-of-developments property, and further, the stronger properties given in Theorem 3.5.12.

A.3 The Church-Rosser property

Strong finiteness of developments gives us the Church-Rosser property exactly as in Section 3.6.

A.4 Standardization

As in Section 3.7, the standardization theorem for the explicit-store calculi requires a lot of work concerned with evaluation contexts and head and internal redexes. We define evaluation contexts for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$ as given in Figures A.1 and A.2; we also augment the subterm ordering with the single new ordering given in Figure A.3. We reinterpret Definitions 3.7.1 and 3.7.2 in terms of these new definitions of evaluation context and subterm ordering.

$$\begin{array}{rcl} E[\,] & ::= & \ldots\text{Figure 3.3}\ldots \\ & | & \sigma\cdot\langle E[\,]\rangle \end{array}$$

Figure A.1: Evaluation contexts for $\lambda[\beta\delta\sigma eag]$

$$\begin{array}{rcl} E[\,] & ::= & \ldots\text{Figure 3.3}\ldots \\ & | & \sigma\cdot\langle S^{laz}[E[\,]]\rangle \end{array}$$

Figure A.2: Evaluation contexts for $\lambda[\beta\delta\sigma laz]$

$$M \;\equiv\; \sigma\cdot\langle M_1\rangle \;\Rightarrow\; M \prec M_1$$

Figure A.3: Additional subterm ordering for $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$

We now tackle the analogues of the lemmas concerning the preservation and separation of head and internal redexes. We first consider the analogue of Lemma 3.7.6:

Lemma A.4.1 (Heads don't sprout) *If $M \rightarrow_{\mathrm{i}} N$ and $[N]_{\underline{\underline{\triangleright}}}$ has a head redex Δ_{h}, then $[M]_{\underline{\underline{\triangleright}}}$ has a head redex.*

Proof: We adopt the notation used in the proof of Lemma 3.7.6, and we refer to that proof for the important discussion of the effect of the interaction of $\rightarrow_{\triangleright}$-reduction with the case analysis. We use the same basic division into three cases based on the relative positions of L' and Δ'_{h}.

Case (1): L' and Δ'_{h} are disjoint.

The argument from the proof of Lemma 3.7.6 applies with little change as far as it can be carried out independently of the actual form of evaluation contexts. For the final part of the argument in which we enumerate a list of possible configurations in which Δ' can be disjoint from Δ_{h}, the first four cases still apply, since they deal with forms of evaluation context that still occur in the explicit-store calculi. The only case we need to add to the list is the form $E[1]\sigma\cdot\langle S^{laz}[\uparrow E[2]\Delta_{\mathrm{h}}]\rangle$, where $\Delta' \subseteq S^{laz}[\,]$. In this case Δ_{h} is an evaluation redex of M, hence there exists a head redex.

Case (2): $\Delta'_{\mathrm{h}} \subseteq L'$.

In this case the reasoning from the proof of Lemma 3.7.6 applies without change, since it makes no use of the actual form of evaluation contexts.

Case (3): $L' \subset \Delta'_{\mathrm{h}}$.

The general reasoning in this case, as well as the reasoning for the subcases for reduction rules β, δ, and *unit* apply without change from Lemma 3.7.6. We now treat the cases resulting from reduction rules that are introduced in $\lambda[\beta\delta\sigma eag]$ and $\lambda[\beta\delta\sigma laz]$. We use σ' to denote a store that reduces to a store σ by reducing a binding for a store-variable in σ'. Since σ' and σ have the same domain there is never any question of a reduction becoming available or becoming forbidden because of the reduction from σ' to σ.

(a) $\sigma\nu$: $\Delta_{\mathrm{h}} \equiv \sigma\cdot\langle \nu w.M_1\rangle$

We construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\sigma\cdot\langle\Delta'\rangle$	Δ'
$\sigma\cdot\langle \nu w.C[\Delta']\rangle$	$\widehat{M}$
$\sigma'\cdot\langle \nu w.M_1\rangle$	$\widehat{M}$

(b) $\sigma{:=}$: $\Delta_{\mathrm{h}} \equiv \sigma\cdot\langle v{:=}N_1\,;\,M_1\rangle$

We construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\sigma\cdot\langle\Delta'\rangle$	Δ'
$\sigma\cdot\langle\Delta' := N_1 ; M_1\rangle$	Δ'
$\sigma\cdot\langle v := C[\Delta']; M_1\rangle$	$\widehat{M}$
$\sigma\cdot\langle v := N_1 ; C[\Delta']\rangle$	$\widehat{M}$
$\sigma'\cdot\langle v := N_1 ; M_1\rangle$	$\widehat{M}$

(c) $\sigma ?$: $\Delta_h \equiv \sigma\cdot\langle v?\triangleright x\cdot M_1\rangle$

We construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\sigma\cdot\langle\Delta'\rangle$	Δ'
$\sigma\cdot\langle\Delta' ?\triangleright x\cdot M_1\rangle$	Δ'
$\sigma\cdot\langle v?\triangleright x\cdot C[\Delta']\rangle$	$\widehat{M}$
$\sigma'\cdot\langle v?\triangleright x\cdot M_1\rangle$	$\widehat{M}$

(d) σ_{block}: $\Delta_h \equiv \mathbf{pure} M_1$

We construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\mathbf{pure}\,\Delta'$	Δ'
$\mathbf{pure}\,C_+[\Delta']$	$\widehat{M}$

(e) σ_{peag}: $\Delta_h \equiv \sigma\cdot\langle\uparrow\lambda x.M_1\rangle$, *etc.*

This case applies in the calculus $\lambda[\beta\delta\sigma eag]$.

We construct the following table of cases:

$\widehat{M}$:	eval. redex:
$\sigma'\cdot\langle\uparrow\lambda x.M_1\rangle$	$\widehat{M}$
$\sigma\cdot\langle\Delta'\rangle$	Δ'
$\sigma\cdot\langle\uparrow\Delta'\rangle$	Δ'
$\sigma\cdot\langle\uparrow\lambda x.C[\Delta']\rangle$	$\widehat{M}$

The other two forms of purifiable expression require similar arguments.

(f) σ_{plaz}: $\Delta_h \equiv \sigma\cdot\langle S^{laz}[\uparrow V]\rangle$, *etc.*

This case applies in the calculus $\lambda[\beta\delta\sigma laz]$.

The following cases are easy to describe:

$\widehat{M}$:	eval. redex:
$\sigma'\cdot\langle S^{laz}[\uparrow V]\rangle$	$\widehat{M}$
$\sigma\cdot\langle\Delta'\rangle$	Δ'
$\sigma\cdot\langle S^{laz}[\Delta']\rangle$	Δ'
$\sigma\cdot\langle S^{laz}[\uparrow\Delta']\rangle$	Δ'

One of the remaining possibilities occur when Δ' produces part of the command sequence notated as $S^{laz}[]$. No matter how this happens, however, $\widehat{M}$ takes the form $\sigma\cdot\langle S_1^{laz}[\uparrow V]\rangle$, and so $\widehat{M}$ is an evaluation redex.

The last possibility is that $\widehat{M} \equiv \sigma\cdot\langle S_1^{laz}[\Delta']\rangle$, in which case Δ' is an evaluation redex.

We have now completed the case analysis required to establish Lemma A.4.1. ∎

The next of the lemmas supporting the standardization theorem is the analogue of Lemma 3.7.7:

Lemma A.4.2 (Heads are preserved) *Let Δ_h be a head redex and Δ_i be an internal redex in $[M]_{\trianglerighteq}$. If $[M]_{\trianglerighteq} \xrightarrow{\Delta_i} [N]_{\trianglerighteq}$, then the residual of Δ_h in $\downarrow_{\triangleright}[N]$ consists of a single redex that is head redex in $[N]_{\trianglerighteq}$.*

Proof: We consider the relative positiosn of Δ_h and Δ_i as in the proof of Lemma 3.7.7.

Case (1) Δ_i and Δ_h are disjoint.

In this case, we reason as in the proof of Lemma 3.7.7, basing our enumeration of subcases on the corresponding case in the proof of Lemma A.4.1 instead of Lemma 3.7.6.

Case (2) $\Delta_i \subset \Delta_h$.

The reasoning used in the corresponding case in Lemma 3.7.6 applies as well in the present case—not because the argument is generic, but because the properties it requires of $\lambda[\beta\delta!\mathit{eag}]$ and $\lambda[\beta\delta!\mathit{laz}]$ are also true of $\lambda[\beta\delta\sigma \mathit{eag}]$ and $\lambda[\beta\delta\sigma \mathit{laz}]$.

∎

The remaining lemma supporting the proof of standardization makes a generic argument, and so it holds for the explicit-store calculi. We can thus prove the standardization result by the same methods as used in Chapter 3.

A.5 Conclusion

The remaining work in Chapter 3, i.e. lifting results about the factored reduction relations up to the original calculus, can now be established by the same arguments for the explicit-store calculi.

B
Changes in notation and terminology from previously published versions

The notation employed in this dissertation differs in several respects from that presented in the antecedent works [Odersky *et al.*, 1993] and [Odersky and Rabin, 1993]. There are two main reasons for the authors decision to change notation. First, the old notation was decidedly odd in some respects, such as in giving the assigned store-variable rightmost in an assignment command. Second, the elaboration of the mathematical exposition in the dissertation has placed a premium on compactness of notation; thus, several keyword-style notations were replaced with non-alphabetic symbols. The author hopes that the exposition surrounding the introduction of these notations compensates for the loss of familiarity.

With respect to terminology, the most troublesome point has been to decide what to call the identifiers for assignable variables. This is a problem because the use of *variable* in the imperative-language culture to denote these expressions is well-entrenched, yet it conflicts with the use of *variable* in the lambda-calculus culture to denote an identifier for which expressions are substituted in the course of computation. We have decided to retain the lambda-calculus usage of the simple term *variable*, and to consistently refer to the former kind as *store*-variables. In the antecedent works we employed such terms as *tag* and *location*, but we now find *tag* to be too unfamiliar and *location* to be overly evocative of implementation issues for the level of discourse of the research presented.

There are also some differences in detail between the calculi λ_{var} and $\lambda[\beta\delta!eag]$. In λ_{var}, there is only one substitutive rule, β, and all other rules that transmit expressions are written in terms of β. In the present work, these rules are written in terms of substitution directly in order to draw a cleaner boundary between functional and imperative rules. In retrospect, the revision does not seem to have gained much.

BIBLIOGRAPHY

[Amadio and Cardelli, 1991] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 104–118. ACM Press, January 1991.

[Ariola *et al.*, 1995] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *Proceedings of the Twenty-second ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 233–246, New York, January 1995. ACM Press.

[Barendregt, 1984] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[Birtwistle *et al.*, 1973] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Auerbach, 1973.

[Chen and Odersky, 1993] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignment. Research Report YALEU/DCS/RR-963, Yale University Department of Computer Science, May 1993.

[Chen and Odersky, 1994] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Science, International Symposium TACS'94, Sendai, Japan, Proceedings*, pages 347–64, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science 789.

[Church, 1951] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.

[Clinger and Rees, 1991] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. Available via World Wide Web as http://www-swiss.ai.mit.edu/ftpdir/scheme-reports/r4rs.ps, November 1991.

[Damas and Milner, 1982] L. Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, January 1982.

[Felleisen and Friedman, 1986] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Descriptions Of Programming Concepts III*. North-Holland, 1986.

[Felleisen and Friedman, 1987a] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages. Papers Presented at the Symposium, Munich, West Germany*, pages 314–325, New York, January 1987. Association for Computing Machinery.

[Felleisen and Friedman, 1987b] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and*

Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, pages 206–223, New York, June 1987. Springer-Verlag. Lecture Notes in Computer Science 259.

[Felleisen and Hieb, 1992] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[Felleisen *et al.*, 1987] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[Felleisen, 1987] Matthias Felleisen. A calculus for assignments in higher-order languages. In *Proceedings, Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 314–325, January 1987.

[Felleisen, 1988] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, ACM Press, January 1988.

[Felleisen, 1991] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[Fischer, 1972] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, January 1972. SIGPLAN Notices, Vol. 7, no. 1 and SIGACT News, No. 14.

[Fischer, 1993] Michael J. Fischer. Lambda calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, November 1993.

[Fradet, 1991] Pascal Fradet. Syntactic detection of single-threading using continuations. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 241–258. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[Girard *et al.*, 1989] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Girard, 1987] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Girard, 1990] Jean-Yves Girard. The system F of variable types, fifteen years later. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, chapter 6. Addison-Wesley Publishing Company, Inc., 1990.

[Gordon, 1994] Andrew D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, New York, 1994.

[Graham and Kock, 1991] T. C. Nicholas Graham and Gerd Kock. Domesticating imperative constructs so that they can live in a functional world. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and*

Logic Programming, pages 51–62. Springer-Verlag, August 1991. Lecture Notes in Computer Science 528.

[Guzmán and Hudak, 1990] Juan Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, Pennsylvania*, pages 333–343, Los Alamitos, California, June 1990. IEEE Computer Society Press.

[Guzmán, 1993] Juan Carlos Guzmán. On expressing the mutation of state in a functional programming language. Research Report YALEU/DCS/RR-962, Yale University, Department of Computer Science, New Haven, Connecticut, May 1993. PhD Thesis.

[Hindley, 1969] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[Huang and Reddy, 1995] H. Huang and U.S. Reddy. Type reconstruction for SCI. In *Proceedings of the Glasgow Functional Programming Workshop*, New York, 1995. Springer-Verlag. To appear.

[Hudak and Sundaresh, 1988] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Research Report YALEU/DCS/RR-665, Yale University, Department of Computer Science, New Haven, Connecticut, December 1988.

[Hudak *et al.*, 1992] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, March 1992. Also in ACM SIGPLAN Notices, Vol. 27(5), May 1992.

[Hudak, 1992] Paul Hudak. Mutable abstract datatypes. Research Report YALEU/DCS/RR-914, Yale University Department of Computer Science, December 1992.

[Huet, 1980] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Jouvelot and Gifford, 1989] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–226. ACM, ACM Press, June 1989.

[Klop, 1992] J.W. Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*, pages 1–116. Oxford University Press, Clarendon Press, New York, 1992.

[Knuth and Bendix, 1970] Donald E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Compuatational Problems in Abstract Algebra*, pages 263–297. Pergamon, Oxford, 1970.

[Lafont, 1988] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Landin, 1964] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

[Launchbury and Peyton Jones, 1994] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI), Orlando, FL*, pages 24–35, New York, June 1994. ACM Press. SIGPLAN *Notices* 29(6).

[Launchbury, 1993] John Launchbury. Lazy imperative programming. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 46–56, June 1993.

[Lucassen and Gifford, 1988] Jon M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57. ACM, ACM Press, January 1988.

[Mason and Talcott, 1991] Ian Mason and Carolyn Talcott. Equivalence in functional languages with side effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[Mason and Talcott, 1992a] Ian A. Mason and Carolyn L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.

[Mason and Talcott, 1992b] Ian A. Mason and Carolyn L. Talcott. References, local variables, and operational reasoning. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 186–197, Los Alamitos, California, June 1992. IEEE Computer Society Press.

[Mason, 1986] Ian A. Mason. *The Semantics of Destructive Lisp*, volume 5 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Ventura Hall, Stanford, CA 94305, 1986.

[Milner *et al.*, 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, Dec 1978.

[Milner, 1991] Robin Milner. The polyadic π-calculus: A tutorial. Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, October 1991.

[Moggi, 1989] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings, Third Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1989.

[Moggi, 1991] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Naur *et al.*, 1960] Peter Naur, John W. Backus, F. L. Bauer, J. Green, C. Katz, John McCarthy, Alan J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3:299–314, 1960.

[Naur *et al.*, 1963] Peter Naur, J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[Odersky and Rabin, 1993] Martin Odersky and Dan Rabin. The unexpurgated call-by-name, assignment, and the lambda-calculus. Research Report YALEU/DCS/RR-930, Department of Computer Science, Yale University, New Haven, Connecticut, May 1993.

[Odersky *et al.*, 1992] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda-calculus. Research Report YALEU/DCS/RR-929, Department of Computer Science, Yale University, New Haven, Connecticut, October 1992.

[Odersky *et al.*, 1993] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 43–56, January 1993.

[Odersky, 1991] Martin Odersky. How to make destructive updates less destructive. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 25–26. ACM Press, January 1991.

[Odersky, 1993a] Martin Odersky. A syntactic method for proving observational equivalences. Research Report YALEU/DCS/RR-964, Department of Compter Science, Yale University, May 1993.

[Odersky, 1993b] Martin Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Yale University Department of Computer Science, New Haven, Connecticut, May 1993.

[Odersky, 1994] Martin Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 48–59, New York, January 1994. ACM Press.

[O'Hearn *et al.*, 1995] P.W. O'Hearn, A.J. Power, M. Takeyama, and R.D. Tennent. Syntactic control of interference revisited. In *Mathematical Foundations of Programming Semantics*, pages 97–136. Elsevier, 1995. Electronic Notes in Theoretical Computer Science 1.

[Peyton Jones and Wadler, 1993] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 71–84. ACM Press, January 1993.

[Plotkin, 1975] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Reddy, 1991] Uday S. Reddy. Acceptors as values: Functional programming in classical linear logic. Available via FTP from cs.uiuc.edu, December 1991.

[Reddy, 1993] Uday S. Reddy. A linear logic model of state. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1993. Distributed electronically.

[Reynolds, 1974] John C. Reynolds. Towards a theory of type structure. In *International Programming Symposium*, pages 408–425. Springer-Verlag, 1974. Lecture Notes in Computer Science 19.

[Reynolds, 1978] John C. Reynolds. Syntactic control of interference. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.

[Reynolds, 1988] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Reynolds, 1989] John C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages, and Programming: 16th International Colloquium, Stresa, Italy*, pages 704–722, New York, July 1989. Springer-Verlag. Lecture Notes in Computer Science 372.

[Riecke and Viswanathan, 1995] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Proceedings of the Twenty-second ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 1–12, New York, January 1995. ACM Press.

[Riecke, 1993] Jon G. Riecke. Delimiting the scope of effects. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 146–155, New York, June 1993. ACM Press.

[Sato, 1994] Masahiko Sato. A purely functional language with encapsulated assignment. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Science, International Symposium TACS'94, Sendai, Japan, Proceedings*, pages 179–202, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science 789.

[Schmidt, 1985] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[Schmidt, 1986] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Stoy, 1977] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Swarup *et al.*, 1991] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[Swarup, 1992] Vipin Swarup. *Type Theoretic Properties of Assignments*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Turner, 1985] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings, Functional Programming Languages and Computer Architecture, Nancy, France*, pages 1–16. Springer-Verlag, September 1985. Lecture Notes in Computer Science 201.

[Wadler, 1990a] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[Wadler, 1990b] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[Wadler, 1991] Philip Wadler. Is there a use for linear logic? In *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, June 1991. SIGPLAN Notices, Volume 26, Number 9.

[Wadler, 1992a] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wadler, 1992b] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14, January 1992.

[Wakeling and Runciman, 1991] David Wakeling and Colin Runciman. Linearity and laziness. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 215–240. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[Weeks and Felleisen, 1992] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in Algol. Technical Report CS 92–193, Department of Computer Science, Rice University, Houston, Texas, 1992.

[Weeks and Felleisen, 1993] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in Algol. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 57–70. ACM Press, January 1993.