DIKU Report 89/11

Declarative Continuations and Categorical Duality

Andrzej Filinski[1]

DIKU – Computer Science Department, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
email: andrzej@diku.dk

August 10, 1989

Abstract

This thesis presents a formalism for reasoning about continuations in a categorical setting. It points out how values and continuations can be seen as categorically dual concepts, and that this symmetry extends to not only data types, but also control structures, evaluation strategies and higher-order constructs. The central idea is a view of continuations as a *declarative* concept, rather than an imperative one, and the implications of this make up the spine of the presentation.

A symmetrical extension of the typed λ-calculus is introduced, where values and continuations are treated as opposites, permitting a mirror-image syntax for dual categorical concepts like products and coproducts. An implementable semantic description and a static type system for this calculus are given. A purely categorical description of the language is also obtained, through a correspondence with a system of combinatory logic, similar to a cartesian closed category, but with a completely symmetrical set of axioms. Finally, a number of possible practical applications and directions for further research are suggested.

[1] Address from Sept. '89: Comp. Sci. Dept., Carnegie Mellon University, Pittsburgh, PA 15213-3890

Preface

This document is the author's "speciale" (Master's thesis), forming part of the credit towards the "cand. scient." (M.S.) degree in Computer Science at DIKU (Computer Science Dept., University of Copenhagen).

Its aim is to bring together two important descriptive formalisms: *category theory* and *continuation semantics*. The former is a very abstract mathematical framework for discussing (among others) functions between structured sets; the latter was introduced mainly for handling 'goto's in programming languages. It would seem that these two subjects had little in common. Yet the present work shows that continuations can not only be forced into a categorical setting (almost everything can), but that they fit in strikingly well, modeling the central categorical concept of *duality*, a precise formulation of "oppositeness".

This thesis was written under the supervision of Dr. Olivier Danvy, who also introduced me to the strange and fascinating world of continuations. His constant encouragement and feedback proved invaluable, and on numerous occasions his initial intuition about a subject turned out to be absolutely correct after lots of hard work on my part. I am deeply grateful for his support, without which this thesis would probably not have been.

Special thanks also go to Prof. Neil D. Jones, whose course on category theory made me "see the light" of this simple yet powerful framework. He pointed out a number of mathematical weaknesses in the early versions of this work, and provided many helpful suggestions at the later stages. I also greatly appreciate the deep comments and ideas from Hans Dybkjær, Karoline Malmkjær, Austin Melton, Torben Mogensen and Philip Wadler, as well as the encouragement and interest shown by John Hannan, John Hughes, John Launchbury and David Schmidt.

A shorter version of this work will appear in [Filinski 89]. It covers much of the material presented in the first two chapters, except evaluation orders, but in a somewhat abridged way.

Copenhagen, July 1989

Contents

1 Introduction **5**

1.1 Background and goals 5
1.2 Overview of the thesis 6
1.3 Category theory 7
1.3.1 Basic definitions 8
1.3.2 Categorical definitions 10
1.4 Continuation semantics 13
1.4.1 The problems of a direct semantics 14
1.4.2 Continuation semantics 16
1.4.3 Semantics of escapes 17
1.4.4 Alternative evaluation orders 19

2 The Symmetric λ-Calculus **21**

2.1 The core of the SLC 21
2.1.1 Functions, values and continuations 21
2.1.2 Basic syntax and conversion rules 22
2.1.3 Denotational semantics 24
2.2 Structured types 25
2.2.1 Products and coproducts 25
2.2.2 Terminal and initial objects 26
2.2.3 Denotational semantics 27
2.3 Summary of the first order language 29
2.3.1 Non-local values and continuations 29
2.3.2 Examples 30
2.4 Higher-order constructs 30
2.4.1 Functions as values and continuations 30
2.4.2 Currying and cocurrying 32
2.4.3 Denotational semantics 33
2.5 Alternative evaluation strategies 34
2.5.1 Pure CBN evaluation 34
2.5.2 CBV with lazy products 37
2.5.3 CBN with eager coproducts 39
2.5.4 A comparison of CBV and CBN 40
2.6 Recursion and iteration 41
2.6.1 Recursive functions, values and continuations 41
2.6.2 Fixpoints and nontermination 42
2.6.3 Denotational semantics 43

2.7 A complete description of the SLC . . . 44
2.7.1 Type system . . . 44
2.7.2 Denotational Semantics . . . 45

3 The Symmetric Combinatory Logic 47
3.1 A categorical view of the SLC . . . 47
3.1.1 Functions, values and continuations . . . 48
3.1.2 Structured types . . . 48
3.1.3 Higher-order features . . . 49
3.2 The SCL category and its axioms . . . 50
3.2.1 Totality and strictness . . . 50
3.2.2 Core axioms . . . 51
3.2.3 Higher-order axioms . . . 52
3.2.4 Summary of morphisms and axioms . . . 53
3.2.5 Derived equalities . . . 53
3.3 Translation from SCL to SLC . . . 55
3.4 Translation from SLC to SCL . . . 55
3.5 SCL and the SLC conversion rules . . . 58
3.6 Evaluation strategies . . . 61
3.6.1 CBV and CBN in the SCL . . . 61
3.6.2 Lazy products in CBV . . . 62
3.6.3 Eager coproducts in CBN . . . 63
3.7 Iteration and recursion . . . 63
3.7.1 Repetition operators . . . 64
3.7.2 Fixpoint morphisms . . . 64
3.7.3 Back to values and continuations . . . 65
3.8 A categorical denotational semantics . . . 66

4 Applications and Examples 68
4.1 Towards a Symmetric Programming Language . . . 68
4.1.1 Notation . . . 69
4.1.2 Labelled datatypes . . . 69
4.1.3 Binding primitives . . . 70
4.1.4 Control primitives . . . 71
4.1.5 Top-level evaluation . . . 71
4.1.6 Asymmetric extensions . . . 71
4.1.7 Restrictions . . . 72
4.2 SLC/SCL as metalanguages . . . 72
4.2.1 The 'goto' . . . 73
4.2.2 Jumps and the type system . . . 74
4.2.3 Multiple returns and label types . . . 76
4.2.4 Exceptions . . . 77
4.2.5 Persistent continuations . . . 79
4.3 Programming in the SLC . . . 81
4.3.1 Non-local exits: tree search . . . 82
4.3.2 Coroutines and streams . . . 83
4.3.3 More on iteration and recursion . . . 85
4.4 Object-oriented programming . . . 86

4.4.1 A symmetry of organizations . . . 86
4.4.2 Specification inheritance . . . 88
4.4.3 Implementation inheritance . . . 89
4.4.4 A look at some object-oriented languages . . . 90
4.5 Abstract interpretation . . . 91
4.6 Other directions for further research . . . 92
4.7 Comparison with related work . . . 93

5 Conclusion **95**

Bibliography **98**

A An Implementation of SLC/SCL **102**
A.1 Concrete syntax and pre-parsing . . . 102
A.2 Abstract syntax and parsing . . . 104
A.3 Translation from SLC to SCL . . . 106
A.4 Type inferencing . . . 108
A.5 Evaluation . . . 112
A.6 Top level interaction . . . 114
A.7 A sample session . . . 115

Chapter 1

Introduction

This chapter presents the background for the thesis. We will start by summarizing the motivation and purpose of the work. After this, we give an overview of the organization and the main results obtained in each of the following chapters. Finally, in order to make the thesis reasonably self-contained, this chapter contains brief introductions to category theory and continuation semantics.

1.1 Background and goals

Category theory has proven a useful formalism for describing semantics of programming languages, since many common language constructs have direct categorical counterparts. However, such descriptions concentrate very much on *data* flow, while many details of the *control* structure, including conditionals, evaluation strategies, *etc.* do not seem to be described adequately in existing presentations. In particular, applicative order reduction (with its termination problems) and imperative features such as "exceptions" and "non-local exits" appear somewhat incompatible with the "declarative" categorical concepts.

On the other hand, a continuation-based denotational semantics is very suitable for describing precisely those phenomena, and enables one to reason formally about programs and their termination properties regardless of the evaluation order of the description language. Also, it provides a clean foundation for handling run-time errors, and for concepts such as backtracking and coroutines. However, it introduces an additional level of abstraction in the semantic description, making it somewhat harder to automate reasoning about programs, for purposes of transformation or correctness proofs.

This thesis is a first step towards building a bridge between these two methods of language description, and in particular to show that, in a suitable framework, first-class continuations can be given a sound declarative meaning, as opposed to being just a powerful but unstructured control primitive. We shall do this by investigating the semantics of a small yet very expressive language, based on the λ-calculus. Unlike conventional programming languages, its main purpose will be to provide the same convenience in reasoning about continuations as about values. In fact, it can be quite accurately characterized as two copies of the λ-calculus, connected back-to-back.

The main idea is based on viewing values and continuations as dual concepts and on generalizing the usual value-based constructs of functional abstraction and application to encompass continuations. This symmetry of language primitives will be mirrored by a symmetry of types, with the two kinds of operations acting on categorically dual types, s.a. products and coproducts. We shall stress this categorical view to the point of developing a combinator-based axiomatic presentation of the same language.

The concept of evaluation order or reduction strategy plays an important role in any language definition. Again, this is easily formulated in a continuation-based framework, but usually harder to express in categorical terms. We will present a view of the language semantics as consisting of a set of "core" axioms, which are satisfied by every reduction strategy, together with additional ones specifying a particular evaluation order. With this separation, we will see that applicative and normal order reduction ("call-by-value" and "call-by-name") are essentially dual strategies. This symmetry is not very apparent for traditional languages, where access to continuations is complicated or non-existent, but becomes much clearer in the symmetric language presented here.

The symmetries exposed in this thesis offer new and somewhat unconventional ways of understanding existing language constructs and concepts. Perhaps even more importantly, it suggests entirely new fields of enquiry for language design and implementation by pointing out dual versions of useful operations and program transformation techniques, for which the theoretical background and practical implications have already been worked out.

As with any descriptive framework, many familiar concepts have to be distorted somewhat to make them fit into the theory. The author believes that this is a small price to pay when compared to the benefits offered by categorical duality, and that the melting pot of programming language design could benefit from a moderate reheating. Many language constructs seem to deviate from the underlying categorical concepts only for historical reasons, rather than by design. Also, the by now almost instinctive aversion to 'goto'-like constructs seems to have inhibited further development of the control primitives required to achieve a better symmetry with data-based operations.

Nevertheless, it is quite likely that the negative implications of deviating from the established path have not been fully understood. After all, very considerable resources have been expended for research in both the theory and practice of programming language design, and some of the ideas presented in the following will probably appear misguided or even plain wrong to a more experienced researcher. It is the author's hope that any such problems will concern the peripheries of the investigation, which could be pruned off as dead branches, rather than the central idea of the work, *i.e.*, treating continuations categorically in a symmetric language.

This thesis represents the current state of work in progress. From the beginning, the emphasis has been on obtaining practically significant results, rather than on precise mathematical structure. Of course, this does not mean that a completely rigorous foundation is not considered important, but that natural time constraints have forced a concentration on investigating broader implications of the preliminary results first. Constructing a well-polished theory without knowing whether it could be used in the real world was considered too big a risk to take. Fortunately, the results obtained so far indicate that a fuller formalization should indeed be both possible and desirable.

1.2 Overview of the thesis

This section gives a chapter-by-chapter overview of the thesis. Structurally, chapters 3 and 4 both depend on material presented in chapter 2, but could be read in any order.

- The rest of this chapter consists of an introduction to Category Theory and Continuation Semantics. Readers familiar with either topic can safely skip the corresponding part, but should be particularly careful about reading the warnings at the end of each section. Only the absolute minimum of required material is covered. Therefore, a reader completely new to both fields may find it quite hard to understand the thesis, or even its purpose, from this introduction alone. Such a reader is strongly advised to consult a standard textbook

on either subject; suitable references will be given at the beginning of each section. Some experience with a modern functional programming language, s.a. Scheme or ML may also be a considerable advantage, but not essential.

- Chapter 2 introduces a notation called the Symmetric λ-calculus, SLC, which we will use to reason about continuations. We will introduce it step-by-step, starting from the core idea, and gradually add structured data, higher-order constructs and recursion. The main theme in the presentation will be duality of values and continuations, with the SLC syntax mirroring the semantic symmetry. The central mode of description will be denotational semantics, although we will mention the categorical foundations of the concepts introduced.

- Chapter 3 covers the Symmetric Combinatory Logic, SCL, which makes up the categorical foundation of the SLC. We will introduce the objects, morphisms and axioms of the SCL as needed to express all SLC concepts. The main result of this chapter is a variable-free representation of the SLC, where flow of both data and control is expressed through a fixed set of categorical combinators. We can see this as factorizing the semantics of the SLC by "compiling away" the concept of substitution. This makes formal reasoning about the SLC much simpler. This chapter will be mostly about category theory, but we will also give a "residual" denotational semantics for the language of categorical terms.

- In chapter 4, we consider some possible applications of the SLC/SCL, and give a number of more substantive examples. We will sketch the design of a practical programming language based on the symmetries presented in the previous two chapters, and point out how many existing language constructs can be expressed in SLC terms. We will also present a few half-baked ideas and suggest directions for future research which look promising but have not been pursued yet. Finally, we point out the similarities with and differences from other related work.

- The conclusion in chapter 5 summarizes the current results and experiences from the SLC/SCL approach to continuations and symmetry. We will point out its particular strengths, but also the weak points and still unsolved problems. Although there are details which could probably benefit from a different organization, it seems certain that the basic ideas for a categorical treatment of continuations are sound and should provide a solid framework for further work in the area.

- Appendix A contains listings of a model implementation of the SLC/SCL, based directly on the equations derived in the thesis. The main purpose of this implementation was to permit evaluation of examples too large to compute by hand, and to enforce precision in the presentation; questions of efficiency have been totally ignored. The implementation consists of modules for parsing, translating, evaluating and type-inferencing of SLC/SCL terms. It is written in the CAML dialect of ML, but does not use any uncommon facilities except a YACC interface for pre-parsing. Thus, provided a convenient way of entering SLC terms can be found, there should be absolutely no problems in translating this to another functional language, such as Standard ML or Scheme.

1.3 Category theory

This section is not intended as a tutorial on category theory. A number of very readable introductory books on the subject exist, *e.g.*, [Arbib & Manes 75,Rydeheard & Burstall 88]. The mathematically inclined reader may prefer [MacLane 71], but this can hardly be recommended

as an introduction for the average computer scientist. For the higher-order constructs the main reference is [Lambek & Scott 86].

Unlike many other branches of mathematics, category theory is basically an organizational framework, and may therefore seem very abstract. While a number of non-trivial theorems can be stated for the "pure" theory, much of the work involving categories consists of applications of the categorical mode of description to other areas of mathematics or even computer science. This thesis is about one such application, and we we will not use any deep theoretical results. Thus, no previous experience with categories should be necessary.

Sometimes, the main text will make reference to rather basic categorical topics, which will not be introduced here, as they do not play a central role. This includes even such central concepts as functors, natural transformations, adjunctions, mono/epimorphisms *etc.* The interested reader can find more details on these in any of the works mentioned above.

1.3.1 Basic definitions

This subsection presents the innermost core of category theory, which is common to virtually all applications. As the reader will see, a category is a rather weak structure, which generalizes many familiar mathematical concepts such as groups or vector spaces. We will give a few examples of how various seemingly unrelated topics fit naturally into the categorical framework.

Definition 1.1 *A category* **C** *consists of a collection of* objects *and* morphisms *or* arrows *between them. A morphism f from object A to object B is written $f : A \to B$. Morphisms can be* composed*: if $f : A \to B$ and $g : B \to C$ are morphisms, there must exist a morphism $(g \circ f) : A \to C$, and this composition must be associative,* i.e.*, if $f : A \to B$, $g : B \to C$ and $h : C \to D$ are morphisms, then it must be the case that $h \circ (g \circ f) = (h \circ g) \circ f$. Finally, for every object A there must exist an* identity *morphism id_A, s.t. for any morphisms $f : A \to B$ and $g : B \to A$, $f \circ id_A = f$ and $id_A \circ g = g$.*

The notation used above suggests functions, and in fact we have the perhaps most obvious example of a category:

Example The category **Set** has sets as objects and (total) functions between them as morphisms. Morphism composition is just function composition, which is associative, and the identity morphism on a set is the function which sends every element to itself. ∎

This example can be extended in various ways, *e.g.*, we have the categories $\mathbf{Set}^{pf}$ of sets and partial functions, **Dom** of domains and continuous functions, **Grp** of groups and group homomorphisms, etc. However, the reader should not be misled into thinking that category theory is just about functions between sets with structure. In fact, a surprising number of other mathematical concepts fit into the framework, and we will mention just two others:

Example Any partially ordered set ("poset") $(M, \sqsubseteq)$ can itself be viewed as a category, with the individual *elements* as objects, and a single (unnamed) morphism between two objects a and b iff $a \sqsubseteq b$. Morphisms can be composed because of the transitivity of $\sqsubseteq$: if $a \sqsubseteq b$ and $b \sqsubseteq c$, then $a \sqsubseteq c$. Associativity of composition is trivially fulfilled, as there can be at most one morphism between two objects. Finally, the existence of identity morphisms is ensured by reflexivity: for every object a, $a \sqsubseteq a$.

We note that the antisymmetry condition is not captured. This is common; virtually every practical category has some extra structure. We can express the antisymmetry in a categorical

way by stating that if there is a morphism from a to b and one from b to a, then $a = b$. In the following, we will write $\mathbf{M}_{\sqsubseteq}$ for the category induced by the partial order (or just preorder, *i.e.*, not necessarily antisymmetric) $\sqsubseteq$ on M. $[\![]$

Example Any theory or deductive system can be viewed as a category with *propositions* or *formulae* as objects and *proofs* as morphisms. We will write $P \vdash Q$ (P *entails* Q) if Q can be proven from P. (The entailment $P \vdash Q$ should not be confused with the *formula* $P \Rightarrow Q$. There is a strong relation between them, however, known as the deduction theorem in formal logic, and functional completeness in category theory. We will return to this towards the end of the next subsection.

Identity amounts to the rule that for every proposition P, $P \vdash P$. Note that this is not an axiom in many formal systems. However, in virtually all, it can be *proven* from the others. Composition means that if $P \vdash Q$ and $Q \vdash R$, then $P \vdash R$. This is known as the transitivity rule, which permits us to build a larger proof out of two smaller ones. Of course, any real deductive system will have more axioms and inference rules than these two.

Unlike the poset example, there can be many different proofs of $P \vdash Q$. However, some of them are *essentially identical*, differing only in form, not content. The categorical framework for deductive systems identifies two proofs with the same *linear structure*, *i.e.*, with the same sequence of non-trivial inference steps. This is precisely what the axioms of identity and associativity express. There may be less obvious reasons why two proofs should be considered identical. For example, they may differ only in the ordering of two independent deductions. Any theory with additional axioms and rules can also specify when two proofs using them are equivalent.

In the following examples we will consider only classical propositional calculus with the usual rules (s.a. de Morgan's laws), but the framework encompasses other systems with more restrictive axioms, *e.g.*, intuitionistic logic. $[\![]$

The aim of category theory is to describe objects only in terms of the arrow structure around them. As we cannot "look inside" objects, we must have another way of expressing their equivalence:

Definition 1.2 *A morphism $f : A \to B$ is an* isomorphism *if there exists a morphism $f^{-1} : B \to A$, s.t. $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$. Two objects A and B are said to be* isomorphic *if there exists an isomorphism $A \to B$.*

In **Set**, isomorphisms are precisely the bijective maps between sets. Thus, all equipotent sets are isomorphic, which may appear undesirable. We must remember, however, that this is only because we have allowed *all* set-theoretical functions as morphisms. So, while, *e.g.*, the sets $\mathbb{R}$ and $\mathbb{R}^2$ are isomorphic in **Set**, they would not be in the category **Top** of topological spaces and *continuous* functions.

In $\mathbf{M}_{\sqsubseteq}$, two elements a and b are isomorphic iff $a \sqsubseteq b$ and $b \sqsubseteq a$. If the preorder $\sqsubseteq$ is in fact a partial order, there are no isomorphisms other than identities.

In a deductive system, two propositions are isomorphic if they entail each other. For example, in a system with conjunctions, the formulas $A \wedge B$ and $B \wedge A$ would usually be isomorphic.

We can now introduce the concept of duality. While not of great direct importance to many applications of category theory, it will be absolutely essential in the rest of the thesis. Unfortunately, it is not easy to give an intuitive explanation of duality in the general case. Hopefully, the examples in this and the following subsection will prove illuminating enough.

Definition 1.3 *For a category* **C**, *the* dual category $\mathbf{C}^{op}$ *is defined as the category with the same objects as* **C**, *but with the direction of all morphisms reversed. In particular, if $f : A \to B$ and*

$g : B \to C$ are morphisms in $\mathbf{C}$*, they will be morphisms* $f : B \to A$ *and* $g : C \to B$ *in* $\mathbf{C}^{op}$*, and their composition in* $\mathbf{C}$*,* $g \circ f : A \to C$ *will be* $f \circ g : C \to A$ *in* $\mathbf{C}^{op}$*.*

The dual category of a poset, $\mathbf{M}_{\sqsubseteq}^{op}$ is the category induced by the reversed ordering $\sqsupseteq$ on the same set.

In the dual category of a deductive system, the original objects are interpreted as *negated* formulae: if $P \vdash Q$ in $\mathbf{D}$ then $\neg Q \vdash \neg P$ in $\mathbf{D}^{op}$. Note that the negation is not part of the formula, but its interpretation.

The dual category of **Set** (or any "functional" category like **Dom**) is hard to imagine, and will not be considered further here. However, much of this thesis is precisely about duality aspects in a **Set**-like category, so we will return to this problem in chapter 2.

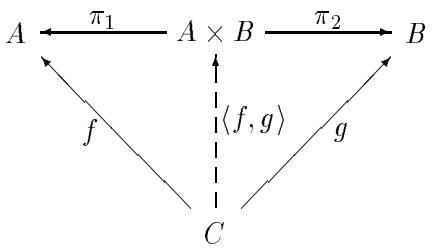
For every concept in a category $\mathbf{C}$, we can speak of the same concept in $\mathbf{C}^{op}$, *i.e.*, with the arrows reversed. Such a concept will also be referred to as the dual of the original one, and its name is often derived by adding the prefix "co-". Some common cases, however, have their own names, and these will also often exhibit some symmetry. We will see lots of examples of dual concepts in the next subsection.

1.3.2 Categorical definitions

Let us consider a categorical characterization of a familiar concept:

Definition 1.4 *A* product *of two objects* A *and* B *in a category consists of an object* $A \times B$ *together with two* projections $\pi_1 : A \times B \to A$ *and* $\pi_2 : A \times B \to B$*, s.t. for every object* C *and morphisms* $f : C \to A$ *and* $g : C \to B$*, there exists a unique* mediating morphism $\langle f, g \rangle : C \to A \times B$*, s.t.* $\pi_1 \circ \langle f, g \rangle = f$ *and* $\pi_2 \circ \langle f, g \rangle = g$*.*

We can express this graphically by means of a *commuting diagram*:



Let us note that the uniqueness condition can be expressed as an unconditional equation:

Theorem 1.1 *The morphism* $\langle f, g \rangle$ *is unique* (i.e., $\pi_1 \circ u = f \wedge \pi_2 \circ u = g \Rightarrow u = \langle f, g \rangle$) *iff for every* $h : C \to A \times B$*,* $\langle \pi_1 \circ h, \pi_2 \circ h \rangle = h$*.*

Proof (If) Let $h : C \to A \times B$ be s.t. $\pi_1 \circ h = f$, $\pi_2 \circ h = g$. Then, by the hypothesis, $h = \langle \pi_1 \circ h, \pi_2 \circ h \rangle = \langle f, g \rangle$.

(Only if) Let $h : C \to A \times B$ be given. Set $f = \pi_1 \circ h$ and $g = \pi_2 \circ h$. Then $h = \langle f, g \rangle = \langle \pi_1 \circ h, \pi_2 \circ h \rangle$. $[\![]$

In **Set**, we can define the object $A \times B$ as the cartesian product of the sets A and B and the projections as $\pi_1 = (a, b) \mapsto a$, $\pi_2 = (a, b) \mapsto b$. For functions $f : C \to A$, $g : C \to B$, we define $\langle f, g \rangle = c \mapsto (f(c), g(c))$. It is easily checked that this satisfies the conditions of the definition.

In the category induced by a poset, the product of two objects (elements) a and b is their greatest lower bound $a \sqcap b$. The projections correspond to the "lower bound" condition, *i.e.*, $a \sqcap b \sqsubseteq a$ and $a \sqcap b \sqsubseteq b$, while the existence of the mediating morphism ensures that if $c \sqsubseteq a$ and $c \sqsubseteq b$, then $c \sqsubseteq a \sqcap b$. The uniqueness condition is trivially fulfilled in this category, as there can be at most one arrow between two objects.

In a deductive system, the product of two objects (propositions) P and Q is their conjunction $P \wedge Q$. The projections are the rules $P \wedge Q \vdash P$ and $P \wedge Q \vdash Q$, while the mediating morphism states that if we have proofs $R \vdash P$ and $R \vdash Q$, we can construct a (unique) proof of $R \vdash P \wedge Q$.

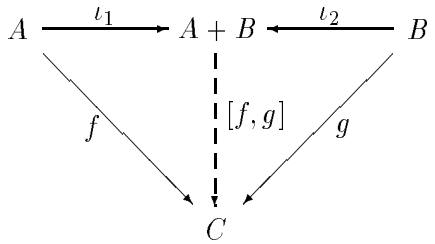
Not all categories have products. For example, not every poset has greatest lower bounds.

For morphisms $f : A \to B$ and $g : C \to D$, it is convenient to define the *morphism* $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle : A \times C \to B \times D$. There is a very good reason for this notation, having to do with functors, but we will not pursue it here.

Let us now consider the dual definition, *i.e.*, the definition of a product with the direction of all morphisms reversed:

Definition 1.5 *A coproduct of two objects A and B in a category consists of an object $A + B$ together with two* injections *$\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$, s.t. for every object C and morphisms $f : A \to C$, $g : B \to C$, there exists a unique* mediating morphism *$[f, g] : A + B \to C$, s.t. $[f, g] \circ \iota_1 = f$ and $[f, g] \circ \iota_2 = g$.*

Diagrammatically:



We have also have a familiar theorem:

Theorem 1.2 *The morphism $[f, g]$ is unique* (i.e., *$u \circ \iota_1 = f \wedge u \circ \iota_2 = g \Rightarrow u = [f, g]$*) *iff for every $h : A + B \to C$, $[h \circ \iota_1, h \circ \iota_2] = h$.*

Proof (If) Let $h : A + B \to C$ be s.t. $h \circ \iota_1 = f$, $h \circ \iota_2 = g$. Then, by the hypothesis, $h = [h \circ \iota_1, h \circ \iota_2] = [f, g]$.

(Only if) Let $h : A+B \to C$ be given. Set $f = h \circ \iota_1$ and $g = h \circ \iota_2$. Then $h = [f, g] = [h \circ \iota_1, h \circ \iota_2]$. $\square$

We see that the proof is virtually identical to the one of theorem 1.1. Therefore we could have stated it as simply "from theorem 1.1 by duality". This shows how duality "cuts the work in half": every time we prove a categorical theorem, we have also proven its dual. We must be careful, however, about the premises of the theorem. For example, we cannot say anything about the existence of coproducts in a category based on the existence or not of products (but we do know that a category has coproducts iff its *dual* has products).

In **Set**, the coproduct of two objects (sets) is their disjoint union $A \uplus B = \{(1, a) \mid a \in A\} \cup \{(2, b) \mid b \in B\}$. The injections are the functions $\iota_1 = a \mapsto (1, a)$ and $\iota_2 = b \mapsto (2, b)$, while the

mediating morphism is the function defined by cases:

$$[f,g] = c \mapsto \begin{cases} f(a) & \text{if } c = (1,a) \\ g(b) & \text{if } c = (2,b) \end{cases}$$

In the category of a poset, the coproduct of two objects a and b is their least upper bound $a \sqcup b$. This is an element with the property that $a \sqsubseteq a \sqcup b$, $b \sqsubseteq a \sqcup b$ (the injections) and for every object c, s.t. $a \sqsubseteq c$, $b \sqsubseteq c$, it is the case that $a \sqcup b \sqsubseteq c$ (the mediating morphism).

In the propositional calculus, the coproduct of two formulae P and Q is their disjunction $P \vee Q$. The injections are the rules $P \vdash P \vee Q$ and $Q \vdash P \vee Q$. The mediating morphism permits the deduction that if $P \vdash R$ and $Q \vdash R$, then $P \vee Q \vdash R$.

For morphisms $f : A \to B$ and $g : C \to D$, we define $f + g : A + C \to B + D$ as $[\iota_1 \circ f, \iota_2 \circ g]$, analogously to products.

It should be clear how the definitions of the binary products and coproducts could be extended to arbitrary finite products and coproducts. This leads to the question of the empty case. Here, the projections disappear, and only the existence and uniqueness of the mediating morphism remain:

Definition 1.6 *A terminal object* 1 *in a category has the property that for every object* A*, there exists a unique morphism* $\Diamond_A : A \to 1$.

We can express the uniqueness of $\Diamond_A$ as the rather drastic-looking equation

$$f = \Diamond_A$$

Of course, this is only meaningful if f has type $A \to 1$.

In **Set**, every singleton set $\{t\}$ is a terminal object, with the unique morphism $A \to 1$ defined as $a \mapsto t$. We note that $A \times 1$ is isomorphic to A for every set A.

In $\mathbf{M}_{\sqsubseteq}$, a terminal object $\top$ has the property that for every object (element) a, $a \sqsubseteq \top$. This is the glb of an empty set (every element is a lower bound, so the glb is just the greatest element of the set).

In a deductive system, the terminal object is the proposition T (truth), and for every proposition P, it is the case that $P \vdash T$. This is the conjunction of zero formulae (the neutral element of $\wedge$).

We can dualize the definition of terminal objects to obtain:

Definition 1.7 *An initial object* 0 *in a category has the property that for every object* A*, there exists a unique morphism* $\Box_A : 0 \to A$

We have an analogous equation

$$f = \Box_A$$

for $f : 0 \to A$.

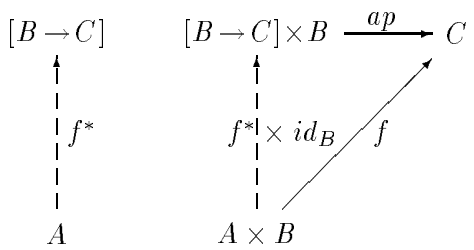
In **Set**, the empty set $\emptyset$ is the only initial object. Its associated unique morphism is the function with the empty domain. We can see $\emptyset$ as the disjoint union of zero sets.

In $\mathbf{M}_{\sqsubseteq}$, an initial object $\bot$ has the property that for every element a, $\bot \sqsubseteq a$. This is the lub of zero elements, where the upper-bound property is trivial.

In a deductive system, the initial object is the formula F (falsehood), and for every formula P, it is the case that $F \vdash P$. F is the disjunction of zero formulae, the neutral element of $\vee$.

Finally, we will consider a slightly more advanced subject, which is nevertheless intuitively understandable. It can be presented in a rather elegant way in terms of adjunctions, but we can easily do without:

Definition 1.8 *An* exponential *of two objects B and C consists of an object $[B \to C]$ and an* application *morphism $ap : [B \to C] \times B \to C$, such that for every morphism $f : A \times B \to C$ there exists a unique morphism $f^* : A \to [B \to C]$, s.t. $ap \circ (f^* \times id) = f$:*



Again, we note that uniqueness can be expressed equationally, for $g : A \to [B \to C]$:

$$(ap \circ (g \times id))^* = g$$

The proof is very similar to the one for uniqueness of products. In fact, they are all instances of a general proof schema, that becomes apparent when the various concepts are expressed as adjunctions.

In **Set**, an exponential object $[B \to C]$ for B and C is the *function space* C^B, *i.e.*, the set of all functions from B to C. ap is the function $(f, a) \mapsto f(a)$. f^* is the function $a \mapsto (b \mapsto f(a, b))$.

In a deductive system, the exponential of two formulae Q and R is the formula $Q \Rightarrow R$. Application is essentially the *modus ponens* rule: $(Q \Rightarrow R) \wedge Q \vdash R$. Also, for every proof $P \wedge Q \vdash R$, there exists a unique proof $P \vdash Q \Rightarrow R$.

In the category of a poset, exponential objects do not correspond to any simple concept, but they may well exist anyway. Let us consider, for example. the poset $\mathcal{P}(S)$, *i.e.*, the set of all subsets of S, ordered by inclusion $\subseteq$. Here, the glb of two subsets M and N is their intersection $M \cap N$. The exponential object for sets B and C is the set $\bar{B} \cup C$, where $\bar{B}$ denotes the complement of B, *i.e.*, $S \setminus C$. It is easily checked that $(\bar{B} \cup C) \cap B \subseteq C$, and that $\forall A : A \cap B \subseteq C \Rightarrow A \subseteq \bar{B} \cup C$.

Definition 1.9 *A category is called* cartesian, *if it has binary products and a terminal object, and* cartesian closed *if it also has exponentials. The abbreviation CCC is commonly used for a Cartesian Closed Category.*

Let us conclude this section by a

Warning While thinking of functional categories in terms of structured sets and functions between them is often a big help, in some cases it may hinder understanding. The category we will work with in the next chapters looks deceptively like **Set** or $\mathbf{Set}^{pf}$, but it is radically different. Thinking of it only in **Set**-based terms will definitely lead into trouble, as some of its fundamental properties will then appear to violate causality. Also, "plausible" results, like the existence of an isomorphism between $A \times (B + C)$ and $A \times B + A \times C$ should certainly not be taken for granted.

1.4 Continuation semantics

The other main prerequisite for this thesis is a knowledge of continuation semantics. Again, we will not really be using any deep results or complicated constructs, but the somewhat peculiar "continuation-passing style" used in the semantic equations may appear quite complicated when seen for the first time.

A good introduction to continuation semantics can be found in most books on denotational semantics, *e.g.*, [Stoy 77], [Schmidt 86]. For a short introduction, see [Reynolds 72] or [Sethi & Tang 80]. In this section, it will be assumed that the reader has at least some familiarity with the general framework of denotational semantics, but no or very little experience with continuations.

Continuations are commonly used to model goto-like constructs. However, as we will only be considering expression-based languages, the presentation here concentrates on "expression continuations", rather than "command continuations". The basic idea is the same, but the concept of jumps in a "declarative" language may appear even stranger than in an "imperative" one. Hopefully, the examples will convince the reader that such a facility is nevertheless both important and useful.

1.4.1 The problems of a direct semantics

The goal of a denotational semantics is to map syntactic constructs of a subject language into semantic terms in a hopefully familiar one (the λ-calculus). A fundamental requirement of such a description is that the meaning of every term should be a function (homomorphism) of the meaning of its subterms. This property is known as compositionality, and makes it possible to reason about denotations in a simple yet precise way, through structural induction.

However, in all but the simplest languages, the meaning of a language construct also depends on its context. For example, the meaning of an expression like $[\![a + 2]\!]$ depends on the value of the variable a when the expression is evaluated. To handle variables in a compositional way, we must make the meaning of a subterm a *function* of the values of the (free) variables occurring in it. We collect these variable values into an *environment*, so that the denotation of an expression becomes a function from environments to values.

Such a semantics of expressions is often called a *direct semantics*. Let us consider a direct semantics for a very simple expression language with the following abstract syntax:

$$E ::= cst \mid V \mid E_1 + E_2 \mid E_1 * E_2 \mid \textbf{let } V = E_1 \textbf{ in } E_2 \mid \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3$$

We will assume that expressions evaluate to integer values, *i.e.*, we have the semantic domains:

$$\begin{aligned} Val &= Int \\ Env &= Var \to Int \end{aligned}$$

and a valuation function

$$\mathcal{E} : Exp \to Env \to Val$$

We will use the letter ρ for environments. We can now give the semantic equations. For simplicity, we will identify syntactic and semantic constants consider any non-zero value as 'true':

$$\begin{aligned} \mathcal{E}[\![cst]\!]\rho &= cst \\ \mathcal{E}[\![V]\!]\rho &= \rho\, V \\ \mathcal{E}[\![E_1 + E_2]\!]\rho &= \mathcal{E}[\![E_1]\!]\rho + \mathcal{E}[\![E_2]\!]\rho \\ \mathcal{E}[\![E_1 * E_2]\!]\rho &= \mathcal{E}[\![E_1]\!]\rho \times \mathcal{E}[\![E_2]\!]\rho \\ \mathcal{E}[\![\textbf{let } V = E_1 \textbf{ in } E_2]\!]\rho &= \mathcal{E}[\![E_2]\!]\,([V \mapsto \mathcal{E}[\![E_1]\!]\rho]\rho) \\ \mathcal{E}[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]\rho &= \mathcal{E}[\![E_1]\!]\rho \neq 0 \to \mathcal{E}[\![E_2]\!]\rho \parallel \mathcal{E}[\![E_3]\!]\rho \end{aligned}$$

This is a simple and readable description of the expression language, and we might use the valuation function $\mathcal{E}$ (somewhat extended, of course) to give the meaning of expressions in a real-sized language. To complete it, we must specify what the initial environment looks like, *e.g.*,

$$\rho_{init} = \lambda V.0$$

This maps all global variables to 0.

Yet this natural way of describing expression evaluation has a fundamental flaw. This is not really apparent for the language above, but let us add a seemingly innocent extension: integer division. We may try to add it directly, by extending the abstract syntax, and giving a semantic equation for the new case:

$$E = \cdots \mid E_1 \textbf{ div } E_2$$

$$\mathcal{E}[\![E_1 \textbf{ div } E_2]\!]\rho \;=\; \lfloor \mathcal{E}[\![E_1]\!]\rho / \mathcal{E}[\![E_2]\!]\rho \rfloor$$

Here $\lfloor x \rfloor$ denotes the floor of x, *i.e.*, the largest integer less than or equal to x. This would probably be an acceptable solution for our toy language, but is too simplistic if we are describing the expression evaluation of a real language. The problem is, of course, the possibility of division by zero. What should be the meaning of $0 * (1 \textbf{ div } 0)$? Even more importantly, is it valid to optimize the expression $x \textbf{ div } x$ to just 1? The semantics above does not answer conclusively to such questions, because it fails to assign any mathematical meaning to a division by zero. We can patch up this problem somewhat by exploiting the special value $\perp$ of our (flat) Val domain:

$$\mathcal{E}[\![E_1 \textbf{ div } E_2]\!]\rho \;=\; \text{let } d = \mathcal{E}[\![E_2]\!]\rho \text{ in } d = 0 \rightarrow \perp \,[\!]\, \lfloor \mathcal{E}[\![E_1]\!]\rho / d \rfloor$$

This is better, but still does not quite solve the problems. For example, what is the meaning of **let** $t = 1$ **div** 0 **in** 2? This is a very real problem, for the two possible answers (2 and $\perp$) are both defensible, and in fact there exist many languages of both kinds. The problem is that the λ-calculus term that we give as the denotation of the let-expression is inherently ambiguous: if it is reduced using applicative order, we obtain $\perp$, but 2 for normal order. This is also a problem when evaluation can fail to terminate because of infinite recursion.

Again, we can limp along by specifying, *e.g.*, that the first expression in a let is always evaluated first, but it does not seem quite right to have to augment a precise mathematical definition by such vague extra information. We can also convey this information through the domain definitions, by precise use of lifting, but there seem to be different conventions regarding this, so that it is not always clear if, *e.g.*, the domain Int already includes a bottom element, or if we must add it ourselves by letting $Val = Int_\perp$.

The problem becomes even worse if we want to recover from run-time errors, instead of abandoning execution completely. For example, let us assume we want to include an expression of the following form:

$$\textbf{try } E_1 \textbf{ else } E_2$$

with the following semantics: first, E_1 is evaluated. If everything goes well, the value of E_1 will be result of the **try** expression, but if evaluation of E_1 encounters a runtime error, E_2 is evaluated instead. For example, the expression:

$$\textbf{try } 100 \textbf{ div } x \textbf{ else } 666$$

would evaluate to $\lfloor 100/x \rfloor$ if x is not zero, and 666 otherwise. Again, the naive attempt

$$\mathcal{E}[\![\textbf{try } E_1 \textbf{ else } E_2]\!]\rho \;=\; \text{let } t = \mathcal{E}[\![E_1]\!]\rho \text{ in } t \neq \perp \rightarrow t \,[\!]\, \mathcal{E}[\![E_2]\!]\rho$$

is not mathematically defensible, as the valuation function is not monotonic, which will cause problems if we ever want to introduce recursion into the language. Instead, we must extend the Val domain with a special "error value", distinct from $\perp$, and then have the semantic equations for arithmetic *etc.* preserve this error value, while try-expressions detect it and substitute something else. This is not unworkable, and we can write an entire semantics in that way [Blikle & Tarlecki 83], but the complexity of the result suggests that maybe our formalism (and perhaps even the entire idea of compositionality) is simply not expressive enough to handle such effects properly.

Fortunately, this is not the case, as pointed out by [Strachey & Wadsworth 74]. The problem is simply that we have not codified from the start the fact that evaluation of an expression can fail and how to deal with it. Also, we have not made precise the order of evaluation, leaving it to the meta-language. Both of these problems are solved by a *continuation semantics*.

1.4.2 Continuation semantics

The key idea is that every expression is evaluated "for a reason". The top-level expression is evaluated because the user wants to know the result. Subexpressions are evaluated because their values will be needed to compute the final result. This seemingly trivial insight is the basis of continuation semantics because it allows the meaning of an expression to depend on (be a function of) what its value will be used for. This leads to a characteristic way of writing semantic equations, where this dependency is made explicit.

We evaluate the top-level expression in order to get an *Answer*. This needs not be the immediate result of the expression. For example, an answer might be a string representing the result as a decimal number. Every time we evaluate a subexpression, it will be a step towards obtaining the answer, and we will always know exactly how to compute the answer, given the value of the subexpression. The function that computes the answer from a subresult is called a *continuation*. The meaning of an expression is then a function from the environment (as usual) and the continuation to the domain of answers. For example, in our toy language, we could have

$$Ans = String$$

and the *initial continuation* being the function

$$\kappa_{init} = IntToString : Val \to Ans$$

We will generally use the letter κ for continuations.

Now, what do the semantic equations look like? We keep in mind that we want to compute the final result of the program, and know how to obtain it from the result of the subexpression. For constants and identifiers, the solution is simple: we just apply the continuation to the result of the subexpression:

$$\begin{aligned} \mathcal{E}[\![cst]\!]\rho\kappa &= \kappa\ cst \\ \mathcal{E}[\![V]\!]\rho\kappa &= \kappa\ (\rho\ V) \end{aligned}$$

We can do this, because evaluating a constant or variable is a simple operation that cannot fail. However, we cannot simply compute the sum of two expressions like this:

$$\mathcal{E}[\![E_1 + E_2]\!]\rho\kappa = \kappa(\mathcal{E}[\![E_1]\!]\rho\kappa_{init} + \mathcal{E}[\![E_2]\!]\rho\kappa_{init})$$

because the denotation of an expression in an environment and with a continuation is a string, not a number. Even if we had $Ans = Val$, and $\kappa_{init} = \lambda x.x$, the above would be incorrect: we are not

evaluating E_1 because its result will be the final answer (as using κ_{init} implies), but because we will need it for the addition. Therefore, to compute the final answer, we must first compute E_1 (or E_2, but we must pick one). After we have the result of E_1, we will evaluate E_2, and when we have that, we can add them. Only then do we apply the continuation, which computes the final result of the program, given the result of $E_1 + E_2$:

$$\mathcal{E}[\![E_1 + E_2]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho\{\lambda a.\mathcal{E}[\![E_2]\!]\rho\{\lambda b.\kappa\,(a+b)\}\}$$

We will often use braces instead of parentheses for continuations. Multiplication is analogous to addition. For a **let**-expression we have to make a choice: do we evaluate E_1 even if the result is not used in E_2? Let us consider the case where we do; we will treat the other possibility in section 1.4.4. We therefore first compute the value of E_1, and then evaluate E_2 in an environment where the variable is bound to this value. After we have evaluated E_2, there is nothing more to do in order to evaluate a **let**, so E_2's continuation will simply be the continuation of the **let**-expression:

$$\mathcal{E}[\![\textbf{let } V = E_1 \textbf{ in } E_2]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho\{\lambda a.\mathcal{E}[\![E_2]\!]\,([V \mapsto a]\rho)\,\kappa\}$$

Finally, let us consider conditional expressions. Again, we first evaluate E_1 and then either E_2 or E_3, with the original continuation:

$$\mathcal{E}[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho\{\lambda b.b \neq 0 \to \mathcal{E}[\![E_2]\!]\rho\kappa \,[\!]\, \mathcal{E}[\![E_3]\!]\rho\kappa\}$$

We have now completed the continuation semantics of our language. The equations are somewhat more complicated, but not very much so. Let us then see how we handle the problems of the direct semantics.

1.4.3 Semantics of escapes

First, let us consider the case when we cannot recover from runtime errors. We want the behavior that if we ever attempt to divide by zero, the program execution will be stopped, and the answer will be (the string) "division by zero". We can express this directly, because every expression is computing the final value. So, we evaluate the dividend and divisor, and if the latter is zero, the error message will be the final result, otherwise we proceed as usual:

$$\mathcal{E}[\![E_1 \textbf{ div } E_2]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho\{\lambda n.\mathcal{E}[\![E_2]\!]\rho\{\lambda d.d = 0 \to \text{“division by zero”} \,[\!]\, \kappa\lfloor n/d \rfloor\}\}$$

This semantics is completely unambiguous. For example, we have

$$\begin{aligned}
&\mathcal{E}[\![\textbf{let } x = 1/0 \textbf{ in } 2]\!]\rho\kappa \;= \\
&\qquad \mathcal{E}[\![1/0]\!]\rho\,\{\lambda t.\mathcal{E}[\![2]\!][x \mapsto t]\rho\,\kappa\} \\
&= \quad \mathcal{E}[\![1]\!]\rho\,\{\lambda n.\mathcal{E}[\![0]\!]\rho\{\lambda d.d = 0 \to \text{“division by zero”} \,[\!]\, \mathcal{E}[\![2]\!][x \mapsto \lfloor n/d \rfloor]\rho\,\kappa\}\} \\
&= \quad (\lambda n.\lambda d.d = 0 \to \text{“division by zero”} \,[\!]\, \kappa\ 2)\ 0\ 1 \\
&= \quad 0 = 0 \to \text{“division by zero”} \,[\!]\, \kappa\ 2 \\
&= \quad \text{“division by zero”}
\end{aligned}$$

and we could not have reduced this in any other way.

Runtime errors are just a special case, however. We can also introduce a more general construct for abandoning the current path of evaluation and returning a new final result. We add the following construct to the language:

$$E = \cdots \mid \textbf{resultis } E$$

If a **resultis** expression is ever evaluated in any context, the value of its argument will become the final result of the program. For example, consider the expression:

$$1 + (\textbf{if } x \textbf{ then } 10 \textbf{ div } x \textbf{ else resultis } 666)$$

If x is non-zero, this evaluates to $1 + \lfloor 10/x \rfloor$. However, if $x = 0$, the result of the expression (and the entire program, if this was a subexpression) will become 666 (not 667). The semantic equation for **resultis** is

$$\mathcal{E}[\![\textbf{resultis } E]\!]\rho\kappa = \mathcal{E}[\![E]\!]\rho\kappa_{init}$$

where we never use the original result continuation κ. Of course, if evaluation of E encounters another **resultis**, we will once again discard the continuation and restart with the initial one. We note that it was not necessary to modify the other semantic equations in order to handle division by zero, or even **resultis**. We only had to specify, for the addition, *e.g.*, which operand is evaluated first, in order to disambiguate expressions like $1/0 + \textbf{resultis } 2$.

Let us now consider an even harder problem: recovery from runtime errors. We will treat errors as a special case of a generalized **resultis** construct, as it gives the simplest presentation. The problem with the existing **resultis** is that always returns the final answer of the program. Sometimes we want to limit its scope, so that it can be used to specify directly the result of a subexpression, not necessarily the whole program. For example, suppose we are multiplying a list of numbers. If we encounter a zero, the result of the entire multiplication will be zero (ignoring the possibility of runtime errors), but we cannot use a simple **resultis**, because the result of the multiplication is not necessarily the final result of the program. We can obtain a delimited version of **resultis** with the two constructs:

$$\textbf{block } B \textbf{ is } E$$

$$\textbf{resultof } B \textbf{ is } E$$

Here, B is a block name (a 'label'). The intent is that the expression in a block is evaluated normally. However, inside that expression we can use the construct **resultof** B **is** E to abandon execution of the block expression and return an alternate result. For example,

$$3 + \textbf{block } b \textbf{ is } (4 + \textbf{resultof } b \textbf{ is } 5)$$

evaluates to 8. For our list multiplication example, we might have something like:

$$\begin{aligned}&...(\textbf{block } mul \textbf{ is}\\ &\quad (\textbf{if } x_1 \textbf{ then } x_1 \textbf{ else resultof } mul \textbf{ is } 0) * \cdots *\\ &\quad (\textbf{if } x_n \textbf{ then } x_n \textbf{ else resultof } mul \textbf{ is } 0))...\end{aligned}$$

This expression may occur in any context, symbolized by the '...'. Of course, in a real language the multiplication would typically be performed by some kind of repetition construct, for example:

```
block mul is
  letrec ml l = if l=[] then 1
                else if hd(l)=0 then resultof mul is 0
                else hd(l)*ml(tl(l))
  in ml [1,2,0,4,5]
```

Block names are identifiers just as let-variables. We could have a separate environment for them, but we can also use the existing one, which handles non-local values. Some identifiers will then denote ordinary values, and some will denote block names. We will adopt the usual syntactic

scope rules, *e.g.*, within a let-expression, it will not be possible to escape to an outer block with the same name as the let-variable.

So what should the environment map a block name to? When we encounter a **resultof**, we want to continue as if we had just finished evaluating the expression part of the designated block. That is, we want to reactivate the continuation of the block. The simplest way to achieve this is simply to let block names denote continuations, *i.e.*, the type of the environment becomes:

$$Env = Var \to Val + Cnt$$

Of course, we have to modify the equations that actually use the environment, to add and remove the injection tags. However, as it is always immediately decidable if an identifier denotes a block or a value, this processing could be done during compilation and need not imply any overhead during execution.

The meaning of a **block** construct is that its expression is evaluated normally, but that the continuation of the block is made accessible for escaping:

$$\mathcal{E}[\![\textbf{block } B \textbf{ is } E]\!]\rho\kappa \;=\; \mathcal{E}[\![E]\!]\,([B \mapsto cnt(\kappa)]\rho)\,\kappa$$

A **resultof** discards the current continuation and installs the one denoted by the block identifier:

$$\mathcal{E}[\![\textbf{resultof } B \textbf{ is } E]\!]\rho\kappa \;=\; \text{let } cnt(\kappa_B) = \rho\ B \text{ in } \mathcal{E}[\![E]\!]\rho\kappa_B$$

Now, how do we handle **try**? We can introduce a special continuation identifier 'error', which will be activated by runtime errors:

$$\mathcal{E}[\![\textbf{try } E_1 \textbf{ else } E_2]\!]\rho\kappa \;=\; \text{let } \kappa_e = \lambda a.\mathcal{E}[\![E_2]\!]\rho\kappa \text{ in } \mathcal{E}[\![E_1]\!]([\text{'}error\text{'} \mapsto cnt(\kappa_e)]\rho)\kappa$$

Here, κ_e denotes a special continuation which is only invoked for runtime errors. It is called with a final answer, (*i.e.*, an error message in our example), which will be discarded because we are handling the error ourselves. The initial 'error' continuation, *i.e.*, the denotation of *error* in the initial environment would return the message as the final answer of the program. We also note that since all try-expressions share the same identifier *error*, we always return to the one which last defined it. However, an error in E_2 will be handled by the enclosing **try**, if any, because it is evaluated in the original environment ρ.

Now, the meaning of a construct which may cause a runtime error becomes:

$$\begin{aligned}\mathcal{E}[\![E_1 \textbf{ div } E_2]\!]\rho\kappa \;=\;& \text{let } cnt(\kappa_e) = \rho\ \text{'}error\text{'} \text{ in} \\ & \mathcal{E}[\![E_1]\!]\rho\{\lambda n.\mathcal{E}[\![E_2]\!]\rho\{\lambda d.d = 0 \to \kappa_e\text{“division by zero”} \,[\!]\, \kappa\,\lfloor n/d \rfloor\}\}\end{aligned}$$

1.4.4 Alternative evaluation orders

Let us finish by showing how we could express another evaluation order for let-expressions. We will now specify that **let** is semantically equivalent to textual substitution (modulo name clashes). In particular, if the defined identifier does not occur in the body, its defining expression will never be evaluated. We do this by stipulating that variables denote residual computations, so that evaluating a variable my be a complex process. Like all such, it may cause a run-time error or an escape, so it should be evaluated with a continuation. We must therefore modify the denotation of value identifiers (ignoring blocks for simplicity)

$$Env = Var \to (Cnt \to Ans)$$

The equation for variable evaluation becomes

$$\mathcal{E}[\![V]\!]\rho\kappa = (\rho\, V)\, \kappa$$

And let-expressions have the denotation:

$$\mathcal{E}[\![\textbf{let } x = E_1 \textbf{ in } E_2]\!]\rho\kappa = \mathcal{E}[\![E_2]\!]\,([x \mapsto \lambda c.\mathcal{E}[\![E_1]\!]\rho c]\rho)\,\kappa$$

Again, this equation is explicit about the denotation of **let** $t = 1$ **div** 0 **in** 2: the body 2 will be evaluated first, and since it does not contain t, the denotation of 1 **div** 0 is of no concern. In fact, it can be shown [Plotkin 75] that the meaning of a term given by a continuation semantics is independent of the evaluation order of the metalanguage, even in the presence of recursive facilities and nontermination. Intuitively, this is because of the property that before every reduction step, there will be at most one β-redex outside of an abstraction, so that the sequence of reductions is completely specified (for a strategy with weak head normal forms).

Like for the category theory section, we conclude by a

Warning User-accessible continuations may look horrible to a reader with a strong functional programming background. They are indeed a very sharp tool, and should not be used without good reason. However, this is mostly because they are introduced as an "imperative" add-on to a "declarative" language. In the next chapter, we will present a language where explicit access to continuations will be a central *declarative* concept, not a parenthesis in the language definition. It will therefore be vital not to consider the continuation-manipulating operations as in any way inferior or more imperative than the traditional value-based constructs. Many of the simpler concepts can indeed be twisted into a continuation-free framework, but doing so will almost certainly impede understanding of the later parts.

Chapter 2

The Symmetric λ-Calculus

This chapter introduces the notation we will use for reasoning about continuations. It takes the form of a simple, functional language, similar in principle to the typed λ-calculus, but containing a continuation-based part of equal importance to the usual expression-oriented half. Because this language will contain the traditional λ-calculus as a subset, together with what might be called its mirror image, we will refer to it as the *Symmetric λ-Calculus, SLC.*

As the full SLC contains many new ideas, it may appear overwhelming if presented all at once. We will therefore start with the core of the language, and gradually add new concepts, such as structured types, higher-order constructs and recursion, trying at all times to point out how familiar, value-based concepts can be dualized in terms of continuations. Each such section will be concluded by a conventional denotational semantics, giving a precise description of the new concepts. A complete semantics, better suited for reference can be found in section 2.7, together with the full type system of the language.

While the SLC will be presented informally in a basically strategy-independent way, its semantical description will be in terms of applicative-order (call-by-value) reduction. While this may be easier to understand, it is in no way essential, and section 2.5 will present an alternative, normal-order (call-by-name) semantics. This section will point out yet another symmetry of SLC, namely the essentially dual properties of the two evaluation strategies. The section on reduction order will in fact appear before the discussion of recursion and iteration, since the treatment of these topics is closely tied to the evaluation strategy.

2.1 The core of the SLC

This section presents the central idea behind the SLC, together with the basic syntax and semantics of the core language.

2.1.1 Functions, values and continuations

Functions are traditionally viewed as value transformers, and the common notation $x \mapsto \varphi(x)$ or syntactic variants thereof reinforce this view. However, a different perspective on functions is given by the data-flow paradigm of computation, namely as *request* transformers. For example, the function $odd : int \to bool$, which is usually thought of as transforming an integer value to a boolean value, could also be seen as transforming a request for a boolean value to a request for an integer.

We note that the roles of values and requests depend on the evaluation strategy. In an eager or *data-driven* evaluation, the flow of control is determined by values while the requests are implicit.

On the other hand, in a lazy or *demand-driven* environment, the requests actively pull forward the evaluation while values have a passive existence.

We can extend this view to continuation semantics, where continuations in a sense act as requests. In a lazy language, they drive forward the computation as nothing is evaluated until someone asks for the result. In an eager language, on the other hand, continuations just accept already computed results. In both cases, however, a continuation can be thought of as the *lack* or *absence* of a value, just as having a negative amount of money represents a debt. This analogy even extends to "debt transformation": one can repay a creditor by borrowing the money from someone else. In fact, some economic situations seem easier to understand in terms of changing debts and deficits rather than a genuine flow of money.

2.1.2 Basic syntax and conversion rules

With this informal view of values and continuations as dual concepts, let us try to construct a language where the two are treated truly symmetrically. First, we must (temporarily) abandon the λ-calculus amalgamation of functions and values and distinguish sharply between three different syntactic classes: functions, values and continuations. Any function can be used either as a value transformer or a continuation transformer. Conversely, a function may be defined either in terms of its effect on the input value or on the result continuation.

We write a function application to a value using an explicit operator: $F \uparrow E$. This denotes the *result* or *output* of E transformed by F and corresponds exactly to juxtaposition in the λ-calculus. They types are also familiar: if F is a function from S to T, and E is an expression of type S, $F \uparrow E$ will be an expression of type T. Symmetrically, we write a continuation C transformed by F as $C \downarrow F$. This denotes a new continuation which will first transform its *input* with F. Let us note how the types are reversed: since F returns a T-typed result, C must also have type T, while $C \downarrow F$ will denote a S-accepting continuation.

We can specify functions either as value abstractions $x \Rightarrow E$ or as continuation abstractions $y \Leftarrow C$. The former describes how to transform an input value into a result value, while the latter specifies how a request for the result is transformed into a request for the input. Again we see a symmetry of types: if x denotes a S-typed variable, and E is a T-typed expression, the function $x \Rightarrow E$ will be of type $S \rightarrow T$. On the other hand, since the y in a function $y \Leftarrow C$ from S to T abstracts the result continuation, it should be T-typed, while the new continuation C should be S-accepting.

Note that since continuations are not treated as functions, we can never explicitly *apply* a continuation to a value, but only substitute the current continuation with another, possibly defined in terms of the original. The importance of this cannot be stressed enough: SLC continuations are black holes from which there is no return. Neither can a function by itself be used as a continuation, because we cannot specify a destination for the result, after it has been processed by the function.

Finally, in addition to explicit abstractions, we include a number of primitive functions (*e.g.*, arithmetic operators), corresponding to δ-rules. Note that these can also be used as either value or continuation transformers, as remarked in section 2.1.1 for the function *odd*. For simplicity, we will assume that these primitive functions cannot themselves escape *etc.*, as any such behavior can be expressed explicitly in the SLC through continuation abstractions.

To justify the name of symmetric λ-calculus, we must at least include all of the the conversion rules of the ordinary λ-calculus together with their mirror images (with the usual restrictions to prevent capturing of free variables):

$$
\begin{array}{rcl@{\qquad}rcl}
a \Rightarrow E & \overset{\alpha}{\leftrightarrow} & b \Rightarrow E[b/a] & a \Leftarrow C & \overset{\bar{\alpha}}{\leftrightarrow} & b \Leftarrow C[b/a] \\
(x \Rightarrow E_1) \uparrow E_2 & \overset{\beta}{\leftrightarrow} & E_1[E_2/x] & C_2 \downarrow (y \Leftarrow C_1) & \overset{\bar{\beta}}{\leftrightarrow} & C_1[C_2/y] \\
(x \Rightarrow F \uparrow x) & \overset{\eta}{\leftrightarrow} & F, \text{ if } x \text{ not free in } F & (y \Leftarrow y \downarrow F) & \overset{\bar{\eta}}{\leftrightarrow} & F, \text{ if } y \text{ not free in } F
\end{array}
$$

Furthermore, as we want to use category theory, we must organize the SLC in a suitable way. A natural choice consists of taking types as objects, and functions (of both kinds!) as morphisms. Note that there is no concept of value flow in the definition of a category, so only the type of the function, not its mode of definition matters. To have a proper category, we must also define identity and function composition, and the symmetry of the conversion rules permits us to do it in two equally acceptable ways, with either value or continuation abstractions. As we do not want to favor any of these, they should be equivalent, giving rise to the following two rules, (where x and y do not occur free in F and G):

$$
\begin{array}{rcl}
x \Rightarrow x & \overset{id}{\leftrightarrow} & y \Leftarrow y \\
x \Rightarrow F \uparrow (G \uparrow x) & \overset{\circ}{\leftrightarrow} & y \Leftarrow (y \downarrow F) \downarrow G
\end{array}
$$

Because of these identities, we can take either side of the rules as definition. It is easily checked that neutrality of identity and associativity of composition follows from the conversion rules.

Strictly speaking, a term like $x \Rightarrow x$ does not by itself denote a function, but rather a *function template*, from which we can make instances by supplying a type. In other words, there is a distinct identity morphism for every object. This will become significant when higher-order constructs are added to the language. We could also view $x \Rightarrow x$ as a single, polymorphically-typed function, but this adds an extra level of complication.

Unfortunately, having both the β and $\bar{\beta}$ rules in the general form presented above leads to an inconsistent system, where totally unrelated terms become interconvertible. For example, the function

$$(x \Rightarrow 2) \circ (y \Leftarrow k)$$

where k is a continuation from an outer scope may be reduced to either $x \Rightarrow 2$ or $y \Leftarrow k$. This is not a problem specific to the SLC, but a general manifestation of a weakness in every applied λ-calculus with partial functions as δ-rules, since runtime errors disturb the continuation. For example, the expression

```
let x=1/0 in 2
```

will evaluate to 2 in a call-by-name (CBN) language, but fail in a call-by-value (CBV) one.

Therefore, virtually all programming language definitions specify an evaluation order, *i.e.*, a set of restrictions on possible reductions. In the following we will assume a CBV strategy, so that β-conversion can only be performed on completely reduced arguments, but with no restrictions on the $\bar{\beta}$-rule. In section 2.5, alternative strategies will be considered.

The significance of the composition rule should now be apparent: it can be used to transform terms to a form where the restricted β-rule can be applied, by shifting nested function applications onto the continuation. This is exactly what happens in a continuation-based interpreter, such as the denotational semantics of the next subsection: during evaluation, an explicit continuation is built up, so that functions are only applied to simple values. It can be directly verified that all the given rules, except full β-reduction, hold in the CBV semantics, and that the identity and composition rules satisfy their associated axioms (neutrality and associativity).

2.1.3 Denotational semantics

We can summarize the syntax of the SLC subset introduced so far as follows:

$$\begin{array}{rcl} E & ::= & cst \mid x \mid F \uparrow E \\ C & ::= & y \mid C \downarrow F \\ F & ::= & x \Rightarrow E \mid y \Leftarrow C \mid prim \\ x, y & ::= & Ide \end{array}$$

For simplicity, we will consider the sets of value and continuation identifiers to be disjoint. In most examples, we will use names like x, a or v for values, and y, c or k for continuations, but in the general case, the kind of an identifier must be inferred from the context of its use and/or its introduction.

Let us also remark on an unfortunate asymmetry of language before we proceed: while we distinguish between *expressions* and *values*, with the former denoting the latter, the term *continuation* covers both uses. Fortunately, this is not a big problem for CBV evaluation, since every *syntactic* continuation also denotes a *semantic* continuation, unlike some expressions which may not denote values because they escape.

We will need two semantic domains, Val and Cnt, representing values and continuations respectively, for assigning meanings to SLC terms. Let us postpone their precise definition for a moment and consider instead the types of the valuation functions for each of the three classes.

First, we note that in order to model substitution denotationally, the meaning of a construct must be given relative to an environment. This should map value identifiers to values and continuation identifiers to continuations. Strictly speaking, it would therefore be represented as a pair of functions, but we will just define it as a single function, mapping an identifier to either a value or a continuation:

$$Env = Ide \rightarrow Val + Cnt$$

In a (CBV) continuation semantics, the meaning of an expression (in an environment) is given relative to a continuation representing the rest of the computation. Dually, therefore, the meaning of a syntactic continuation should be given relative to a value, representing the result computed so far. Finally, a SLC function is invoked with both an input value and an output continuation, of which one will be explicit, depending on the application type. We can summarize the types of the semantic functions in a strategy-independent way:

$$\begin{array}{rcl} \mathcal{E} & : & E \rightarrow Env \rightarrow Cnt \rightarrow Ans \\ \mathcal{C} & : & C \rightarrow Env \rightarrow Val \rightarrow Ans \\ \mathcal{F} & : & F \rightarrow Env \rightarrow Val \rightarrow Cnt \rightarrow Ans \end{array}$$

We have presented the types in curried form, to reduce the clutter of parentheses brought about by tupling. However, in the semantic equations, we will never apply a function to only a subset of its parameters, so uncurried definitions could equally well be used.

Let us now consider a particular evaluation strategy, CBV, and give a proper denotational semantics of the language. The valuation functions will necessarily look somewhat asymmetrical, since we are translating a symmetrical language into an asymmetrical one, while fixing a particular evaluation strategy. Of course, as with all continuation semantics, the evaluation order of the defining language (the λ-calculus) does not affect the meaning of the defined language (SLC).

For the semantic domain of values, we will use an unspecified domain of basic values (*e.g.*, integers), with continuations mapping values to answers:

$$\begin{aligned} a, t, v \in Val &= Basic \\ c, \kappa \in Cnt &= Val \to Ans \end{aligned}$$

For expressions, we have:

$$\begin{aligned} \mathcal{E}[\![cst]\!]\rho\kappa &= \kappa\ cst \\ \mathcal{E}[\![x]\!]\rho\kappa &= \text{let } val(v) = \rho\ x \text{ in } \kappa\ v \\ \mathcal{E}[\![F \uparrow E]\!]\rho\kappa &= \mathcal{E}[\![E]\!]\rho(\lambda t.\mathcal{F}[\![F]\!]\rho t\kappa) \end{aligned}$$

To keep things simple, we have identified syntactic and semantic constants, and the same for identifiers.

For continuations, there are no constants, so we just get

$$\begin{aligned} \mathcal{C}[\![y]\!]\rho v &= \text{let } cnt(c) = \rho\ y \text{ in } c\ v \\ \mathcal{C}[\![C \downarrow F]\!]\rho v &= \mathcal{F}[\![F]\!]\rho v(\lambda t.\mathcal{C}[\![C]\!]\rho t) \end{aligned}$$

And finally, for functions:

$$\begin{aligned} \mathcal{F}[\![x \Rightarrow E]\!]\rho v\kappa &= \mathcal{E}[\![E]\!]\ ([x \mapsto val(v)]\rho)\kappa \\ \mathcal{F}[\![y \Leftarrow C]\!]\rho v\kappa &= \mathcal{C}[\![C]\!]\ ([y \mapsto cnt(\kappa)]\rho)v \\ \mathcal{F}[\![prim]\!]\rho v\kappa &= \kappa(prim\ v) \end{aligned}$$

Here, the last rule should be viewed as a specific equation for every primitive function.

2.2 Structured types

So far, we had only considered one type of basic values. Virtually all strongly-typed programming languages provide support for building new types out of existing ones. The goal of this subsection is to present the syntax for doing this in the SLC. We will see how continuations play a crucial role in this definition.

2.2.1 Products and coproducts

The two basic schemas for constructing new types out of old ones are the tuple (record, structure) and the disjoint sum (variant, union). Their relation with the categorical concepts of products and coproducts is obvious, but the syntax and semantics associated with them seldom exhibit many similarities, perhaps mainly for historical reasons.

Products are expressed naturally in a value-based view, and in the SLC, we will directly adopt the simple, traditional way to write categorical projections:

$$\pi_1 \equiv (a, b) \Rightarrow a : A \times B \to A \qquad \pi_2 \equiv (a, b) \Rightarrow b : A \times B \to B$$

Here, the projection π_1 transforms a *value* of type $A \times B$ into a value of type A by selection from the pair (a, b). Similarly, given functions $f : C \to A$ and $g : C \to B$, we write the mediating morphism:

$$\langle f, g \rangle \equiv c \Rightarrow (f \uparrow c, g \uparrow c) : C \to A \times B$$

The value-based reasoning behind this notation is clear: the function $\langle f, g \rangle$ constructs a pair of values, from which one can later be extracted using a projection.

Let us now consider coproducts in terms of continuations: the injection $\iota_1 : A \to A + B$ transforms a request for either an A or a B into a request for an A. Now, a request for a coproduct type is really a pair of requests, of which one should be satisfied, *i.e.*, a selection between two continuations. In the SLC, the selection aspect is made explicit by the following syntax:

$$\iota_1 \equiv \{a, b\} \Leftarrow a : A \to A + B \qquad\qquad \iota_2 \equiv \{a, b\} \Leftarrow b : B \to A + B$$

Curly braces are used for clarity, to emphasize the fact that the definition is not expressed in terms of values but continuations. However, the direction of the arrow is sufficient to determine the meaning, so simple parentheses could be used instead.

Furthermore, given functions $f : A \to C$ and $g : B \to C$, the function $[f, g] : A + B \to C$ transforms a request for a C into a request for either an A or a B. In the first case, the function f is applied to obtain the result, otherwise, the function g:

$$[f, g] \equiv c \Leftarrow \{c \downarrow f, c \downarrow g\} : A + B \to C$$

2.2.2 Terminal and initial objects

Now, let us examine the border cases of products and coproducts: terminal and initial objects. Unfortunately, there seems to be no universally accepted way of writing the associated unique morphisms. We will therefore use a notation suggestive of the binary case, writing $\Diamond_A : A \to 1$ for the empty product and $\Box_A : 0 \to A$ for the empty sum, omitting the type subscripts when no confusion is possible.

The terminal object has an important role in programming. In some languages, it has an associated explicit type (usually called 'unit' or 'void'); in others it is represented by an empty (or absent) argument list on input, and 'procedure' instead of 'function' for output. Actually, 'void' is a misnomer, since the type does contain a value, which may just not be directly expressible. We will use the names *unit* and 1 synonymously in the following.

Considering 1 as an empty product, we write the unique value of type *unit* as $()$. For every type A, we have a function

$$\Diamond_A \equiv a \Rightarrow () : A \to unit$$

which ignores its parameter. This is important from a theoretical point of view, as it provides a basis for non-strict functions. In practical languages, however, functions with this type are only used for any side effects they might have, since the value they return will always be the same and thus contains no information.

On the other hand, we have a bijection between T-typed expressions E, and functions $E' : unit \to T$, given by:

$$E' = () \Rightarrow E : unit \to T \qquad\qquad E = E' \uparrow () : T$$

This correspondence will become important later, when higher-order constructs are added to the language, and we can pass functions like E' around. Often, such parameterless functions are called *suspensions* or *thunks*, and are used to obtain CBN-like behavior in a CBV language by delaying evaluation.

The initial object 0 rarely manifests itself in programming, but has a natural interpretation as well. In the category **Set**, it is simply the empty set, *i.e.*, the type *null* with *no* values. ('void'

would be a better name, but this has already been taken). Again, the unique morphism $\Box : 0 \to A$, naturally written like this:

$$\Box_A \equiv a \Leftarrow \{\} : null \to A$$

is not very interesting by itself, but its significance is that it discards a *continuation*, and thus can be used to model imperative constructs, such as escapes, jumps, *etc.* Thus, both $\Diamond$ and $\Box$ play central roles in the categorical definition of SLC.

We also have a correspondence between S-accepting continuations C and functions $C' : S \to null$:

$$C' = \{\} \Leftarrow C : S \to null \qquad\qquad C = \{\} \downarrow C' : S$$

$S \to null$ is the type of a function which *never* returns to the point of call, as there is no possible value it can return with. Usually, no functions like that are written (on purpose), but some languages, notably C, present a number of control primitives as standard functions, *e.g.*, `exit` and `longjmp`. The return type of these could be conveniently declared as *null*, and we will return to this in section 4.2.2.

Note that these constructs only present a syntactic formalism for products, coproducts and initial/terminal objects; while the associated morphisms are defined in all cases, whether they actually satisfy the axioms depends on the evaluation order of the language. For example, the SLC with CBV semantics does not have true products unless morphisms are restricted to "well-behaved" functions. In Lisp terms, the McCarthy axiom `car[cons[A,B]] = A` does not hold for *all* expressions `A` and `B`, *e.g.*, evaluation of `B` may fail to terminate. In fact, not even empty products exist under CBV: even ignoring side effects such as assignments, I/O, *etc.*, there may be many different functions from a given type to *unit*, differing in termination behavior and escapes.

On the other hand, the CBV SLC does have true coproducts and a proper initial object, even in the presence of continuation abstractions or non-termination. The precise problems with existence and uniqueness are deferred to the next chapter, however.

2.2.3 Denotational semantics

Let us add structured types to our CBV semantics. First, we extend the abstract syntax, to permit structured expressions and continuations. We also introduce two new syntactic domains to denote structured value and continuation parameters:

$$\begin{array}{rcl} E & ::= & cst \mid x \mid () \mid (E_1, E_2) \mid F \uparrow E \\ C & ::= & y \mid \{\} \mid \{C_1, C_2\} \mid C \downarrow F \\ F & ::= & X \Rightarrow E \mid Y \Leftarrow C \mid prim \\ X & ::= & x \mid () \mid (X_1, X_2) \\ Y & ::= & y \mid \{\} \mid \{Y_1, Y_2\} \end{array}$$

Of course, no variable may be repeated in a pattern.

The two new classes have their own valuation functions, which extend the environment with values and continuations respectively:

$$\begin{array}{rcl} \mathcal{X} & : & X \to Val \to Env \to Env \\ \mathcal{Y} & : & Y \to Cnt \to Env \to Env \end{array}$$

To handle structured types, we also need more refined semantic domains. Again, as this is CBV, everything happens in the value part. We will write the domain as a large sum, but it should

be kept in mind that the language is strongly-typed, and we could instead express the definition as a number separate Val domains, indexed by type, and give semantic equations for each of them.

$$Val = Basic + Unit + Pair(Val \times Val) + In_1(Val) + In_2(Val)$$

Here, $Unit$ denotes the domain with exactly one (proper) element. We use the concise notation $Val = \cdots + Pair(Val \times Val) + \cdots$ as a shorthand for the two equations $Pair = Val \times Val$ and $Val = \cdots + Pair + \cdots$.

We can now give the new valuation equations. Again, because of the static typing, we will not check explicitly if a value is of the correct type.

For expressions, we get

$$\begin{aligned} \mathcal{E}[\![()]\!]\rho\kappa &= \kappa\, unit() \\ \mathcal{E}[\![(E_1, E_2)]\!]\rho\kappa &= \mathcal{E}[\![E_1]\!]\rho\,(\lambda v_1.\mathcal{E}[\![E_2]\!]\rho\,(\lambda v_2.\kappa\, pair(v_1, v_2))) \end{aligned}$$

Note in particular that in pure CBV, we evaluate both arguments of a tuple, and the equation above arbitrarily specifies that E_1 must be evaluated first. This is an example of the overspecification sometimes imposed by a continuation semantics. As fixing a particular evaluation order for tuple elements breaks the symmetry between the first and second component, we would prefer to leave it unspecified. This is quite hard to express properly in a precise way, so we will simply keep the equation above, but specify that SLC expressions may not depend on any particular evaluation order among tuple elements.

For continuations, we get the two new equations:

$$\begin{aligned} \mathcal{C}[\![\{\}]\!]\rho v &= \text{case } v \text{ of esac} \\ \mathcal{C}[\![\{C_1, C_2\}]\!]\rho v &= \text{case } v \text{ of } in_1(t) : \mathcal{C}[\![C_1]\!]\rho t \;[]\; in_2(t) : \mathcal{C}[\![C_2]\!]\rho t \text{ esac} \end{aligned}$$

We note that the continuation denoted by $\{\}$ can never be applied in a type-correct program, as there are no values of type $null$, so the semantic equation gives its denotation as a "case expression" with no choices.

The two equations look very different from the corresponding ones for expressions. In particular, there is no problem with evaluation order among the two continuations in the first equation. Again, this is because we are explicitly specifying a CBV strategy, where products and coproducts are both expressed in terms of values, and thus appear quite different "under the hood", just like cartesian products and disjoint unions do in **Set**. In the symmetrical world of SLC, we can still think of coproducts in terms of continuations.

The equations for functions are almost unchanged. The only difference is that the parameter binding does not happen directly, but in the equations for $\mathcal{X}$ and $\mathcal{Y}$. We will even keep the "function updating" notation for multiple values:

$$\begin{aligned} \mathcal{F}[\![X \Rightarrow E]\!]\rho v\kappa &= \mathcal{E}[\![E]\!]\,([\mathcal{X}[\![X]\!] \mapsto v]\rho)\kappa \\ \mathcal{F}[\![Y \Leftarrow C]\!]\rho v\kappa &= \mathcal{C}[\![C]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)v \end{aligned}$$

Finally, we consider the parameter binding functions. For values, they are the expectable:

$$\begin{aligned} [\mathcal{X}[\![x]\!] \mapsto v]\rho &= [x \mapsto val(v)]\rho \\ [\mathcal{X}[\![()]\!] \mapsto v]\rho &= \text{let } unit() = v \text{ in } \rho \\ [\mathcal{X}[\![(X_1, X_2)]\!] \mapsto v]\rho &= \text{let } pair(v_1, v_2) = v \text{ in } [\mathcal{X}[\![X_1]\!] \mapsto v_1, \mathcal{X}[\![X_2]\!] \mapsto v_2]\rho \end{aligned}$$

In particular, if the parameter is a single variable, the binding happens precisely as before.

For continuation parameters, we obtain:

$$\begin{aligned}
[\mathcal{Y}[\![y]\!] \mapsto \kappa]\rho &= [y \mapsto cnt(\kappa)]\rho \\
[\mathcal{Y}[\![\{\}]\!] \mapsto \kappa]\rho &= \rho \\
[\mathcal{Y}[\![\{Y_1, Y_2\}]\!] \mapsto \kappa]\rho &= [\mathcal{Y}[\![Y_1]\!] \mapsto (\lambda v.\kappa\ in_1(v)),\ \mathcal{Y}[\![Y_2]\!] \mapsto (\lambda v.\kappa\ in_2(v))]\rho
\end{aligned}$$

Again, for structured continuations, this looks very different from value bindings, because of the CBV strategy.

2.3 Summary of the first order language

Before proceeding to higher-order functions, let us consider briefly the expressive power of the subset of SLC we have seen already.

2.3.1 Non-local values and continuations

In the language presented so far, functions must be syntactical abstractions, and cannot be passed around. However, the body of an abstraction may refer to values and continuations abstracted at an outer lexical scope, *i.e.*, the language has a block structure. This is more important theoretically than it may seem, as we will show in the next subsection. In the following paragraphs, we will consider the more general implications of non-locality.

Non-local values are usually considered a clear and efficient way of avoiding explicit function parameters that would otherwise just be carried around throughout a computation, but only used in a few places. A good example would be the environment of a simple interpreter: it will only be used for binding and looking up of variables, but must be passively transmitted through all the language structure which does not access it directly. If the language of the interpreter is block-structured, the environment would therefore be a natural candidate for globalizing.

On the other hand, non-local exits are commonly regarded as quick-and-dirty tricks, acceptable only in very special circumstances, *e.g.*, for handling fatal errors. Even though many programming problems are naturally expressed in terms of continuations, there is a strong preference for value-based solutions.

Consider, for example, a (recursive) function that searches a tree for a given value, returning either 'found' or 'not found'. The "clean" implementation will pass a 'found' answer back through all the nested recursive calls, which is clearly redundant, as it will not be modified on the way. If the searched tree is non-homogeneous, (*e.g.*, a syntax tree), and different parts must be searched with different functions, clarity suffers as well. In such a case, a non-local exit with the answer 'found' is in fact more natural, just as a non-local value might be used to specify the element to be searched for. We will show how to write such a function in section 4.3.1

Unfortunately, non-local exits have strong connotations of 'error' in most languages, and are often performed using a powerful but unstructured construct like 'goto' in imperative languages, or 'call-with-current-continuation' in Scheme. In others, notably ML and Ada, 'exceptions' have an essentially dynamic scope, as control returns not to a statically-determined point, but to the last encountered 'handler'. While a practical language where a non-local continuation would be considered just as natural, simple, and declarative as a non-local value probably still remains to be seen, the SLC takes large step towards this goal, by providing a symmetrical, structured, and statically-scoped framework for such a facility.

2.3.2 Examples

It might seem that the variable-based notation is just syntactic sugar for the basic categorical constructs (product, coproduct, *etc.*), but this is not true: by allowing nested function definitions, we have made it possible to define a number of morphisms which could not otherwise be expressed.

Let us first note that having non-local values permits us to define (in a CBV semantics) the *isomorphism*

$$\delta : A \times (B + C) \to A \times B + A \times C$$

for any types A, B, and C. In traditional notation, it would be expressed as *e.g.*:

$$\lambda(a,bc).\text{case } bc \text{ of } in_1(b) : in_1(a,b) \parallel in_2(c) : in_2(a,c) \text{ esac}$$

which corresponds to the following function in the SLC's "arrow notation":

$$(a,bc) \Rightarrow (\{ab,ac\} \Leftarrow \{ab\downarrow(b \Rightarrow (a,b)), ac\downarrow(c \Rightarrow (a,c))\})\uparrow bc$$

Such a morphism cannot be defined using only the first-order categorical constructs introduced so far (composition, products, *etc.*). Interestingly, however, the inverse function can be expressed as the following categorical term:

$$\delta^{-1} \equiv [id \times \iota_1, id \times \iota_2] : A \times B + A \times C \to A \times (B + C)$$

which can of course be expanded into pure SLC.

By reversing the arrows, we can also define a morphism

$$\bar{\delta} : (A + B) \times (A + C) \to A + (B \times C)$$

as the SLC function

$$\{a,bc\} \Leftarrow bc\downarrow((ab,ac) \Rightarrow ((b \Leftarrow \{a,b\})\uparrow ab, (c \Leftarrow \{a,c\})\uparrow ac))$$

$\bar{\delta}$ is actually an isomorphism when a different (CBN-like) evaluation strategy is used. This is a consequence of a more general, and even more surprising property of the SLC, presented in the next section.

2.4 Higher-order constructs

We removed higher-order functions from the SLC because of the fundamentally asymmetric identification of functions and values in the λ-calculus. In this section, we will see how to restore them in a symmetrical way.

2.4.1 Functions as values and continuations

The typed λ-calculus does not distinguish conceptually between functions and values; everything is a value, and some values (those of a functional type) can be applied to others. The SLC accommodates this model, but draws attention to the implicit "change of viewpoint" that permits a function to be treated as a value and vice versa.

Symmetry dictates that there should be a similar correspondence between functions and continuations as well, and not surprisingly this is indeed the case. Yet while this correspondence is technically simple to add in a continuation semantics, its manifestation in the categorical interpretation of the SLC gives rise to a number of highly unusual and at first sight very counterintuitive

properties. It is therefore important to remember that every result obtained can be verified in a simple and unambiguous way by reference to the semantic equations of section 2.7.

In a typed λ-calculus, the foundation of higher-order functions is provided by the fact that for all objects (types) A and B there exists an *exponential* or *function space* object, usually written B^A or $[A \to B]$, with a correspondence between values of type $[A \to B]$ and functions $A \to B$. In the SLC, a dual relationship holds as well, namely as a *coexponential object* A_B or $[B \leftarrow A]$, and a correspondence between functions $A \to B$ and *continuations* of type $[B \leftarrow A]$.

Computationally, a value of type $[A \to B]$ is known as a functional closure. In a continuation semantics, it denotes a function that will map an argument value and a result continuation (but *not* an environment) to the final result of the program. We can therefore equivalently view a function $f : A \to B$ as a continuation accepting a pair consisting of an A-typed value and a B-accepting continuation. Such a pair will be called the *context* of a function application, and its type written as $[B \leftarrow A]$.

Perhaps the easiest way to think about a context-typed continuation is to view it as a Smalltalk object, (or, even better an *actor* [Hewitt 79]). Such an object accepts requests ("messages") to compute a function. It does not know, however, where the request came from. Thus, if we want an answer, we must also supply a "return address", *i.e.*, a continuation for it.

In the SLC, explicit conversions between the three syntactic classes are not necessary in the concrete syntax, but are present as implied operators in the abstract one. There, we will use the notation:

$$\begin{array}{rcl} \ulcorner F \urcorner & : & \text{Function } F : S \to T \text{ as } [S \to T]\text{-typed value} \\ \overline{E} & : & \text{Value } E : [S \to T] \text{ as function } S \to T \\ \llcorner F \lrcorner & : & \text{Function } F : S \to T \text{ as } [T \leftarrow S]\text{-typed continuation} \\ \underline{C} & : & \text{Continuation } C : [T \leftarrow S] \text{ as function } S \to T \end{array}$$

In a sense, these conversions are are very much like the *coercions* found in many languages, permitting *e.g.*, an integer value to be used in a context where a real-typed one is required. However, in the SLC they are not based on coercing values but entire syntactic classes, *i.e.*, permitting a ($[A \to B]$-typed) expression to be used where a function (of type $A \to B$) is expected. We will therefore rely on the parsing algorithm to insert coercions where needed, but not anywhere else, just like we would not expect a real-typed value in a real-typed context to be coerced into an integer and back, losing precision in the process. Similarly, unwarranted coercions can change the meaning of an SLC term, as we shall see in a moment.

When writing SLC terms without parentheses, we need a set of disambiguating rules like the ones for the λ-calculus. We will extend the usual syntactic conventions, letting applications (of both kinds) bind stronger than abstractions, *e.g.*, $x \Rightarrow E_1 \uparrow E_2$ should be parsed as $x \Rightarrow (E_1 \uparrow E_2)$. Abstractions of both kinds associate to the right, *e.g.*, $a \Rightarrow b \Leftarrow c \Rightarrow b$ means $a \Rightarrow (b \Leftarrow \{c \Rightarrow b\})$. Finally, value applications associate to the left, as usual ($F \uparrow E_1 \uparrow E_2 = (F \uparrow E_1) \uparrow E_2$, and continuation applications associate to the right for symmetry: $C_1 \downarrow C_2 \downarrow F = C_1 \downarrow \{C_2 \downarrow F\}$.

Having the coercions permits us to directly transliterate any typed lambda-term into an SLC function. For example, the combinator $S = \lambda fgx.fx(gx)$ would be written as simply

$$f \Rightarrow g \Rightarrow x \Rightarrow f \uparrow x \uparrow (g \uparrow x) : [A \to [B \to C]] \to [[A \to B] \to [A \to C]]$$

With the coercions made explicit, it would become the rather less readable

$$f \Rightarrow \ulcorner g \Rightarrow \ulcorner x \Rightarrow \overline{(\overline{f} \uparrow x)} \uparrow (\overline{g} \uparrow x) \urcorner\urcorner$$

However, we can also write entirely new terms, with no simple relation to common concepts, *e.g.*

$$f \Rightarrow y \Leftarrow f \uparrow y$$

which it may be more helpful to see as

$$f \Rightarrow \ulcorner y \Leftarrow \llcorner \overline{\overline{f} \uparrow \ulcorner \underline{y} \urcorner} \lrcorner \urcorner$$

because it shows all the implicit coercions. In particular, we note that treating the continuation y as a value by passing it to a function with $\uparrow$ implies first converting it into a function and then into a value. Thus, we have an correspondence between closure-typed expressions and context-typed continuations, but not between expressions and continuations in general.

Ideally, the coercions would be invertible, but this is not quite the case, just as in the integer/real example. For functional values in a CBV setting the problem is well-known: Evaluation of the function part of an application may fail to terminate, so that the number of closure-typed expressions is strictly larger than the number of functions of the corresponding type. This means that while we can always convert a function into a value and back, preserving meaning, we cannot in general do the reverse.

For a similar reason, while converting a function into a context-typed continuation and back is always an identity operation, doing it in the reverse order does not necessarily preserve the meaning (although it does in CBV). Again, we will treat this problem more rigorously in the next chapter.

It is important to note that because we have insisted on separating functions and values, the η-rule is still generally valid, unlike, *e.g.*, Scheme, where adding or removing an η-redex can change the meaning of a program (*e.g.*, when defining an applicative-order fixpoint combinator). The key point is that the SLC η-rule applies to functions, not expressions. If we insert the coercions explicitly, we see that the reduction is actually:

$$x \Rightarrow E \uparrow x \;\equiv\; \ulcorner x \Rightarrow \overline{E} \uparrow x \urcorner \;\xrightarrow{\eta}\; \ulcorner \overline{E} \urcorner \;\neq\; E$$

i.e., the E stays suspended. But, as there are no explicit coercions in the concrete syntax, $\ulcorner \overline{E} \urcorner$ must be written using an η-redex to specify the intended meaning.

2.4.2 Currying and cocurrying

In functional programming languages, there is a well-known correspondence between functions from a product type, $f : A \times B \to C$, and their curried forms $f^* : A \to [B \to C]$. Naturally, we can define f^* in terms of f in the higher-order SLC as follows:

$$f^* \equiv a \Rightarrow b \Rightarrow f \uparrow (a, b)$$

Here, we treat the function $b \Rightarrow f \uparrow (a, b)$ as a value. This is the essence of building a functional closure: the value of type $[B \to C]$ must account for the value of the free variable a as well as the function body itself.

Conversely, given a value of type $[B \to C]$, we can apply it to a value of type B using the morphism ap:

$$ap \;\equiv\; (g, b) \Rightarrow g \uparrow b : [B \to C] \times B \to C$$

Here, we treat the value parameter g as a function, by applying it to b. With these definitions, we have the equality:

$$(a, b) \Rightarrow ap \uparrow (f^* \uparrow a, b) \;=\; ap \circ \langle f^* \circ \pi_1, \pi_2 \rangle \;=\; f$$

Note that this is a purely first-order equation, *i.e.*, with no implicit coercions. In fact, having a f^* for every f, and a single morphism ap is enough to handle functional values in a categorical framework, as we will see in the next chapter.

While the above definitions and results were hardly surprising, the symmetry of SLC permits us to define some very unfamiliar morphisms as well. The point is that the function body can also contain a free continuation variable. Again, we can dualize by just reversing the arrows in the definition. Thus, for every coproduct-returning function $f : C \to A + B$, there exists a *cocurried* version:

$$f_* \equiv a \Leftarrow b \Leftarrow \{a, b\} \downarrow f : [B \leftarrow C] \to A$$

and a fixed co-application morphism

$$pa \equiv \{g, b\} \Leftarrow b \downarrow g : C \to [B \leftarrow C] + B$$

such that

$$\{a, b\} \Leftarrow \{a \downarrow f_*, b\} \downarrow pa = [\iota_1 \circ f_*, \iota_2] \circ pa = f$$

In fact, for CBV evaluation, the morphism f_* will also be unique. This bijection between functions $C \to A + B$ and $[B \leftarrow C] \to A$ is quite striking. In particular it defies set-theoretical reasoning because the cardinalities look impossible (*e.g.*, if A is a singleton set), but then again, we are not working in **Set**, but in a very unfamiliar category where functions have access to continuations.

In fact, even without the bijection aspect, the equation defies set-theoretic intuition. The morphism $(f_* + id_B)$ does not change the injection tag, and in the case of the second inject not even the value. Nevertheless, a single morphism pa can be used for all f! This just shows that the usual element-based reasoning techniques must not be used indiscriminately in a categorical setting.

Operationally, the equation is explained by the fact that the context-typed result of pa includes the continuation in effect just before the case construct, which permits the computation to backtrack slightly when f_* is applied. So, if $f = \iota_2$, pa will first return with the first inject, but make it possible for f_* to re-enter the case expression with the correct tag. However, this is just another example of "under-the-hood" behavior, which is invisible from within the category. The proper way to think of coexponentials is still as the mirror image of exponentials, not as strange noises from the machinery below.

Not surprisingly, cocurrying will be central to the concept of functions as continuations, and we will return to the equation above in section 3.1.3.

Let us now consider a simpler practical application of cocurried functions, by analogy to partial (value) application: given a function $add : int \times int \to int$, we can define a specialized version $add3 \equiv add^* \uparrow 3 : int \to int$, which will add 3 to its argument. The dual case is analogous: let the function $elog : real \to unit + real$ be defined as returning the logarithm of its argument, if positive, and () on error. Let there also be given a continuation $fail$, accepting a unit-typed value. We can then define the function $log \equiv fail \downarrow elog_* : real \to real$ which will return the logarithm of a positive argument, but invoke the continuation $fail$ on error.

2.4.3 Denotational semantics

We add higher-order constructs to the CBV language as follows. First, we extend the abstract syntax:

$$E ::= \ldots \mid \ulcorner F \urcorner$$

$$\begin{array}{rcl} C & ::= & \ldots \mid \llcorner F \lrcorner \\ F & ::= & \ldots \mid \overline{E} \mid \underline{C} \end{array}$$

Again, it must be stressed that the coercions are inserted as part of the parsing process and do not normally appear in concrete SLC terms.

In the semantic domain, we add two more types, closures and contexts:

$$Val = \cdots + Closr(Val \to Cnt \to Ans) + Contx(Val \times Cnt)$$

The semantics of treating functions as values are familiar from the λ-calculus:

$$\begin{array}{rcl} \mathcal{E}[\![\ulcorner F \urcorner]\!]\rho\kappa & = & \kappa\ closr(\lambda vc.\mathcal{F}[\![F]\!]\rho vc) \\ \mathcal{F}[\![\overline{E}]\!]\rho v\kappa & = & \mathcal{E}[\![E]\!]\rho\ (\lambda t.\text{let } closr(f) = t \text{ in } fv\kappa) \end{array}$$

We recall that in the equation for (value) application, the argument is evaluated before the function. This becomes significant when the function can itself be another expression, rather than a syntactic abstraction, because in an expression like $E_1 \uparrow E_2$, E_2 will be evaluated first. Unlike the evaluation order for tuple components, this behavior *is* considered formally specified, and will be essential for the translation to categorical combinators in the next chapter.

The two coercions are not inverses. While $\mathcal{F}[\![\overline{\ulcorner F \urcorner}]\!] = \mathcal{F}[\![F]\!]$, we have

$$\mathcal{E}[\![\ulcorner \overline{E} \urcorner]\!] = \lambda\rho\kappa.\kappa\ closr(\lambda vc.\mathcal{E}[\![E]\!]\rho\ \{\lambda t.\text{let } closr(f) = t \text{ in } fvc\}) \neq \mathcal{E}[\![E]\!]$$

i.e., E has been suspended as if we had written an η-redex around it.

Similarly, we have the conversions between functions and continuations:

$$\begin{array}{rcl} \mathcal{C}[\![\llcorner F \lrcorner]\!]\rho v & = & \text{let } contx(a,c) = v \text{ in } \mathcal{F}[\![F]\!]\rho ac \\ \mathcal{F}[\![\underline{C}]\!]\rho v\kappa & = & \mathcal{C}[\![C]\!]\rho\ contx(v,\kappa) \end{array}$$

However, here we *do* have a complete correspondence, *i.e.*, both $\mathcal{C}[\![\llcorner \underline{C} \lrcorner]\!] = \mathcal{C}[\![C]\!]$ and $\mathcal{F}[\![\llcorner \underline{F} \lrcorner]\!] = \mathcal{F}[\![F]\!]$. Again, this is a consequence of CBV evaluation; in CBN, the situation is reversed, as we shall see in the next section.

2.5 Alternative evaluation strategies

So far, we have only considered the semantics of CBV evaluation. However, the SLC can equally well be considered in a CBN setting, and we will explore this possibility in this section. We will see how the CBV semantics can be almost mechanically inverted to obtain a CBN-like evaluation strategy, and how this change affects the language. Perhaps the most important point is that while moving to CBN evaluation, we will retain the SLC's first-class continuations. Thus, the benefits of having a symmetrical syntax are perhaps even greater here, when we can can no longer rely on our experience with CBV languages.

2.5.1 Pure CBN evaluation

Let us first consider the exact dual of CBV evaluation, which we will call *pure CBN*. In the next subsections, we will examine a slightly modified version, which is closer to the strategy found in most lazy functional languages.

In CBV, values are always completely evaluated, while continuations represent the rest of the computation. In pure CBN, it will be the yet unevaluated part of the expression which determines

the rest of the computation, while the continuations do not contain any residual code. We can think of values in CBN as inputless functions (cf. the parameterless procedures or 'thunks' in Algol 60 terminology), just as continuations in CBV were conceptually outputless functions. Thus, we have continuations as atomic entities, and values mapping continuations to answers.

Exchanging the two semantic domains Val and Cnt, replacing products by coproducts, *etc.*, we directly obtain the CBN domains:

$$\begin{aligned} Val &= Cnt \to Ans \\ Cnt &= Bcont + Zero() + Case(Cnt \times Cnt) + Pr_1(Cnt) + Pr_2(Cnt) + \\ &\quad Contx(Val \to Cnt \to Ans) + Closr(Val \times Cnt) \end{aligned}$$

The domain $Bcont$ represents a basic continuation. In principle, it could be defined arbitrarily. However, as the primitive functions of our semantic metalanguage (the λ-calculus) can only be used as value transformers, we will be forced to define it as

$$Bcont = Basic \to Ans$$

where $Basic$ is the same domain of basic values as in CBV. We will also use this when giving the denotation of constants, which can be seen as a special kind of primitive function that accepts no input.

We note that the type of the initial (top-level) continuation depends on the type of the expression we are evaluating. This is not a problem, as every SLC expression is statically typeable. For example, when evaluating a top-level expression of a coproduct type, the initial continuation would be a 'case'-tagged pair of continuations, each for handling one of the injects. When evaluating a product-typed top-level expression, we sequentially evaluate and print the first component using a pr_1-tagged continuation, and then the second. This is also the typical behavior of a lazy language, where infinite lists are printed incrementally.

We also see that the role of the two higher-order types is also exchanged: A context-typed continuation denotes a function, while a continuation accepting a closure is really a context for the application.

Let us now consider the semantic equations. The type of the environment and of all valuation functions are the same as for CBV, but with new definitions of Val and Cnt. To emphasize this, we will use the letters ν and k, instead of v and κ, for values and continuations respectively. Except for the special cases mentioned above, the new equations in figure 2.1 are the exact mirror image of the CBV ones, with the role of expressions and continuations interchanged.

Perhaps the most surprising of these equations is the denotation of a coproduct-accepting continuation, $\{C_1, C_2\}$. Just as for CBV we do not want to force a particular "consideration order" for the two alternatives, so we restrict the operation to continuations which can be interchanged. For example, we do *not* allow functions like $y \Leftarrow \{y\downarrow(x \Rightarrow v_1), y\downarrow(x \Rightarrow v_2)\}$ *i.e.*, $[x \Rightarrow v_1, x \Rightarrow v_2]$ in pure CBN, because both branches of the 'case' discard the value, just as we do not permit $x \Rightarrow ((y \Leftarrow k_1)\uparrow x, (y \Leftarrow k_2)\uparrow x)$, *i.e.*, $\langle y \Leftarrow k_1, y \Leftarrow k_2\rangle$ (where both components of the tuple escape), in pure CBV. For both cases, the result in SLC should be considered undefined. Note, however that we *do* permit the case where only one of the elements "misbehaves", *i.e.*, both $\langle id, y \Leftarrow k_2\rangle$ and $[x \Rightarrow v_1, id]$ are legal.

While we would usually accept the CBV restriction on products as very reasonable, the CBN restriction on coproducts looks much too severe. This is because the pure CBN coproduct is quite different from the one found in typical lazy programming languages. For example, we have an isomorphism $A+1 \xrightarrow{\approx} 1$, just like $A \times 0$ is isomorphic to 0 in CBV. This does not mean, however, that

$$
\begin{aligned}
Val &= Cnt \to Ans \\
Cnt &= Bcont + Zero() + Case(Cnt \times Cnt) + Pr_1(Cnt) + Pr_2(Cnt) + \\
&\quad Contx(Val \to Cnt \to Ans) + Closr(Val \times Cnt)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} : E \to Env \to Cnt &\to Ans \\
\mathcal{E}[\![cst]\!]\rho k &= \text{let } bcont(b) = k \text{ in } b\ cst \\
\mathcal{E}[\![x]\!]\rho k &= \text{let } val(\nu) = \rho\ x \text{ in } \nu\ k \\
\mathcal{E}[\![(E_1, E_2)]\!]\rho k &= \text{case } k \text{ of } pr_1(t) : \mathcal{E}[\![E_1]\!]\rho t \parallel pr_2(t) : \mathcal{E}[\![E_2]\!]\rho t \text{ esac} \\
\mathcal{E}[\![()]\!]\rho k &= \text{case } k \text{ of esac} \\
\mathcal{E}[\![F \uparrow E]\!]\rho k &= \mathcal{F}[\![F]\!]\rho(\lambda c.\mathcal{E}[\![E]\!]\rho c)k \\
\mathcal{E}[\![\ulcorner F \urcorner]\!]\rho k &= \text{let } closr(a, c) = k \text{ in } \mathcal{F}[\![F]\!]\rho ac
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C} : C \to Env \to Val &\to Ans \\
\mathcal{C}[\![y]\!]\rho\nu &= \text{let } cnt(\nu) = \rho\ x \text{ in } \nu\ k \\
\mathcal{C}[\![\{C_1, C_2\}]\!]\rho\nu &= \mathcal{C}[\![C_1]\!]\rho\ (\lambda c_1.\mathcal{C}[\![C_2]\!]\rho\ (\lambda c_2.\nu\ case(k_1, k_2))) \\
\mathcal{C}[\![\{\}]\!]\rho\nu &= \nu\ zero() \\
\mathcal{C}[\![C \downarrow F]\!]\rho\nu &= \mathcal{C}[\![C]\!]\rho\ (\lambda c.\mathcal{F}[\![F]\!]\rho\nu c) \\
\mathcal{C}[\![\llcorner F \lrcorner]\!]\rho\nu &= \nu\ contx(\lambda ac.\mathcal{F}[\![F]\!]\rho ac)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F} : F \to Env \to Val \to Cnt &\to Ans \\
\mathcal{F}[\![p]\!]\rho\nu k &= \text{let } bcont(c) = k \text{ in } \nu\ bcont(\lambda a.c\ (p\ a)) \\
\mathcal{F}[\![X \Rightarrow E]\!]\rho\nu k &= \mathcal{E}[\![E]\!]\ ([\mathcal{X}[\![X]\!] \mapsto \nu]\rho)k \\
\mathcal{F}[\![Y \Leftarrow C]\!]\rho\nu k &= \mathcal{C}[\![C]\!]\ ([\mathcal{Y}[\![Y]\!] \mapsto k]\rho)\nu \\
\mathcal{F}[\![\overline{E}]\!]\rho\nu k &= \mathcal{E}[\![E]\!]\rho\ closr(\nu, k) \\
\mathcal{F}[\![\overline{C}]\!]\rho\nu k &= \mathcal{C}[\![C]\!]\rho\ (\lambda t.\text{let } contx(f) = t \text{ in } f\nu k)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{X} : X \to Val \to Env &\to Env \\
[\mathcal{X}[\![x]\!] \mapsto \nu] &= [x \mapsto val(\nu)]\rho \\
[\mathcal{X}[\![()]\!] \mapsto \nu] &= \rho \\
[\mathcal{X}[\![(X_1, X_2)]\!] \mapsto \nu] &= [\mathcal{X}[\![X_1]\!] \mapsto (\lambda c.\nu\ pr_1(c)),\ \mathcal{X}[\![X_2]\!] \mapsto (\lambda c.\nu\ pr_2(c))]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{Y} : Y \to Cnt \to Env &\to Env \\
[\mathcal{Y}[\![y]\!] \mapsto k]\rho &= [y \mapsto cnt(\kappa)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto k]\rho &= [] \\
[\mathcal{Y}[\![\{Y_1, Y_2\}]\!] \mapsto k]\rho &= \text{let } case(k_1, k_2) = k \text{ in } [\mathcal{Y}[\![Y_1]\!] \mapsto k_1, \mathcal{Y}[\![Y_2]\!] \mapsto k_2]
\end{aligned}
$$

Figure 2.1: A pure CBN semantics for the SLC

the pure CBN coproduct is useless; as we shall see in the next chapter, it has some very important properties which will be crucial for the categorical definition of the SLC. And, fortunately, we can define the more familiar coproduct (with no strange restrictions) in terms of it, which is the subject of the next two subsections.

We note that in the CBN semantics, the β conversion rule of the SLC holds in full generality, while the $\bar{\beta}$ must be restricted. Also, both the existence and uniqueness properties of products hold unconditionally. The price is, as we have seen, that coproducts behave very strangely. We get similar problems with the initial objects and the coexponentials, in that they all take over the weaknesses of their value-based counterparts in CBV.

Let us conclude by noting the semantics of identity and function composition in CBN. For both definitions, we obtain:

$$\mathcal{F}[\![id]\!] = \mathcal{F}[\![x \Rightarrow x]\!] = \mathcal{F}[\![y \Leftarrow y]\!] = \lambda\rho\nu k.\nu\ k$$

$$\mathcal{F}[\![F \circ G]\!] = \mathcal{F}[\![x \Rightarrow F \uparrow (G \uparrow x)]\!] = \mathcal{F}[\![y \Leftarrow \{y \downarrow F\} \downarrow G]\!] = \lambda\rho\nu k.\mathcal{F}[\![F]\!]\rho(\lambda c.\mathcal{F}[\![G]\!]\rho\nu c)k$$

Again, we can immediately verify that identity is neutral and composition is associative. The equation for composition is particularly illuminating: the evaluation order is reversed from CBV. In particular, if F does not use its argument, it will be completely discarded. We can summarize this in the following two equations:

$$\begin{array}{rcl} F \circ (y \Leftarrow k) & = & (y \Leftarrow k) \text{ in CBV} \\ (x \Rightarrow v) \circ G & = & (x \Rightarrow v) \text{ in CBN} \end{array}$$

2.5.2 CBV with lazy products

As we have seen, pure CBN does not have general coproducts. We can, however, define an alternative coproduct-like operator which guarantees existence but not uniqueness of the mediating morphism in all cases. But as we will see, the price is quite high. Not surprisingly, a dual concept exists in CBV, and we will consider this first, as the intuition should be much stronger here. We will give a rather informal presentation here; a more rigorous treatment will be given in section 3.6.

The problem with pure CBV products was that the equation

$$\pi_1 \uparrow (E_1, E_2) = E_1$$

does not hold if evaluation of E_2 escapes, fails to terminate *etc.* We might therefore want to introduce a CBN-like tuple into CBV, whose elements would not be evaluated until their values were actually requested by a projection. Such a *lazy product* [Friedman & Wise 76] facility is provided by a number of otherwise eager programming languages in some form, *e.g.*, streams in Scheme, and "lazy cons" in Hope.

Building on the concept of functions as values and the suspension effect, we can introduce such products as an extension, without changing the core semantics of the language. We will do it first by introducing the new product constructs as syntactic abbreviations (macros) to be expanded by the parsing process.

First, however, we must introduce the suspension primitives. In Scheme, they are known as *delay* and *force*. We will adopt these names, but not the memoization effect, since this introduces the concept of an updateable store in the description. We will return to this problem in section 2.5.4.

Let us recall that for every expression $E : T$, we have a function $E' : unit \to T$, defined as $E' \equiv () \Rightarrow E$. Furthermore, such a function is a genuine value, which can be passed around and

used as argument even in restricted β-reduction. We will call a functional value of this kind a *promise* or an *explicit thunk*). The correspondence motivates the definition of the two operators:

$$\text{delay:}\quad ?E \equiv () \Rightarrow E$$

$$\text{force:}\quad E'! \equiv E' \uparrow ()$$

Strictly speaking, we only need to define 'delay' as a special operator, as 'force' could instead be defined as just a function application:

$$E'! \equiv ev \uparrow E',\ \text{where } ev \equiv e' \Rightarrow e' \uparrow ().$$

Using these, we can define a lazy product in terms of the original as follows:

$$\begin{array}{rcl} (\!|E_1, E_2|\!) & \equiv & (?E_1, ?E_2) \\ (\!|a, b|\!) \Rightarrow E & \equiv & (a, b) \Rightarrow E[a!/a, b!/b] \end{array}$$

Here, the second equation specifies that all free occurrences of a and b in E should be replaced with $a!$ and $b!$ respectively. In CBN, this lazy product is in fact isomorphic to the normal one, because the coercions between functions and values are each others inverses.

Having defined the lazy products in terms of the existing ones, we can now integrate them directly into the semantics. We could construct the type of lazy products out of existing semantic domains (closures and strict products), but as the value passed to a suspension carries no information, we can introduce a special "suspended value" type, and define a lazy pair as follows:

$$\begin{array}{rcl} SVal & = & Cnt \to Ans \\ Val & = & \cdots + LPair(SVal \times SVal) \end{array}$$

The denotation of a lazy tuple becomes:

$$\mathcal{E}[\![(\!|E_1, E_2|\!)]\!]\rho\kappa = \kappa\ lpair(\lambda c.\mathcal{E}[\![E_1]\!]\rho c, \lambda c.\mathcal{E}[\![E_2]\!]\rho c)$$

Projections are slightly more complicated. If we want to keep the destructuring syntax on the left hand side, we will have to deal with two syntactic classes of value identifiers: those introduced by "normal" abstractions and denoting already evaluated values and those representing components of a tuple. If we distinguish the latter in the abstract syntax by adding primes to their names, we get an additional equation for variable lookup, as well as a modification of the environment:

$$Env = Ide \to Val + SVal + Cnt$$

$$\mathcal{E}[\![x']\!]\rho\kappa = \text{let } sval(s) = \rho\ x' \text{ in } s\ \kappa$$

In particular, the meaning of a projection becomes

$$\mathcal{F}[\![(\!|a', b'|\!) \Rightarrow a']\!]\rho v\kappa = \text{let } lpair(s_1, s_2) = v \text{ in } s_1\ \kappa$$

It might be argued that the lazy products are a better foundation for the language than the strict ones. However, in the variable-free presentation of SLC in the next chapter, we will need the old products to represent the strict (value) environment of CBV, and we will not be able to define them in terms of the lazy ones.

2.5.3 CBN with eager coproducts

Let us now return to the problem of adding traditional coproducts to pure CBN. In CBV, a lazy pair introduced a delimited piece of CBN-like evaluation. Delimited, because we switch back to CBV when control reaches a projection. Similarly, for the CBN 'case' construct, we want to switch to CBV evaluation of the argument until the injection tag is known, but not further.

Guided by experience from the value-based operations, we introduce two operators:

$$\text{codelay: } !C \equiv \{\} \Leftarrow C$$

$$\text{coforce: } C'? \equiv \{\} \downarrow C'$$

which we will use to define *eager coproducts* in CBN:

$$\begin{array}{rcl} \{C_1, C_2\} & \equiv & \{!C_1, !C_2\} \\ \{a, b\} \Leftarrow C & \equiv & \{a, b\} \Leftarrow C[a?/a, b?/b] \end{array}$$

Note that the syntax is reversed from the CBV case: ! is now a prefix operator, while ? is postfix. The meanings are also opposite: 'codelay' in a sense initiates eager evaluation, while 'coforce' stops as soon as the injection tag is known. Again, we can integrate this definition into the pure CBN semantics, by defining:

$$\begin{array}{rcl} SCnt & = & Val \to Ans \\ Cnt & = & \cdots + ECase(SCnt \times SCnt) \end{array}$$

We recognize in the "suspended continuation" the traditional definition as a function mapping values to answers. However, the values here are themselves suspensions.

The semantic equation for an "eager case" becomes:

$$\mathcal{C}[\![\{C_1, C_2\}]\!]\rho\nu \;=\; \nu\ ecase(\lambda t.\mathcal{C}[\![C_1]\!]\rho t, \lambda t.\mathcal{C}[\![C_2]\!]\rho t)$$

We do not examine the two continuations C_1 and C_2 further, but instead start evaluating the value ν, leaving the choice between C_1 and C_2 to it.

For the special continuation variables introduced by eager case abstractions, we get analogously to the value case:

$$\mathcal{C}[\![y']\!]\rho\nu \;=\; \text{let } scnt(s) = \rho\ y' \text{ in } s\ \nu$$

with the denotation of an injection becoming:

$$\mathcal{F}[\![\{a', b'\} \Leftarrow a']\!]\rho\nu k \;=\; \text{let } ecase(s_1, s_2) = k \text{ in } s_1\ \nu$$

Here, we finally select one of the continuations and revert to CBN evaluation.

It should be noted that this new construct is still not a true coproduct. The missing uniqueness property manifests itself as "loss of laziness" in seemingly innocent transformations involving coproducts or coproduct-like types like booleans. As usual, this is true for all lazy languages, not just the SLC. For example, the equation

```
((if ... then 1 else 2), 3) = if ... then (1,3) else (2,3)
```

which holds for standard (eager) ML, does not in general hold for lazy ML: if the '...' fails to terminate or escapes, the result of applying the second projection to the left-hand side will not be equal to the second projection of the right-hand side, thus outlawing a potentially useful optimization. Lazy languages attempt to limit such problems by not including escapes as explicit constructs, but they cannot prevent runtime errors (s.a. division by zero) or non-termination.

2.5.4 A comparison of CBV and CBN

As we have tried to demonstrate, CBV and CBN have essentially dual properties. Nevertheless, they are not usually seen as complementary strategies. Rather, CBN is considered the ideal, referentially transparent and maximally terminating strategy, which we must sometimes approximate by CBV for efficiency reasons. The special merits of CBV, *e.g.*, its proper coproducts are seldom mentioned as advantages, and its semantic fitness (because of the unrestricted $\bar{\beta}$-rule) for integrating user-accessible continuations s.a. `call/cc` in Scheme is usually scorned upon by theoreticians.

This asymmetry arises mainly because we are usually comparing a pure CBV strategy to CBN with eager coproducts. Furthermore, as the λ-calculus does not have first-class continuations as a declarative concept, many otherwise dual properties appear unrelated, and counterexamples to claims of CBN's unconditional superiority cannot be expressed directly, but manifest themselves as semantic pitfalls, as demonstrated by the 'if' example above. On the other hand, even proponents of CBN admit that it is usually somewhat slower than CBV, because it risks evaluating the same thing twice.

Let us therefore note that *pure* CBN is not inherently less efficient than CBV. Any re-evaluation that may occur is caused by our introduction of eager coproducts or primitive functions (δ-rules) operating on coproduct-like values, s.a. arithmetic. We get the same problem when we add lazy products to CBV. In a sense, it is not a particular strategy as such which causes the inefficiencies, but the implicit *changes* from one to the other.

In both cases, for actual implementations, one solution is memoization, to prevent repeated traversals of the same code. As this introduces the concept of an updateable store, it falls out of the scope of this thesis. However, there are no fundamental problems in adding a store or even general side-effects s.a. I/O to the SLC. In fact, the axiomatic description of the next chapter seems well-suited for handling functions with side-effects. Another technique applies abstract interpretation (notably strictness analysis), to determine when one strategy can in fact be replaced by the other without changing the meaning.

As the language we have discussed so far does not have any recursive facilities, all computations terminate (the Y combinator, *etc.* cannot be expressed with existing primitives in a type-correct way). However, for the possibly non-terminating computations introduced by the next chapter, CBN does not necessarily terminate more often then CBV. Let us namely consider a simple scenario where CBN is superior to CBV:

$$(x \Rightarrow v)\uparrow \bot$$

Here, $\bot$ represents an expression whose evaluation does not terminate, while v is constant (or an expression not containing x). In CBN, we simply discard the argument, but CBV will loop forever trying to evaluate something that will never be used anyway. Hopefully, the dual example will be obvious by now:

$$\top \downarrow (y \Leftarrow k)$$

Here, $\top$ represents a continuation that does not terminate, while y does not occur in k. In CBV, we will proceed to k, avoiding the bottomless pit, but CBN evaluation will needlessly attempt to "reduce" it to an atomic form, before it can continue. In ML terms, for the expression

`let _=raise k in` ...*loop*...

standard ML will avoid the path that would prevent a lazy ML (even if it had exceptions or a similar facility) from terminating.

2.6 Recursion and iteration

In this section, we will consider adding general recursion (fixpoints) to the SLC. Once more, we will see a duality between values and continuations, and between recursion for CBN and iteration for CBV. In the area of recursively-defined types, the presentation will be somewhat informal, as it is not yet clear what the best way to integrate them in the SLC would be.

2.6.1 Recursive functions, values and continuations

As the SLC is typed, we need a special operator to write recursive definitions. With it, we introduce the possibility of nontermination. However, this presents surprisingly few new problems, as nontermination in CBV can be viewed locally as a special case of escaping: from the point of call it makes no difference whether the called function is looping forever or has jumped somewhere else. A dual view holds for CBN, and the axioms presented in the next chapter will automatically cope with it. We will see that the concept of a recursive function generalizes to both recursive values and recursive continuations.

As usual, we will consider value-based concept first, but in CBN this time. We can introduce recursive values into the language be means of a single operator $\overline{\mathbf{rec}}$; intuitively, we want the expression $\overline{\mathbf{rec}}\ x = E$ to mean 'let $x = \bot$ in let $x = E$ in ... in E'. Under CBN, such a definition makes sense, because evaluation proceeds inside-out from the last E; if an x is encountered, the preceding E is evaluated and so on, and the $\bot$ will never be reached.

We can therefore form infinite, structured values which will only be evaluated on need. For example, we can obtain an infinite list:

$$\overline{\mathbf{rec}}\ x = (1, x) \ \equiv \ (1, (1, (1, ...)))$$

Its type is a solution to the recursive domain equation $D = int \times D$. We can also use the $\overline{\mathbf{rec}}$ operator to define recursive values from the function space, *i.e.*, recursive functions. Again, the laziness of CBN prevents excessive unfolding of the definition.

In CBV, the intuitive explanation of recursion in terms of an infinite let does not quite hold. CBV evaluation proceeds outside in, which would require evaluating $\bot$. Even if this could somehow be avoided, there would still be an infinite number of expressions to be evaluated before the last E, which will be the result. Therefore, general recursive values such as infinite lists are not possible in pure CBV. We can still have recursive functions, of course, but then they become a special case, imposing the artificial restriction on rec-expressions, that the argument must be a lambda-abstraction.

Using the insights of the previous sections, we might therefore try to express a recursive function in CBV through continuations, and in fact this is possible and gives a very natural model. We will write recursive continuations as $\underline{\mathbf{rec}}\ y = C$, and treat a recursive function as a recursive, context-typed continuation. The implicit conversions between such continuations and functions permit traditional recursive definitions, *e.g.*

$$\underline{\mathbf{rec}}\ f = n \Rightarrow \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times f\!\uparrow\!(n-1)$$

However, recursive functions are only a special case of recursive continuations. In fact, we can write useful recursive continuations that are not functions, in the same way as we use recursive values such as lazy lists under CBN. Such continuations represent unbounded (while-type) iteration.

For example, let us consider again the CBN domain of infinite lists of integers, $D = int \times D$. We can write a function which will map an integer to an infinite list of that integer like this:

$$\overline{\mathbf{rec}}\ f = n \Rightarrow (n, f\!\uparrow\! n) : int \to D$$

Categorically, this represents the morphism $f = \langle id, f \rangle$.

However, we can also express the result directly as a recursive value, so we may write the same function as

$$n \Rightarrow \overline{\mathbf{rec}}\, d = (n, d) : int \to D$$

Now, let us consider the dual case. Our domain becomes the solution to $\bar{D} = int + \bar{D}$, *i.e.*, an integer embedded in a (finite) series of injection tags. (If we had used *unit* instead of *int* above, $\bar{D}$ would be isomorphic to the natural numbers). We can write a function $g : \bar{D} \to int$, which will extract the integer from a $\bar{D}$-typed value:

$$\underline{\mathbf{rec}}\, g = n \Leftarrow \{n, n \downarrow g\} : \bar{D} \to int$$

This uses the pure SLC syntax for writing a "case expression". In categorical terms, we have $g = [id, g]$. However, we can also express g directly, using a recursive continuation:

$$n \Leftarrow \underline{\mathbf{rec}}\, d = \{n, d\} : \bar{D} \to int$$

The continuation d will accept an element of the domain $\bar{D}$, and pass it either to the result continuation n or to itself, stripping off the injection tag first. This is essentially an iterative computation, and we shall treat it more formally as such in the next chapter.

Mutual recursion in CBV also works in a dual way to CBN. While in CBN we can define a pair of closure-typed values together and project out the one we want, *e.g.*,

$$((f, g) \Rightarrow f) \uparrow (\overline{\mathbf{rec}}\, (f, g) = (\ldots, \ldots))$$

in CBV, we inject a computational context into the coproduct expected by the recursive continuation:

$$\{\underline{\mathbf{rec}}\, \{f, g\} = \{\ldots, \ldots\}\} \downarrow (\{f, g\} \Leftarrow f)$$

In the object-oriented view presented in subsection 2.4.1, we can think of recursion as sending messages to oneself. To distinguish between requests to compute f and g, we also supply an injection tag (the "message constructor").

Let us finish by remarking that when we are defining recursive functions (as a special case of recursive values or continuations), it is often irrelevant whether they are seen as CBN values or CBV continuations. So, when the evaluation order does not matter (as in the definition of factorial above), we will simply write $\mathbf{rec}\ f = F$. This should be interpreted as $\overline{\mathbf{rec}}$ in CBN and $\underline{\mathbf{rec}}$ in CBV.

2.6.2 Fixpoints and nontermination

Let us note that in CBN, the principal type (as defined in section 2.7.1) of the fixpoint combinator is the familiar *normal-order* one:

$$fix_n \;\equiv\; f \Rightarrow \overline{\mathbf{rec}}\, a = f \uparrow a : [A \to A] \to A$$

This means that having fix_n together with higher-order functions is equivalent to the $\underline{\mathbf{rec}}$ operator. However, if we consider the same definition in CBV, we obtain the less general type of the *applicative-order* fixpoint combinator:

$$fix_v \;\equiv\; f \Rightarrow \underline{\mathbf{rec}}\, a = f \uparrow a : [[A \to B] \to [A \to B]] \to [A \to B]$$

This is because the implied coercions force a to be of a functional type. Thus, the CBN 'fix' can find fixpoints of functions s.a. $x \Rightarrow (1, x)$, but the CBV one can not, because of *type* incompatibility. It can, however, be applied to functions s.a. $f \Rightarrow n \Rightarrow ...$, or even $f \Rightarrow f$, where f is assigned a functional type.

On the other hand, we can define a CBV-type "fixpoint" like this:

$$xif \;\equiv\; f \Leftarrow \underline{\mathbf{rec}}\; a = a \downarrow f : A \to [A \leftarrow A]$$

and in section 3.7 we will show that this morphism is equivalent in power to **rec**, and in particular can be used to define recursive functions.

We further note that having recursion in the language means that every type A has a *global* value (element) [Poigné 83], defined as follows:

$$\bot_A \equiv \overline{\mathbf{rec}}\; a = a : A$$

We can use such values to construct a type-correct right inverse of a projection (in categorical terms, projections are split epimorphisms). We define

$$\pi_1^{-1r} \equiv a \Rightarrow (a, \bot_B) : A \to A \times B \qquad\qquad \pi_2^{-1r} \equiv b \Rightarrow (\bot_A, b) : B \to A \times B$$

and obtain the equalities

$$\pi_1 \circ \pi_1^{-1r} = id_A \qquad\qquad \pi_2 \circ \pi_2^{-1r} = id_B$$

Similarly, in CBV, we have a global *continuation* for every type:

$$\top_A \;\equiv\; \underline{\mathbf{rec}}\; a = a : A$$

Injections are split monomorphisms, and we can define their left inverses:

$$\iota_1^{-1l} \;\equiv\; a \Leftarrow \{a, \top_B\} : A + B \to A \qquad\qquad \iota_2^{-1l} \;\equiv\; b \Leftarrow \{\top_A, b\} : A + B \to B$$

with the properties

$$\iota_1^{-1l} \circ \iota_1 = id_A \qquad\qquad \iota_2^{-1l} \circ \iota_2 = id_B$$

This operation is in fact common in CBV languages, called 'out', 'as', or simply a reference to a specific variant outside of a cases construct. For lists, defined as $L(A) = 1 + A \times L(A)$, we can also express this as

$$\begin{aligned} head &\equiv h \Leftarrow \{\top_1, h \downarrow ((a, l) \Rightarrow a)\} \;=\; \pi_1 \circ \iota_2^{-1l} : L(A) \to A \\ tail &\equiv t \Leftarrow \{\top_1, t \downarrow ((a, l) \Rightarrow l)\} \;=\; \pi_2 \circ \iota_2^{-1l} : L(A) \to L(A) \end{aligned}$$

2.6.3 Denotational semantics

Naturally, the denotations of both recursive values and recursive continuations are given in terms of the semantic fixpoint. The equations are in fact very simple, even when we permit variable patterns in **rec**. We add recursive continuations to CBV as follows:

$$\mathcal{C}[\![\underline{\mathbf{rec}}\; Y = C]\!]\rho v = fix(\lambda c.\lambda a.\mathcal{C}[\![C]\!]([\mathcal{Y}[\![Y]\!] \mapsto c]\rho)\; a)\; v$$

And recursive values to CBN like this:

$$\mathcal{E}[\![\overline{\mathbf{rec}}\; X = E]\!]\rho k = fix(\lambda a.\lambda c.\mathcal{E}[\![E]\!]([\mathcal{X}[\![X]\!] \mapsto a]\rho)\; c)\; k$$

Both allow us to define recursive functions. It is quite instructive to see how the CBV one works, so let us find the denotation of the factorial function expressed in terms of recursive continuations. For simplicity, we will assume that an 'if' is available, with the usual semantic equation (*i.e.*, the one presented in the introduction). In section 4.1.4, we will show how it can be defined as a continuation abstraction in the SLC. From the semantic equations, we directly obtain

$$\mathcal{F}[\![\underline{\textbf{rec}\ f = {\llcorner}n \Rightarrow \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \underline{f}\uparrow(n-1){\lrcorner}}]\!]\rho v\kappa = \cdots =$$

$$(fix\ \lambda k.\lambda t.\text{let } contx(a,c) = t \text{ in } a = 0 \rightarrow c\ 1 \parallel k\ contx(a-1, \lambda s.c\ (a \times s)))\ contx(v,\kappa)$$

This shows how the evaluation proceeds as iteration over contexts. It starts from the original input value and result continuations, and treats each recursive call as another pass through the loop denoted by k with a new context, in which the value part has been decremented, and a multiplication added to the continuation. In implementational terms, this corresponds exactly to adding activation frames to the stack, and is also very close to the view of recursion presented in [Hewitt 79].

2.7 A complete description of the SLC

2.7.1 Type system

As mentioned earlier, the SLC has a simple, static type system that encompasses both values and continuations. We present it here in a concise notation, obtained by decorating the abstract syntax grammar with type symbols. In figure 2.2, subscripts represent output or result types, while superscripts denote input or entry types. For example, $C^{\alpha+\beta}$ denotes a continuation accepting a value of type $\alpha + \beta$. For symmetry reasons and to emphasize the distinction between functions and elements, we will also use the neutral notation F_τ^σ for a function with domain σ and codomain τ, and reserve the arrow notation for closure-typed expressions and context-typed continuations.

The usual scope rules of the λ-calculus apply, so that an abstraction body may also contain variables (of both kinds) bound at an outer lexical level. We also note that when variables occur on the left side of an abstraction, *i.e.*, as formal parameters, they behave differently, so that value variables denote inputs to the function, while continuation variables denote outputs.

The rules can either describe a simple, monotyped system, where every type variable in the rules must be instantiated to a concrete type (*i.e.*, a ground term), or a polymorphic one, where the type of a function like identity can contain free type variables, as in ML. In fact, the results obtained in [Damas & Milner 82] can be carried over directly, together with the concept of most general or principal type. For example, it is easily verified that the *expression* $\zeta \equiv (f,g) \Rightarrow c \Leftarrow \{c\downarrow f, c\downarrow g\}$, (expressing the coproduct morphism constructor as a constant), has the principal type $[[A \rightarrow C] \times [B \rightarrow C] \rightarrow [A + B \rightarrow C]]$, where A, B and C are free.

We will only use simpler, monotyped view in definitions and examples. Therefore, a typing relation like $a \Rightarrow () : A \rightarrow 1$ should be read as $\forall A.\ a \Rightarrow () : A \rightarrow 1$, not $a \Rightarrow () : \forall A.A \rightarrow 1$, However, the semantic equations are also valid for polymorphic types. Explicitly parameterized types can also be considered, with functions between them as natural transformations, but again this complicates the concepts somewhat, while not directly affecting the evaluation aspects s.a. termination properties. The main problems with the type system presented here concern recursive types, and adding any extensions without resolving this aspect first would be dangerous.

$$
\begin{array}{rcl@{\qquad}rcl}
E_\tau & ::= & cst_\tau \mid x_\tau \mid F^\sigma_\tau \uparrow E_\sigma & C^\sigma & ::= & y^\sigma \mid C^\tau \downarrow F^\sigma_\tau \\
E_{unit} & ::= & () & C^{null} & ::= & \{\} \\
E_{\alpha\times\beta} & ::= & (E_\alpha, E_\beta) & C^{\alpha+\beta} & ::= & \{C^\alpha, C^\beta\} \\
E_{[\sigma\to\tau]} & ::= & F^\sigma_\tau & C^{[\tau\leftarrow\sigma]} & ::= & F^\sigma_\tau \\
F^\sigma_\tau & ::= & X^\sigma \Rightarrow E_\tau \mid E_{[\sigma\to\tau]} & F^\sigma_\tau & ::= & Y_\tau \Leftarrow C^\sigma \mid C^{[\tau\leftarrow\sigma]} \\
X^\tau & ::= & x_\tau & Y_\sigma & ::= & y^\sigma \\
X^{unit} & ::= & () & Y_{null} & ::= & \{\} \\
X^{\alpha\times\beta} & ::= & (X^\alpha, X^\beta) & Y_{\alpha+\beta} & ::= & \{Y_\alpha, Y_\beta\}
\end{array}
$$

Figure 2.2: Abstract Syntax and Type System of the SLC

2.7.2 Denotational Semantics

Figure 2.3 gives the semantic domains and equations for the CBV version of the SLC; the abstract syntax is as in figure 2.2.

$$
\begin{array}{rcl}
Val & = & Basic + Unit() + Pair(Val \times Val) + In_1(Val) + In_2(Val) + \\
 & & Closr(Val \to Cnt \to Ans) + Contx(Val \times Cnt) \\
Cnt & = & Val \to Ans \\
Env & = & Ide \to (Val + Cnt)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} : E \to Env \to Cnt & \to & Ans \\
\mathcal{E}[\![cst]\!]\rho\kappa & = & \kappa\ cst \\
\mathcal{E}[\![x]\!]\rho\kappa & = & \text{let } val(v) = \rho\ x \text{ in } \kappa\ v \\
\mathcal{E}[\![(E_1, E_2)]\!]\rho\kappa & = & \mathcal{E}[\![E_1]\!]\rho\ (\lambda v_1.\mathcal{E}[\![E_2]\!]\rho\ (\lambda v_2.\kappa\ pair(v_1, v_2))) \\
\mathcal{E}[\![()]\!]\rho\kappa & = & \kappa\ unit() \\
\mathcal{E}[\![F \uparrow E]\!]\rho\kappa & = & \mathcal{E}[\![E]\!]\rho\ (\lambda v.\mathcal{F}[\![F]\!]\rho v\kappa) \\
\mathcal{E}[\![\ulcorner F \urcorner]\!]\rho\kappa & = & \kappa\ closr(\lambda vc.\mathcal{F}[\![F]\!]\rho vc)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} : C \to Env \to Val & \to & Ans \\
\mathcal{C}[\![y]\!]\rho v & = & \text{let } cnt(\kappa) = \rho\ y \text{ in } \kappa\ v \\
\mathcal{C}[\![\{C_1, C_2\}]\!]\rho v & = & \text{case } v \text{ of } in_1(t) : \mathcal{C}[\![C_1]\!]\rho t \parallel in_2(t) : \mathcal{C}[\![C_2]\!]\rho t \text{ esac} \\
\mathcal{C}[\![\{\}]\!]\rho v & = & \text{case } v \text{ of esac} \\
\mathcal{C}[\![C \downarrow F]\!]\rho v & = & \mathcal{F}[\![F]\!]\rho v\ (\lambda t.\mathcal{C}[\![C]\!]\rho t) \\
\mathcal{C}[\![\llcorner F \lrcorner]\!]\rho v & = & \text{let } contx(a, c) = v \text{ in } \mathcal{F}[\![F]\!]\rho ac
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F} : F \to Env \to Val \to Cnt & \to & Ans \\
\mathcal{F}[\![p]\!]\rho v\kappa & = & \kappa(pv) \\
\mathcal{F}[\![X \Rightarrow E]\!]\rho v\kappa & = & \mathcal{E}[\![E]\!]\ ([\mathcal{X}[\![X]\!] \mapsto v]\rho)\kappa \\
\mathcal{F}[\![Y \Leftarrow C]\!]\rho v\kappa & = & \mathcal{C}[\![C]\!]\ ([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)v \\
\mathcal{F}[\![\overline{E}]\!]\rho v\kappa & = & \mathcal{E}[\![E]\!]\rho\ (\lambda t.\text{let } closr(f) = t \text{ in } fv\kappa) \\
\mathcal{F}[\![\underline{C}]\!]\rho v\kappa & = & \mathcal{C}[\![C]\!]\rho\ contx(v, \kappa)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{X} : X \to Val & \to & Env \to Env \\
[\mathcal{X}[\![x]\!] \mapsto v]\rho & = & [x \mapsto val(v)]\rho \\
[\mathcal{X}[\![()]\!] \mapsto v]\rho & = & \text{let } unit() = v \text{ in } \rho \\
[\mathcal{X}[\![(X_1, X_2)]\!] \mapsto v]\rho & = & \text{let } pair(v_1, v_2) = v \text{ in } [\mathcal{X}[\![X_1]\!] \mapsto v_1, \mathcal{X}[\![X_2]\!] \mapsto v_2]\rho
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : Y \to Cnt & \to & Env \to Env \\
[\mathcal{Y}[\![y]\!] \mapsto \kappa]\rho & = & [y \mapsto cnt(\kappa)]\rho \\
[\mathcal{Y}[\![\{\}]\!] \mapsto \kappa]\rho & = & \rho \\
[\mathcal{Y}[\![\{Y_1, Y_2\}]\!] \mapsto \kappa]\rho & = & [\mathcal{Y}[\![Y_1]\!] \mapsto (\lambda v.\kappa\ in_1(v)),\ \mathcal{Y}[\![Y_2]\!] \mapsto (\lambda v.\kappa\ in_2(v))]\rho
\end{array}
$$

Figure 2.3: A CBV semantics for the SLC

Chapter 3

The Symmetric Combinatory Logic

This chapter concerns the categorical aspects of the SLC. In chapter 2, we have referred to purely categorical concepts, s.a. products, coexponentials, *etc.* However, the presentation of the SLC was fundamentally element-based, and no categorical interpretation was assigned to the rather fundamental concepts of expressions and continuations, only to functions. We will start by pointing out how all SLC concepts can be integrated properly in the categorical interpretation, and what structure that imposes on the category.

We have already defined a number of categorical constructs as SLC terms, *e.g.*, id, $f \circ g$, $\langle f, g \rangle$, *etc.* In this chapter we will show that we can in fact express the entire SLC in terms of these few *combinators*, *i.e.*, terms without free variables. In principle, this is analogous to traditional combinatory logic, where the combinators $S = \lambda fgx.fx(gx)$, $K = \lambda xy.x$ and $I = \lambda x.x$ (which is not strictly necessary, because $I = SKK$) are sufficient to express any λ-calculus term. However, the SKI system is inherently higher-order, and does not handle structured data.

A more immediate similarity is therefore provided by the Categorical Combinatory Logic (CCL) of [Curien 86], which builds on the equivalence between the λ-calculus with products and cartesian closed categories (or more precisely, C-monoids). While closer in spirit, the CCL still does not handle coproducts, much less escapes. We will need additional combinators to express these, and we will call the resulting system a *Symmetric Combinatory Logic, SCL.*

Because the translation to combinators eliminates the somewhat complex concepts of abstraction and substitution, the flow of both control and data is made much more explicit in the SCL. This permits us to give a strategy-independent axiomatic presentation of the language, rather than the denotational one of chapter 2. With this, we can treat the CBV restrictions on products, *etc.* in a simple yet formal way and make precise such concepts as evaluation order and lazy products.

Finally, we will consider the categorical terms themselves as a new programming language and give a traditional denotational semantics for it. This will also be continuation-based, but with the environment "compiled away". We can in fact view it as a factorization of the original semantics, passing through the intermediate code of the combinator-based language.

3.1 A categorical view of the SLC

In this section, we will demonstrate how all of the SLC can be treated in purely categorical terms. This will be a rather informal presentation, and in particular, we will not yet consider the restrictions that a particular evaluation strategy may put on existence and uniqueness properties of products, exponentials, *etc.* We will treat these aspects formally in section 3.2, and give the exact translations between SLC and SCL in sections 3.3 and 3.4.

3.1.1 Functions, values and continuations

Let us start by recalling from section 2.1.2 that we could treat the SLC types as objects in a category, with the functions between them as morphisms. However, values and continuations were not assigned a categorical interpretation, but were treated intuitively as "elements" and "requests for elements" of various types. This approach was chosen in order to explain the SLC in terms of traditional (at least for values) programming language constructs and concepts, that could be directly related to the denotational semantics. This section will present a morphism-based view of the SLC core, *i.e.*, the concepts of abstraction and application, as applying to both values and continuations.

As usual, we will start with the familiar value-based side. There is a natural view of expressions as degenerate functions, with no (information-carrying) inputs *from the point of application.* An expression of type T can therefore be thought of as a morphism $1 \to T$. (Not $0 \to T$; $\Box_T$ is the only such morphism). In this view, (value) application corresponds to simply morphism composition, *e.g.*, if the function F from S to T denotes a morphism $f : S \to T$, and expression E of type S a morphism $e : 1 \to S$, then the expression $F \uparrow E$ denotes the morphism $f \circ e : 1 \to T$.

The dual case is analogous. A continuation *seen from within SLC* is like a one-ended function that accepts input, but does not (and can not) return *to the point of application.* We can express this by viewing an S-typed continuation as a morphism $S \to 0$ (not $S \to 1$; $\Diamond_S$ is the only one there). In **Set**, there are no morphisms of this type, but neither is there any concept of escaping. Again, (continuation) application corresponds to composition: if C is a T-accepting continuation, denoting the morphism $c : T \to 0$, and the function $F : S \to T$ represents a morphism $f : S \to T$, then the S-accepting continuation $C \downarrow F$ denotes the morphism $c \circ f : S \to 0$.

Value and continuation abstractions are more complicated, but are based on adjoining indeterminate arrows, denoted by variable symbols, to the category. We will see that an expression $E : T$, with morphism $e : 1 \to T$ in which the symbol x occurs as an S-typed value determines a unique x-free morphism $xe : S \to T$, s.t. $xe \uparrow x = e$. Similarly, a continuation $C : S$, *i.e.*, morphism $c : S \to 0$ expressed in terms of a continuation variable $y : T$ defines a categorical morphism $yc : S \to T$, s.t. $y \downarrow yc = c$. This important property, known as functional completeness is the subject of section 3.4.

3.1.2 Structured types

As we have mentioned in section 2.2, the structured types of SLC are based directly on categorical concepts. There we also showed how the categorical constructs s.a. projections and mediating morphisms could be expressed in the SLC. In the following, we will see that the reverse translations are equally simple.

For products, if two expressions $E_1 : T_1$ and $E_2 : T2$ denote morphisms $e_1 : 1 \to T_1$ and $e_2 : 1 \to T_2$ respectively, then the expression $(E_1, E_2) : T_1 \times T_2$ denotes precisely the morphism $\langle e_1, e_2 \rangle : 1 \to T_1 \times T_2$. Similarly, for continuations $C_1 : S_1$ and $C_2 : S_2$ denoting morphisms $c_1 : S_1 \to 1$ and $c_2 : S_2 \to 0$, the $S_1 + S_2$-accepting continuation $\{C_1, C_2\}$ corresponds exactly to the morphism $[c_1, c_2] : S_1 + S_2 \to 0$.

The "variable pattern" syntax used in destructuring abstractions like $(a, b) \Rightarrow a + b$ also has a simple categorical interpretation. We can see it as simply a shorthand for explicit projections and injections, so that we have

$$(a, b) \Rightarrow E \;\equiv\; x \Rightarrow E[(\pi_1 \uparrow x)/a, (\pi_2 \uparrow x)/b]$$

i.e., all free occurrences of a and b can be replaced by projections from the value parameter x. Nested patterns correspond to longer sequences of projections. Similarly, but perhaps less intuitively,

continuation patterns on the left side of $\Leftarrow$ correspond to explicit injections in the body of the abstraction:

$$\{a,b\} \Leftarrow C \;\equiv\; y \Leftarrow C[\{y\downarrow\iota_1\}/a, \{y\downarrow\iota_2\}/b]$$

The destructuring syntax is used in the SLC to avoid the need for predefined functions. Also, it emphasizes the symmetry between products and coproducts, by dualizing the common notation for functions from a product to functions into a coproduct.

The treatment of terminal and initial objects is even simpler. The 1-typed expression () denotes the morphism $\Diamond_1 : 1 \to 1$ (or just id_1). Analogously, the 0-typed continuation $\{\}$ denotes the morphism $\Box_0 : 0 \to 0$ (or id_0).

Empty variable patterns are almost trivial. In fact, the *expression* $E : T$ and the *function* $() \Rightarrow E : 1 \to T$ denote exactly the same morphism. Analogously, the S-typed continuation C and the function $\{\} \Leftarrow C : S \to 0$ are equivalent. Note, however, that these interpretations must not be confused with the *coercions* mentioned in section 2.4.1. The result of coercing the closure-typed expression $E : [S \to T]$ into a function $\overline{E} : S \to T$ is very different from the function $() \Rightarrow E : 1 \to [S \to T]$.

As we have remarked in section 2.3.2, having just products and coproducts is not quite enough to express every first-order SLC term with categorical combinators. To handle non-local variables appearing inside morphism constructors, we will need additional combinators. In fact, just the two "distributive law morphisms" δ and $\overline{\delta}$ from that section are sufficient for this, as we will show in section 3.4. We will not, however, adopt δ and $\overline{\delta}$ as primitive combinators, because they can also be defined in terms of the higher-order constructs we will introduce now:

3.1.3 Higher-order features

In the categorical view, functions are arrows *between* objects, and it is not at all clear how to handle higher-order functions properly. The key is provided by the coercions of section 2.4.1, together with the views of values and continuations as morphisms from 1 and into 0 respectively.

Naively, we want a correspondence between functions $B \to C$ and values of type $[B \to C]$, *i.e.*, between morphisms:

$$\frac{f : B \to C}{\ulcorner f \urcorner : 1 \to [B \to C]}$$

Similarly, the correspondence between functions $B \to C$ and $C \leftarrow B$-typed continuations could be based on the categorical interpretation of the coercions

$$\frac{f : B \to C}{\llcorner f \lrcorner : [C \leftarrow B] \to 0}$$

These bijections describe the closure and context objects solely in terms of the morphisms around them, not their internal structure, which is precisely what we want. However, in order to handle free variables in a function body, we will need slightly more general versions. For functional values we must have the properties of a cartesian closed category: a bijection between morphisms and their curried forms:

$$\frac{f : A \times B \to C}{f^* : A \to [B \to C]}$$

We can also express this as the existence of a right adjoint to the product functor. The special case $A = 1$ gives the original bijection, because of the isomorphisms

$$\pi_2 : 1 \times B \to B \qquad\qquad \langle \Diamond, id \rangle : B \to 1 \times B$$

The general version of the dual case gives a bijection between morphisms into a coproduct and their cocurried versions:

$$\frac{f : C \to A + B}{f_* : [B \leftarrow C] \to A}$$

Equivalently, we require the existence of a left adjoint to the coproduct functor. Here, the bridge back to the original case is given by $A = 0$, because of the isomorphisms

$$\iota_2 : B \to 0 + B \qquad\qquad [\Box, id] : 0 + B \to B$$

Unfortunately, the SLC with higher-order functions is still more expressive than the category described above, because it is also possible to have references to non-local continuations in value abstractions and vice versa. To handle these, we will need two special morphisms, very similar to the distributive laws for products and coproducts, but we defer the details to section 3.4.

3.2 The SCL category and its axioms

This section gives a formal specification of the SCL category: its objects, morphisms and axioms. In particular, we will treat properly the existence and uniqueness problems that we ignored in section 3.1.

We will need a category with a fair amount of structure, *i.e.*, a number of distinguished objects and morphisms that will function as products, exponentials, projections, *etc.* While we will use the traditional names and notation for these, it must be remembered that their associated axioms are somewhat weaker than the usual ones, in that they sometimes only quantify over *well-behaved* morphisms. In fact, a category that obeyed all the axioms in full generality would collapse to a Boolean algebra, and with recursion added to that, it would become inconsistent.

The objects include all the basic types (integers, booleans *etc.*). In addition to those, there must be a designated *initial object* 0 and *terminal object* 1. Furthermore, for every pair of objects A and B, there must exist a specific *product* $A \times B$, *coproduct* $A + B$, *exponential* $[A \to B]$ and *coexponential* $[A \leftarrow B]$ object.

The morphisms consists of the primitive functions, and morphisms derived from the categorical constructs, *e.g.*, injections, curried forms, *etc.* Furthermore, for each type, the category must include a denumerable set of value and continuation variables, corresponding to free variables in SLC terms. SLC constants can be thought of as either primitive functions $1 \to T$ or global (value) variables.

The choice of axioms is very delicate. They should be weak enough to allow the imperfections of CBV products, *etc.*, and not give problems when recursion is added. On the other hand, they should be strong enough to prove as much as possible, in particular functional completeness of the category. We will present and motivate them one by one in this section; they are also summarized in figure 3.2.

Some of the axioms will look rather strange. An important question is therefore whether a real SCL category exists, and what it looks like. An affirmative answer to the former question, together with one possibility for the latter is given by the denotational semantics in section 3.8.

3.2.1 Totality and strictness

Before we can present the axioms, we must formalize the concept of well-behaved morphisms. Basically, they are precisely the ones which respect the designated initial and terminal objects of the category:

Definition 3.1 *A morphism $f : A \to B$ will be called* strict, *if $f \circ \Box_A = \Box_B$, and* total *if $\Diamond_B \circ f = \Diamond_A$.*

These names are based on familiar terms, but have slightly different meanings. We use them because they convey the essence of the definition in an intuitively understandable way, but they should be regarded as precise technical terms.

A total morphism is essentially one which always produces an output, while a strict one always consumes its input. Note that this is slightly stronger than the usual definition of strictness ($f \uparrow \bot = \bot$) in that, *e.g.*, the morphism denoted by $x \Rightarrow \bot$ is not necessarily strict according to our definition, precisely because it does not evaluate x.

Whether a morphism is total or strict depends not only on its definition as an SLC or SCL term, but also on the evaluation strategy. For example, in CBV all functions, even ones like $x \Rightarrow 2$ evaluate their arguments and so are strict. This is a direct consequence of the $\bar{\beta}$ rule, because $F \circ \Box_A = F \circ (a \Leftarrow \{\}) = (b \Leftarrow \{\})$. More surprisingly, all functions are considered total in CBN, because $\Diamond_B \circ F = (b \Rightarrow ()) \circ F = a \Rightarrow ()$.

We can also make precise two key concepts of the SLC:

Definition 3.2 *A* value *of type T is a total morphism $1 \to T$. A* (semantic) continuation *of type S is a strict morphism $S \to 0$.*

3.2.2 Core axioms

Of course, the usual axioms of a category (neutrality of identity and associativity of composition) must be satisfied for *all* morphisms, even "badly-behaved" ones. We remark that identity is naturally total and strict (by definition), and that the composition of two total (strict) morphisms is also total (strict).

We now present the axioms specific to SCL. Here we can be more restrictive.

The key is to keep the axiom system symmetrical, *i.e.*, for every axiom its dual must hold as well. Apart from aesthetic considerations, this will also reduce the work in proving subsequent theorems by half, because every result can be immediately dualized. Also, some continuation-based axioms are somewhat counterintuitive in CBN; We should use experience with CBV to develop them. Thus, we will only present the value-based axioms here, but remember that their duals must hold as well.

Let us first consider the terminal object. While we cannot require the $\Diamond$ morphisms to be unique, they should certainly be unique among total morphisms. This leads to our first axiom:

$$\Diamond_1 = id_1$$

As we shall see in section 3.2.5, this is enough to prove the general case.

The products are more complicated. We start by requiring that if f and g are total (strict), so should $\langle f, g \rangle$ be. In CBV, we cannot expect $\pi_1 \circ \langle f, g \rangle$ to be the same as f, if g is not well-behaved. But if g is total, we must be able to project it away. This motivates the following two axioms:

$$\pi_1 \circ \langle f, g \rangle = f \text{ if } g \text{ total} \qquad \pi_2 \circ \langle f, g \rangle = g \text{ if } f \text{ total}$$

We do not have general uniqueness ($\langle \pi_1 \circ f, \pi_2 \circ f \rangle = f$) either. But we must at least have uniqueness if f is total. We will codify the essence of this as one axiom, similar to the one for the terminal object:

$$\langle \pi_1, \pi_2 \rangle = id.$$

Next, we must be able to compose $\langle f, g\rangle$ with other morphisms. We want to interleave evaluation of f and g with total morphisms. This leads to the following two axioms:

$$\begin{aligned} \langle f, g\rangle \circ h &= \langle f \circ h, g \circ h\rangle, \text{ if h total} \\ (h \times k) \circ \langle f, g\rangle &= \langle h \circ f, k \circ g\rangle, \text{ if h,k total} \end{aligned}$$

We see that a satisfactory handling of products in SCL required 5 axioms, rather than the usual 3. This is precisely because they are weaker, so that properties that could otherwise be proven from the strong axioms must be asserted explicitly.

3.2.3 Higher-order axioms

The higher-order facilities of the SCL are based on exponentials and currying. We will present the relevant (value) axioms in this subsection.

Let us first consider the totality and strictness properties of the involved morphisms. f^* is always total, even when f itself is not, but it is not necessarily strict (in CBN) even if f is, *e.g.*, the function $a \Rightarrow b \Rightarrow b = {\pi_2}^*$ should not evaluate a. On the other hand, ap always strict (in its first argument): if the result of ap is needed, it will have to evaluate its input. It is not necessarily total (*e.g.*, when applying the closure of an escaping function).

For values, there is no problem with existence of a curried f, so we can adopt one (the only) CCC axiom in full generality:

$$ap \circ (f^* \times id) = f.$$

where we recall that the notation $f^* \times id$ is an abbreviation for $\langle f^* \circ \pi_1, \pi_2\rangle$. On the other hand, uniqueness of the curried form is not guaranteed, *i.e.*, there may be many different functions satisfying the above equation. Again, this is true for any CBV-type language with runtime errors or escapes: the uncurried forms of the two following two ML functions

```
fn x=>let t=1/x in (fn y=>y+t)
```

```
fn x=>fn y=>y+1/x
```

are indistinguishable (both compute the function $\phi(x, y) = y + 1/x$), but behave differently (the first will fail if applied to zero, the second will not) in original form. Therefore, we cannot demand uniqueness among *all* functions, only among total ones. This leads to the weakened axiom:

$$(ap \circ \langle g \circ \pi_1, \pi_2\rangle)^* = g, \text{ if } g \text{ total}$$

Ideally, we would be finished now. However, in order to express every SLC function we will need yet another special morphism, to mediate between values and continuations. We can express it in a number of equivalent ways, but we will take the form that simplifies the following presentations the most:

$$\phi : A \times [C \leftarrow B] \rightarrow [C \leftarrow A \times B]$$

It must satisfy the following two axioms:

$$\phi \circ \langle f \circ \Diamond, id\rangle = (pa \circ \langle f \circ \Diamond, id\rangle)_*$$

$$(f \circ \pi_2)_* \circ \phi = f_* \circ \pi_2$$

We will see that this is very similar to what we require of the "distributive law" δ. It seems plausible, in fact, that both δ and ϕ are instances of a more general schema, but this has not

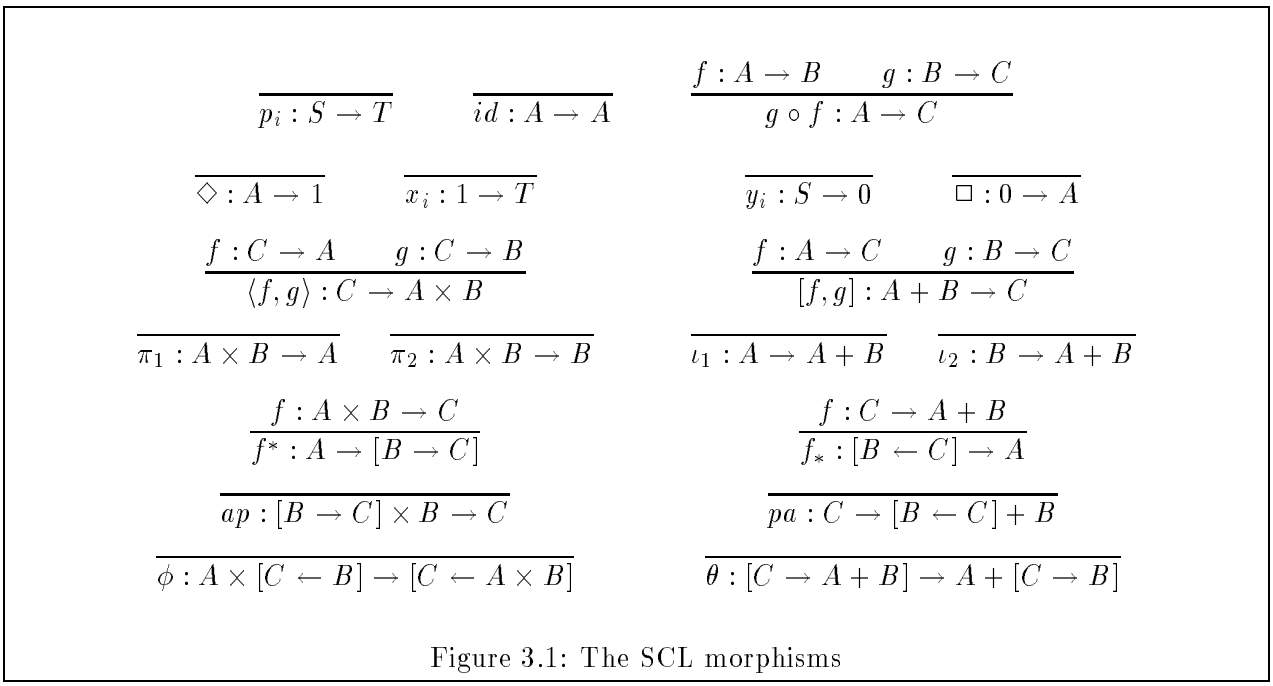


Figure 3.1: The SCL morphisms

been investigated yet. Just as for the distributive law, the morphism in the other direction can be expressed using only existing combinators:

$$\phi^{-1} = \langle (\iota_1 \circ \pi_1)_*, (pa \circ \pi_2)_* \rangle : [C \leftarrow A \times B] \to A \times [C \leftarrow B]$$

But in general this is not the inverse, although it is for CBV evaluation.

3.2.4 Summary of morphisms and axioms

We summarize the morphisms of SCL in figure 3.1; this also specifies its type system. The axioms are collected in section 3.2. In the equations, t represents a total morphism, and s a strict one.

3.2.5 Derived equalities

This section presents a number of derived laws. Again, we will only state and prove the value part.

Lemma 3.1 *If $f : A \to 1$, is total, $f = \Diamond_A$.*

Proof $f = id_1 \circ f = \Diamond_1 \circ f = \Diamond_A$. []

Lemma 3.2 *If $f : C \to A \times B$ is total, $f = \langle \pi_1 \circ f, \pi_2 \circ f \rangle$.*

Proof $f = id \circ f = \langle \pi_1, \pi_2 \rangle \circ f = \langle \pi_1 \circ f, \pi_2 \circ f \rangle$. []

Lemma 3.3 $ap \circ \langle f^* \circ g, h \rangle = f \circ \langle g, h \rangle$.

Proof $ap \circ \langle f^* \circ g, h \rangle = ap \circ (f^* \times id) \circ \langle g, h \rangle = f \circ \langle g, h \rangle$. []

Lemma 3.4 *If k is total, $h^* \circ k = (h \circ (k \times id))^*$.*

Total			Strict		
$p_i, id, t_1 \circ t_2, \diamond, \langle t_1, t_2\rangle, \pi_1, \pi_2,$ $\Box, [t_1, t_2], \iota_1, \iota_2, \phi, \theta,$ x_i, pa, f^*			$p_i, id, t_1 \circ t_2, \diamond, \langle t_1, t_2\rangle, \pi_1, \pi_2,$ $\Box, [t_1, t_2], \iota_1, \iota_2, \phi, \theta,$ y_i, ap, f_*		
$\diamond_1$	$=$	id_1	$\Box_0$	$=$	id_0
$\pi_1 \circ \langle f, t\rangle$	$=$	f	$[f, s] \circ \iota_1$	$=$	f
$\pi_2 \circ \langle t, g\rangle$	$=$	g	$[s, g] \circ \iota_2$	$=$	g
$\langle \pi_1, \pi_2\rangle$	$=$	id	$[\iota_1, \iota_2]$	$=$	id
$\langle f, g\rangle \circ t$	$=$	$\langle f \circ t, g \circ t\rangle$	$s \circ [f, g]$	$=$	$[s \circ f, s \circ g]$
$(t_1 \times t_2) \circ \langle f, g\rangle$	$=$	$\langle t_1 \circ f, t_2 \circ g\rangle$	$[f, g] \circ (s_1 + s_2)$	$=$	$[f \circ s_1, g \circ s_2]$
$ap \circ (f^* \times id)$	$=$	f	$(f_* + id) \circ pa$	$=$	f
$(ap \circ (t \times id))^*$	$=$	t	$((s + id) \circ pa)_*$	$=$	s
$\phi \circ \langle f \circ \diamond, id\rangle$	$=$	$(pa \circ \langle f \circ \diamond, id\rangle)_*$	$[\Box \circ f, id] \circ \theta$	$=$	$([\Box \circ f, id] \circ ap)^*$
$(f \circ \pi_2)_* \circ \phi$	$=$	$f_* \circ \pi_2$	$\theta \circ (\iota_2 \circ f)^*$	$=$	$\iota_2 \circ f^*$

Figure 3.2: The SCL axioms

Proof $h^* \circ k = (ap \circ ((h^* \circ k) \times id))^* = (ap \circ \langle h^* \circ k \circ \pi_1, \pi_2\rangle)^* = (h \circ \langle k \circ \pi_1, \pi_2\rangle)^* = (h \circ (k \times id))^*$ ▯

Lemma 3.5 *If f is total, $\langle f \circ \diamond, g\rangle = \langle f \circ \diamond, id\rangle \circ g$. The equation also holds (trivially) if g is total.*

Proof (This depends centrally on the fact that projections are total)

$$\langle f \circ \diamond, g\rangle = \langle f \circ \diamond \circ \pi_1, \pi_2\rangle \circ \langle id, g\rangle = \langle f \circ \diamond \circ \pi_2, \pi_2\rangle \circ \langle id, g\rangle =$$

$$\langle f \circ \diamond, id\rangle \circ \pi_2 \circ \langle id, g\rangle = \langle f \circ \diamond, id\rangle \circ g$$

▯

Lemma 3.6 *If f is strict, $f \circ ap \circ \langle [g^*, h^*] \circ \pi_1, \pi_2\rangle = ap \circ \langle [(f \circ g)^*, (f \circ h)^*] \circ \pi_1, \pi_2\rangle$.*

Proof

$$\begin{aligned}
f \circ ap \circ \langle [g^*, h^*] \circ \pi_1, \pi_2\rangle &= ap \circ \langle (f \circ ap \circ \langle [g^*, h^*] \circ \pi_1, \pi_2\rangle)^* \circ \pi_1, \pi_2\rangle \\
&= ap \circ \langle (f \circ ap)^* \circ [g^*, h^*] \circ \pi_1, \pi_2\rangle \\
&= ap \circ \langle [(f \circ ap)^* \circ g^*, (f \circ ap)^* \circ h^*] \circ \pi_1, \pi_2\rangle \\
&= ap \circ \langle [(f \circ ap \circ (g^* \times id))^*, (f \circ ap \circ (h^* \times id))^*] \circ \pi_1, \pi_2\rangle \\
&= ap \circ \langle [(f \circ g)^*, (f \circ h)^*] \circ \pi_1, \pi_2\rangle
\end{aligned}$$

▯

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{L}[\![p]\!] & = & p \;=\; x \Rightarrow p\uparrow x & \mathcal{L}[\![p]\!] & = & p \;=\; y \Leftarrow y\downarrow p \\
\mathcal{L}[\![id]\!] & = & x \Rightarrow x & \mathcal{L}[\![id]\!] & = & y \Leftarrow y \\
\mathcal{L}[\![f \circ g]\!] & = & x \Rightarrow \mathcal{L}[\![f]\!]\uparrow(\mathcal{L}[\![g]\!]\uparrow x) & \mathcal{L}[\![f \circ g]\!] & = & y \Leftarrow (y\downarrow\mathcal{L}[\![f]\!])\downarrow\mathcal{L}[\![g]\!] \\
\mathcal{L}[\![x]\!] & = & () \Rightarrow x & \mathcal{L}[\![y]\!] & = & \{\} \Leftarrow y \\
\mathcal{L}[\![\diamond]\!] & = & x \Rightarrow () & \mathcal{L}[\![\Box]\!] & = & y \Leftarrow \{\} \\
\mathcal{L}[\![\langle f,g\rangle]\!] & = & x \Rightarrow (\mathcal{L}[\![f]\!]\uparrow x, \mathcal{L}[\![g]\!]\uparrow x) & \mathcal{L}[\![[f,g]]\!] & = & y \Leftarrow \{y\downarrow\mathcal{L}[\![f]\!], y\downarrow\mathcal{L}[\![g]\!]\} \\
\mathcal{L}[\![\pi_1]\!] & = & (a,b) \Rightarrow a & \mathcal{L}[\![\iota_1]\!] & = & \{a,b\} \Leftarrow a \\
\mathcal{L}[\![\pi_2]\!] & = & (a,b) \Rightarrow b & \mathcal{L}[\![\iota_2]\!] & = & \{a,b\} \Leftarrow b \\
\mathcal{L}[\![f^*]\!] & = & a \Rightarrow b \Rightarrow \mathcal{L}[\![f]\!]\uparrow(a,b) & \mathcal{L}[\![f_*]\!] & = & a \Leftarrow b \Leftarrow \{a,b\}\downarrow\mathcal{L}[\![f]\!] \\
\mathcal{L}[\![ap]\!] & = & (g,b) \Rightarrow g\uparrow b & \mathcal{L}[\![pa]\!] & = & \{g,b\} \Leftarrow b\downarrow g \\
\mathcal{L}[\![\phi]\!] & = & & \mathcal{L}[\![\theta]\!] & = & \\
 & & (x,q) \Rightarrow (p \Leftarrow \{c \Rightarrow p\uparrow(x,c)\})\uparrow q & & & \{y,p\} \Leftarrow p\downarrow(q \Rightarrow (c \Leftarrow \{y,c\}\downarrow q))
\end{array}
$$

Figure 3.3: Translation from SCL to SLC

3.3 Translation from SCL to SLC

Throughout chapter 2, it was demonstrated how categorical constructs such as products, exponentials, *etc.* could be naturally expressed with SLC terms. In figure 3.3, we formalize this idea, giving a translation from every SCL morphism to an equivalent SLC function.

Naturally, the variables introduced by the rules must not capture any free variables in the categorical term. Note that for identity and composition, we have two exactly equivalent translations, either as value or continuation abstractions. The translations for primitive functions demonstrate the η and $\bar{\eta}$ conversion rules of the SLC. The only complicated part consists of the translations of ϕ and θ. The idea is that ϕ adds a value to a context, while θ extracts a continuation from a closure.

3.4 Translation from SLC to SCL

While the above result was reasonably straightforward, the inverse translation is somewhat more complicated. We will present a (reasonably) simple translation scheme to demonstrate the idea; in practice, it produces rather large SCL terms that can usually be simplified considerably, just like the naive SKI abstraction algorithm for the λ-calculus.

The main difference between SLC and SCL is that the latter has no variable bindings. We must therefore be able express all value and continuation abstractions using only the SCL combinators. This is possible because of the following central theorem:

Theorem 3.1 (Functional Completeness) *Let $f : A \to B$ be a morphism expressed in terms of the S-typed (value) variable x. Then there exists a unique morphism $f^x : S \times A \to B$ not containing x, such that $f^x \circ \langle x \circ \diamond, id\rangle = f$. Dually, for a variable $y : T \to 0$, there exists a unique morphism $f_y : A \to T + B$, such that $[\Box \circ y, id] \circ f_y = f$.*

Proof The detailed verifications are rather verbose, but simple in principle. In fact, many of the CCC results from [Lambek & Scott 86] can be carried over directly, verifying that only the weaker forms of the axioms are used. Since the SCL axioms are symmetrical, the results dualize immediately to continuation abstractions.

First, we define a number of auxiliary morphisms, expressing associativity, commutativity and distributivity of products and coproducts:

$$\begin{array}{lcl}
\alpha & \equiv & \langle \pi_1 \circ \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle \rangle : (A \times B) \times C \to A \times (B \times C) \\
\bar{\alpha} & \equiv & [\iota_1 \circ \iota_1, [\iota_1 \circ \iota_2, \iota_2]] : A + (B + C) \to (A + B) + C \\
\gamma & \equiv & \langle \pi_2, \pi_1 \rangle : A \times B \to B \times A \\
\bar{\gamma} & \equiv & [\iota_2, \iota_1] : A + B \to B + A \\
\delta & \equiv & ap \circ ([(\iota_1 \circ \gamma)^*, (\iota_2 \circ \gamma)^*] \times id) \circ \gamma : A \times (B + C) \to (A \times B) + (A \times C) \\
\bar{\delta} & \equiv & \bar{\gamma} \circ (\langle (\bar{\gamma} \circ \pi_1)_*, (\bar{\gamma} \circ \pi_2)_* \rangle + id) \circ pa : (A + B) \times (A + C) \to A + (B \times C)
\end{array}$$

Of these, the first four are isomorphisms in any SCL category and conceptually trivial. δ and $\bar{\delta}$, on the other hand, require some work. Let us verify the two properties we will need of δ:

Lemma 3.7 $\delta \circ \langle f \circ \Diamond, id \rangle = \langle f \circ \Diamond, id \rangle + \langle f \circ \Diamond, id \rangle$ *if* f *total.*

Proof

$$\begin{array}{rcl}
\delta \circ \langle f \circ \Diamond, id \rangle & = & ap \circ \langle [(\iota_1 \circ \gamma)^*, (\iota_2 \circ \gamma)^*] \circ \pi_1, \pi_2 \rangle \circ \gamma \circ \langle f \circ \Diamond, id \rangle \\
& = & ap \circ \langle [(\iota_1 \circ \gamma)^*, (\iota_2 \circ \gamma)^*], f \circ \Diamond \rangle \\
& = & ap \circ \langle id, f \circ \Diamond \rangle \circ [(\iota_1 \circ \gamma)^*, (\iota_2 \circ \gamma)^*] \\
& = & [ap \circ \langle id, f \circ \Diamond \rangle \circ (\iota_1 \circ \gamma)^*, ap \circ \langle id, f \circ \Diamond \rangle \circ (\iota_2 \circ \gamma)^*] \\
& = & [ap \circ \langle (\iota_1 \circ \gamma)^*, f \circ \Diamond \rangle, ap \circ \langle (\iota_2 \circ \gamma)^*, f \circ \Diamond \rangle] \\
& = & [\iota_1 \circ \gamma \circ \langle id, f \circ \Diamond \rangle, \iota_2 \circ \gamma \circ \langle id, f \circ \Diamond \rangle] \\
& = & \langle f \circ \Diamond, id \rangle + \langle f \circ \Diamond, id \rangle
\end{array}$$

□

Lemma 3.8 $(\pi_2 + \pi_2) \circ \delta = \pi_2$

Proof

$$\begin{array}{rcl}
(\pi_2 + \pi_2) \circ \delta & = & (\pi_2 + \pi_2) \circ ap \circ \langle [(\iota_1 \circ \gamma)^*, (\iota_2 \circ \gamma)^*] \circ \pi_1, \pi_2 \rangle \circ \gamma \\
& = & ap \circ \langle [((\pi_2 + \pi_2) \circ \iota_1 \circ \gamma)^*, ((\pi_2 + \pi_2) \circ \iota_2 \circ \gamma)^*] \circ \pi_2, \pi_1 \rangle \\
& = & ap \circ \langle [(\iota_1 \circ \pi_2 \circ \gamma)^*, (\iota_2 \circ \pi_2 \circ \gamma)^*] \circ \pi_2, \pi_1 \rangle \\
& = & ap \circ \langle [(\iota_1 \circ \pi_1)^*, (\iota_2 \circ \pi_1)^*] \circ \pi_2, \pi_1 \rangle \\
& = & ap \circ \langle [(\pi_1 \circ \langle \iota_1 \circ \pi_1, \pi_2 \rangle)^*, (\pi_1 \circ \langle \iota_2 \circ \pi_1, \pi_2 \rangle)^*] \circ \pi_2, \pi_1 \rangle \\
& = & ap \circ \langle [{\pi_1}^* \circ \iota_1, {\pi_1}^* \circ \iota_2] \circ \pi_2, \pi_1 \rangle = ap \circ \langle {\pi_1}^* \circ [\iota_1, \iota_2] \circ \pi_2, \pi_1 \rangle \\
& = & ap \circ \langle {\pi_1}^* \circ \pi_2, \pi_1 \rangle = \pi_1 \circ \langle \pi_2, \pi_1 \rangle = \pi_2
\end{array}$$

□

We can now define the *abstraction rules* of figure 3.4.

As the rules are overlapping (if x does not occur in a complex term, two equations are applicable), we must first show that this really is a definition, *i.e.*, that the rules determine a unique result. We use structural induction. The base case is trivial: no simple term can contain x, so only

$$
\begin{array}{rcl@{\qquad}rcl}
x^x & = & \pi_1 & y_y & = & \iota_1 \\
f^x & = & f \circ \pi_2 \text{ (if } x \text{ not in } f) & f_y & = & \iota_2 \circ f \text{ (if } y \text{ not in } f) \\
(f \circ g)^x & = & f^x \circ \langle \pi_1, g^x \rangle & (f \circ g)_y & = & [\iota_1, f_y] \circ g_y \\
\langle f, g \rangle^x & = & \langle f^x, g^x \rangle & \langle f, g \rangle_y & = & \bar{\delta} \circ \langle f_y, g_y \rangle \\
[f, g]^x & = & [f^x, g^x] \circ \delta & [f, g]_y & = & [f_y, g_y] \\
f^{*x} & = & (f^x \circ \alpha)^* & f^*{}_y & = & \theta \circ (f_y)^* \\
f_*{}^x & = & (f^x)_* \circ \phi & f_{*y} & = & (\bar{\alpha} \circ f_y)_*
\end{array}
$$

Figure 3.4: Abstraction rules

one rule applies. For each possibility of a composite term, we get:

$$
\begin{array}{rcl}
h = f \circ g & : & (f \circ g)^x = f^x \circ \langle \pi_1, g^x \rangle = f \circ \pi_2 \circ \langle \pi_1, g \circ \pi_2 \rangle = f \circ g \circ \pi_2 = h^x \\
h = \langle f, g \rangle & : & \langle f, g \rangle^x = \langle f^x, g^x \rangle = \langle f \circ \pi_2, g \circ \pi_2 \rangle = \langle f, g \rangle \circ \pi_2 = h^x \\
h = [f, g] & : & [f, g]^x = [f^x, g^x] \circ \delta = [f \circ \pi_2, g \circ \pi_2] \circ \delta = [f, g] \circ (\pi_2 + \pi_2) \circ \delta = \\
 & & [f, g] \circ \pi_2 = h^x \\
h = f^* & : & f^{*x} = (f^x \circ \alpha)^* = (f \circ \pi_2 \circ \alpha)^* = (f \circ \langle \pi_2 \circ \pi_1, \pi_2 \rangle)^* = f^* \circ \pi_2 = h^x \\
h = f_* & : & f_*{}^x = (f^x)_* \circ \phi = (f \circ \pi_2)_* \circ \phi = f_* \circ \pi_2 = h^x
\end{array}
$$

We then show that $f^x \circ \langle x \circ \diamond, id \rangle = f$ by structural induction on f:

$$
\begin{array}{rcl}
x^x \circ \langle x \circ \diamond, id \rangle & = & \pi_1 \circ \langle x \circ \diamond, id \rangle = x \circ \diamond = x \circ id_1 = x \\
f^x \circ \langle x \circ \diamond, id \rangle & = & f \circ \pi_2 \circ \langle x \circ \diamond, id \rangle = f \text{ (if } x \text{ not in } f) \\
(f \circ g)^x \circ \langle x \circ \diamond, id \rangle & = & f^x \circ \langle \pi_1, g^x \rangle \circ \langle x \circ \diamond, id \rangle \\
 & = & f^x \circ \langle \pi_1 \circ \langle x \circ \diamond, id \rangle, g^x \circ \langle x \circ \diamond, id \rangle \rangle \\
 & = & f^x \circ \langle x \circ \diamond, g \rangle = f^x \circ \langle x \circ \diamond, id \rangle \circ g = f \circ g \\
\langle f, g \rangle^x \circ \langle x \circ \diamond, id \rangle & = & \langle f^x, g^x \rangle \circ \langle x \circ \diamond, id \rangle = \langle f^x \circ \langle x \circ \diamond, id \rangle, g^x \circ \langle x \circ \diamond, id \rangle \rangle \\
 & = & \langle f, g \rangle \\
[f, g]^x \circ \langle x \circ \diamond, id \rangle & = & [f^x, g^x] \circ \delta \circ \langle x \circ \diamond, id \rangle \\
 & = & [f^x, g^x] \circ [\iota_1 \circ \langle x \circ \diamond, id \rangle, \iota_2 \circ \langle x \circ \diamond, id \rangle] \\
 & = & [f^x \circ \langle x \circ \diamond, id \rangle, g^x \circ \langle x \circ \diamond, id \rangle] = [f, g] \\
f^{*x} \circ \langle x \circ \diamond, id \rangle & = & (f^x \circ \alpha)^* \circ \langle x \circ \diamond, id \rangle = (f^x \circ \alpha \circ \langle \langle x \circ \diamond, id \rangle \circ \pi_1, \pi_2 \rangle)^* \\
 & = & (f^x \circ \langle x \circ \diamond \circ \pi_1, \langle \pi_1, \pi_2 \rangle \rangle)^* = (f^x \circ \langle x \circ \diamond, id \rangle)^* = f^* \\
f_*{}^x \circ \langle x \circ \diamond, id \rangle & = & (f^x)_* \circ \phi \circ \langle x \circ \diamond, id \rangle = (f^x)_* \circ (pa \circ \langle x \circ \diamond, id \rangle)_* \\
 & = & ((f^x)_* + id) \circ pa \circ \langle x \circ \diamond, id \rangle)_* = (f^x \circ \langle x \circ \diamond, id \rangle)_* = f_*
\end{array}
$$

Finally, we prove uniqueness of f^x, *i.e.*, given an x-free morphism g, s.t. $g \circ \langle x \circ \diamond, id \rangle = f$, we show that $g = f^x$:

$$
\begin{array}{c}
f^x = (g \circ \langle x \circ \diamond, id \rangle)^x = g^x \circ \langle \pi_1, \langle x \circ \diamond, id \rangle^x \rangle = g \circ \pi_2 \circ \langle \pi_1, \langle (x \circ \diamond)^x, id^x \rangle \rangle = \\
g \circ \langle x^x \circ \langle \pi_1, \diamond^x \rangle, \pi_2 \rangle = g \circ \langle \pi_1 \circ \langle \pi_1, \diamond \circ \pi_2 \rangle, \pi_2 \rangle = g \circ \langle \pi_1, \pi_2 \rangle = g
\end{array}
$$

$\square$

We can now reap the fruits:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}[\![x]\!] & = & x & \mathcal{C}[\![y]\!] & = & y \\
\mathcal{E}[\![()]\!] & = & \diamond & \mathcal{C}[\![\{\}]\!] & = & \Box \\
\mathcal{E}[\![(E_1, E_2)]\!] & = & \langle \mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!] \rangle & \mathcal{C}[\![\{C_1, C_2\}]\!] & = & [\mathcal{C}[\![C_1]\!], \mathcal{C}[\![C_2]\!]] \\
\mathcal{E}[\![F \uparrow E]\!] & = & \mathcal{F}[\![F]\!] \circ \mathcal{E}[\![E]\!] & \mathcal{C}[\![C \downarrow F]\!] & = & \mathcal{C}[\![C]\!] \circ \mathcal{F}[\![F]\!] \\
\mathcal{E}[\![\ulcorner F \urcorner]\!] & = & (\mathcal{F}[\![F]\!] \circ \pi_2)^* & \mathcal{C}[\![\llcorner F \lrcorner]\!] & = & (\iota_2 \circ \mathcal{F}[\![F]\!])_* \\
\\
\mathcal{F}[\![p]\!] & = & p & \mathcal{F}[\![p]\!] & = & p \\
\mathcal{F}[\![X \Rightarrow E]\!] & = & \mathcal{E}[\![E]\!]^{\mathcal{X}[\![X]\!]} \circ \langle id, \diamond \rangle & \mathcal{F}[\![Y \Leftarrow C]\!] & = & [id, \Box] \circ \mathcal{C}[\![C]\!]_{\mathcal{Y}[\![Y]\!]} \\
\mathcal{F}[\![\overline{E}]\!] & = & ap \circ \langle \mathcal{E}[\![E]\!] \circ \diamond, id \rangle & \mathcal{F}[\![\underline{C}]\!] & = & [\Box \circ \mathcal{C}[\![C]\!], id] \circ pa \\
\\
f^{\mathcal{X}[\![x]\!]} & = & f^x & f_{\mathcal{Y}[\![y]\!]} & = & f_y \\
f^{\mathcal{X}[\![()]\!]} & = & f \circ \pi_2 & f_{\mathcal{Y}[\![\{\}]\!]} & = & \iota_2 \circ f \\
f^{\mathcal{X}[\![(X_1, X_2)]\!]} & = & (f^{\mathcal{X}[\![X_1]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha & f_{\mathcal{Y}[\![\{Y_1, Y_2\}]\!]} & = & \bar{\alpha} \circ (f_{\mathcal{Y}[\![Y_2]\!]})_{\mathcal{Y}[\![Y_1]\!]}
\end{array}
$$

Figure 3.5: Translation from SLC to SCL

Corollary 3.1 *For every morphism $e : 1 \to T$ containing a (value) variable $x : S$, there exists a unique morphism, $xe : S \to T$, free of x, s.t. $xe \circ x = e$. Similarly, for every morphism $c : S \to 0$, in a (continuation) variable $y : T$, there exists a unique y-free $yc : S \to T$, s.t. $y \circ yc = c$.*

Proof Set $xe = e^x \circ \langle id, \diamond \rangle$. Then $xe \circ x = e^x \circ \langle id, \diamond \rangle \circ x = e^x \circ \langle x \circ id_1, \diamond \rangle = e^x \circ \langle x \circ \diamond, id \rangle = e$. This gave us existence. For the uniqueness, assume that $f \circ x = e$, x not in f. Then $f = f \circ \pi_2 \circ \langle \pi_1, \pi_1 \rangle \circ \langle id, \diamond \rangle = f^x \circ \langle \pi_1, x^x \rangle \circ \langle id, \diamond \rangle = (f \circ x)^x \circ \langle id, \diamond \rangle = e^x \circ \langle id, \diamond \rangle = xe$.

Analogously, we let $yc = [id, \Box] \circ c_y$. ∎

This result makes it possible to give a full translation from SLC terms to morphisms in SCL, where every function $F : S \to T$, expression $E : T$ and continuation $C : S$ denotes a unique $f : S \to T$, $e : 1 \to T$ and $c : S \to 0$, respectively. The equations are given in figure 3.5. Since the functional completeness theorem was expressed in terms of a single variable, we must abstract pattern variables one at a time. In practice, the abstraction algorithm can easily be generalized to handle variable patterns directly.

3.5 SCL and the SLC conversion rules

Now let us see that SCL really models the SLC. We will state most definitions and theorems only for the value side; the continuation-based results are analogous.

Definition 3.3 *Let f be a categorical term, possibly containing the (value) variable $x : 1 \to T$, and let e by a value (*i.e.*, a total morphism $1 \to T$). We define $f[e/x]$ as the term f with all occurrences of x replaced by e. (There is no concept of free occurrence in SCL, because there are no explicit abstraction constructs).*

Lemma 3.9 *Substitution preserves equality in SCL,* i.e., *if $f, g : A \to B$ are two terms, s.t. $f = g$, then $f[e/x] = g[e/x]$ and $f[c/y] = g[c/y]$.*

Proof Structural induction on the proof of $f = g$. Because we are substituting a total (strict) morphism for x (y), the restrictions on the axioms are still fulfilled, and the proof goes through unmodified. ∎

We then see that the translation rules commute with SLC substitution:

Definition 3.4 *For any SLC term T, (value) variable x and expression E, we define $T[x := E]$ as the term where all free occurrences of x have been replaced by E (with implicit renaming of any captured variables).*

Lemma 3.10 *For any SLC term T, (value) variable x and total expression E, $\mathcal{T}[\![T[x := E]]\!] = \mathcal{T}[\mathcal{E}[\![\mathcal{E}]\!]/\S]$. Here, $\mathcal{T}$ denotes the appropriate translation function for T,* i.e., *$\mathcal{E}$, $\mathcal{C}$ or $\mathcal{F}$.*

Proof We can assume that any bound variables in T have already been renamed so as not to collide with variables in E. We proceed by structural induction on T; the only non-trivial cases are those directly involving value variables:

$$\begin{aligned}
\mathcal{E}[\![x[x := E]]\!] &= \mathcal{E}[\![E]\!] = \mathcal{E}[\![x]\!][\mathcal{E}[\![E]\!]/x] \\
\mathcal{E}[\![x'[x := E]]\!] &= \mathcal{E}[\![x']\!] = \mathcal{E}[\![x']\!][\mathcal{E}[\![E]\!]/x] \text{if } x \neq x' \\
\mathcal{F}[\![(X \Rightarrow E')[x := E]]\!] &= \mathcal{F}[\![X \Rightarrow E']\!] = \mathcal{E}[\![E']\!]^{\mathcal{X}[\![X]\!]} \circ \langle id, \Diamond \rangle \\
&= (\mathcal{E}[\![E']\!]^{\mathcal{X}[\![X]\!]} \circ \langle id, \Diamond \rangle)[\mathcal{E}[\![E]\!]/x] \\
&= \mathcal{F}[\![X \Rightarrow E']\!][\mathcal{E}[\![E]\!]/x] \text{ if } x \text{ in } X \\
\mathcal{F}[\![(X' \Rightarrow E')[x := E]]\!] &= \mathcal{F}[\![X' \Rightarrow E'[x := E]]\!] = \mathcal{E}[\![E'[x := E]]\!]^{\mathcal{X}[\![X']\!]} \circ \langle id, \Diamond \rangle \\
&= (\mathcal{E}[\![E']\!][\mathcal{E}[\![E]\!]/x])^{\mathcal{X}[\![X']\!]} \circ \langle id, \Diamond \rangle \\
&= (\mathcal{E}[\![E']\!]^{\mathcal{X}[\![X']\!]} \circ \langle id, \Diamond \rangle)[\mathcal{E}[\![E]\!]/x] \\
&= \mathcal{F}[\![X' \Rightarrow E']\!][\mathcal{E}[\![E]\!]/x] \text{ if } x \text{ not in } X'
\end{aligned}$$

□

Definition 3.5 *For a categorical term f, variable pattern X and value e, we define $f[e/X]$ (when X is not a single variable) inductively as:*

$$\begin{aligned}
f[e/()] &= f \\
f[e/(X_1, X_2)] &= f[(\pi_1 \circ e)/X_1][(\pi_2 \circ e)/X_2]
\end{aligned}$$

This allows us to state a more general form of the functional completeness theorem:

Theorem 3.2 *For every morphism, $f : A \to B$ and variable pattern $X : S$, the morphism $f^{\mathcal{X}[\![X]\!]} : S \times A \to B$ is the unique morphism not containing any variable from X with the property that $f^{\mathcal{X}[\![X]\!]} \circ \langle \mathcal{E}[\![X]\!] \circ \Diamond, id \rangle = f$. (Where $\mathcal{E}$ translates X to a SCL term built out of variables from X.)*

Proof Structural induction on X. The base case $X = x$ is precisely the functional completeness theorem, because $f^{\mathcal{X}[\![x]\!]} = f^x$. For the other two possibilities, we verify the existence part of the theorem:

$$\begin{aligned}
f^{\mathcal{X}[\![()]\!]} \circ \langle \mathcal{E}[\![()]\!] \circ \Diamond, id \rangle &= f \circ \pi_2 \circ \langle \Diamond, id \rangle = f \\
f^{\mathcal{X}[\![(X_1, X_2)]\!]} \circ \langle \mathcal{E}[\![(X_1, X_2)]\!] \circ \Diamond, id \rangle &= (f^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \circ \langle \langle \mathcal{E}[\![X_1]\!], \mathcal{E}[\![X_2]\!] \rangle \circ \Diamond, id \rangle \\
&= (f^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \circ \langle \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, \mathcal{E}[\![X_2]\!] \circ \Diamond \rangle, id \rangle \\
&= (f^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, id \rangle \circ \langle \mathcal{E}[\![X_2]\!] \circ \Diamond, id \rangle \\
&= f^{\mathcal{X}[\![X_2]\!]} \circ \langle \mathcal{E}[\![X_2]\!] \circ \Diamond, id \rangle = f
\end{aligned}$$

For uniqueness, we assume that $g \circ \langle \mathcal{E}[\![X]\!] \circ \Diamond, id \rangle = f$. Again, case analysis on X gives:

$$\begin{aligned}
f^{\mathcal{X}[\![()]\!]} &= (g \circ \langle \mathcal{E}[\![()]\!] \circ \Diamond, id \rangle)^{\mathcal{X}[\![()]\!]} = g \circ \langle \Diamond \circ \Diamond, id \rangle \circ \pi_2 = g \circ \langle \Diamond, id \rangle \circ \pi_2 \\
&= g \circ \langle \Diamond, \pi_2 \rangle = g \circ \langle \pi_1, \pi_2 \rangle = g \\
f^{\mathcal{X}[\![(X_1,X_2)]\!]} &= (g \circ \langle \mathcal{E}[\![(X_1, X_2)]\!] \circ \Diamond, id \rangle)^{\mathcal{X}[\![(X_1,X_2)]\!]} \\
&= ((g \circ \langle \langle \mathcal{E}[\![X_1]\!], \mathcal{E}[\![X_2]\!] \rangle \circ \Diamond, id \rangle)^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \\
&= ((g \circ \langle \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, \mathcal{E}[\![X_2]\!] \circ \Diamond \rangle, id \rangle)^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \\
&= ((g \circ \alpha^{-1} \circ \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, \langle \mathcal{E}[\![X_2]\!] \circ \Diamond, id \rangle \rangle)^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \\
&= ((g \circ \alpha^{-1} \circ \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, id \rangle \circ \langle \mathcal{E}[\![X_2]\!] \circ \Diamond, id \rangle)^{\mathcal{X}[\![X_2]\!]})^{\mathcal{X}[\![X_1]\!]} \circ \alpha \\
&= (g \circ \alpha^{-1} \circ \langle \mathcal{E}[\![X_1]\!] \circ \Diamond, id \rangle)^{\mathcal{X}[\![X_1]\!]} \circ \alpha \\
&= g \circ \alpha^{-1} \circ \alpha = g
\end{aligned}$$

□

We define generalized substitution in SLC as follows:

Definition 3.6 *For any SLC term T, (value) variable pattern X and expression E, with the same structure as X, we define $T[X := E]$ (when X is not a single variable) inductively as:*

$$\begin{aligned}
T[() := ()] &= T \\
T[(X_1, X_2) := (E_1, E_2)] &= T[X_1 := E_1][X_2 := E_2]
\end{aligned}$$

Lemma 3.11 *For any SLC term T, (value) variable pattern X and X-shaped total expression E, it holds that $\mathcal{T}[\![T[X := E]]\!] = \mathcal{T}[\![T]\!][\mathcal{E}[\![E]\!]/X]$.*

Proof Induction on the structure of X. We have already treated the base case $X = x$. For the remaining two possibilities, we check:

$$\begin{aligned}
\mathcal{T}[\![T[()/()]]\!] &= \mathcal{T}[\![T]\!] = \mathcal{T}[\![T]\!][\mathcal{E}[\![()]\!]/()] \\
\mathcal{T}[\![T[(E_1, E_2)/(X_1, X_2)]]\!] &= \mathcal{T}[\![T[X_1 := E_1][X_2 := E_2]]\!] = \mathcal{T}[\![T[X_1 := E_1]]\!][\mathcal{E}[\![E_2]\!]/X_2] \\
&= \mathcal{T}[\![T]\!][\mathcal{E}[\![E_1]\!]/X_1][\mathcal{E}[\![E_2]\!]/X_2] \\
&= \mathcal{T}[\![T]\!][(\pi_1 \circ \langle \mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!] \rangle)/X_1][(\pi_2 \circ \langle \mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!] \rangle)/X_2] \\
&= \mathcal{T}[\![T]\!][(\pi_1 \circ \mathcal{E}[\![(E_1, E_2)]\!])/X_1][(\pi_2 \circ \mathcal{E}[\![(E_1, E_2)]\!])/X_2] \\
&= \mathcal{T}[\![T]\!][\mathcal{E}[\![(E_1, E_2)]\!]/(X_1, X_2)]
\end{aligned}$$

□

We can now finally verify that SLC β-reduction is modeled properly in th SCL:

Theorem 3.3 *If the X-structured expression E_2 denotes a value,* i.e., *if $\mathcal{E}[\![E_2]\!]$ is total, then $\mathcal{E}[\![(X \Rightarrow E_1) \uparrow E_2]\!] = \mathcal{E}[\![E_1[X := E_2]]\!]$.*

Proof

$$\mathcal{E}[\![(X \Rightarrow E_1) \uparrow E_2]\!] = \mathcal{E}[\![E_1]\!]^{\mathcal{X}[\![X]\!]} \circ \langle id, \Diamond \rangle \circ \mathcal{E}[\![E_2]\!] = \mathcal{E}[\![E_1]\!]^{\mathcal{X}[\![X]\!]} \circ \langle \mathcal{E}[\![E_2]\!], \Diamond \rangle =$$

$$(\mathcal{E}[\![E_1]\!]^{\mathcal{X}[\![X]\!]} \circ \langle \mathcal{E}[\![X]\!], \Diamond \rangle)[\mathcal{E}[\![E_2]\!]/X] = \mathcal{E}[\![E_1]\!][\mathcal{E}[\![E_2]\!]/X] = \mathcal{E}[\![E_1[X := E2]]\!]$$

□

We can also check that η-conversion is preserved:

Theorem 3.4 *For any SLC function F, in which x does not occur free, $\mathcal{F}[\![x \Rightarrow F \uparrow x]\!] = \mathcal{F}[\![F]\!]$.*

Proof

$$\begin{aligned}
\mathcal{F}[\![x \Rightarrow F \uparrow x]\!] &= \mathcal{E}[\![F \uparrow x]\!]^x \circ \langle id, \Diamond \rangle = (\mathcal{F}[\![F]\!] \circ \mathcal{E}[\![x]\!])^x \circ \langle id, \Diamond \rangle \\
&= \mathcal{F}[\![F]\!]^x \circ \langle \pi_1, x^x \rangle \circ \langle id, \Diamond \rangle = \mathcal{F}[\![F]\!] \circ \pi_2 \circ \langle \pi_1, \pi_1 \rangle \circ \langle id, \Diamond \rangle \\
&= \mathcal{F}[\![F]\!] \circ \pi_1 \circ \langle id, \Diamond \rangle = \mathcal{F}[\![F]\!]
\end{aligned}$$

$[\![$

The identity and composition rules are also preserved, of course; this follows directly from the equations.

Among other things, these results justify our casual mixing of the two styles, using categorical morphisms and morphism constructors as SLC functions, or value and continuation abstractions as SCL morphisms. For instance, the mixed term $f \Rightarrow f^* : [A \times B \to C] \to [A \to [B \to C]]$ can be seen as abbreviating either the SLC function $f \Rightarrow a \Rightarrow b \Rightarrow f \uparrow (a, b)$ or the SCL morphism $((ap \circ \alpha)^*)^*$.

3.6 Evaluation strategies

Let us now return to the subject of evaluation strategies. In the SLC chapter, we saw how parameterless value abstractions could be used to implement lazy products in CBV, and parameterless continuation abstractions gave eager coproducts in CBN. We will now treat this more rigorously.

3.6.1 CBV and CBN in the SCL

Definition 3.7 *A* CBV *SCL category satisfies all the SCL axioms, and furthermore* 0 *is a (proper) initial object,* i.e., $\Box_A$ *is in fact the only morphism* $A \to 0$. *Similarly, a* CBN *SCL category is an SCL category, in which* 1 *is a (proper) terminal object,* i.e., $\Diamond_A$ *is unique.*

We immediately note that in a CBV strategy, all morphisms are strict, and in a CBN strategy, all are total (with the definitions of total and strict from definition 3.1). This means that a CBN SCL category is in fact a proper CCC, with all axioms holding generally, and the substitution theorem of last section applies to unrestricted β-reduction. In addition to the usual CCC types, the CBN category has a rather weak coproduct, which does not make it bicartesian closed (but neither is the category of other lazy languages with recursion).

On the other hand, and probably more surprising, all the continuation-based axioms hold without restriction in CBV. In particular, we have the full existence and uniqueness axioms for coexponentials. This demonstrates once more that even this strategy (in which δ is an isomorphism) is radically different from **Set**. In fact, it could be called co-Cartesian closed, as it has a proper initial object, binary coproducts and coexponentials, but not true products and exponentials.

A strategy specification added to the axioms of SCL is inherited by the SLC through the translations. Consider for example the SLC function $(x \Rightarrow 2) \circ (y \Leftarrow k)$, where k is a continuation constant or variable from an outer scope. For both definitions of composition in SLC, its translation is the morphism $2 \circ \Diamond \circ \Box \circ k$. In a CBN strategy, where 1 is a terminal object, this is equivalent to $2 \circ \Diamond$, while under CBV, where 0 is initial, it is the same as $\Box \circ k$.

Let us now see what the opposite-strategy products and eager coproducts of section 2.5 correspond to in SCL:

3.6.2 Lazy products in CBV

Borrowing some notation from domain theory, we define the *lifted* object:

$$A_\perp \equiv [1 \to A]$$

We have a natural morphism from from the lifted object back to the original:

$$ev_A \equiv a' \Rightarrow a' \uparrow () = ap \circ \langle id, \Diamond \rangle : A_\perp \to A$$

In CBN this is in fact an isomorphism. In CBV, however, $A_\perp$ contains more values, corresponding to all the non-total morphisms $1 \to A$.

Furthermore, for a morphism $f : A \to B$, we define the *lazy* version:

$$f_\perp \equiv x \Rightarrow () \Rightarrow f \uparrow x = (f \circ \pi_1)^* : A \to B_\perp$$

By the properties of $-^*$, $f_\perp$ is always total. As a special case, we get the inclusion of A into $A_\perp$ as $id_\perp$.

For every $f : A \to B$, we have:

$$ev_B \circ f_\perp = ap \circ \langle id, \Diamond \rangle \circ (f \circ \pi_1)^* = ap \circ \langle (f \circ \pi_1)^*, \Diamond \rangle = f \circ \pi_1 \circ \langle id, \Diamond \rangle = f$$

This makes possible a definition of lazy products, in terms of the old. For two objects A and B we define the object

$$A \otimes B \equiv A_\perp \times B_\perp$$

We have an unfortunate reversal of notation here, since $\otimes$ is usually used for the smash product. However, as the (eager) product $\times$ is more fundamental for CBV, it was given the simpler symbol.

We also define two morphisms

$$\boldsymbol{\pi}_1 \equiv ev_A \circ \pi_1 : A \otimes B \to A \qquad \boldsymbol{\pi}_2 \equiv ev_B \circ \pi_2 : A \otimes B \to B$$

and finally, for $f : C \to A$, $g : C \to B$:

$$\langle f, g \rangle \equiv \langle f_\perp, g_\perp \rangle : C \to A \otimes B$$

We see immediately that for *every* f, g:

$$\begin{aligned} \boldsymbol{\pi}_1 \circ \langle f, g \rangle &= ev_A \circ \pi_1 \circ \langle f_\perp, g_\perp \rangle = ev_A \circ f_\perp = f \\ \boldsymbol{\pi}_2 \circ \langle f, g \rangle &= ev_B \circ \pi_2 \circ \langle f_\perp, g_\perp \rangle = ev_B \circ g_\perp = g \end{aligned}$$

Unfortunately, we still do not have

$$\langle \boldsymbol{\pi}_1 \circ f, \boldsymbol{\pi}_2 \circ f \rangle = f$$

for non-total fs. Even worse, the two projections $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$ are no longer total. The functional completeness of the category relies heavily on that, because lemma 3.5 is used for abstracting variables out of function composition. With only lazy products, we cannot abstract a out of a function like $a \circ \Diamond \circ f$ (where f is not total), because there will be no way to evaluate f if its result is discarded. Also, $A \otimes 1$ is not isomorphic to A.

3.6.3 Eager coproducts in CBN

Let us now consider CBN, and eager coproducts. We recall that these are in fact the "usual" coproducts, which evaluate the argument to a case construct, until its tag is known.

Again, for an object we define the "lowered" one:

$$A^\top \equiv [0 \leftarrow A]$$

and a "co-evaluation" morphism

$$ve_A \equiv a' \Leftarrow \{\} \downarrow a' = [id, \Box] \circ pa : A \to A^\top$$

which is an isomorphism under CBV. We also define, for $f : A \to B$:

$$f^\top \equiv b \Leftarrow \{\} \Leftarrow b \downarrow f = (\iota_1 \circ f)_* : A^\top \to B$$

which is an always strict morphism, and

$$f^\top \circ ve_A = (\iota_1 \circ f)_* \circ [id, \Box] \circ pa = [(\iota_1 \circ f)_*, \Box] \circ pa = [id, \Box] \circ \iota_1 \circ f = f$$

We can now immediately write down the definition of eager coproducts:

$$A \oplus B \equiv A^\top + B^\top$$

$$\boldsymbol{\iota}_1 \equiv \iota_1 \circ ve_A : A \to A \oplus B \qquad \boldsymbol{\iota}_2 \equiv \iota_2 \circ ve_B : B \to A \oplus B$$

$$[f, g] \equiv [f^\top, g^\top] : A \oplus B \to C$$

With the desired properties:

$$[f, g] \circ \boldsymbol{\iota}_1 = f \qquad [f, g] \circ \boldsymbol{\iota}_2 = g$$

As we would expect, the new injections are not strict. Among other things, this means that we again lose functional completeness, as well as the isomorphism between A and $0 + A$. On the other hand, we can use eager coproducts to define a number of useful datatypes, *e.g.*, $bool = 1 \oplus 1$ and $nat = 1 \oplus nat$, and functions operating on them *within the language*, rather than as external primitives.

3.7 Iteration and recursion

Let us now consider iteration and recursion in a categorical setting.

It is known [Huwig & Poigné 86] that adding fixpoints to a sufficiently powerful category (*e.g.*, CCC with proper initial object) makes it inconsistent (*i.e.*, isomorphic to the trivial 1-object category). However, the axioms of SCL are weak enough so that this does not become a problem. For example, even though the CBN SCL is a full CCC, the restriction that $\Box$ need only be unique among *strict* morphisms prevents the construction of an isomorphism between 1 and 0.

3.7.1 Repetition operators

In the following, we will only consider pure CBV and CBN, for simplicity. The operator-based presentation is similar to [Poigné 83], but the notation is slightly different. In particular, we use a *subscript* '†' to denote iteration, for consistency with the other coproduct-based operators.

The principal idea is that a "repeatable" function $f : A \to A$, determines a recursive value $1 \to A$ in CBN and a recursive continuation $A \to 0$ in CBV. As with the functional vales and continuations we will, however, need a more general formulation to handle free variables. For a change, we will consider the coproduct-based case first, in CBV:

Let $f : A \to B + A$ be a morphism. Intuitively, f maps a value of type A to either a value of type A or one of type B. If the result is of type A, we can apply f to it again, *etc.* until we obtain a B-typed result. We call this iterated function $f_\dagger$. Note that if f always returns the second inject, $f_\dagger$ will never return. However, from the point where $f_\dagger$ is called this is expectable, and $f_\dagger$ is just another non-total function.

Formally, for every morphism $f : A \to B + A$ we require the existence of a morphism $f_\dagger : A \to B$, s.t.

$$f_\dagger = [id, f_\dagger] \circ f$$

$$((g + id) \circ f)_\dagger = g \circ f_\dagger$$

Note that the first equation in general has many solutions, *e.g.*, for $f = \iota_2 : A \to A + A$, $f_\dagger = id$ also satisfies it. The second equation severs to eliminate such solutions, by ensuring that $f_\dagger$ only terminates when f returns a first inject.

Dually, let us consider a CBN morphism $f : B \times A \to A$. It defines the output a in terms of both b and itself, so we seek a morphism $f^\dagger : B \to A$, defining a in terms of b only. Dualizing the axioms above, we obtain:

$$f^\dagger = f \circ \langle id, f^\dagger \rangle$$

$$(f \circ (g \times id))^\dagger = f^\dagger \circ g$$

3.7.2 Fixpoint morphisms

The operators of the previous subsection gave use a purely categorical view of iteration and recursion. However, we do not really want to introduce new morphism constructors, as we do not know how to abstract value and continuation variables out from them. Fortunately, that will not be necessary; instead, we can define single morphisms with the same effect, because of the (co-)cartesian closedness of the category. Let us consider the value-based (CBN) case first this time, as it is probably more familiar.

Let us assume that we have a recursion operator $-^\dagger$. We start by defining, from an instance of $ap : [A \to A] \times A \to A$,

$$fix \equiv ap^\dagger : [A \to A] \to A$$

For every function $f : B \times A \to A$, we also have its curried version $f^* : B \to [A \to A]$. This means that we can define

$$f^\ddagger \equiv fix \circ f^* : B \to A$$

and we immediately get (in CBN, all morphisms are total):

$$\begin{aligned} f^\ddagger &= fix \circ f^* = ap^\dagger \circ f^* = ap \circ \langle id, ap^\dagger \rangle \circ f^* = ap \circ \langle f^*, fix \circ f^* \rangle \\ &= f \circ \langle id, fix \circ f^* \rangle = f \circ \langle id, f^\ddagger \rangle \\ (f \circ (g \times id))^\ddagger &= fix \circ (f \circ (g \times id))^* = fix \circ f^* \circ g = f^\ddagger \circ g \end{aligned}$$

i.e., $f^{\ddagger\ddagger}$ satisfies the same axioms as $f^{\dagger}$, but $f^{\ddagger\ddagger}$ is expressed only in terms of existing morphism constructors and a single morphism fix, so that the existing abstraction rules apply, and the functional completeness theorem still holds.

Let us now return to the coproduct/iteration case for CBV and see how coexponentials can permit us to write iterative definitions with a single morphism. For a function $f : A \to B + A$, there is a (unique) cocurried version, $f_* : [A \leftarrow A] \to B$. This immediately suggests defining the morphism

$$xif \equiv pa_{\dagger}^{\dagger} : A \to [A \leftarrow A],$$

and from that, the iterated version of f:

$$f_{\ddagger}^{\ddagger} \equiv f_* \circ xif : A \to B$$

As for recursion, we immediately verify that $f_{\ddagger}^{\ddagger}$ satisfies the equations for $f_{\dagger}^{\dagger}$.

3.7.3 Back to values and continuations

As usual, we can interpret these definitions in the SLC as well, using the function fix to express recursive values, and xif for recursive continuations. Like in the SLC, we can define global values and continuations of any type as fixpoints of strict and total functions respectively:

$$\begin{array}{rcl} \bot_A & = & fix \uparrow id_A = fix \circ {\pi_2}^* = {\pi_2}^{\dagger}{}^{\dagger} : 1 \to A \\ \top_A & = & id_A \downarrow xif = \iota_{2*} \circ xif = \iota_2{}_{\dagger}^{\dagger} : A \to 0 \end{array}$$

We note that from the uniqueness of arrows $\Diamond$ and $\Box$ in CBN and CBV, respectively, we get:

$$\begin{array}{rcl} \bot_1 & = & \Diamond_1 = id_1 = \mathcal{E}[\![()]\!] \text{ in CBN} \\ \top_0 & = & \Box_0 = id_0 = \mathcal{C}[\![\{\}]\!] \text{ in CBV} \end{array}$$

This motivates the definitions of $\mathcal{E}[\![()]\!]$ and $\mathcal{C}[\![\{\}]\!]$ as runtime errors in CBN and CBV respectively. It also shows that the SCL definition of strictness is not in conflict with usual one by considering $\Diamond_A$ to be strict in CBN, because $(a \Rightarrow ()) \uparrow \bot = () = \bot$.

Finally, let us extend the equivalence between SLC and SCL to the recursive case, by giving the translation rules. We will consider both definitions of recursion in SCL, as it is instructive to see their translations. For the translation from SCL to SLC, we have:

$$\begin{array}{rcl} \mathcal{L}[\![fix]\!] & = & f \Rightarrow \overline{\mathbf{rec}}\ a = f \uparrow a \\ \mathcal{L}[\![f^{\dagger}]\!] & = & b \Rightarrow \overline{\mathbf{rec}}\ a = \mathcal{L}[\![f]\!] \uparrow (b, a) \end{array} \qquad \begin{array}{rcl} \mathcal{L}[\![xif]\!] & = & f \Leftarrow \underline{\mathbf{rec}}\ a = a \downarrow f \\ \mathcal{L}[\![f_{\dagger}]\!] & = & b \Leftarrow \underline{\mathbf{rec}}\ a = \{b, a\} \downarrow \mathcal{L}[\![f]\!] \end{array}$$

In the other direction, we get

$$\begin{array}{rcl} \mathcal{E}[\![\overline{\mathbf{rec}}\ X = E]\!] & = & \left(\mathcal{E}[\![E]\!]^{\mathcal{X}[\![X]\!]} \circ \gamma\right)^{\dagger} = fix \circ \left(\mathcal{E}[\![E]\!]^{\mathcal{X}[\![X]\!]} \circ \langle \pi_2, \Diamond \rangle\right)^* \\ \mathcal{C}[\![\underline{\mathbf{rec}}\ Y = C]\!] & = & \left(\bar{\gamma} \circ \mathcal{C}[\![C]\!]_{\mathcal{Y}[\![Y]\!]}\right)_{\dagger}^{\dagger} = \left([\iota_2, \Box] \circ \mathcal{C}[\![C]\!]_{\mathcal{Y}[\![Y]\!]}\right)_* \circ xif \end{array}$$

This also handles mutually recursive definitions. As mentioned, the second translation (in terms of fix/xif) is preferable, since we already know how to abstract value and continuation variables out from it.

3.8 A categorical denotational semantics

In this section, we present a CBV denotational semantics for categorical terms. It is basically a slightly modified version of the CBV SLC semantics of section 2.7.2, where the environment has been eliminated. They are indeed equivalent under the translations in section 3.4. The abstract syntax and type system is as in figure 3.1, while the semantic domains and equations are given in figure 3.6. We note that there is only one valuation function, for morphisms.

We can see this semantics as a model of a SCL category, where objects are types, and morphisms are equivalence classes of terms, factored by the equations.

It is easily checked that this semantics satisfies all of the axioms required for a SCL category. In fact, the continuation-based axioms (coproduct, *etc.*) hold in full generality, as all morphisms are strict.

We can also remark that there seems to be a close connection between this semantics and the the Kleisli category [Lambek & Scott 86,Moggi 89] of the triple (T, η, μ), where $T(A) = (A \to Ans) \to Ans$, $T(f) = \lambda tc.t(\lambda a.c(fa))$, $\eta(A) = \lambda ac.ca$, and $\mu(A) = \lambda pc.p(\lambda t.tc)$. In particular, the composition of two morphisms $f : A \to T(B)$ and $g : B \to T(C)$ is given by $g * f = \mu(C) \circ T(g) \circ f : A \to T(C)$, and $\eta(A) : A \to T(A)$ becomes the identity morphism in the Kleisli category.

It is known that the Kleisli category is the initial object in the category of resolutions of the triple. It seems that its terminal object, the Eilenberg-Moore category is strongly connected with CBN evaluation.

$$
\begin{array}{rcl}
Val & = & Bas + Unit() + Pair(Val \times Val) + In_1(Val) + In_2(Val) + \\
 & & Closr(Val \to Cnt \to Ans) + Contx(Val \times Cnt) \\
Cnt & = & Val \to Ans
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M} : M & \to & Val \to Cnt \to Ans \\
\mathcal{M}[\![x]\!]v\kappa & = & \text{let } val(a) = \rho_{init}\ x \text{ in } \kappa\ a \\
\mathcal{M}[\![y]\!]v\kappa & = & \text{let } cnt(c) = \rho_{init}\ y \text{ in } c\ v \\
\mathcal{M}[\![p]\!]v\kappa & = & \kappa(pv) \\
 & & \\
\mathcal{M}[\![id]\!]v\kappa & = & \kappa\ v \\
\mathcal{M}[\![f \circ g]\!]v\kappa & = & \mathcal{M}[\![g]\!]v(\lambda t.\mathcal{M}[\![f]\!]t\kappa) \\
 & & \\
\mathcal{M}[\![\diamond]\!]v\kappa & = & \kappa\ unit() \\
\mathcal{M}[\![\langle f, g\rangle]\!]v\kappa & = & \mathcal{M}[\![f]\!]v(\lambda s.\mathcal{M}[\![g]\!]v(\lambda t.\kappa\ pair(s,t))) \\
\mathcal{M}[\![\pi_1]\!]v\kappa & = & \text{let } pair(v_1, v_2) = v \text{ in } \kappa\ v_1 \\
\mathcal{M}[\![\pi_2]\!]v\kappa & = & \text{let } pair(v_1, v_2) = v \text{ in } \kappa\ v_2 \\
 & & \\
\mathcal{M}[\![\Box]\!]v\kappa & = & \text{case } v \text{ of esac} \\
\mathcal{M}[\![[f, g]]\!]v\kappa & = & \text{case } v \text{ of } in_1(t) : \mathcal{M}[\![f]\!]t\kappa \ [\!]\ in_2(t) : \mathcal{M}[\![g]\!]t\kappa \text{ esac} \\
\mathcal{M}[\![\iota_1]\!]v\kappa & = & \kappa\ in_1(v) \\
\mathcal{M}[\![\iota_2]\!]v\kappa & = & \kappa\ in_2(v) \\
 & & \\
\mathcal{M}[\![f^*]\!]v\kappa & = & \kappa\ closr(\lambda bc.\mathcal{M}[\![f]\!]pair(v,b)\ c) \\
\mathcal{M}[\![ap]\!]v\kappa & = & \text{let } pair(g, a) = v \text{ in let } closr(f) = g \text{ in } fa\kappa \\
\mathcal{M}[\![\phi]\!]v\kappa & = & \text{let } pair(x, q) = v \text{ in let } contx(a, c) = q \text{ in } \kappa\ contx(pair(x, a), c) \\
 & & \\
\mathcal{M}[\![f_*]\!]v\kappa & = & \text{let } contx(a, c) = v \text{ in } \mathcal{M}[\![f]\!]a(\lambda t.\text{case } t \text{ of } in_1(r) : \kappa r \ [\!]\ in_2(s) : cs \text{ esac}) \\
\mathcal{M}[\![pa]\!]v\kappa & = & \kappa\ in_1(contx(v, \lambda t.\kappa\ in_2(t))) \\
\mathcal{M}[\![\theta]\!]v\kappa & = & \text{let } closr(f) = v \text{ in} \\
 & & \quad \kappa\ in_2(closr(\lambda ac.fa(\lambda t.\text{case } t \text{ of } in_1(r) : \kappa\ t \ [\!]\ in_2(s) : cs \text{ esac}))) \\
 & & \\
\mathcal{M}[\![xif]\!]v\kappa & = & (fix\lambda q.\lambda a.\kappa\ contx(a, q))v
\end{array}
$$

Figure 3.6: A CBV semantics of SCL

Chapter 4

Applications and Examples

In this chapter, we will consider some applications of the presented framework for reasoning categorically about continuations. The results seem to fall in two classes. First, there is a number applications of the SLC/SCL itself, as a method of expressing and describing various programming constructs. Secondly, and perhaps even more importantly, we may consider the broader implications of the value/continuation symmetry in suggesting entirely new concepts and constructs as duals of existing ones.

We begin by a section about extending the SLC towards a "real" programming language that could be used for practical applications. After that, we will compare some existing language concepts with the SLC ones. We will also give a number of small programming examples. They come rather late, because we will use some notational conventions presented in the earlier sections. Also, since we will compare them to similar programs in other languages, we must introduce the relevant facilities of those first. Finally, we compare the SLC/SCL approach to related work, and suggest directions for further research.

4.1 Towards a Symmetric Programming Language

The SLC as presented is quite terse, and unfit for any larger programming task, just as no-one would program seriously in the pure λ-calculus. In this section, we will consider some extensions towards a real programming language with a more readable ML-like syntax, but keeping the symmetrical features. We can see most of the notation introduced here as simply syntactic sugar for the underlying SLC concepts.

Instead of building on the SLC, we could rather try to extend the SCL towards a more practical combinator-based language, similar to APL [Iverson 62], FP [Backus 78], or CPL [Hagino 87a]. Indeed, the SCL constructs are often more concise and clear than the corresponding SLC terms, and better suited for formal reasoning. However, experience has shown that it is usually easier to understand larger SLC terms than SCL ones, particularly when higher-order functions are involved, because variable names are easier to follow over long distances than nested combinators.

As the extensions to SLC are needed precisely to handle larger programs, it seems that a completely variable-free approach would be impractical. Therefore, while we will certainly keep the SCL combinators, and introduce new ones for common patterns, the goal will not be to eliminate all uses of the SLC value and continuation abstractions, and they would still remain part of the language core.

4.1.1 Notation

This is a theoretically trivial yet practically important problem. To write SLC/SCL terms in a normal ASCII-based system, we will need a few conventions. Some of these considerations also relate to the model implementation of SLC in Appendix A.

The only SLC characters not commonly available are $\uparrow$ and $\downarrow$. There are no symmetric characters left, so we can pick two basically at random; the model implementation uses `^` and `?`. Perhaps `_` would be a better choice than `?`, but it is usually used in variable names for readability. The symbols $\Rightarrow$ and $\Leftarrow$ can be directly replaced by `=>` and `<=`, but it may be preferable to use a less concise notation for abstractions, like ML's `fn x=>`... to simplify parsing.

For SCL terms, with the many special symbols, the situation is worse. The "pair" and "case" constructs are so common that it would probably be reasonable to use the symbols `<, >, [ and ]` to express them. This extends perfectly to the morphisms associated with initial and terminal morphisms, where we can use `<>` and `[]` for $\diamond$ and $\Box$ respectively. On the other hand, currying and recursion are so relatively uncommon, that it seems most natural to introduce them as keywords, together with suitable names for the various "greek letters" used for projections *etc.*

4.1.2 Labelled datatypes

Perhaps the worst readability problems of the SLC come from having only unlabelled products and coproducts, so that unrelated pieces of data are hard to distinguish. Also, for practical purposes, having only binary and nullary products/coproducts is very inconvenient. We can easily introduce arbitrary, labelled products (records with fields):

$$\textbf{type}\ rr = (f_1 : T_1, \ldots)$$

and variants as

$$\textbf{type}\ vv = \{v_1 : T_1, \ldots\}$$

We could build these types out of the binary primitives of SLC, but it is probably more natural to add arbitrary finite products and coproducts directly to it. The labels are just a notational convenience, which may have a influence of the type checking, but not execution.

Expressions of type rr are written $(f_1 = E_1, \ldots)$, and continuations of type vv as $\{v_1 = C_1, \ldots\}$. The same syntax is used for patterns: $(f_1 = x_1, \ldots) \Rightarrow E$ and $\{v_1 = y_1, \ldots\} \Leftarrow C$. Furthermore, we can allow the field/variant tags to be used as projections/injections, writing simply f_i as an abbreviation for the value abstraction $(\ldots, f_i = x_i, \ldots) \Rightarrow x_i$ and v_i instead of $\{\ldots, v_i = y_i, \ldots\} \Leftarrow y_i$. We will call these functions *constructors* for coproducts, and *destructors* for products.

As constructors (injections) are always total, we can further abbreviate *unit*-typed variants, by treating them as constants rather than *unit*-accepting functions and writing

$$\textbf{type}\ color = \{red, green, blue\}$$

instead of $\{red : unit, green : unit, blue : unit\}$. Of course, such a definition only makes sense for eager coproducts. Similarly, because destructors (projections) are always strict, we can abbreviate null-typed fields in records by omitting the '$: null$' in declarations. Note that while a record with a *null*-typed field is isomorphic to just *null* for strict products, the two are different for lazy products, as found in CBN.

This labelled syntax extends naturally to recursive types, where the type being defined can itself appear in the defining term. The injections/projections would then become isomorphisms between the type and its defining type expression, as for `datatype` in ML and `codatatype` in [Hagino 87b].

Of course, we can mix labelled and unlabelled types as convenient. For example, we could define the type of binary trees with integer leaves as follows:

$$\textbf{type}\ \mathit{inttree} = \{\mathit{empty} : (), \mathit{leaf} : \mathit{int}, \mathit{node} : \mathit{inttree} \times \mathit{inttree}\}$$

Another possibility would be to allow only labelled types, but permit positional association as in Ada, *e.g.*

$$\textbf{type}\ \mathit{triple} = (a : \mathit{int}, b : \mathit{bool}, c : \mathit{int})$$

with the following expressions all denoting the same value:

$$(c = 2, b = \mathit{true}, a = 1) \qquad (1, c = 2, b = \mathit{true}) \qquad (1, \mathit{true}, 2)$$

Finally, we can allow parameterized, polymorphic types such as Tree(A), in a way analogous to ML.

4.1.3 Binding primitives

Perhaps the most obvious binding extension is the **let**-expression. We define it as follows:

$$\textbf{let}\ X = E_1\ \textbf{in}\ E_2 \ \equiv\ (X \Rightarrow E_2) \uparrow E_1$$

This also handles "destructuring", *e.g.*, $\textbf{let}\ (a, b) = E\ \textbf{in}\ a + b$. With respect to a polymorphic type inference system, the above definition may be too restrictive (cf.,*e.g.*, [Peyton Jones 87, sect. 7]), but that is not a problem specific to the SLC. Of course, we should provide an analogous definition facility for continuations:

$$\textbf{colet}\ Y = C_1\ \textbf{in}\ C_2 \ \equiv\ C_1 \downarrow (Y \Leftarrow C_2)$$

Simultaneous definitions are no problem:

$$\textbf{let}\ X_1 = E_1, X_2 = E_2\ \textbf{in}\ E_3 \ \equiv\ \textbf{let}\ (X_1, X_2) = (E_1, E_2)\ \textbf{in}\ E_3$$

Similarly, we can introduce recursive definitions:

$$\textbf{let rec}\ X = E_1\ \textbf{in}\ E_2 \ \equiv\ \textbf{let}\ X = (\overline{\textbf{rec}}\ X = E_1)\ \textbf{in}\ E_2$$

and implicit abstractions:

$$\textbf{let}\ f \uparrow X = E_1\ \textbf{in}\ E_2 \ \equiv\ \textbf{let}\ f = X \Rightarrow E_1\ \textbf{in}\ E_2$$

Again, the same extensions apply to **colet**.

A common pattern consists of applying a continuation abstraction to a value or vice versa. For this, we can use the syntax:

$$E\ \textbf{as}\ C\ \textbf{into}\ Y \ \equiv\ (Y \Leftarrow C) \uparrow E$$

$$C\ \textbf{as}\ E\ \textbf{from}\ X \ \equiv\ C \downarrow (X \Rightarrow E)$$

e.g., $E\ \textbf{as}\ a\ \textbf{into}\ \{a, b\}$ injects E as first inject, while $C\ \textbf{as}\ (a, a)\ \textbf{from}\ a$ denotes the continuation which will pass two copies of its input to C. It is a matter of taste, however, whether this is more readable than the SLC arrows.

4.1.4 Control primitives

For (eager) coproducts, we can define the familiar **case** expression:

$$\textbf{case } E \textbf{ of } (\iota_1 \uparrow X_1) \Rightarrow E_1 \mid (\iota_2 \uparrow X_2) \Rightarrow E_2 \textbf{ esac } \equiv [X_1 \Rightarrow E_1, X_2 \Rightarrow E_2] \uparrow E$$

Of course, with the labelled coproducts we can use any constructor for selection, not only ι_1 and ι_2. If we expand the SCL notation $[-,-]$ into SLC, we could also express it with the binding constructs of the previous section:

$$E \textbf{ as } \{r \textbf{ as } E_1 \textbf{ from } X_1, r \textbf{ as } E_2 \textbf{ from } X_2\} \textbf{ into } r$$

Similarly, for lazy products, we define the *continuation*:

$$\textbf{pair } C \textbf{ of } \{Y_1 \downarrow \pi_1\} \Leftarrow C_1 \;\&\; \{Y_2 \downarrow \pi_2\} \Leftarrow C_2 \textbf{ riap } \equiv C \downarrow \langle Y_1 \Leftarrow C_1, Y_2 \Leftarrow C_2 \rangle$$

This dispatches according to the element of the product that C wants to evaluate. Again, the extension to labelled products is immediate.

It seems natural to extend these constructs to general pattern-matching, as found in ML *etc.*, but we must probably require patterns to be both exhaustive and non-overlapping, in order to give a direct translation into the SLC.

4.1.5 Top-level evaluation

However symmetrical the language may be, it will always have the fundamental asymmetry that we are evaluating top-level expressions. In this section, we will consider some practical extensions that do not correspond to any SLC concept, but are probably essential for a practical language. For example, we might include the *command*

$$\textbf{def } X = E$$

With the semantics that E is evaluated in the initial environment, and the variables in X bound to the result. Recursive top-level definitions are easily added:

$$\textbf{def rec } X = E \;\equiv\; \textbf{def } X = (\textbf{rec } X = E)$$

Note that this works even in a CBV language, where recursive functions are defined as recursive continuations, because of the coercions.

We can permit the pattern-matching syntax in function definitions as well, so that we could express the functions $h = [f, g]$ and $k = \langle f, g \rangle$ like this (using the pure ASCII syntax for a change):

```
def h^(in1^x) = f^x        def (y?pr1)?k = y?f
 |  h^(in2^x) = g^x         &  (y?pr2)?k = y?g
```

4.1.6 Asymmetric extensions

Finally, we can include a number of conventional, asymmetric constructs which do not dualize nicely, s.a. conditionals. Also, some data types, *e.g.*, lists and Booleans are more commonly used than their duals, and it may be worth defining a special syntax for them.

We define the type $Bool$ as semantically equivalent to $unit + unit$ ($unit \oplus unit$ for CBN). With the labelling of section 4.1.2, we would write this as simply

$$\textbf{type } Bool = \{true, false\}$$

The conditional expression then becomes a special case of **case**:

$$\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \equiv \textbf{case } E_1 \textbf{ of } true \Rightarrow E_2 \mid false \Rightarrow E_3 \textbf{ esac}$$

We can also define an operator **or** : $Bool \times Bool \to Bool$, which does not evaluate its second argument if the first is true as follows:

$$p \textbf{ or } q \equiv (\{t, f\} \Leftarrow \{t, \{t, f\} \downarrow (() \Rightarrow q)\}) \uparrow p$$

This definition makes explicit the flow of control: the Bool-expecting result continuation is captured as $\{t, f\}$. Then the first operand p is evaluated with a success continuation of t (*i.e.*, if p is true, return immediately with true), and a failure continuation of $\{t, f\} \downarrow (() \Rightarrow q)$. This continuation discards the *unit*-typed false result of p, and evaluates q with the original continuation $\{t, f\}$, *i.e.*, q's result will become the result of p **or** q. Let us also give the definition of **and** for reference:

$$p \textbf{ and } q \equiv (\{t, f\} \Leftarrow \{\{t, f\} \downarrow (() \Rightarrow q), f\}) \uparrow p$$

Of course, this is not the "dual definition" in the SLC sense. We recall that **and** and **or** were duals in the category of propositions. In the SLC, however, they are both *Bool*-returning operators, and we get the kind of symmetry exhibited above, not the arrow-reversal kind.

The main reason for introducing booleans and **if** in the first place is their relation to toposes, where the *subobject classifier* Ω plays a role very similar to $Bool$, and under the right conditions it is even the case that $\Omega = 1+1$. While toposes and their connections to type theory appear very related to programming language design, they do not seem to agree very well with the symmetrical world of SLC/SCL. It would therefore be interesting to see if there exists a more symmetrical approach to the fundamental concept of equality than the "comparison operator" $(=) : A \times A \to Bool$, based on *e.g.*, equalizers and coequalizers.

4.1.7 Restrictions

Despite the goal of symmetry, it is not obvious whether having the full power of the SLC is really desirable in a practical language. For example, the "lazy coproducts" of pure CBN, while theoretically analogous to the ordinary "eager products" of CBV, are quite counterintuitive. It may therefore be reasonable to have only the "eager coproducts" in the subject language, even at the price of losing functional completeness. A number of other concepts like the "inverse fixpoint" xif or even the whole idea of coexponentials may also be too weird to include directly.

Nevertheless, the SLC is a valuable starting point for the task, by pointing out that continuations should play a fundamental role in the language definition, and how any restrictions on their use will cut off symmetrical pieces of the language. Also, if a language includes both forms of a construct, *e.g.*, products and coproducts, they should have symmetrical or at least clearly related syntax, and concepts such as modularity or inheritance should apply to both.

4.2 SLC/SCL as metalanguages

In this section we will see how some "imperative" constructs in other programming languages could be expressed in the SLC, and conversely, how various SLC primitive concepts are latent in many existing languages. Oddly enough, some rather old languages appear more symmetrical than the newer ones. In fact, the "modern functional languages" are perhaps the most asymmetrical.

4.2.1 The 'goto'

The 'goto' statement is probably the single most criticized programming language construct [Dijkstra 68], and not without cause. Nevertheless, in the rush to eliminate it wherever possible, a number of more innocent applications seem to have suffered as well. In this subsection, we will consider jumps and related constructs from the SLC viewpoint, and try to separate their "imperative" aspects from the basically declarative ones. However, as control flow in many languages is command-based rather than expression-based, and thus involving centrally the concept of a store, the correspondence will not be perfect. We will also mention the SLC view of "tail calls", as this concept is often intuitively connected with jumps.

First, let us note that it is useful to distinguish between two kinds of goto: *forwards* and *backwards*. Basically, a forwards goto is one which does not involve any looping, *i.e.*, while the destination need not appear textually after the jump, sequential execution should not reach the jumping instruction again. While backwards goto has been adequately superseded by structured constructs, such as 'while', no single construct seems to capture the full essence of a forwards goto. We shall consider forwards goto first, as it is conceptually simpler; we will return to backwards goto towards the end of this subsection.

While 'goto' is the prototypical jumping construct, many languages also include a number of "semi-structured" escaping primitives, usually with names like 'exit', 'return' or 'break'. We will refer to all of these collectively as simply jumps. The idea of a jump is to effect a transfer of control away from the sequence of statements it appears in. Basically, this consists of discarding the current continuation, and substituting a new one, and this operation can be expressed directly with a SLC continuation abstraction.

It is very important to keep in mind, however, that SLC continuations in themselves do not embody any concept of discarding, because they are inherently one-sided, just like values: we do not usually treat values as functions ignoring any other value passed to it. When we do want this behavior, we write it explicitly as $x \Rightarrow v$. Similarly, when we really want to discard a continuation and install another, we write $y \Leftarrow k$, which is not really more "imperative" than throwing away a value. It is the *implicit* discarding of the old continuation that makes up the imperative essence of jumps, not the installation of the new one, and we shall return to this in the next subsection.

The implicit *merging* operation of jumps is also emphasized by the SLC. We can reach a label either "normally", by executing the sequence of statements preceding it textually, or "exceptionally" by jumping to it from somewhere else. Similarly, we can exit from a loop or a procedure by simply reaching the bottom of it sequentially (and fulfilling the exit condition for a loop), or by an escaping construct. At the destination of the jump, it is usually not apparent in which of the two ways control reached it.

The SLC expresses this "untagged duplication" of the continuation as $a \Leftarrow \{a, a\} : A + A \to A$, which explicitly discards the information about how control reached a given point. Often, the above is used in the style $l \Leftarrow l \downarrow (\ldots (c \Leftarrow l) \ldots)$. This corresponds precisely to the the normal/abnormal entry to a label. Note that the SLC expression would have been a $\bar{\eta}$-redex if the l did not also occur in the body. Such a redex mirrors the eliminable sequence 'goto l; l:...', where l is not used anywhere else.

Even the first-order SLC is a fundamentally block-oriented language. Referring to non-local continuations corresponds to non-local gotos in block-structured languages (*e.g.* Algol 60 [Naur 62], Pascal [Jensen & Wirth 74] or Ada [ANSI 83]). The implementation of non-local gotos is usually quite simple when a system for handling non-local values (*e.g.*, a display or static links) exists. This is also true when functions and procedures can be used as "downward funargs", *i.e.*, as formal

parameters.

On the other hand, "upward funargs" (returned functions) give the same problems for values as for continuations, namely that a stack-allocated activation record ("context") referred to by a non-local variable may no longer exist if a name can be returned out of its scope. We will return to this in section 4.2.5.

Tail(-recursive) calls are sometimes referred to as "goto with value" This is a little misleading, because nothing is discarded as with a traditional goto. Rather, it is a suppression of adding an identity operation to the representation of the continuation, *i.e.*, the control stack or tree.

What may cause this confusion is that in assembly language, the "declarative" instruction sequence 'call p; return' can be abbreviated to the "imperative" 'jump p'. We must remember, however, that assembly language is inherently a higher-order framework because return addresses *etc.* are treated as values. In the CBV evaluation of traditional architectures, a function is in fact naturally seen as a *continuation* accepting pairs (argument value, return address). Such a continuation cannot itself "return", except possibly by executing a 'halt' instruction; it can only transfer control to another continuation, *e.g.*, the one designated by the context it received.

In this framework, tail calls correspond to passing the received context (or at least its continuation part) on to the callee, rather than building a new one with the same effect. Precisely because the model of functions is a low-level one, such an optimization is completely declarative, while a 'goto' in an otherwise functional programming language would not be.

Perhaps the most serious problem with the original 'goto' is its close connection to repetition. Every label declaration involves a fixpoint, because it may be referred to from anywhere in the statement sequence it appears in, not only from the preceding part. This encourages "spaghetti" code where loops are not properly nested. Often, this is only the result of sloppy programming, but sometimes the basic structure of the algorithm is really best expressed as states of an automaton.

As is well known, such pieces of code can be rewritten into a single 'while' and a 'case', with the original 'goto' labels becoming case selectors. This replaces the many intermixed fixpoints with a single one and a coproduct-based construct (the 'case' statement'), which corresponds exactly to the "mutually recursive continuations" of SLC, *i.e.*, the construct $\underline{\mathbf{rec}}\ \{c_1, c_2, \ldots\} = \{\ldots\}$.

4.2.2 Jumps and the type system

Traditionally, jumping primitives are considered part of the control structure of a programming language, and independent of its typing aspects. However, the close coupling of continuations with the type system in the SLC framework makes it possible to reason precisely about the type of such constructs. The following discussion also applies to languages that are not strongly typed, *e.g.*, Scheme [Rees & Clinger 86], in the same way that considerations about the types of values are meaningful even if the language does not (or can not) enforce them statically. We will finish this subsection by a moderate (at least as compared to the first-class continuations) suggestion for new language designs.

In the SLC, all computation patterns are abstracted as functions, which can be either value or continuation abstractions. Naturally, jumping constructs are defined as continuation abstractions. Any function has two types associated with it, a domain (input) and a codomain (output). We will consider the input type of jumping constructs first, as it is simpler. It is also usually dealt with properly by typed languages.

The input type of a jump has to do with information transfer from the source of the jump to its destination. Most jumping constructs in command-based languages, s.a. Algol cannot pass any data directly during a jump, *i.e.*, the domain the domain or source type of 'goto' is *unit*. Any data

that should be transferred must be stored in an auxiliary variable. For the same reason, labels appear on statements, not expressions.

However, even command-based languages can have information-transferring jumps. The most obvious example is the 'return' statement in *e.g.*, Ada or C. This transfers control to the end of the innermost enclosing function body. In a (non-void) function, it takes an argument which will be returned to the point of call as the result of the function.

Also, the `setjmp/longjmp` facility in C can transfer a value. Remarkably, it even enforces a distinction between the normal and exceptional returns from `setjmp`, in that the former always returns zero, and the latter a non-zero value.

On the other hand, jumps in expression-based languages usually encourage a transfer of information, like the **resultof** operator of the introduction. Other examples are ML's exceptions and Scheme's `call/cc`, but we will return to those in the next sections. Some languages also have a "pseudo-tail" call, which calls a function (with parameters) "as if" it was the last one in the body, *e.g.*, 'chain' in Pop-11. This discards the rest of the function body, by invoking the called function with the return continuation of the calling, which may not be the same as the continuation at the point of call.

The output type of jumping constructs is more problematic. What makes them appear imperative is that most languages misrepresent their type, so that the jumping behavior is not made explicit.

This is most apparent when the jumping construct is presented as a function or procedure. For example, in C, some non-local exits (`exit`, `longjmp`) look just like function calls. Also some Pascal implementations, to avoid introducing new reserved words, present escapes like 'return' as standard procedures. However, the type problems also affects command-based jumps like 'goto', because the same syntax is used for both "sequential" statements, such as assignments and procedure calls, and "escaping" ones, that discard the current continuation. This makes the latter look imperative, because they behave differently.

For example, in most languages it is perfectly correct to write a sequence like 'goto lab; a := 1', where the statement 'a := 1' is unreachable. However, such code usually represents a mistake, and in fact the C language style checker 'lint' will give a warning for it.

In the SLC formalism, we can view such a usage of 'goto' as a *type* error. First, note that we can treat a statement as a function $unit \to unit$, with ';' acting as function composition in reverse order ($f;g \equiv g \circ f$). This also applies to procedure calls with arguments: the arguments are expressions (*i.e.*, functions $unit \to T$), and application corresponds to composition, so the entire statement is still of type $unit \to unit$, although its components may not be. In fact, 'lint' will even complain about a call to a non-void returning function, *e.g.*, `printf`, appearing in a statement list. To explicitly discard the returned status value, it must be called as `(void)printf(...)`.

Let us now see how 'goto' fits in. If we see it as a function that does not (and can not) return, it should have type $unit \to null$. This is different than a function that is expected to return but fails to do so, because we need not provide a return address for it. The type of 'goto' makes it composable on the left, *i.e.*, 'S; goto L' is also a function $unit \to null$; but 'goto L; S' is a type error because we feed the *null*-typed output of 'goto' to the unit-expecting 'S'.

To make the composition type-correct we must insert a function $0 \to 1$ between the two. There exist at least two such functions: $\diamond_0$ and $\square_1$. The former corresponds to a 'cast to void' in C terminology, and this is done implicitly by most languages, so that 'goto' appears as a normal statement. The latter solution is expressed as the 'lint' command `/*NOTREACHED*/`, which changes a *unit*-accepting sequence into a *zero*-accepting one.

A conclusion is: programming languages should recognize the concept of a *null* type with *no* values. This would not have to be explicit, *e.g.*, a distinction like Pascal's function/procedure would be acceptable, but for function-based languages s.a. C or ML, a special, named type might be most natural.

Functions that did not return, *e.g.*, the `exec` system call in Unix could then be properly declared as 'null'-returning, rather than the inaccurate 'unit' or 'void'. Apart from aiding a reader of the code by specifying the behavior, this would also help the compiler detect problems, s.a. unreachable statements following a `longjmp`. Also, it would make propagation of jumping behavior automatic, *i.e.*, if one function calls another that does not return, neither can the first, and its type should express that.

A *null* type also has an implementational significance: for *unit*-accepting functions, no input value is actually passed, because it would not carry any information. Similarly, a *null*-returning function needs no return address, *i.e.*, would always be entered by a jump. Apart from being more efficient, this also makes it possible to express *in the type system* that a group of functions should call each other only tail-recursively, *i.e.*, with no stack growth, and have the compiler detect violations of this pattern automatically.

4.2.3 Multiple returns and label types

A fundamental primitive of the SLC is the "continuation pattern" syntax for continuation abstractions used, *e.g.*, in the definition of injections: $\{a, b\} \Leftarrow a$. This selection construct is present in a number of other languages as "alternate returns" from a function or procedure.

For example, a Fortran 77 [ANSI 78] subroutine can be called with a list of line labels as extra arguments. In the body of the called subroutine an alternate return is then invoked with `RETURN` n, where n is the number of the alternate exit. The purpose is, of course, to abbreviate call/return patterns like this:

```
      ...
      CALL P(..., ISTAT)
      GOTO (100,200) ISTAT
C     normal return here
      ...
      SUBROUTINE P(..., ISTAT)
      ...
      ISTAT = 2
      RETURN
      ...
```

into simply

```
      ...
      CALL P(..., *100, *200)
C     normal return here
      ...
      SUBROUTINE P(...)
      ...
      RETURN 2
      ...
```

This mechanism is also used to handle errors in other statements, *e.g.*

```
      READ(..., END=100, ERR=200) ...
```

will jump to line 100 on end of file and line 200 on other errors.

A similar feature can be found in Algol 60, where labels can appear as explicitly named procedure parameters:

procedure *p*(..., *l1*, *l2*); **label** *l1*,*l2*;
begin
  ...**goto** *l1*; ...
end *p*;

If *p* is later invoked like this:

begin
  ...*p*(..., *lab1*, *lab2*); ...
  ...*lab1*: ...
end

then the **goto** *l1* in *p*'s body will transfer control to the label *lab1*. A "value" of a label type is very similar to a SLC continuation, except that labels can only be *unit*-typed. In most later languages, label types were dropped, probably partly because their effect can easily be simulated through procedural parameters:

procedure *p*(..., *pl1*, *pl2*); **procedure** *pl1*, *pl2*;
begin
  ...*pl1*; ...
end *p*;
⋮
begin
  procedure *golab1*; **begin goto** *lab1* **end**;
  ...
  p(..., *golab1*, *golab2*);
  ...
  lab1: ...
end

In SLC terms, this is possible precisely because we can treat unit-typed continuations as functions $1 \to 0$. Again, as Algol does not have a 'null' type, the function must be declared as a 'proper procedure' (*i.e.*, unit-returning), which obscures its nature: the call to *pl1* in *p*'s body looks just as a procedure call, but is really a **goto** in disguise.

Another characteristic of multiple exits in existing languages is that one is singled out as a "normal" return, with the others being "special". The SLC can express this through cocurried functions: $a \Leftarrow b \Leftarrow \ldots$ where the b return is 'ordinary', while the a becomes a non-local exit, *cf.* the *elog* example of section 2.4.2. Such a distinction may be natural in some circumstances, where the nature of the two exits is very different. However, just as a function accepting a coordinate pair would not usually be curried, there are functions where the two exits are very closely related. In these cases, the symmetric syntax $\{a, b\} \Leftarrow \ldots$ is preferable.

4.2.4 Exceptions

Now, let us consider a rather different approach to escaping. A number of programming languages (*e.g.*, Ada, SML) include *exceptions* as a "structured" form of the goto. This is not a very accurate

view, for as we shall see, exceptions are essentially a dynamically-scoped feature in an otherwise statically statically-scoped language. Symmetry suggests that a language with dynamic continuations should also have dynamic values, which are generally regarded a problematical concept.

We will consider the exceptions mechanism found in Standard ML [Harper *et al.* 88]. Its syntax has changed recently, and we will present the new version, but the old one is conceptually very similar. As exceptions are statically *typed*, any exception must be declared before use (there is also a number of predeclared exceptions corresponding to run-time errors). We declare an exception with the syntax

`exception` *en* `of` *T*`;`

e.g., `exception exc of int` declares an integer-typed exception `exc`.

The exception is activated with a `raise` expression:

`raise` *en E*

e.g., `raise exc 4` activates exception `exc` with the value 4. A `raise` can appear in any context, so it is legal to write `1+raise` .... After a `raise`, an *exception packet* (containing the name and associated value of the exception) is propagated outward to the nearest *dynamically* enclosing handler for the exception.

We attach a handler to an expression with the syntax:

E_N `handle (`*en x*`) =>` E_E

This is a new expression that will normally evaluate to E_N. However, if execution of E_N raises exception *en* with a value v, (and does not itself contain a handler for it), E_E will be evaluated instead, with x bound to v. If an exception is not handled, it will propagate to the top level and stop execution of the program.

Exceptions in Ada are almost identical, the main difference being that they are command-based rather than expression-based, so that they cannot transfer information. Like in SML, they are declared explicitly, activated by a `raise` statement, and handled in the 'exception part' of a block. Several Lisp variants also have a similar facility, known as catch/throw. Here, an exception is raised using `(throw` *en* E_T`)`, and handled by `(catch` *en* E_C`)`, with the semantics that the result of `catch` is the value of E_C, unless its evaluation causes a `throw` to *en*, in which case the result will be the value of E_T.

We see that while exceptions can be typed statically (the exception declaration must be visible at the `raise`), they are bound dynamically, because the `raise` need not occur textually within E_N, but could be in a (parameterless) function called by it. This is exactly the continuation-based counterpart of dynamic value bindings. The similarity even extends to implementation details:

In *deep binding*, as used originally in the LISP 1.5 definition [McCarthy *et al.* 62], the current value of a variable is found by going back through the A-list, locating the first (*i.e.*, latest added) binding. While this is a side-effect-free implementation, it can end up being quite slow, especially when the variable is bound outside of a deeply nested recursive call.

Therefore [Baker 79] introduced the implementation technique known as *shallow binding*: to every symbol (or every symbol declared a fluid variable) is associated a *value cell*, which can be accessed in constant time. When a variable is bound to a new value, the old contents of its value cell is saved away on the stack, and overwritten with the the new one. When control returns to the place of binding, the old value is restored. The current value of the variable is then always immediately accessible.

The same technique applies to exceptions. To every exception name we can associate a cell containing the address of its last encountered handler frame (and 'unhandled' for new exceptions). Every time an expression (or block in Ada) is evaluated (executed) with a new handler, *i.e.*, E_1 `handle` exc `=>` E_2 or `begin` S_1 `exception when` exc `=>` S_2 `end`, a new handler address is installed, with the old one saved on the stack, to be restored upon exiting the protected block. A `raise` can then be executed instantaneously, by simply fetching the current handler address, rather than going back through all activation frames along the dynamic links, looking for a handler.

Dynamic scoping may be very convenient on some occasions, and is used by a number of languages, *e.g.* ("old") Lisp and APL. However, the semantic problems are severe and most languages, including Scheme and Common Lisp use static scoping for the basic binding primitives; among other things, this makes efficient compilation easier. The SLC as presented does not have dynamically scoped variables. But if it did (as an extension, not a replacement!), exceptions (dynamic continuations) would be an exact dual to fluid-let (dynamic values).

It is interesting to note that the resent change in SML's exceptions was made precisely in order to provide a uniform treatment of constructors (coproduct injections) and exceptions. This is very much in accordance with the SLC view of these concepts as being closely related, except for the dynamic binding aspect. It definitely seems possible to create a variant of ML with exceptions replaced by a statically-scoped construct, but retaining the new unification of coproduct types and escapes.

4.2.5 Persistent continuations

Persistent continuations are the natural extension of non-local exits to languages with higher-order functions, *e.g.*, Scheme or ML. The idea of such a facility is that the usual handling of non-local values in returned functions is extended to non-local continuations. This is very close to the SLC view, and we will be able to give quite accurate definitions of the various constructs as SLC terms.

There seem to be two slightly different approaches to providing persistent continuations in a language. The first presents the continuation as a "normal" function that will perform a jump when applied, while the other conceptually distinguishes between continuations and functions. The first method introduces fewer new concepts but is also semantically more complex, and inherently higher-order. On the other hand, the separated syntax like the **block/resultof** facility of the introduction is itself first order, but can be directly integrated in a higher-order framework.

The first way is exemplified by Scheme, which provides access to the current continuation trough a single procedure `call-with-current-continuation` or `call/cc`. This functional will pass an imperative functional abstraction q of the current continuation to its argument f. If f does not apply q, the result of `call/cc` will be f's result. But if f applies q to a value a, evaluation of f's body will be abandoned, and a returned as the result of `call/cc`. For example, we have:

```
(+ 5 (call/cc (lambda (q) (+ 3 4)))) ⟹ 12
(+ 5 (call/cc (lambda (q) (+ 3 (q 4))))) ⟹ 9
```

`Call/cc` is much more general than an exception mechanism, because if f does not call q, but returns it as (part of) a functional value, the captured continuation may be re-invoked at any time, restoring control to the expression embedding the `call/cc`. This means that function activation records cannot in general be stack-allocated. But this is also the case if the language only has non-local values. Therefore, adding first-class continuations to a higher-order language does not in general give any serious implementability problems, just like the runtime organization of stack-based languages usually directly admits non-local exits.

We can express `call/cc` accurately in the SLC:

$$call/cc \equiv k \Leftarrow k\downarrow(f \Rightarrow f\uparrow(c \Leftarrow k)) : [[A \to B] \to A] \to A$$

This definition highlights the two key features of *call/cc*: the *duplication* of the continuation k at the point where *call/cc* is called, and the *discarding* of the current continuation c, where the k-representing function (f's argument) is applied. The type also makes it clear that the result of *call/cc* will be either f's normal result or q's argument, so that the two must have the same type A. Furthermore, the continuation-representing function can be called in any context B, just like ML's `raise`.

We can also give an alternative definition, which makes it clear that the continuation-representing function will not return to its place of call. Rather than discarding the continuation, we explicitly specify that there can be no meaningful continuation, by making q return a *null*-typed result:

$$call/cc' \equiv k \Leftarrow k\downarrow(f \Rightarrow f\uparrow(\{\} \Leftarrow k)) : [[A \to 0] \to A] \to A$$

A slight syntactic variation on `call/cc` consists of performing the binding with a special form, *e.g.*, **escape** [Reynolds 72], T's `catch` (not to be confused with MacLisp catch), *etc.* The two styles are completely equivalent, as shown by the inter-definability relations:

$$\textbf{escape}\ k\ \textbf{in}\ E \equiv call/cc\uparrow(k \Rightarrow E)$$

$$call/cc \equiv f \Rightarrow \textbf{escape}\ k\ \textbf{in}\ f\uparrow k$$

A more significant variation consists of splitting the duplication aspect from the discarding, by treating continuations as special objects. This is somewhat closer to the SLC approach, precisely because the primitives are conceptually simpler. An example is of this organization is the `callcc` facility found in newer versions of Standard ML of New Jersey (NJML) [Appel & Jim 89]. However, this still insists on treating continuations as values, which actually makes the SLC definitions of the primitives slightly more complicated than Scheme's `call/cc`.

NJML gives access to continuation through the two functions

```
callcc:  ('a cont->'a)->'a
 throw:  'a cont->'a->'b
```

Here `'a cont` represents the type of `'a`-accepting continuations *treated as values*. We note that NJML's `callcc` differs from Scheme's `call/cc` by providing an opaque representation of the continuation, that can only be applied using a `throw`:

```
2+callcc (fn k=>1+throw k 3) ==> 5
```

How do we represent the type `'a cont` in the SLC? As usual, we must go through functions, which gives two equivalent possibilities, either $[A \to 0]$ or $[A \leftarrow 1]$. The former corresponds to viewing a continuation as a function that does not return, the latter to seeing it as an application context with no (meaningful) value part. For the closure-based view, we get

$$callcc_1 \equiv a \Leftarrow a\downarrow(f \Rightarrow f\uparrow(\{\} \Leftarrow a)) : [[A \to 0] \to A] \to A$$

$$throw_1 \equiv p \Rightarrow b \Leftarrow (\{\}\downarrow p) : [A \to 0] \to A \to B$$

And for the context-based:

$$callcc_2 \equiv a \Leftarrow a\downarrow(f \Rightarrow (a\downarrow f)\uparrow()) : [[A \leftarrow 1] \to A] \to A$$

$$throw_2 \equiv p \Rightarrow a \Rightarrow (b \Leftarrow () \Rightarrow a) \uparrow p : [A \leftarrow 1] \rightarrow A \rightarrow B.$$

We recognize $callcc_1$ as identical to the *call/cc′* definition above.

Let us finally note that we can implement a Scheme-like `call/cc` with NJML's primitives:

```
fun scallcc f = callcc (fn k=>f (throw k))
```

In particular, the type inferred by NJML for this is the same as the one for *call/cc* in the SLC. The reverse definition is also possible, in a sense, but it concentrates everything in `mlcallcc`:

```
(define mlcallcc (lambda (f) (call/cc f)))
      (define mlthrow (lambda (k) k))
```

It is interesting to note that a statically-scoped primitive like `callcc` can expose the fundamentally dynamic nature of exceptions. In fact, we can use the interchangeability of values and continuations through functions to define a fluid-let for values using exceptions:

Construct	Translation
`fluid-dcl` v`:`T	`exception` v`e of` T `cont`
`fluid-let` v`=`E_1 `in` E_2	E_2 `handle (`v`e k)=>throw k` E_1
`fluid-get` v	`callcc (fn k=>raise` v`e with k)`

For example,

```
fluid-dcl x:int;
fun addx n = (fluid-get x) + n;
(fluid-let x=3 in addx 1) * (fluid-let x=4 in addx 2)
```

is translated into

```
exception xe of int cont;
fun addx n = callcc (fn k=>raise xe with k) + n;
(addx 1 handle (xe c)=>throw c 3) * (addx 2 handle (xe c)=>throw c 4)
```

which evaluates to 24 in NJML.

In principle, a construct like `call/cc` is powerful enough to express any SLC function, but in practice the translation is non-obvious, particularly when higher-order constructs are involved, and the semantic complexity of `call/cc` completely hides the nature of the SLC definition. This suggests that it may sometimes be advantageous to think of a problem in SLC terms, and then translate it into Scheme or NJML. We shall see one such situation in section 4.3.3.

Finally, let us note that facilities like coroutines in Simula 67 [Dahl *et al.* 70] or tasking in Ada are also closely related to persistent continuations, because the activation records are heap-allocated. While the SLC in its current form cannot express true parallelism, it can handle coroutines, as we shall see in section 4.3.2.

4.3 Programming in the SLC

The SLC does not enforce any special "programming style". Every well-typed program in a higher-order value-based language is also correct in SLC, modulo syntactic differences. A typical SLC program will also encapsulate most continuation abstractions as "mode-neutral" injections *etc*, but sometimes a function is really best described with explicit continuations, and we will see a few such examples in this section.

We assume CBV evaluation everywhere, although there should be no real problems in going to CBN with eager coproducts. We will also use some of the notation introduced in section 4.1, but none of the proper language extensions presented there. Therefore, all examples can be directly translated back into the pure SLC, and indeed, they have all been executed, using an implementation of SLC derived directly from the semantic equations.

4.3.1 Non-local exits: tree search

As mentioned in section 2.3.1, tree search is a very natural candidate for using non-local continuations. Optimizing it with a goto or a call/cc is usually considered a rather questionable idea, but in the SLC it can be expressed as a precise declarative specification, not an imperative implementation.

Let us first define the type of binary trees:

$$\textbf{type } T_A = A + T_A \times T_A$$

A natural problem consists of deciding whether a tree contains a given element. We want a function

$$search : A \times T_A \to Bool$$

that returns true (first inject) if the tree contains the element, and false otherwise. Let us first give the direct solution:

$$\begin{aligned} search_1 \quad \equiv \quad & \textbf{rec } s = (x,t) \Rightarrow \\ & \quad \textbf{case } t \textbf{ of } (\iota_1 \uparrow a) \Rightarrow (a = x) \mid (\iota_2 \uparrow (l,r)) \Rightarrow s \uparrow (x,l) \textbf{ or } s \uparrow (x,r) \textbf{ esac} \end{aligned}$$

This uses the operator $or : Bool \times Bool \to Bool$ of section 4.1.6 to avoid traversing the entire tree if the element is found quickly. We see that every recursive call is made with the same x as first argument. We can therefore take it out of the recursion as a non-local value:

$$\begin{aligned} search_2 \quad \equiv \quad & (x,t) \Rightarrow (\textbf{rec } s = t \Rightarrow \\ & \quad \textbf{case } t \textbf{ of } (\iota_1 \uparrow a) \Rightarrow (a = x) \mid (\iota_2 \uparrow (l,r)) \Rightarrow s \uparrow l \textbf{ or } s \uparrow r \textbf{ esac}) \uparrow t \end{aligned}$$

While this definition is arguably better than the first, because it does not involve x in the recursion, there is still a common part: the true continuation. If any comparison succeeds, the result of 'search' is immediately known to be 'true', but this answer must be passed back through all the nested recursive calls. We can eliminate this copying in the third definition, where we also abstract the 'found' continuation outside of the recursion:

$$\begin{aligned} search_3 \quad \equiv \quad & (x,t) \Rightarrow t \textbf{ as } nfnd \downarrow (\textbf{rec } s = t \Rightarrow \textbf{case } t \textbf{ of} \\ & \quad (\iota_1 \uparrow a) \Rightarrow (nf \Leftarrow \{nf, fnd\}) \uparrow (a = x) \mid \\ & \quad (\iota_2 \uparrow (l,r)) \Rightarrow \textbf{let } () = s \uparrow l, () = s \uparrow r \textbf{ in } () \textbf{ esac) into } \{fnd, nfnd\} \end{aligned}$$

A normal return from a recursive call will mean that the search failed. Therefore, if the element is not found anywhere, the failure continuation $nfnd$ will eventually be activated. Because we are only returning a yes/no answer, the order in which the tree is searched is unimportant. This means that we can use the "parallel let" to search both branches in an unspecified way. The two negative results are combined into one before returning.

We note that abstracting out the x in going from $search_1$ to $search_2$ would probably not improve efficiency much, if at all. The reason is that we are using x a lot, which means that the extra implementation complexity of accessing a non-local variable (display or static links) can more

than outweigh the copying operation. In fact, some compilers would "optimize" $search_2$ back into $search_1$ (λ-lifting). On the other hand, the non-local success continuation fnd will only be applied once or not at all. Therefore, we can really expect a speedup from the second transformation.

We could also express the definition of $search_3$ using more pieces of SCL. In particular, a function $r \Leftarrow \{f, r\} \downarrow F$ is in fact a 'partial coapplication' of F, so we can write an equivalent definition of $search_3$ as:

$$search_{3a} \equiv (x,t) \Rightarrow (\{f, nf\} \Leftarrow nf \downarrow \mathbf{rec}\ s = [f \downarrow (a \Rightarrow (a = x))_*, \Diamond \circ (s \times s)]) \uparrow t$$

This is not quite as intuitive as the definition above, but still readable. If we treat '=' as a binary function, rather than an infix operator, we can also eliminate the a, by writing $a \Rightarrow (=) \uparrow (a, x)$ as $(=)^* \uparrow x$. However, if we take the final step and abstract out the variables x and f from the body of the recursive function and interpret it as a recursive continuation, we obtain a rather large and unreadable SCL term, pointing out once more that a totally variable-free notation is not really practical for writing programs, even though it may be well-suited for implementation.

4.3.2 Coroutines and streams

The example above was basically first-order, and could easily be translated into a language such as Algol or Pascal with non-local gotos. In this section, we will consider a more complicated problem, involving higher-order SLC concepts. The example can still be considered in Scheme or NJML, or a language with similar capabilities for heap-allocating activation records, *e.g.*, Simula coroutines or Ada tasking.

The same-fringe problem [Hewitt *et al.* 74] is a well-known example of the use of coroutines. The problem is to decide whether two binary trees (*e.g.*, as defined in the previous example) have the same sequence of values at the leaves in a left-to-right traversal. The problem lies in the fact that the two trees can have a completely different structure, so that a simultaneous recursion on the structure is not possible.

One way of solving this would be to simply flatten the trees into two lists, and then compare them iteratively or recursively. However, this potentially wastes a lot of time, if the two trees differ in the beginning, not to mention the extra space used for the lists.

Therefore, a better solution would be to generate the successive elements of the trees incrementally, and compare them as they are found. A CBN language solves this automatically, as we can rely on the laziness not to produce more of the lists than is actually needed. We can easily express lazy lists in a CBV language by using lifted types, but their generation is more problematical. For example, the natural definition of 'fringe' (or 'flatten') with an accumulator will not work as intended:

```
(define (fringe t a)
  (if (atom? t)
      (stream-cons t a)
      (fringe (car t) (fringe (cdr t) a))))
```

The problem is that `stream-cons` must be given something that can benefit from a delaying as the second argument. Delaying a variable is useless, so we do not gain any laziness from the above definition. Of course, we can simply explicitly delay the second parameter (a) of `fringe`, but this may not always be so simple, if we are examining a more complicated tree.

In a language with first-class continuations, we can give an alternative model of streams, based on coroutines, rather than delayed arguments. While we will still access a stream as a lazy list, it will be built incrementally by capturing the continuation of the generator as a suspension, again by exploiting the coercions continuation $\rightarrow$ function $\rightarrow$ value. Also, the primitive operation on

streams will be appending of substreams rather than adding elements one at a time. In fact, stream appending will be modeled by (naturally associative) function composition.

We define lazy lists in the usual way:

$$\textbf{type}\ LL_A = 1 + A \times [1 \to LL_A]$$

However, we will not define the operator stream-cons. Instead, we introduce two functions *emit* and *collect*. *Emit* is used every time we are returning a value from the generator. It can be thought of as creating a substream of length one. *Collect* transforms the stream built by the generator into a lazy list, which can be accessed sequentially.

For example, the fringe of a tree can be found as:

$$fringe \equiv t \Rightarrow collect \uparrow ((\textbf{rec}\ f = [emit,\ (l,r) \Rightarrow (f \uparrow r) \circ (f \uparrow l)]) \uparrow t) : T_A \to LL_A$$

The recursive function f traverses the tree. For leaf elements, it returns a singleton substream obtained from *emit*, while the stream generated by an interior node consists of the substreams generated by the subtrees. Because of the right-to-left evaluation of CBV function composition, the above definition does specify that the substream generated by from the left subtree should be returned first.

Emit is called with a value, which will become the head of a lazy list cell. For the tail part, *emit* should capture the current continuation of the generator as a suspension, *i.e.*, a value of type $[1 \to LL_A]$. By the coercions, this corresponds to a continuation of type $[LL(A) \leftarrow 1]$, which must therefore be the return type of $emit \uparrow a$. It must also be the input type, if we want to combine an unknown number of successive emits through function composition. This leads to the following definition of the stream type

$$S_A = [LL_A \leftarrow 1] \to [LL_A \leftarrow 1]$$

We define *emit* as follows:

$$emit \equiv n \Rightarrow s \Leftarrow () \Rightarrow \iota_2 \uparrow (n, s) : A \to S_A$$

This describes the effect precisely: *emit* accepts a value n and a continuation s, and packages them together as a second inject of the lazy list type. The generator will remain suspended until the lazy list consumer reactivates it, by applying s to ().

Collect in a sense starts the process, by supplying the end of the stream:

$$collect \equiv p \Rightarrow (\iota_1 \downarrow p) \uparrow () : S_A \to LL_A$$

Collect implicitly captures the current continuation by using a function as the continuation of p.

We can then compare the fringes of two trees as lazy lists:

$$sf \ \equiv \ (t_1, t_2) \Rightarrow cll \uparrow (fringe \uparrow t_1, fringe \uparrow t_2) : T_A \times T_A \to Bool$$

where the function *cll* (compare lazy lists) is defined in the usual way as

$$\begin{aligned} cll \ \equiv \ & \textbf{rec}\ c = (l_1, l_2) \Rightarrow \textbf{case}\ l_1\ \textbf{of} \\ & (\iota_1 \uparrow ()) \Rightarrow \textbf{case}\ l_2\ \textbf{of}\ (\iota_1 \uparrow ()) \Rightarrow true \mid (\iota_2 \uparrow (a_2, ll_2)) \Rightarrow false\ \textbf{esac} \mid \\ & (\iota_2 \uparrow (a_1, ll_1)) \Rightarrow \textbf{case}\ l_2\ \textbf{of} \\ & \quad (\iota_1 \uparrow ()) \Rightarrow false \mid \\ & \quad (\iota_2 \uparrow (a_2, ll_2)) \Rightarrow (a_1 = a_2)\ \textbf{and}\ c \uparrow (l_1 \uparrow (), l_2 \uparrow ())\ \textbf{esac esac} \end{aligned}$$

There is no need for non-local exits when comparing two lists, because the function is expressed tail-recursively.

The stream system presented here is just as general as Scheme's. for example, the stream representing the infinite sequence of integers is written like this:

$$ints \equiv (\mathbf{rec}\ from = n \Rightarrow (from \uparrow (n+1)) \circ (emit \uparrow n)) \uparrow 0$$

and it can be converted into a lazy list by $collect \uparrow ints$.

4.3.3 More on iteration and recursion

Iteration is commonly thought of as a special case of recursion, because it is so simple to translate an iterative function into a recursive one. While it is well-known that any partial recursive function can also be rewritten into an iterative one (in fact one while-loop is enough), such a conversion may be far from obvious if the recursion pattern is complicated. It may therefore seem somewhat surprising that having first-class continuations in a language permits us to define a traditional fixpoint combinator (for functions) in terms of iteration only.

Let us start by noting that for recursive values, we have the equation:

$$(\overline{\mathbf{rec}}\ x = E) \ = \ ((),x) \Rightarrow E)^{\dagger} \uparrow ()$$

Here, we simply use the instance of $-^{\dagger}$ that gives a $1 \to T$ typed morphism (function) from one with type $1 \times T \to T$. To obtain a T-typed value, we apply the function to (). Similarly, for recursive continuations, we have:

$$\{\underline{\mathbf{rec}}\ y = C\} \ = \ \{\} \downarrow (\{\{\},y\} \Leftarrow C)_{\dagger}$$

Now, we can simply rewrite the definition of fix with a CBV $\underline{\mathbf{rec}}$, exploiting the coercion between functions and values:

$$fix \ = \ (f \Rightarrow \underline{\mathbf{rec}}\ a = f \uparrow a) \ = \ \{\} \downarrow (\{\{\},a\} \Leftarrow f \uparrow a)_{\dagger}$$

Thus, fix works by iteration on contexts. This is very similar to the way a traditional (iteration-based) machine architecture executes recursive programs by managing the stack of contexts (application frames) explicitly.

Perhaps even more surprisingly, we can translate this definition into any language with persistent continuations, s.a. NJML. We assume we have an iterator like the following:

```
datatype ('a,'b) COPROD = in1 of 'a | in2 of 'b;

(* f:A->B+A => iter(f):A->B *)
fun iter f a = case f a of (in1 b)=>b | (in2 a1)=>iter f a1;
```

Here, we have written `iter` as a tail-recursive function but could equally well have used a while-like construct, except that this is quite awkward to express in ML, requiring updateable auxiliary variables, explicit sequencing with ';' *etc.* With an iterator like the above we can easily define some recursive functions by using an accumulator, *e.g.*:

```
fun fac n = (iter (fn (n,a)=>if n=0 then in1(a) else in2(n-1,a*n))) (n,1)
```

But it is not so obvious how to obtain a general fixpoint combinator. We could probably find it through operational reasoning in terms of stacks, *etc.*, but this is not a very pleasing solution. If we instead translate the above SCL term, we obtain the somewhat unreadable, but theoretically well-founded NJML definition:

```
exception null;
fun fix f x =
    callcc (fn r=>
      (iter (fn (v,c)=>callcc (fn na=>
              throw c (f (fn x=>callcc (fn q=>throw na (in2(x,q)))) v)))
            (x,r);
       raise null));
```

Here, we represent a context explicitly as a pair consisting of a value v and a continuation c. The coercions and continuation abstractions of the SLC turn into a tangle of `callcc`s and `throw`s, but the underlying execution pattern is the same for both languages. The exception `null` will never actually be raised, just like the continuation $\{\}$ in the SLC version will never be activated. In both cases they are only needed to make the expression type-correct.

It is possible to give an intuitive, operational explanation of how the NJML `fix` works by adding new activation frames to the "control stack" as needed. However, this would miss the issue of its declarative content, obtained directly from dualizing a simple value-based SLC equation.

As expected from a true fixpoint combinator, the expression

```
fix (fn f=>fn n=>if n=0 then 1 else n*f(n-1)) 5
```

evaluates to 120. Of course, we could just as well write the definition above using `call/cc` in Scheme, but the NJML definition also highlights the fact that the expression is well-typed.

4.4 Object-oriented programming

Object-oriented programming (OOP) is often thought of as radically different from "traditional" programming. This distinction is based on two important characteristics of OOP languages: an object/message paradigm of computation, and the concept of inheritance. In this section, we will show how categorical duality relates to these ideas. We do not treat any particular OOP language in depth, but the last subsection points out how concepts and constructs in various such languages are affected by the symmetrical framework presented here.

The presentation adopts a function-based view, concentrating on the symmetry between products and coproducts rather than between values and continuations directly. We will, however, introduce relevant SLC/SCL concepts as necessary to demonstrate "hidden" symmetries. Some familiarity with the basic ideas of OOP might be a considerable advantage for reading this section but is not strictly necessary.

To avoid any confusion, let us point out immediately that the "objects" of object-oriented programming are not related to the categorical objects, at least not for the kind of categories we are concerned with here. In this section, the term "object" will always refer to the OOP meaning of the term; we will use "type" for objects in the SCL category.

4.4.1 A symmetry of organizations

A common conceptual structure in many problems of data organization can be summarized as a choice between representing a two-dimensional table in either row-major or column-major order. The purpose of this section is to show that the difference between "traditional" and "object-oriented" style is essentially of this kind.

Let there be given a number of *input* or *instance* types $I^1, \ldots, I^m$, from which we want to compute a series of *output properties* or *operations* $O_1, \ldots, O_n$. As an example, we could take an interpreter for a programming language and consider the type of values in this language. This is

precisely a set of instances: integers, booleans, strings, functions, *etc.* Similarly, there are many operations we want to perform on these values: read, print, create, destroy, compare, *etc.*

As another example, we have the typical OOP scenario: the instances of a type vehicle are, *e.g.*, bicycle, bus, truck, *etc.*, with properties like weight, maximum speed, number of passengers, *etc.* It is important that these properties are not necessarily present as explicit parts of the instance ("instance variables"), but are computed from it. For example, the instance itself could be an index into a database, and the outputs found by a lookup operation.

Let us assume that for every instance number i and property number j, we have a function ("method") $f^i_j : I^i \to O_j$. (This is an oversimplification; we will consider the case where not all f^i_js exist or are meaningful in the next subsections). We want to organize these functions in a "logical" way, grouping "related" code together, to ease further maintenance and modifications. This is precisely where we must make the fundamental choice between two paths.

In "traditional style", we first define explicitly a *variant type* ("datatype", union) $I = I^1 + \cdots + I^m$ (eager coproduct in CBN), *i.e.*, fix the set of instances. We can then give the n functions $f_j : I \to O_j$ as $f_j = [f^1_j, \ldots, f^m_j]$, where the definitions of f^i_j are all placed together, in a case-like construct. Finally, we implicitly define the desired function f for computing any property from any instance as $\langle f_1, \ldots, f_n \rangle$.

This organization makes it easy to add a new output (operation, property) O_{n+1}, by just supplying a definition of $f_{n+1} = [f^1_{n+1}, \ldots, f^m_{n+1}]$. For the interpreter example, we may want to compute the size of an instance of Val: we need only specify how to find the size of an integer, a string, *etc.*; no existing code needs to be modified. However, adding a new input type s.a. reals is very inconvenient, as we have to add new cases to all existing functions on values: read, print, compare, *etc.*

In "object-oriented style", we instead group all the outputs as a *class specification* or *signature* $O = O_1 \times \cdots \times O_n$ (lazy product in CBV). We then define the m functions $f^i : I^i \to O$ as $f^i = \langle f^i_1, \ldots, f^i_n \rangle$ (the class definitions). Here, the definitions of the f^js are given together as "member functions". Finally, we give f as an implicit $[f^1, \ldots, f^m]$.

Such an organization makes it very easy to add a new instance type I^{m+1}: we need just add a definition of $f^{m+1} = \langle f^{m+1}_1, \ldots, f^{m+1}_n \rangle$, with no changes to existing code. On the other hand, adding a new operation is just as problematic as adding a new input to the traditional organization: we must go through all existing class definitions and add descriptions of how to perform the new operation on each instance.

From a theoretical point of view, the two approaches are completely equivalent, because of the equality

$$\langle [f^1_1, \ldots, f^m_1], \ldots, [f^1_n, \ldots, f^m_n] \rangle = [\langle f^1_1, \ldots, f^1_n \rangle, \ldots, \langle f^m_1, \ldots, f^m_n \rangle] : I \to O$$

This follows directly from the general properties of products and coproducts, unlike, *e.g.*, the distributive law (the morphism δ), which requires further axioms.

This symmetry means that none of the alternatives is inherently superior. The best choice of organization is clearly dependent on the nature of the problem: if the set of operations is relatively stable, but new instances are often added, an OOP organization would be preferable, but if the main extensions consist of adding new operations on a fixed set of instances, such as new functions over a tree type, the traditional organization should be used. Often, a large program will use the OOP organization for some data and code, and the traditional one for others. A language with a heavy bias towards either formalism would cause severe problems in such a case.

Perhaps somewhat surprisingly, the description of "OOP style" above did not mention *objects*. This is because the essence of the object-oriented organization seems to consist of a particular

grouping of methods; an object is just a value-based manifestation of this principle. We can think of an object as an O-typed value obtained by applying f^i to an I^i-typed instance, *i.e.*, a lazy tuple of the properties. A "message" is then a selection among the components of the object. Note that a property need not be a simple value, but can itself be a function (closure), so that additional information may be present in the message.

We could similarly define a "co-object" as an I-typed *continuation*, obtained by transforming a O_j-typed continuation with f_j. Again, the summands of a co-object can be context-typed, so that messages sent to it can contain "return addresses" as well as input data. In fact, the *actors* in [Hewitt 79] are more like co-objects than objects, because they are associated with *properties* rather than *instances* as in other object-oriented languages.

4.4.2 Specification inheritance

The other main characteristic of OOP is the concept of inheritance. In fact, some would say that this is an even more important property than the organizational difference mentioned above. Two benefits of inheritance are often cited: clarity and potential for code reuse. We shall see that these two are essentially independent, and that both dualize to "traditional" programming. The former, which we will call *specification inheritance*, concerns the concept of subtyping, while the latter, *implementation inheritance* has to do with method sharing. Unfortunately, many OOP languages do not make an explicit distinction between these two kinds of inheritance, and often force an implementational relationship to express a specification-based one.

In the following, we will treat specification inheritance, and implementation inheritance in the next subsection. Basically, specification inheritance permits a more general function to be used directly where a restricted one is expected. For OOP, this means a function computing more properties than necessary; for "traditional" programming, it is a function over a larger number of instances.

More concretely, for OOP, a class O' is said to be a *specification subclass* of a class O if $O' = O \times \cdots$. Any function $f' : I \to O'$ can then be viewed as a function $f : I \to O$, by letting $f = \pi_1 \circ f'$. Note that this ordering is based exclusively on properties and does not imply any common instance variables or algorithms. In fact, the instances of the subclass can often be represented *more* efficiently and compactly, precisely because they are more specialized.

For example, integer numbers can be thought of as a specification subclass of reals, because every property that a real number has, an integer also has, through the inclusion of integers into reals. But it would rarely be a good idea to implement integer arithmetic in terms of floating point operations. Likewise, for many "container classes", similar specifications do not imply similar implementations, *e.g.*, an array can be seen as a special case of an "lookup function" but is usually implemented in a much more efficient way than a general association list.

The subclass ordering of properties induces a reverse ordering on inputs when functions are composed. In particular, $id_{O'}$ can be viewed as a function $O' \to O$, so that a function $g : O \to T$ is also a function $g' : O' \to T$. This is just another way of stating that the result of an O'-returning function can be used where a O-typed value is required.

In "traditional" style, we can equivalently view an instance variant type I' as a *specification supertype* of I if $I' = I + \cdots$. A function $f' : I' \to O$ is also a function $f : I \to O$, by $f = f' \circ \iota_1$. Similarly, a function $g : S \to I$ is also a function $g' : S \to I'$, because the function $id_{I'}$ is also a function $I \to I'$. Such a coproduct-based inheritance relation can be used to avoid some of the "useless" injection tags, so common in ML programs, by making type inclusions implicit. As a simple example, a tree with possibly "empty" leafs is a supertype of a simple binary one. This means that we can use an existing binary tree directly where a more general one is expected.

In general, a class can have many subclasses, and a variant type can have many supertypes; both of these represent *single inheritance*. The concept of specification inheritance extends directly to *multiple inheritance*, where a class can inherit the properties of several others. In that case, however, we must modify the definition somewhat: we say that O' is a subclass of O if O' includes all components of O, *i.e.*, if O' is *isomorphic* (not *identical*) to $O \times \cdots$. This permits us to treat, *e.g.*. $O_1 \times O_2 \times O_3$ as a subclass of both $O_1 \times O_2$ and $O_1 \times O_3$, which would not be the case with the earlier definition of subclassing. Inheritance for coproduct types is extended analogously, so that a variant type may be a supertype of several others at the same time.

In implementational terms, going from identities to isomorphisms of types introduces an extra level of indirection in method lookup. However, there are no theoretical problems from overlap between several superclasses, as far as *specification* is concerned. Only when we want to inherit *implementations* do conceptual problems arise.

4.4.3 Implementation inheritance

The second aspect of inheritance in object-oriented languages concerns code reuse. In principle, this is independent of specification inheritance, although the two concepts supplement each other nicely. The purpose of this subsection is to present the essence of implementation inheritance in isolation.

While specification inheritance concentrates on the instance and property types I and O, implementation inheritance concerns the methods f^i_j. We recall that these are usually not independent functions, but are grouped together according to either instance (f^i) or property (f_j). Often, we want to reuse parts of such a composite function, without physically duplicating the code.

For OOP, it may be the case that some properties of a *new instance* I^{m+1} are computed in the same way way as those of an existing instance I^i. We can then define I^{m+1} as an *implementation subclass* of I^i. I^{m+1} may have additional properties, but this is a question of (output) specification inheritance; the code for the new operations cannot be inherited anyway.

Also, the instance I^{m+1} need not be identical to I^i. However, to reuse the methods of I^i, I^{m+1} must be a *specification subclass* of I^i, *i.e.*, I^{m+1} must include all of I^i's *instance variables*. Then (input) specification inheritance permits us to use the f^i_js directly as f^{m+1}_js. Of course, the methods not inherited can be functions of all of I^{m+1}, not only the part present in I^i (although visibility rules in particular languages may limit access to the instance variables of a superclass). Also, a new method replacing one from the superclass can often be conveniently expressed in terms of the old.

For "traditional style" there is an analogous concept of implementation inheritance where a *new property* O_{n+1} is computed similarly to an existing one O_j. Again, specification inheritance permits some differences between types: the new property may be defined for a larger number of instances than the old. In this case, we must supply the new methods explicitly. Also, O_{n+1} can be a supertype of O_j, *i.e.*, a variant with more cases. Specification inheritance ensures that the old methods can still be used directly.

Surprisingly, the essence of implementation inheritance is already present in most "traditional" programming languages, namely as the 'otherwise' clause or "wildcard pattern" in a 'case' statement or expression. For example, in the interpreter example, we can define a function 'print_ext', almost like 'print' except that it prints strings in quotes, to permit the output to be read in later. We can define 'print_ext' in terms of 'print' by a case construct that singles out strings for special treatment, but defers all other value types (without listing them explicitly) to the original 'print' function. The new printing method for strings can also be expressed in terms of the old one.

Like specification inheritance did not in general imply implementation inheritance, the reverse is not true either. It is sometimes the case that some inherited properties of an implementation superclass are meaningless for the new class. For instance, we might implement booleans in terms of integers, with 0 for false and 1 for true. A number of fundamental properties can be inherited, *e.g.*, comparison operations, size, creation/destruction *etc.* On the other hand, arithmetic operations on booleans do not make sense.

As a slightly larger example, we might implement finite multisets (bags) as unsorted lists. A large number of operations on lists also apply to multisets, *e.g.*, 'length' (cardinality), 'map' (extension of function to multisets) and 'append' (union). However, an operation like 'head' ("first element") makes no sense for a multiset. In a language where implementation subclassing implies specification subclassing, problems like this must be solved by overriding the inapplicable inherited method with an error message.

Implementation inheritance can also be multiple, *i.e.*, a new instance can inherit methods from several others. However, this is somewhat more problematical in the case of overlaps, because no single strategy for dealing with them is likely to be generally applicable. Often, we want to inherit a multiply-defined method from a particular superclass, but sometimes, *e.g.*, for object finalization, we would usually invoke the cleanup methods of all superclasses. Similarly, multiply inherited instance variables should usually not be duplicated but may have to be in certain cases.

It is therefore important to remember that implementation inheritance is not a precisely defined concept like specification inheritance, but a generic term for any language constructs that make method sharing simpler. Thus, there is no "natural" way to handle overlaps; the correct solution must be decided on a case-by-case basis.

4.4.4 A look at some object-oriented languages

In this subsection, we will see how some existing OOP languages fit into the framework outlined above. As we have mentioned, the concepts of specification and implementation inheritance are often tightly interwoven, and different languages group them in various ways. We will try, however, to separate them as much as possible.

The perhaps best-known object-oriented language Smalltalk [Goldberg & Robson 83] has a purely implementation-based class organization. The specification class of an object is determined implicitly by the set of messages it accepts (without error). Classes in languages like Simula [Dahl *et al.* 70] and C++ [Stroustrup 86] are also based on common implementation but use (often *pro forma*) implementation subclassing to express specification conformance as well: the possibility of different instance types with the same properties is expressed through *virtual* member functions, *i.e.*, methods in the implementation superclass that are expected (or even required) to be redefined in subclasses.

In other languages, specifications form the basis of class definitions. As examples of this kind we note Trellis/Owl [Schaffert *et al.* 86] and the language presented in [Cardelli 84]. Here, it is the behavior of a class, not its set of instance variables or methods that determines its place in the type hierarchy.

The specification subclassing relation may be explicit or implicit. In the former case, a class specification explicitly names its superclass. Examples of this approach include C++ and Trellis/Owl. In other languages, subclassing is implicit and relies on a consistent naming of properties. This is the case for Smalltalk and Cardelli's language. The latter also has a name-based specification inheritance for coproducts.

C++ (at least the latest available version 1.2) and Simula are based on single inheritance, while languages like Trellis/Owl and Cardelli's notation use multiple. Somewhat surprisingly, *specification*

inheritance in Smalltalk is also multiple: an object is a member of a specification class simply by virtue of having the required properties, so that it can easily be in several classes at the same time. And in fact, message lookup in Smalltalk is somewhat more complicated than for the single-inheritance languages.

Another important aspect is the possibility of static typing. Once again, this is independent of the other characteristics, s.a. implicit/explicit subclassing. In Smalltalk, an object may fail to "understand" a message at runtime; the type systems of all the other languages mentioned prevent this.

However, even a statically typed language can include features that breach type security. In C++, it is possible to cast a pointer to an object of a class type to one of a derived (subclass) type. Such a cast is well-defined only if we can somehow ensure that the object pointed to is really of the indicated type, *i.e.*, has the additional properties of the derived class. When this is not the case, we have effectively applied the "inverse projection" $\pi_1^{-1r} : A \to A \times B$ of section 2.6.2: the result appears to have both properties A and B, but trying to access the B-typed one constitutes a runtime error.

Now unrestricted casting is a well-known unsafe feature of both C and C++. However, even Cardelli's language includes a very similar feature for coproducts, namely the 'as' operation, which converts a coproduct-typed value into one with *fewer* possibilities, *i.e.*, only 1. This is precisely an "inverse injection" $\iota_1^{-1l} : A + B \to A$. If we know (*e.g.*, from the 'is' operator) that the converted instance is the right variant, this operation is safe but not otherwise.

So, from a theoretical point of view, the C++ class pointer casting is just as defensible as an 'as'. In practice, however, there is the important difference that casting represents an *unchecked* conversion, *i.e.*, an illegal cast will give unspecified results, rather than a well-defined runtime error. This is analogous to a Lisp implementation not checking, *e.g.*, that the argument of a 'car' operations is in fact a "cons-cell".

4.5 Abstract interpretation

We can consider interpretations of SLC/SCL programs over a non-standard domain, inducing a natural symmetry between forward analysis as interpretation with abstract values and backward analysis as interpretation with abstract continuation. This also seems strongly connected with the view of statements in an imperative language as either state or predicate transformers [Gries 81].

Typically, dual analyses will concern dual evaluation strategies. For example, *strictness analysis* [Hughes 87] in CBV would be meaningless, as all functions are strict. It would rather be replaced by *totality analysis*, which could be used to ensure that transformations involving the conditional axioms did not change the meaning of the program.

As a simpler example of duality, we can consider *reachability* analysis in CBV. Here we interpret a program over the abstract domain $\{R, UR\}$, with R denoting "maybe reachable" and UR meaning "definitely unreachable". This is a rather crude domain, but can refine it by adding two more abstract elements for coproduct-typed values, I_1, and I_2, with I_1 denoting "only reachable with first inject", and I_2 analogously for the second. These four elements would be arranged in the usual "diamond lattice", as for, *e.g.*, an odd/even interpretation.

We start the interpretation with UR assigned to all values except the input of the program, and propagate reachability information forwards until a fixpoint is reached. In particular, knowledge about injection tags (I_1/I_2) can sometimes be used to see whether iterations terminate. A non-trivial (*i.e.*, not all R's) fixpoint may show us that parts of the program will never be traversed,

and can be eliminated without changing the meaning.

Let us now consider the dual case, *neededness analysis* in CBN. Here, the goal is to find out which parts of the computation will not be used. The basic domain is the two-point $\{N, UN\}$, denoting "maybe needed" and "definitely unneeded" respectively. Again, we can refine it by adding two more elements for *product* types: P_1 and P_2, meaning that at most the first (second) component will be needed.

Here, we start the interpretation from all UN, except the output of the program, which will be needed. We then propagate neededness information backwards, towards the input. Like for iteration, knowledge that a recursive result is expressed only in terms of itself, *i.e.*, that it will not need its other input, can be propagated through recursive definitions.

However, it seems that more advanced analyses than the ones sketched here are also possible. In particular, higher-order programs would be treated by considering abstract closures and contexts. A number of the other analyses suggested in [Hughes 88] also seem to fit well into the SLC/SCL framework.

4.6 Other directions for further research

As mentioned before, this investigation is not yet very deep. We have already suggested a number of subjects to be developed further, and in this section we will present a few more topics to explore. Again, some of these concern the SLC/SCL directly, while others involve more general implications of the symmetries pointed out in this thesis.

- **Further categorical concepts:** The SCL is based on a fixed set of combinators and associated axioms. It would be very interesting to see if the framework presented in [Hagino 87a], where virtually all of the structure is defined through adjunctions, could be extended to encompass continuations. From the other side, maybe the value/continuation duality could present categorical concepts s.a. pushouts and pullbacks in a new light. The idea of introducing symmetrical "subset" and "quotient" types through equalizers and coequalizers also looks promising, although computability may then become a problem.

- **Mathematical applications:** Unlike many applications of category theory, this one seems to provide a little feedback. The rather simple concept of a coexponential, expressed as the left adjoint of the coproduct functor could easily be dismissed as "meaningless" or of only theoretical interest by a mathematician. However, its importance in providing a higher-order framework for a generalization of partial functions might make it worth taking a second look at. Also, just as the usual CCC axioms can be "flattened" into intuitionistic logic, there may be a reasonable deductive system corresponding to the category investigated here.

- **Logic programming:** It seems that the ideas presented here concerning functional programming can be generalized to a category of relations, and logic programming languages like Prolog. Of course, this would involve the "procedural" interpretation of logic programs, just like the SLC builds on a computational view of functions, not a set-theoretical one. For reasoning about the control aspects of logic programming, *e.g.*, backtracking, continuations could be a natural tool. In fact, the SLC/SCL already contains elements of backtracking, as witnessed by functions like *pa*. Some work on continuations in a logic programming framework has already been undertaken by [Sato & Tamaki 89]. The bidirectional data flow in Prolog predicates may also be strongly connected with the view of functions as either value or request transformers in SLC.

- **Parallelism:** The SLC/SCL has been presented in an inherently sequential way. However, there seem to be no major obstacles in extending the framework to parallel execution, and the unspecified evaluation order of eager CBV products and lazy CBN coproducts suggests a natural duality between AND- and OR-parallelism. Continuation abstractions in the language need not impede parallelization, precisely because flow of control becomes explicit, so that independent pieces of the computation may be easier to detect.

- **SCL as an abstract machine:** With some refinements, the SCL might be used as the basis of an abstract machine, similar to the CAM [Curien 86], but with a firm categorical foundation for handling "imperative" aspects of control flow, s.a. run-time errors, escapes, *etc.* In fact, a CBV version of SLC might be more natural for traditional architectures than the CAM, whose theoretical background seems to imply availability of proper products and exponentials.

  Most current machine architectures have a well-developed set of operations for expressing flow of control, and an abstract machine that could deal well with such operations would be desirable. The CBV SLC view of functions as continuations accepting contexts (activation frames) also appears well-suited for implementation.

4.7 Comparison with related work

A number of authors have worked on designing more symmetric languages, using category theory for language design and implementation, and taming continuations. Yet no-one seems to have combined all three ideas. In this section, we will point out some of this work, and compare it to the present approach.

- [Hagino 87a] uses category theory to obtain a symmetrical language. He starts with very few primitive concepts, and builds virtually all of the type structure and programming constructs as categorical adjoints. The dialgebra approach gives a nice formulation of recursive types and primitive recursion. However, when used as a programming language the framework is not quite symmetrical. Although the left adjoint to the coproduct functor (*i.e.*, coexponentials) can be easily expressed in the general notation, it is not a valid construct in the "computable" subset of the language. Also, the restriction to primitive recursion, although seldom a problem, excludes a number of applications, s.a. interpreters.

  [Hagino 87b] also proposes a symmetric extension to ML, similar to the one discussed in section 4.1. However, because it concentrates exclusively on values, the symmetry is not perfect. For example, the construct 'merge', suggested as the dual to 'case' is not really its mirror image, neither syntactically nor semantically.

- [Curien 86] builds on the equivalence between λ-calculusand CCCs (or more precisely, C-monoids). One result is a set of categorical combinators leading to an efficient implementation (the Categorical Abstract Machine, CAM). However, the full axioms of CCCs seem to require CBN evaluation if semantic problems with nontermination are to be avoided, while at least some versions of the CAM are CBV-based. The implications of this do not appear to be treated in depth.

  Also, concentrating on the untyped λ-calculus gives conceptual problems because of the requirement of surjective pairing (basically that *every* value can be split into two components and recombined). Furthermore, the work does not seem to consider at all the "other side" of category theory, *e.g.*, coproducts, which are equally important in a practical language.

- [Girard 87] presents a formalism called Linear Logic, which exhibits many similarities with SCL. It has a form of negation as a fundamental primitive, and this induces a question/answer symmetry similar to SCL's. However, this work (on *classical* linear logic) appears to involve only proof theory, not general computation. In particular, it does not impose any structure on different morphisms between objects. Another important difference with SCL is that the latter does not consider a type and its negation as two different objects in the same category, but expresses the duality through the morphism structure instead.

  On the other hand, the Linear Abstract Machine [Lafont 88] builds on *intuitionistic* linear logic, in an essentially bicartesian closed category (*i.e.*, CCC with coproducts). Also it gives a traditional computational model in terms of values. Like the SCL, it distinguishes between strict products ($\otimes$) and lazy ones (&).However, the LAM loses much of the symmetry of classical linear logic. For example, it only has eager coproducts ($\oplus$), unlike the classical LL, which also has objects ("inverted &") which seem to correspond to the lazy coproducts of pure CBN. Furthermore, unlike the CAM, the LAM does not deal with general recursion.

- [Cardelli 84] investigates the symmetry between products and coproducts as related to inheritance, and a number of his ideas about types and subtyping have influenced section 4.1.2. The type system and the subtyping relation he presents is in fact completely symmetrical in products and coproducts. However, the actual language (*i.e.*, the evaluation part) is basically a syntactic variant of ML, and the symmetry of data types does not extend properly to symmetry of the constructs, although it goes further than many other languages.

- [Felleisen 88,Danvy & Filinski 89] approach continuations from an entirely different perspective. The emphasis is also on formalizing their declarative aspects, but is based on abstracting control as composable continuations, *i.e.*, essentially two-ended objects, in a traditional, Scheme-like language. While a number of deep symmetries between control and data appear here as well, they seem somewhat different from the SLC/SCL kind, and it has not yet been possible to unify the SLC view of continuations as fundamentally one-ended entities with the "delimited context" view presented here.

Chapter 5

Conclusion

The previous chapters have presented a framework for reasoning about continuations as a declarative concept. First-class continuations in a language are not a new idea. However, to the best of the authors knowledge, no other formalism has integrated them so deeply into the very core of the semantics and type system that the operations and concepts defined in terms of them could be accurately said to make up half of the language.

Perhaps the best way of summarizing the results obtained is therefore by pointing out a number of the fundamental symmetries exposed by the SLC/SCL:

- **Values and continuations:** This is the fundamental duality, on which all the following are built. It leads to the view that any function can act as either a value transformer or a continuation transformer, regardless of its mode of definition. From an SLC perspective, Scheme's slogan "continuations as first-class citizens" therefore appears somewhat misleading, because it implicitly forces a continuation to look and act as a value, rather then being an equal partner. So, while any particular evaluation strategy may well define values and continuations very differently, the SLC syntax treats them identically, which brings us to the next point:

- **Call-by-value and call-by-name:** It seems that these two reduction strategies have essentially dual properties and problems. In a basically asymmetric language, like the λ-calculus, this is not readily apparent, but with first-class continuations the duality becomes much clearer. In particular, while the CBN strategy directly admits a terminal object, products and exponentials, a CBV one has instead a proper initial object, coproducts and coexponentials. The exact dual of ordinary CBV, *i.e.*, "pure CBN", has the same problems with coproducts that CBV has with products. On the other hand, the "eager coproducts" in a traditional CBN strategy dualize directly to "lazy products" in a CBV language.

- **Exponentials and coexponentials:** Exponential objects provide a categorical framework for treating functions as values; they solve the problem of free variables in a function body by tying higher-order functions to products. Coexponential objects make possible an analogous treatment of functions as continuations, with non-local continuations handled through a connection with coproducts.

  While counter-intuitive at first sight, because they appear incompatible with set-theoretical concepts s.a. cardinality, coexponentials may in fact offer a more accurate view of higher-order functions in CBV languages than exponentials. Also, both of these constructs provide a basis for treating continuations as values and vice versa, but with the route through functions made explicit.

- **Totality and strictness:** These two concepts play a central role in the definition of SCL. It must be kept in mind that the familiar names denote slightly unconventional definitions. In popular terms, a total function respects a continuation passed to it, while a strict function respects the value. We note that the SCL's totality is a weaker condition than the one imposed by the traditional definition, while strictness is slightly stronger than usual. Yet in some ways the new, symmetrical definitions seem more accurate when reasoning about the properties of programs, *e.g.*, whether a function always evaluates its argument.

- **Iteration and recursion:** While (unbounded) iteration is traditionally seen as a special case of (general) recursion, the SLC/SCL view of them is as dual concepts. Their theoretical equipotence assumes a very direct form in the SLC, with iteration becoming the SCL foundation for self-referential continuations, and recursion for self-referential values (although the generic term "recursive"is used for both of these).

  Also, the SLC/SCL approach to the problem of nontermination is somewhat unconventional. Because it has to deal routinely with functions that do not return or that discard their input, the axiomatic framework can handle non-termination as just a special case of these phenomena. Thus, it does not impose any explicit information ordering relation on terms, but represents non-termination as a jump to (in CBV) or input from (in CBN) a special continuation $\top$ and value $\bot$ respectively.

It may interest the reader to know in which order the various parts of this thesis were developed. Perhaps surprisingly, the entire denotational semantics and type system for the SLC, including structured types and higher-order functions was written down very early, and have remained virtually unmodified for the entire duration of the work. Once the symmetrical syntax was fixed, there was in fact very little choice for the semantic equations, if the λ-calculus conversion rules and their duals were to hold.

However, for a very long time, absolutely no intuition could be attached to terms like $a \Leftarrow b \Leftarrow a$, and virtually every result involving higher-order functions in conjunction with continuations had to be derived by "cranking the handle" of the semantics. This provoked a strong requirement for direct implementability, and an SLC interpreter (in Scheme) and a type inferencer (in Prolog) have existed almost from the beginning.

On the other hand, the SCL and the entire axiomatic presentation is a comparatively recent addition, obtained as the result of a long search for the "underlying category" of the SLC. While the categorical concept of duality was the guiding light during the initial development of SLC, the imperfections of products and exponentials in CBV evaluation defied all attempts of formalization for a long time. The breakthrough idea of a strategy-independent core and a strategy-dependent specialization based on initial and terminal objects took surprisingly long to mature.

The applications suggested in chapter 4 also date back to the early days, although they have been refined somewhat lately. In particular, design and efficient implementation of a symmetrical programming language possibly including inheritance was the original goal of this thesis. The SLC was only intended as a throwaway notation, to try out a few ideas before moving on to more important and fundamental problems. Gradually, however, the deep semantic implications of having directly accessible continuations in a language became apparent, and this thesis resulted.

Let the author be the first to point out that a number of loose ends remain; while it is hoped that these problems can be solved in a way compatible with the present framework, it may become necessary to backtrack slightly in order to obtain a coherent presentation of the whole theory.

One remaining task is to develop a complete set of strategy-independent conversion rules for the SLC. While the denotational semantics is completely unambiguous about the equality of two

terms, it is connected intimately with a particular evaluation order, which makes a formal proof of equivalence with the SCL hard to express.

Further, the axioms of SCL are *conversion* rather than *reduction* rules. Thus, there is no well-defined concept of normal form, and deciding strategy-independent equality of even non-recursive terms amounts to general theorem proving. Maybe an automated procedure like Knuth-Bendix completion could be used to obtain a set of directed rewrite rules, but some axioms, notably that of morphism composition are so inherently bidirectional, that a more general concept of reduction may be necessary. Anyway, the present axiom set is rather ad-hoc in nature, and it would be desirable to find some underlying pattern.

Finally, the whole area of recursive types has not received the proper treatment it deserves. In particular, the implicit equality between a type and its recursive definition will probably have to be replaced by a named isomorphism. Moreover, the existence of recursive values and continuations of a given type seems closely connected with the existence of initial and final fixpoints of the type-defining functor [Manes & Arbib 86, chapt. 10,11], and this will have to be investigated.

Yet none of these points, important though they may be, undermine the basic theme of this thesis: that first-class continuations are not only categorically sound, but can be seen as a powerful *declarative* tool exposing the deep structure and symmetry of many common language constructs, as well as suggesting new ones. If the present work has convinced the reader that the beauty of category theory and the truth (not beast!) of continuation semantics can be reconciled, it has served its purpose.

Bibliography

[ANSI 78] *American National Standard: Programming Language FORTRAN.* American National Standards Institute (1978). ANSI Std. X3.9-1978.

[ANSI 83] *The Programming Language ADA Reference Manual.* American National Standards Institute (1983). ANSI/MIL-STD-1815A-1983, LNCS 155.

[Appel & Jim 89] Andrew W. Appel and Trevor Jim: *Continuation-Passing, Closure-Passing Style.* In Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 293–302, Austin, Texas (January 1989).

[Arbib & Manes 75] Michael A. Arbib and Ernest G. Manes: *Arrows, Structures and Functors: The Categorical Imperative.* Academic Press, London (1975).

[Backus 78] John Backus: *Can Programming be Liberated from the von Neumann Style?* Communications of the ACM, 21:613–641 (1978).

[Baker 79] Henry G. Baker, Jr.: *Shallow Binding in LISP 1.5.* In Artificial Intelligence: An MIT Perspective, Vol. 2, pp. 375–387, MIT Press (1979).

[Blikle & Tarlecki 83] Andrzej Blikle and Andrzej Tarlecki: *Naive Denotational Semantics.* In Information Processing 83, R. E. A. Mason (ed.), pp. 345–355, IFIP (1983).

[Cardelli 84] Luca Cardelli: *A Semantics of Multiple Inheritance.* In Proceedings of International Symposium on the Semantics of Data Types, pp. 51–67 (1984). LNCS 173.

[Curien 86] P-L. Curien: *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science, Vol. 1, Pitman (1986).

[Dahl *et al.* 70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard: *Simula: Common Base Language.* Norwegian Computing Center (October 1970).

[Damas & Milner 82] L. Damas and Robin Milner: *Principal Type-schemes for Functional Languages.* In Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages, pp. 207–212 (January 1982).

[Danvy & Filinski 89] Olivier Danvy and Andrzej Filinski: *A Functional Abstraction of Typed Contexts.* DIKU Rapport 89/12, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (July 1989).

[Dijkstra 68] Edger W. Dijkstra: *Goto Statement Considered Harmful.* Communications of the ACM, 11(3) (March 1968).

[Felleisen 88] Matthias Felleisen: *The Theory and Practice of First-Class Prompts.* In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 180–190, San Diego, California (January 1988).

[Filinski 89] Andrzej Filinski: *Declarative Continuations: An Investigation of Duality in Programming Language Semantics.* In Proceedings of Summer Conference on Category Theory and Computer Science, Manchester, U.K. (September 1989). To appear in LNCS.

[Friedman & Wise 76] Daniel P. Friedman and David S. Wise: *CONS Should Not Evaluate its Arguments.* In Automata, Languages, and Programming, S. Michaelson and R. Milner (eds.), pp. 257–284, Edinburgh University Press, Edinburgh, Scotland (1976).

[Girard 87] Jean-Yves Girard: *Linear Logic.* Theoretical Computer Science, 50:1–102 (1987).

[Goldberg & Robson 83] Adele Goldberg and David Robson: *Smalltalk-80: The Language and its Implementation.* Addison-Wesley (1983).

[Gries 81] David Gries: *The Science of Programming.* Texts and Monographs in Computer Science, Springer-Verlag (1981).

[Hagino 87a] Tatsuya Hagino: *A Categorical Programming Language.* PhD thesis, University of Edinburgh, Edinburgh, Scotland (1987). ECS-LFCS-87-38.

[Hagino 87b] Tatsuya Hagino: *A Typed Lambda Calculus with Categorical Type Constructors.* In Proc. Summer Conference on Category Theory and Computer Science, pp. 141–156, Edinburgh (1987). LNCS 283.

[Harper *et al.* 88] Robert Harper, Robin Milner, and Mads Tofte: *The Definition of Standard ML, Version 2.* Report ECS-LFCS-88-62, University of Edinburgh, Edinburgh, Scotland (August 1988).

[Hewitt 79] Carl Hewitt: *Control Structure as Patterns of Passing Messages.* In Artificial Intelligence: An MIT Perspective, Vol. 2, pp. 434–465, MIT Press (1979).

[Hewitt *et al.* 74] Carl Hewitt *et al.*: *Behavioral Semantics of Nonrecursive Control Structures.* In Proceedings of Programming Symposium, B. Robinet (ed.), pp. 385–407, Paris, France (April 1974). LNCS 19.

[Hughes 87] John Hughes: *Analysing Strictness by Abstract Interpretation of Continuations.* In Abstract Interpretation of Declarative Languages, Samson Abramsky and Chris Hankin (eds.), chapter 4, Ellis Horwood (1987).

[Hughes 88] John Hughes: *Backwards Analysis of Functional Programs.* In Partial Evaluation and Mixed Computation, D. Bjørner, A. P. Ershov, and N. D. Jones (eds.), pp. 187–208, North-Holland (1988).

[Huwig & Poigné 86] Hagen Huwig and Axel Poigné: *A Note on Inconsistencies Caused by Fixpoints in a Cartesian Closed Category* (1986). Unpublished manuscript.

[Iverson 62] K. E. Iverson: *A Programming Language.* John Wiley and Sons (1962).

[Jensen & Wirth 74] Kathleen Jensen and Niklaus Wirth: *PASCAL: User Manual and Report.* Springer-Verlag (1974). LNCS 18.

[Lafont 88] Yves Lafont: *The Linear Abstract Machine.* Theoretical Computer Science, 59:157–180 (1988).

[Lambek & Scott 86] Joachim Lambek and P.J. Scott: *Introduction to Higher Order Categorical Logic.* Cambridge studies in advanced mathematics, Vol. 7, Cambridge University Press (1986).

[MacLane 71] Saunders MacLane: *Categories for the Working Mathematician.* Graduate Texts in Mathematics, Vol. 5, Springer-Verlag (1971).

[Manes & Arbib 86] Ernest G. Manes and Michael A. Arbib: *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science, Springer-Verlag (1986).

[McCarthy *et al.* 62] John McCarthy *et al.*: *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts (1962).

[Moggi 89] Eugenio Moggi: *Computational Lambda-calculus and Monads.* In Proceedings of 4th Conference on Logic in Computer Science, pp. 14–23, IEEE (1989).

[Naur 62] Peter Naur (ed.): *Revised Report on the Algorithmic Language Algol 60.* Communications of the ACM, 6(1):1–17 (1962).

[Peyton Jones 87] Simon L. Peyton Jones: *The Implementation of Functional Programming Languages.* Prentice-Hall (1987).

[Plotkin 75] Gordon D. Plotkin: *Call-by-name, Call-by-value and the λ-calculus.* Theoretical Computer Science, 1:125–159 (1975).

[Poigné 83] Axel Poigné: *On Semantic Algebras: Higher Order Structures* (1983). Unpublished manuscript.

[Rees & Clinger 86] Jonathan Rees and William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme.* SIGPLAN Notices, 21(12):37–79 (December 1986).

[Reynolds 72] John C. Reynolds: *Definitional Interpreters for Higher-Order Programming Languages.* In Proceedings 25th ACM National Conference, pp. 717–740, New York (1972).

[Rydeheard & Burstall 88] David E. Rydeheard and Rod M. Burstall: *Computational Category Theory.* Prentice Hall International Series in Computer Science, Prentice Hall (1988).

[Sato & Tamaki 89] Taisuke Sato and Hisao Tamaki: *Existential Continuation.* New Generation Computing, 6:421–438 (1989).

[Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt: *An Introduction to Trellis/Owl.* In Proceedings of OOPSLA'86, pp. 9–16 (1986).

[Schmidt 86] David A. Schmidt: *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc. (1986).

[Sethi & Tang 80] Ravi Sethi and Adrian Tang: *Constructing Call-by-Value Continuation Semantics.* Journal of the ACM, 27(3):580–597 (July 1980).

[Stoy 77] Joseph E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press (1977).

[Strachey & Wadsworth 74] Christopher Strachey and Christopher P. Wadsworth: *Continuations: A Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1974).

[Stroustrup 86] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley (1986).

[Weis *et al.* 87] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Ascànder Suárez: *The CAML Reference Manual, Version 2.5*. INRIA-ENS, Paris, France (December 1987).

Appendix A

An Implementation of SLC/SCL

This chapter contains listings of a model implementation of the SLC in the CAML dialect [Weis *et al.* 87] of ML. Note that the name (the letters CAM stand for for Categorical Abstract Machine) refers to the implementation of the language, not to any categorical features in it. The only somewhat non-standard facility we will use is an interface to the YACC parser generator, and even then most of the parsing is done explicitly, because the machine-produced parser is too limited.

It should be noted that the aim of this implementation has been simplicity, not efficiency. Also, some of the support code (e.g., parser, unification algorithm) have been implemented in a rather quick-and-dirty way, and are currently being rewritten to improve the structure. Most of the actual SLC-related code should correspond directly to the equations in the thesis, and thus are rather sparsely commented.

A.1 Concrete syntax and pre-parsing

A major obstacle to writing language processors in ML is that the abstract syntax becomes very verbose. Therefore, a parser of some kind is essential for entering terms larger than a few lines. The CAML system provides an interface to the YACC parser generator, which permits use of concrete syntax for entering programs.

Unfortunately, the built-in (and non-replaceable) lexical analyzer cannot distinguish between different kinds of identifiers, *i.e.*, value and continuation variables. Such a distinction is essential for parsing an SLC program, since the correct coercions must be inserted, *e.g.*, whenever an identifier is used as a function.

We therefore split the parsing process into two phases, so that the YACC-based parser only converts an input string to a simple tree-based representation of it, according to operator precedences and associativity. This is then processed further by a tree parser written directly in CAML. The concrete syntax is parsed according to the following grammar:

```
/* <= => rec */
%token LEFT RIGHT REC
%token NUM IDENT
%right LEFT RIGHT
%nonassoc '='
%left '+' '-'
%left '*'
%left '^'
%right '?'
%%
```

```
form    : sform         {$1}
        | num           {sNum($1)}
        | form '+' form {sAdd($1,$3)}
        | form '-' form {sSub($1,$3)}
        | form '*' form {sMul($1,$3)}
        | form '^' form {sUp($1,$3)}
        | form '?' form {sDwn($1,$3)}
        | form '=' form {sEql($1,$3)}
        | '[' forms ']' {sSqb($2)}
        | '<' forms '>' {sAnb($2)}
        | sform LEFT form {sLft($1,$3)}
        | sform RIGHT form  {sRgt($1,$3)}
        | REC sform '=' form {sRec($2,$4)}
        ;

sform   : ide           {sIde($1)}
        | '(' forms ')' {sPar($2)}
        | '{' forms '}' {sBrc($2)}
        ;

forms   : /* empty */   {[]}
        | flist         {$1}
        ;

flist   : form          {[$1]}
        | form ',' flist {$1 :: $3}
        ;

num     : NUM           {$1}
        ;

ide     : IDENT         {$1}
        ;

%%
```

The syntactic domain `form` associated with this "pre-abstract syntax" is given by:

```
(* form.ml: syntactic forms recognized by the YACC pre-parser *)

type form = sNum of num
          | sIde of string
          | sAdd of form&form
          | sSub of form&form
          | sMul of form&form
          | sUp  of form&form
          | sDwn of form&form
          | sEql of form&form
          | sLft of form&form
          | sRgt of form&form
          | sRec of form&form
          | sPar of form list
          | sBrc of form list
          | sSqb of form list
```

```
          | sAnb of form list;;
```

A.2 Abstract syntax and parsing

The concrete syntax could be used for both SLC and SCL. However, only a parser for SLC exists at present. The SLC abstract syntax is given by the following ML datatype:

```
(* slc.ml: abstract syntax of the SLC *)

type var == string;;

type cst = cInt of num;;

type rec
    exp = eCst of cst
        | eIde of var
        | eUnit
        | ePair of exp&exp
        | eUp  of fnc&exp
        | eRec of pax&exp
        | eFnc of fnc
and
    cnt = cIde of var
        | cZero
        | cCase of cnt&cnt
        | cDwn of cnt&fnc
        | cRec of pay&cnt
        | cFnc of fnc
and
    fnc = fRgt of pax&exp
        | fLft of pay&cnt
        | fPrm of var
        | fExp of exp
        | fCnt of cnt
and
    pax = xVar of var
        | xPair of pax&pax
        | xUnit
and
    pay = yVar of var
        | yCase of pay&pay
        | yZero;;
```

A tree-based parser converts from the pre-abstract syntax to the abstract one. A few aspects of this transformation are not yet handled quite properly. In particular, the CBV view of recursive functions as continuations is embedded in the parsing algorithm; a more general approach would be preferable.

```
(* parse.ml: parser for SLC terms (CBV recursion) *)

#use "form";;
#use "slc";;

type varkind = kVal | kCnt ;;
```

```
let rec parseE =
    fun (sNum n) r -> eCst (cInt n)
    |   ((sIde v) as s) r ->
           (case r v of kVal -> eIde v | kCnt -> eFnc (parseF s r))
    |   (sAdd (n,m)) r -> eUp ((fPrm "+"), ePair ((parseE n r),(parseE m r)))
    |   (sSub (sNum 0,m)) r ->
           eUp ((fPrm "~"), (parseE m r))
    |   (sSub (n,m)) r -> eUp ((fPrm "-"), ePair ((parseE n r),(parseE m r)))
    |   (sMul (n,m)) r -> eUp ((fPrm "*"), ePair ((parseE n r),(parseE m r)))
    |   (sEql (n,m)) r -> eUp ((fPrm "="), ePair ((parseE n r),(parseE m r)))
    |   (sUp (f,e)) r -> eUp ((parseF f r),(parseE e r))
    |   (sPar []) r -> eUnit
    |   (sPar [e]) r -> parseE e r
    |   (sPar [e1;e2]) r -> ePair ((parseE e1 r),(parseE e2 r))
    |   (sBrc [f]) r -> eFnc (parseF f r)
    |   ((sLft _) as s) r -> eFnc (parseF s r)
    |   ((sRgt _) as s) r -> eFnc (parseF s r)
    |   ((sDwn _) as s) r -> eFnc (parseF s r)
    |   ((sRec _) as s) r -> eFnc (parseF s r)
    |    _ -> failwith "bad E syntax"
and     parseC =
    fun ((sIde v) as s) r ->
           (case r v of kVal -> cFnc (parseF s r) | kCnt -> cIde v)
    |   (sDwn (c,f)) r -> cDwn ((parseC c r),(parseF f r))
    |   (sBrc []) r -> cZero
    |   (sBrc [c]) r -> parseC c r
    |   (sBrc [c1;c2]) r -> cCase ((parseC c1 r),(parseC c2 r))
    |   (sRec (y,c)) r -> let (r1,py) = parseY y r in cRec (py,parseC c r1)
    |   (sPar [f]) r -> cFnc (parseF f r)
    |   ((sLft _) as s) r -> cFnc (parseF s r)
    |   ((sRgt _) as s) r -> cFnc (parseF s r)
    |   ((sUp  _) as s) r -> cFnc (parseF s r)
    |   _ -> failwith "bad C syntax"
and     parseF =
    fun (sRgt (x,e)) r -> let (r1,px) = parseX x r in fRgt (px,parseE e r1)
    |   (sLft (y,c)) r -> let (r1,py) = parseY y r in fLft (py,parseC c r1)
    |   (sPar [f]) r -> parseF f r
    |   (sIde v) r ->
           (case r v of kVal -> fExp (eIde v) | kCnt -> fCnt (cIde v))
    |   ((sUp _) as s) r -> fExp (parseE s r)
    |   ((sDwn _) as s) r -> fCnt (parseC s r)
    |   ((sRec _) as s) r -> fCnt (parseC s r)
    |   _ -> failwith "bad F syntax"
and     parseX =
    fun (sIde v) r -> ((fun x->if x=v then kVal else r x), (xVar v))
    |   (sPar []) r -> (r, xUnit)
    |   (sPar [x]) r -> parseX x r
    |   (sPar [x1;x2]) r ->
          let (r1,p1) = parseX x1 r in let (r2,p2) = parseX x2 r1 in
            (r2, xPair (p1,p2))
    |   _ -> failwith "bad X syntax"
and     parseY =
    fun (sIde v) r -> ((fun y->if y=v then kCnt else r y), (yVar v))
    |   (sBrc []) r -> (r, yZero)
```

```
    |   (sBrc [y]) r -> parseY y r
    |   (sBrc [y1;y2]) r ->
          let (r1,p1) = parseY y1 r in let (r2,p2) = parseY y2 r1 in
            (r2, yCase (p1,p2))
    |   _ -> failwith "bad Y syntax";;
```

A.3 Translation from SLC to SCL

The abstract syntax for SCL mirrors figure 3.1, extended with the fixpoints of section 3.7.2:

```
(* scl.ml: abstract syntax of the SCL *)

type rec morph =
    lCst of cst                 (* cst *)
|   lVal of var                 (* x *)
|   lCnt of var                 (* y *)
|   lPrm of var                 (* p *)

|   lId                         (* id *)
|   lCmp of morph&morph         (* f o g *)

|   lDia                        (* <> *)
|   lPair of morph&morph        (* <f,g> *)
|   lPr1                        (* pr1 *)
|   lPr2                        (* pr2 *)

|   lSqr                        (* [] *)
|   lCase of morph&morph        (* [f,g] *)
|   lIn1                        (* in1 *)
|   lIn2                        (* in2 *)

|   lCur of morph               (* f^* *)
|   lAp                         (* ap *)
|   lPhi                        (* phi *)

|   lCcr of morph               (* f_* *)
|   lPa                         (* pa *)
|   lTta                        (* theta *)

|   lFix                        (* fix *)
|   lXif                        (* xif *)
;;
```

The translator is derived directly from the abstraction rules of figure 3.4 and the translation equations of figure 3.5:

```
(* trans.ml: translator from SLC to SCL *)

#use "slc";;
#use "scl";;

let rec occurs v =
    fun (lVal x) -> x=v
    |   (lCnt y) -> y=v
    |   (lCmp (f,g)) -> (occurs v f) or (occurs v g)
```

```
    |   (lPair (f,g)) -> (occurs v f) or (occurs v g)
    |   (lCase (f,g)) -> (occurs v f) or (occurs v g)
    |   (lCur f) -> occurs v f
    |   (lCcr f) -> occurs v f
    |   _ -> false;;

let alphav = lPair(lCmp(lPr1,lPr1),lPair(lCmp(lPr2,lPr1),lPr2))
and alphac = lCase(lCmp(lIn1,lIn1),lCase(lCmp(lIn1,lIn2),lIn2));;

let gammav = lPair(lPr2,lPr1)
and gammac = lCase(lIn2,lIn1);;

let deltav = lCmp(lAp,lPair(lCmp(lCase(lCur(lCmp(lIn1,gammav)),
                                       lCur(lCmp(lIn2,gammav))),lPr2),lPr1))
and deltac = lCmp(lCase(lCmp(lIn2,lPair(lCcr(lCmp(gammac,lPr1)),
                                        lCcr(lCmp(gammac,lPr2)))),lIn1),lPa);;
let rec absv x f =
    if occurs x f then
       case f of
          (lVal x1) -> lPr1 (*x1=x*)
       |  (lCmp (f,g)) -> lCmp (absv x f, lPair(lPr1, absv x g))
       |  (lPair (f,g)) -> lPair (absv x f, absv x g)
       |  (lCase (f,g)) -> lCmp (lCase(absv x f, absv x g), deltav)
       |  (lCur f) -> lCur (lCmp (absv x f, alphav))
       |  (lCcr f) -> lCmp (lCcr (absv x f), lPhi)
       |  _ -> failwith "absv: cannot abstract"
    else
       lCmp(f, lPr2)
and absc y f =
    if occurs y f then
       case f of
          (lCnt y1) -> lIn1 (*y1=y*)
       |  (lCmp (f,g)) -> lCmp (lCase(lIn1, absc y f), absc y g)
       |  (lPair (f,g)) -> lCmp (deltac, lPair(absc y f, absc y g))
       |  (lCase (f,g)) -> lCase (absc y f, absc y g)
       |  (lCur f) -> lCmp (lTta, lCur (absc y f))
       |  (lCcr f) -> lCcr (lCmp (alphac, absc y f))
       |  _ -> failwith "absc: cannot abstract"
    else
       lCmp(lIn2, f);;

let rec transE =
    fun (eCst a) -> lCst a
    |   (eIde x) -> lVal x
    |   (eUnit) -> lId (* = lDia *)
    |   (ePair (e1,e2)) -> lPair(transE e1, transE e2)
    |   (eUp (f,e)) -> lCmp (transF f, transE e)
    |   (eRec (x,e)) -> lCmp(lFix, transE (eFnc (fRgt (x,e))))
    |   (eFnc f) -> lCur (lCmp(transF f, lPr2))
and transC =
    fun (cIde y) -> lCnt y
    |   (cZero) -> lId (* = lSqr*)
    |   (cCase (c1,c2)) -> lCase(transC c1, transC c2)
    |   (cDwn (c,f)) -> lCmp(transC c, transF f)
```

```
    |   (cRec (y,c)) -> lCmp(transC (cFnc (fLft (y,c))), lXif)
    |   (cFnc f) -> lCcr (lCmp(lIn2, transF f))
and transF =
    fun (fPrm p) -> lPrm p
    |   (fRgt (x,e)) -> lCmp(transX (transE e) x, lPair(lId,lDia))
    |   (fLft (y,c)) -> lCmp(lCase(lId,lSqr), transY (transC c) y)
    |   (fExp e) -> lCmp(lAp, lPair(lCmp(transE e, lDia), lId))
    |   (fCnt c) -> lCmp(lCase(lCmp(lSqr, transC c), lId), lPa)
and transX f =
    fun (xVar x) -> absv x f
    |   (xUnit) -> lCmp(f, lPr2)
    |   (xPair (x1,x2)) -> lCmp(transX (transX f x2) x1, alphav)
and transY f =
    fun (yVar y) -> absc y f
    |   (yZero) -> lCmp(lIn2, f)
    |   (yCase (y1,y2)) -> lCmp(alphav, transY (transY f y2) y1);;
```

A.4 Type inferencing

While all SLC examples in the thesis have been simply typed, a practical implementation should probably support type polymorphism, so that it would not be necessary, *e.g.*, to define a new identity function for every type. The type system in the model implementation is therefore polymorphic, although the semantic consequences of this have not yet been fully investigated. It does not, however, allow recursively-defined types. SLC programs involving such types, *e.g.*lists, must currently be executed without type checking.

The actual type inferencer uses a number of auxiliary functions. Most importantly, the function `unify` implements a simple but rather inefficient unification algorithm. It returns the MGU of two terms or fails when unification is not possible. Building on this, a "resolution theorem prover" `solve` instantiates the variables in a deterministically-built "proof tree". Finally, a term generalizer `gen` converts a term with possibly unbound variables into a "normal form", from which new instances with renamed variables can be made:

```
(* unify.ml: unification and resolution theorem prover *)

type rec term =
  tVar of num
| tFun of string&(term list)
and subst == (num->term);;

let emptysubst = tVar;;

let rec unify (x1,x2) =
    if x1=x2 or x1=tVar 0 or x2=tVar 0 then emptysubst
    else case (x1,x2) of
             ((tVar x1), t2) -> bind x1 t2
           | (t1, (tVar x2)) -> bind x2 t1
           | (tFun(f1,l1), tFun(f2,l2)) ->
                if f1=f2 then unifylist (l1,l2) else failwith "functor"
and unifylist =
    fun ([],[]) -> emptysubst
    |   (x1::l1, x2::l2) -> let sl=unifylist (l1,l2) in
          let s=unify(substitute sl x1, substitute sl x2) in
             (substitute s) o sl
```

```
    |   _ -> failwith "arity"
and bind x t =
    if occurs x t then failwith "occur"
    else (fun x1->if x1=x then t else tVar x1)
and substitute s =
    fun (tVar x) -> s x
    |   (tFun (f,l)) -> tFun(f, map (substitute s) l)
and occurs x =
    fun (tVar x1) -> x1=x
    |   (tFun (_,l)) -> foldor (occurs x) l
and foldor f =
    fun [] -> false
    |   (h::t) -> (f h) or (foldor f t);;

let instance n = substitute (fun k->if k=0 then tVar 0 else tVar (k+n));;

let tCst t = tFun(t, [])
and V_ = tVar 0
and V1 = tVar 1
and V2 = tVar 2
and V3 = tVar 3
and V4 = tVar 4
and V5 = tVar 5;;

type rec ProofTree = pt of num & term & ((ProofTree&term) list);;

let rec solve (pt(n, t1, l)) t x s =
    let s1=(substitute (unify(instance x t1, substitute s t))) o s in
        let l1=map (fun (p,t)->(p,(instance x t))) l in
            solvelist l1 (x+n) s1
and solvelist l x s = case l of
      [] -> (s, x)
    | ((p,t)::tr) -> let (s1,x1)=solve p t x s in solvelist tr x1 s1;;

(* generalize term to clause *)

let rec addl x =
    fun [] -> (1,[x])
    |   (h::t) -> if x=h then (1,h::t) else let (n,t1)=addl x t in (n+1,h::t1)
and gen vl =
    fun (tVar v) -> let (n,vl1) = addl v vl in (vl1,tVar n)
    |   (tFun (f,a)) -> let (vl1,a1) = genl vl a in (vl1,tFun(f,a1))
and genl vl =
    fun [] -> (vl,[])
    |   (h::t) -> let (vl1,h1)=gen vl h in let (vl2,t1)=genl vl1 t in
          (vl2,h1::t1);;
```

The type system itself is common for SLC and SCL. It is given together with two functions for converting between types and general terms:

```
(* type.ml: SLC/SCL type system *)

#use "unify";;

type rec Type =
```

```
     tVV of num
|    tBas of string
|    tUnit
|    tProd of Type&Type
|    tExp of Type&Type
|    tZero
|    tCoprod of Type&Type
|    tCoexp of Type&Type;;

let rec tterm =
    fun (tVV n) -> tVar n
    |   (tBas t) -> tFun(t, [])
    |   (tUnit) -> tFun("unit", [])
    |   (tProd (t1,t2)) -> tFun("*", [tterm t1; tterm t2])
    |   (tExp (t1,t2)) -> tFun("->", [tterm t1; tterm t2])
    |   (tZero) -> tFun("zero", [])
    |   (tCoprod (t1,t2)) -> tFun("+", [tterm t1; tterm t2])
    |   (tCoexp (t1,t2)) -> tFun("<-", [tterm t1; tterm t2]);;

let rec termt =
    fun (tVar n) -> (tVV n)
    |   (tFun ("unit", [])) -> tUnit
    |   (tFun ("*", [t1;t2])) -> tProd (termt t1,termt t2)
    |   (tFun ("->", [t1;t2])) -> tExp (termt t1,termt t2)
    |   (tFun ("zero", [])) -> tZero
    |   (tFun ("+", [t1;t2])) -> tCoprod (termt t1, termt t2)
    |   (tFun ("<-", [t1;t2])) -> tCoexp (termt t1, termt t2)
    |   (tFun (b, [])) -> tBas b
    |   (tFun (s, _)) -> failwith "termt: unknown type constructor "^s;;
```

The SLC type inferencer implements the rules of figure 2.2, extended with recursively-defined expressions and continuations. In principle, it s quite simple, but the many explicit unifications make it somewhat harder to read than an equivalent version in Prolog. The types are written directly as unifiable terms to avoid the extra conversions; they would make the result even harder to read.

```
(* slctype.ml: type inferencer for SLC terms *)

#use "slc";;
#use "unify";;

let bt (n, t, l) = pt(n+1, tFun("|-", tVar(n+1)::t),
                     map (fun (x,y)->(x,tFun("|-", tVar(n+1)::y))) l);;

let ext r v =
    fun i->if i=v then (1, tFun(",", [V1;V_]), V1)
    else let (n,G,T)=r i in (n, tFun(",", [V_;G]), T);;

let rec typeE e r = case e of
       (eCst (cInt n)) -> bt(0, [tCst "int"], [])
   |   (eIde x) -> let (n,G,T)=r x in pt(n, tFun("|-", [G;T]), [])
   |   (eUnit) -> bt(0, [tCst "unit"], [])
   |   (ePair (e1,e2)) ->
          bt(2, [tFun("*", [V1;V2])], [typeE e1 r,[V1]; typeE e2 r,[V2]])
   |   (eUp (f,e)) -> bt(2, [V2], [typeF f r,[V1;V2]; typeE e r,[V1]])
```

```
    |   (eFnc f) -> bt(2, [tFun("->",[V1;V2])], [typeF f r,[V1;V2]])
    |   (eRec (x,e)) ->
          pt(3, tFun("|-", [V1;V2]), let (r1,ptx) = typeX x r in
           [ptx, tFun("|:", [V1;V2;V3]); typeE e r1, tFun("|-", [V3;V2])])
and typeC c r = case c of
       (cIde y) -> let (n,G,T)=r y in pt(n, tFun("|-", [G;T]), [])
    |   (cZero) -> bt(0, [tCst "zero"], [])
    |   (cCase (c1,c2)) ->
           bt(2, [tFun("+", [V1;V2])], [typeC c1 r,[V1]; typeC c2 r,[V2]])
    |   (cDwn (c,f)) -> bt(2, [V1], [typeF f r,[V1;V2]; typeC c r,[V2]])
    |   (cFnc f) -> bt(2, [tFun("<-",[V2;V1])], [typeF f r,[V1;V2]])
    |   (cRec (y,c)) ->
          pt(3, tFun("|-", [V1;V2]), let (r1,pty) = typeY y r in
            [pty, tFun("|:", [V1;V2;V3]); typeC c r1, tFun("|-", [V3;V2])])
and typeF f r = case f of
       (fPrm "+") ->
          bt(0, [tFun ("*", [tCst "int"; tCst "int"]); tCst "int"], [])
    |   (fPrm "-") ->
          bt(0, [tFun ("*", [tCst "int"; tCst "int"]); tCst "int"], [])
    |   (fPrm "*") ->
          bt(0, [tFun ("*", [tCst "int"; tCst "int"]); tCst "int"], [])
    |   (fPrm "=") ->
          bt(1, [tFun ("*", [V1;V1]); tFun("+",[tCst "unit";tCst "unit"])], [])
    |   (fRgt (x,e)) ->
          let (r1,ptx) = typeX x r in
           pt(4, tFun("|-", [V1; V2; V3]),
            [ptx,tFun("|:", [V1;V2;V4]); typeE e r1,tFun("|-", [V4; V3])])
    |   (fLft (y,c)) ->
          let (r1,pty) = typeY y r in
           pt(4, tFun("|-", [V1; V2; V3]),
            [pty,tFun("|:", [V1;V3;V4]); typeC c r1,tFun("|-", [V4; V2])])
    |   (fExp e) -> bt(2, [V1;V2], [typeE e r,[tFun("->", [V1;V2])]])
    |   (fCnt c) -> bt(2, [V1;V2], [typeC c r,[tFun("<-", [V2;V1])]])
and typeX x r = case x of
       (xVar x) -> (ext r x, pt(2, tFun("|:", [V1;V2;tFun(",", [V2;V1])]), []))
    |   (xUnit) -> (r, pt(1, tFun("|:", [V1;tCst "unit";V1]), []))
    |   (xPair (x1,x2)) ->
          let (r1,pt1)=typeX x1 r in let(r2,pt2)=typeX x2 r1 in (r2,
           pt(5, tFun("|:", [V1;tFun("*", [V2;V3]);V4]),
            [pt1, tFun("|:", [V1;V2;V5]); pt2, tFun("|:", [V5;V3;V4])]))
and typeY y r = case y of
       (yVar y) -> (ext r y, pt(2, tFun("|:", [V1;V2;tFun(",", [V2;V1])]), []))
    |   (yZero) -> (r, pt(1, tFun("|:", [V1;tCst "zero";V1]), []))
    |   (yCase (y1,y2)) ->
          let (r1,pt1)=typeY y1 r in let(r2,pt2)=typeY y2 r1 in (r2,
           pt(5, tFun("|:", [V1;tFun("+", [V2;V3]);V4]),
            [pt1, tFun("|:", [V1;V2;V5]);  pt2, tFun("|:", [V5;V3;V4])]));;

let infer e gr =
    let (s1,_) = solve (typeE e (fun x->let (n,t)=gr x in (n,V_,t)))
                       (tFun ("|-", [V_; V1])) 1 emptysubst in
    let (l,t) = gen [] (s1 1) in
    (length l,t);;
```

A.5 Evaluation

The semantic domains for CBV evaluation are:

```
(* cbveval.ml: semantic domains for CBV evaluation *)

#use "type";;

type rec Val =
   vInt of num
|  vUnit
|  vPair of Val&Val
|  vIn1 of Val
|  vIn2 of Val
|  vClosr of Val->Cnt->Ans
|  vContx of Val&Cnt
and Cnt == Val->Ans
and Ans =
   aResult of Val
|  aError;;

type rvar =
   rVal of Val
|  rCnt of Cnt
|  rErr
and Env == var -> rvar
and TopEnv == var->(rvar & (num & Type));;

let ir = ref ((fun v -> (rErr,(0,tBas "err"))):TopEnv);;

let prim p v = case p of
       "+" -> let (vPair (vInt n, vInt m)) = v in vInt (n+m)
   |   "-" -> let (vPair (vInt n, vInt m)) = v in vInt (n-m)
   |   "*" -> let (vPair (vInt n, vInt m)) = v in vInt (n*m)
   |   "=" -> let (vPair (a, b)) = v in
                 if a=b then vIn1(vUnit) else vIn2(vUnit)
   |   "~" -> let (vInt n) = v in vInt (- n)
   |   _ -> failwith ("unknown primitive: "^p);;
```

The evaluator for SLC terms is derived directly from the semantic equations of figure 2.3:

```
(* CBV interpreter for SLC terms *)

#use "slc";;
#use "cbveval";;

let rec E_ e r k = case e of
       (eCst (cInt n)) -> k (vInt n)
   |   (eIde x) -> let (rVal v) = r x in k v
   |   (eUnit) -> k vUnit
   |   (ePair (e1,e2)) -> E_ e1 r (fun v1->E_ e2 r (fun v2->k (vPair (v1,v2))))
   |   (eUp (f,e)) -> E_ e r (fun v->F_ f r v k)
   |   (eRec (x,e)) -> aError
   |   (eFnc f) -> k (vClosr(fun v c->F_ f r v c))
and C_ c r v = case c of
```

```
       (cIde y) -> let (rCnt k) = r y in k v
   |   (cZero) -> aError
   |   (cCase (c1,c2)) ->
          (case v of (vIn1 a)->C_ c1 r a | (vIn2 b)->C_ c2 r b)
   |   (cDwn (c,f)) -> F_ f r v (fun t->C_ c r t)
   |   (cRec (y,c)) -> let rec k a=C_ c (Y_ y k r) a in k v
   |   (cFnc f) -> let (vContx(a,c)) = v in F_ f r a c
and F_ f r v k = case f of
       (fPrm p) -> k (prim p v)
   |   (fRgt (x,e)) -> E_ e (X_ x v r) k
   |   (fLft (y,c)) -> C_ c (Y_ y k r) v
   |   (fExp e) -> E_ e r (fun t->let (vClosr f)=t in f v k)
   |   (fCnt c) -> C_ c r (vContx(v,k))
and X_ x v r = case x of
       (xVar x) -> (fun x1->if x=x1 then rVal v else r x1)
   |   (xPair (x1,x2)) -> let (vPair(v1,v2)) = v in X_ x1 v1 (X_ x2 v2 r)
   |   (xUnit) -> let vUnit = v in r
and Y_ y k r = case y of
       (yVar y) -> (fun y1->if y=y1 then rCnt k else r y1)
   |   (yCase (y1,y2)) ->
         let k1=(k o vIn1) and k2=(k o vIn2) in Y_ y1 k1 (Y_ y2 k2 r)
   |   (yZero) -> r;;

let eval e =
    let (aResult a) = E_ e (fun x->let (v,_)=!ir x in v) aResult in a;;
```

Similarly, the SCL interpreter mirrors figure 3.6:

```
(* CBV interpreter for SCL terms *)

#use "scl";;
#use "cbveval";;

let rec M_ m v k = case m of
        (lCst (cInt n)) -> let vUnit=v in k (vInt n)
    |   (lVal x) -> let ((rVal a),_) = (!ir) x in k a
    |   (lCnt y) -> let ((rCnt c),_) = (!ir) y in c v
    |   (lPrm p) -> k (prim p v)

    |   (lId) -> k v
    |   (lCmp (f,g)) -> M_ g v (fun t-> M_ f t k)

    |   (lDia) -> k vUnit
    |   (lPair (f,g)) -> M_ f v (fun a->M_ g v (fun b->k (vPair (a,b))))
    |   (lPr1) -> let (vPair(a,b)) = v in k a
    |   (lPr2) -> let (vPair(a,b)) = v in k b

    |   (lSqr) -> aError
    |   (lCase (f,g)) -> (case v of (vIn1 a)->M_ f a k | (vIn2 b)->M_ g b k)
    |   (lIn1) -> k (vIn1 v)
    |   (lIn2) -> k (vIn2 v)

    |   (lCur f) -> k (vClosr(fun b c->M_ f (vPair (v,b)) c))
    |   (lAp) -> let (vPair (q,a))=v in let (vClosr f) = q in f a k
    |   (lPhi) -> let (vPair(x,q))=v in let (vContx (a,c))=q in
```

```
          k (vContx (vPair(x,a), c))

    |   (lCcr f) -> let (vContx(a,c))=v in
           M_ f a (fun t->case t of (vIn1 r)->k r | (vIn2 s)->c s)
    |   (lPa) -> k (vIn1 (vContx (v, (fun t->k (vIn2 t)))))
    |   (lTta) -> let (vClosr f) = v in
           k (vIn2 (vClosr (fun a c->
             f a (fun t->case t of (vIn1 r)->k (vIn1 r) | (vIn2 s) -> c s))))

    |   (lXif) -> let rec q a = k (vContx(a,q)) in q v;;

let evalm m = let (aResult a) = M_ m vUnit aResult in a;;
```

A.6 Top level interaction

Finally, the implementation contains a friendlier user interface for printing SLC types and values in a more traditional way. Unfortunately, version 2.5 of CAML does not allow user-defined concrete syntax for I/O, so the interaction must be based on the existing top-level evaluator, using auxiliary functions `z` and `zd` for evaluating and defining SLC terms respectively. Similarly, the SCL-based evaluator is invoked using functions `m` and `md`:

```
(* toplev.ml: top level interaction *)

#use "parse";;
#use "type";;
#use "slctype";;
#use "slcint";;
#use "trans";;
#use "sclint";;
load_syntax "slgram";;

let vtop r v = case r v of
   ((rVal _),_)->kVal
 | ((rCnt _),_)->kCnt
 | (rErr, _) -> failwith "Unknown identifier: "^v;;

let rec printv v =
    fun (tVV _) -> failwith "polymorphic value"
    |   (tBas "int") -> let (vInt n)=v in print_num n
    |   (tBas _) -> print_string "<Bas>"
    |   (tUnit) -> let vUnit = v in print_string("()")
    |   (tProd (t1,t2)) -> let (vPair (v1,v2)) = v in
           print_string "("; printv v1 t1; print_string ",";
           printv v2 t2; print_string ")"
    |   (tExp (t1,t2)) -> print_string "<clsr>"
    |   (tZero) -> failwith "zero-typed value"
    |   (tCoprod (t1,t2)) -> (case v of
             (vIn1 v1) -> print_string "(in1^";printv v1 t1;print_string ")"
           | (vIn2 v2) -> print_string "(in2^";printv v2 t2;print_string ")")
    |   (tCoexp (t1,t2)) -> print_string "<cntx>";;

let rec printt =
    fun (tVV n) -> print_string (extract_string "_ABCDEFGHIJKLMNOP" n n)
    |   (tBas s) -> print_string s
```

```
    |   (tUnit) -> print_string "unit"
    |   (tProd (t1,t2)) ->
          print_string "("; printt t1; print_string "*"; printt t2;
          print_string ")"
    |   (tExp (t1,t2)) ->
          print_string "["; printt t1; print_string "->"; printt t2;
          print_string "]"
    |   (tZero) -> print_string "null"
    |   (tCoprod (t1,t2)) ->
          print_string "("; printt t1; print_string "+"; printt t2;
          print_string ")"
    |   (tCoexp (t1,t2)) ->
          print_string "["; printt t1; print_string "<-"; printt t2;
          print_string "]";;

let prep s =
    let e = parseE s (vtop (!ir)) in
    let (n,t1)=infer e (fun x->let (_,(m,t))=!ir x in (m,tterm t)) in
    (e, (n, termt t1));;
let printvt v t = printv v t; print_string " : "; printt t; print_newline();;
let def x v nt =
    let oir = !ir in ir := (fun i->if i=x then (rVal v, nt) else oir i);;

let z s = let (e,(_,t))=prep s in let r=eval e in printvt r t
and zd x s = let (e,(n,t))=prep s in let r=eval e in
    def x r (n,t); print_string ("defined "^x^" = "); printvt r t
and m s = let (e,(_,t))=prep s in let r=evalm (transE e) in printvt r t
and md x s = let (e,(n,t))=prep s in let r=evalm (transE e) in
    def x r (n,t); print_string ("defined "^x^" = "); printvt r t;;
```

A.7 A sample session

Finally, let us give an example of an actual CAML session with the model implementation. We have used `z`/`zd` throughout, but `m`/`md` give identical results. It is even possible to mix the two interpreters freely, since the semantic domains for CBV evaluation are exactly the same.

```
   CAML (sun) (V 2-5) by INRIA Fri Jan 22

#echo_types false; echo_values false;
#load "toplev";;
/home/gevn/andrzej/thesis/caml/slgram_mly.lo loaded
/home/gevn/andrzej/thesis/caml/toplev.lo loaded
#
#zd "id" <<x=>x>>;;
defined id = <clsr> : [A->A]
#
#zd "prod" <<(f,g)=>(a,b)=>(f^a,g^b)>>;;
defined prod = <clsr> : [([A->B]*[C->D])->[(A*C)->(B*D)]]
#zd "curry" <<f=>a=>b=>f^(a,b)>>;;
defined curry = <clsr> : [[(A*B)->C]->[A->[B->C]]]
#zd "ap" <<(g,b)=>g^b>>;;
defined ap = <clsr> : [([A->B]*A)->B]
#
#zd "add" <<(a,b)=>a+b>>;;
```

```
defined add = <clsr> : [(int*int)->int]
#zd "pp" <<prod^(curry^add,id)>>;;
defined pp = <clsr> : [(int*A)->([int->int]*A)]
#z <<pp^(3,4)>>;;
(<clsr>,4) : ([int->int]*int)
#z <<ap^(pp^(3,4))>>;;
7 : int
#
#zd "sum" <<(f,g)=>{a,b}<={a?f,b?g}>>;;
defined sum = <clsr> : [([A->B]*[C->D])->[(A+C)->(B+D)]]
#zd "cocurry" <<f=>a<=b<={a,b}?f>>;;
defined cocurry = <clsr> : [[A->(B+C)]->[[C<-A]->B]]
#zd "pa" <<{g,b}<=b?g>>;;
defined pa = <clsr> : [A->([B<-A]+B)]
#
#zd "is3" <<x=>x=3>>;;
defined is3 = <clsr> : [int->(unit+unit)]
#zd "ss" <<sum^(cocurry^is3,id)>>;;
defined ss = <clsr> : [([unit<-int]+A)->(unit+A)]
#z <<pa^4>>;;
(in1^<cntx>) : ([A<-int]+A)
#z <<ss^(pa^3)>>;;
(in1^()) : (unit+unit)
#z <<ss^(pa^4)>>;;
(in2^()) : (unit+unit)
#
#zd "callcc" <<k<=k?(f=>f^(c<=k))>>;;
defined callcc = <clsr> : [[[A->B]->A]->A]
#z <<3+callcc^(f=>4+f^5)>>;;
8 : int
#
#zd "xif" <<f<=rec a=a?f>>;;
defined xif = <clsr> : [A->[A<-A]]
#zd "pf" <<f<=n=>(r<={r?(()=>1), r?(()=>n*f^(n-1))})^(n=0)>>;;
defined pf = <clsr> : [[int<-int]->[int<-int]]
#zd "fac" <<pf?xif>>;;
defined fac = <clsr> : [int->int]
#z <<fac^5>>;;
120 : int
#
#quit();;
A bientot ...
```