An Investigation of Abadi and Cardelli's Untyped Calculus of Objects

Master's Thesis[1]

Jacob Johannsen, 19990920
Department of Computer Science
University of Aarhus[2]

June 2, 2008

[1]Advisor: Olivier Danvy

[2]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: cnn@daimi.au.dk, jajocnn@gmail.com

Abstract

We study the relationship between the natural (big-step) semantics and the reduction (small-step) semantics of Abadi and Cardelli's untyped calculus of objects. By applying Danvy et al.'s functional correspondence to the natural semantics, we derive an abstract machine for this calculus, and by applying Danvy et al.'s syntactic correspondence to the reduction semantics, we also derive an abstract machines for this calculus. These two abstract machines are identical. The fact that the machines are identical, and the fact that they have been derived using meaning-preserving program transformations, entail that the derivation constitutes a proof of equivalence between natural semantics and the reduction semantics. The derivational nature of our proof contrasts with Abadi and Cardelli's soundness proof, which was carried out by pen and paper. We also note that the abstract machine is new.

To move closer to actual language implementations, we reformulate the calculus to use explicit substitutions. The reformulated calculus is new. By applying the functional and syntactic correspondences to natural and reduction semantics of this new calculus, we again obtain two abstract machines. These two machines are also identical, and as such, they establish the equivalence of the natural semantics and the reduction semantics of the new calculus.

Finally, we prove that the two abstract machines are strongly bisimilar. Therefore, the two calculi are computationally equivalent.

Dansk referat

Vi studerer sammenhængen mellem den naturlige semantik (big-step semantikken) og reduktions-semantikken (small-step semantikken) for Abadi og Cardellis typeløse objekt-kalkule. Ved anvendelse af Danvy o.a.'s funktionelle korrespondance på den naturlige semantik udleder vi en abstrakt maskine for denne kalkule, og ved anvendelse af Danvy o.a.'s syntaktiske korrespondance på reduktions-semantikken udleder vi ligeledes en abstrakt maskine. De to abstrakte maskiner er identiske. Det faktum, at maskinerne er identiske, samt det faktum, at de er blevet udledt ved brug af semantikbevarende programtransformationer, betyder, at udledningen udgør et bevis for, at den naturlige semantik og reduktions-semantikken er ækvivalente. Bevisets udledte natur står i kontrast til Abadi og Cardellis bevis for sundhed, som er blevet udført ved brug af pen og papir. Vi lægger også mærke til, at den abstrakte maskine er ny.

For at nærme os konkrete sprog-implementationer reformulerer vi kalkulen til at anvende eksplicitte substitutioner. Den reformulerede kalkule er ny. Ved anvendelse af den funktionelle og den syntaktiske korrespondance på den naturlige semantik og reduktions-semantikken denne nye kalkule, udleder vi igen to abstrakte maskiner. Disse to maskiner er ligeledes identiske, og etablerer dermed ækvivalensen mellem den naturlige semantik og reduktions-semantikken for den nye kalkule.

Til slut beviser vi, at de to abstrakte maskiner er stærkt bisimilære. Dermed er de to kalkuler beregningsmæssigt ækvivalente.

Acknowledgements

This dissertation, along with the subsequent defence, concludes my studies at the University of Aarhus. My work behind all of it would not have been possible, had it not been for the help and support of a number of people.

First and foremost, I am forever grateful to my thesis advisor, Olivier Danvy. His enthusiasm, his pedagogical insights and his scientific rigour and wisdom have been an inspiration throughout the work of this thesis, but most important of all, he has been a good friend. A day is never truly bad when he is around.

Extra special thanks with cream on the top (cherry optional) are also due to my good friend and former fellow student Erik Søe Sørensen, for continued inspiration and support during my entire studies, as well as for meticulous proofreading of this dissertation.

Thanks are also due to my personal friends Tatiana Barfod, Maria Kristensen, Aske Christensen, Thomas Jensen, and to my parents, Kirsten and Bjarne, my brother Claus and the rest of my family for their constant support and patience.

Finally, I would like to extend my grateful thanks to the technical, administrative, systems and library staff at DAIMI, in particular Ole Østerby, Michael Glad, Hanne Friis Jensen, Karen Kjær Møller, Ann Eg Mølhave, Ellen Kjemtrup Lindstrøm and Oksana Orlenko.

Jacob Johannsen (CNN)
Århus, June 2, 2008

Contents

List of Figures

Chapter 1

Introduction

This dissertation shows how natural semantics, abstract machines and reduction semantics for Abadi and Cardelli's untyped calculus of objects can be inter-derived using Danvy et al.'s functional and syntactic correspondences.

For many years, natural semantics, abstract machines and reduction semantics have been considered independent semantic formalisms. As a consequence, proving equivalence between semantic descriptions using different formalisms have been a tedious process, a fact which has made equivalence proofs unrealistic for anything but small-scale languages.

The functional correspondence allows for the inter-derivation of natural semantics and abstract machines, and the syntactic correspondence allows for the inter-derivation of reduction semantics and abstract machines. Together, these correspondences mediate between the three formalisms and provide not only a means to prove equivalence between the formalisms, but also a means to derive a semantic description in one formalism from a description in another formalism [2, 5, 13].

The functional and syntactic correspondences have previously been applied to several variations of the λ-calculus [13], including the λ-calculus with explicit substitutions [3] and the λ-calculus with effects [5, 8], and has given rise to a number of doctoral theses [3, 6, 9, 14, 23–25], so the methods are well established in the area of functional languages. However, as this dissertation shows, the correspondences can also be applied to object-oriented languages, which shows the robustness of the methodology.

Our choice of object-oriented language is the untyped ς-calculus as proposed by Abadi and Cardelli [1, Chapter 6]. The purpose of this calculus is to provide a basic computational model for the object-oriented paradigm, similar to the role that the λ-calculus fills for the functional paradigm.

First contribution: We apply the functional and syntactic correspondences to the ς-calculus, thereby deriving two abstract machines; one from the natural semantics using the functional correspondence, and one from the reduction semantics using the syntactic correspondence. The abstract machines are identical, which confirms the equivalence of the natural semantics and the reduction semantics given by Abadi and Cardelli. The derivational proof of equivalence (as opposed to a pen-and-paper proof), along with the derived abstract machine itself, constitute the first contribution of this thesis.

Second contribution: To move closer to realistic language implementations, we then present a new version of the ς-calculus, defined using *explicit substitutions* (the $\varsigma\rho$-calculus). We apply the functional and syntactic correspondences to the $\varsigma\rho$-calculus, thereby deriving two abstract machines for this new calculus. These two abstract machines are also identical, which proves the equivalence of the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus. The derivational proof of equivalence, along with the derived abstract machine, constitute the second contribution of this thesis.

Third contribution: The abstract machine for the ς-calculus and the abstract machine for the $\varsigma\rho$-calculus are strongly bisimilar, a fact for which we present a formal proof. Our bisimilarity result formally connects the ς-calculus and the $\varsigma\rho$-calculus, and shows that the calculi are computationally equivalent. Since the calculi are computationally equivalent, the $\varsigma\rho$-calculus provides a definition for an object-oriented computational model using explicit substitutions. The $\varsigma\rho$-calculus constitutes the third contribution of this thesis.

The structure of this dissertation: Part I presents two calculi of objects: Chapter 2 presents the ς-calculus (as described by Abadi and Cardelli), and Chapter 3 presents the $\varsigma\rho$-calculus. Chapter 4 concludes.

Part II presents the tools used to derive the abstract machines: Chapter 5 explains the functional correspondence, and Chapter 6 explains the syntactic correspondence. Chapter 7 concludes.

Part III contains the derivations and equivalence proofs between the semantic descriptions of the calculi. We derive the abstract machine for the ς-calculus in Chapter 8, and the abstract machine for the $\varsigma\rho$-calculus in Chapter 9. In Chapter 10, we present the formal connection between the two calculi, by proving bisimilarity between the derived abstract machines. Chapter 11 concludes.

Part IV concludes, and presents areas for further work.

Prerequisites We assume that the reader is familiar with the concepts of natural semantics (big-step semantics, interpreters as evaluation functions), reduction semantics (small-step semantics/one-step reductions, BNFs of terms and of reduction contexts, a notion of redex, evaluation by iterated reduction), abstract machines (state-transition functions, and initial, intermediate, and final states), and of bisimulation.

We also assume a basic familiarity with the SML programming language.

Terminology: In line with Plotkin [11], we use the term 'programming language' (or just 'language') to mean 'a calculus equipped with a reduction strategy (for small-step semantics) or an evaluation order (for big-step semantics)'.

We sometimes refer to the notion of 'a functional interpreter' or 'a functional evaluator', i.e., 'an interpreter (or evaluator, respectively) written in a functional programming language'.

As pointed out by Biernacka and Danvy, the word 'substitution' is overloaded [3]. In an attempt at clarity, we use the word 'substitution' to mean 'the action of substituting a value or term for a variable', and the term 'delayed substitutions' to refer to explicit substitutions, to stress that the substitutions have not been performed yet (and possibly never will be).

Typically, the word 'environment' is used to describe delayed substitutions only in connection with concrete implementations of languages, and the term 'explicit substitutions' is reserved for descriptions of syntax and semantics. Nevertheless, we use 'environment' to mean 'a collection of delayed substitutions', regardless of the context in which these delayed substitutions occur.

We use the word 'bisimilarity' (e.g., 'by bisimilarity between A and B') to state that there exists a bisimulation relating two entities (e.g., A and B). The word 'bisimulation' only refers to the actual relation defining the bisimilarity.

Context: An article based on this thesis is to appear in the LNCS proceedings of WoLLIC in July 2008 under the title "Inter-deriving Semantic Artifacts for Object-Oriented Programming", and is co-authored with Olivier Danvy [6].

Part I

Calculi

Chapter 2

The ς-calculus

This chapter introduces the untyped ς-calculus, as proposed by Abadi and Cardelli [1, Chapter 6].

We begin by presenting the syntax description of the calculus, as well as the description of values. The presentation includes an informal explanation of the intended meaning of each syntactic construct.

We then present the natural semantics and reduction semantics of the calculus, as defined by Abadi and Cardelli [1, Chapter 6]. We also include an informal description of the evaluation of a ς-term.

Finally, we briefly touch upon the subject of equivalence between natural semantics and reduction semantics.

2.1 Formal Definitions

This section presents the formal definitions of the ς-calculus. The formal definitions consist of a syntax definition for terms and values, a definition of a reduction semantics and a definition of a natural semantics for the calculus.

2.1.1 Syntax

In the ς-calculus, objects are the only values. An object is a collection of named methods. Names are labels, and all labels within an object must be distinct. Each method takes exactly one parameter, which represents self. The methods of an object can be updated. Here are the BNFs of terms and values in the ς-calculus:

$$\begin{array}{lrcl} (\mathsf{Term}) & t & ::= & x \mid [l = \varsigma(x)t,\ \ldots,\ l = \varsigma(x)t] \mid t.l \mid t.l \Leftarrow \varsigma(x)t \\ (\mathsf{Value}) & v & ::= & [l = \varsigma(x)t,\ \ldots,\ l = \varsigma(x)t] \end{array}$$

This grammar for terms defines the same language as in Abadi and Cardelli's book. It only differs in our choice of more uniform naming for non-terminals. The ellipses are only here for clarity: objects may contain any finite number of methods, including zero and one.

Fields do not exist directly in the calculus. They are modelled by methods that do not access self. Creation of a new object occurs only in case of object literals. In other words, classes and traits do not exist (but they can be modelled [1, pages 73-74]).

In addition to the formal syntax, we use parentheses to group subterms, to avoid ambiguity. Also, we often index a value with the number of methods it contains, so that $v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}]$.

2.1.2 Semantics

Invoking a method on an object v binds the method parameter to v and evaluates the method body. If the evaluation of the method body does not diverge, a value is returned.

Updating a method named l in an object replaces the object with a copy in which l has been updated to refer to the new method. New labels cannot be added to an existing object, nor can labels be dropped from an object. Object references do not exist, so updates have only local effect.

The evaluation of a term is stuck if it reaches an invocation or an update of a label that does not exist in the target object. Furthermore, the evaluation is stuck if it reaches any variable (which never happens when evaluating closed terms).

Natural semantics

The methods of an object can be invoked or updated, which gives rise to the following natural semantics [1, page 64]:

$$(\mathrm{INV}_\varsigma) \quad \frac{\vdash t \rightsquigarrow v^n \quad \vdash t_j\{v^n/x_j\} \rightsquigarrow v}{\vdash t.l_j \rightsquigarrow v} \qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}]$$

$$(\mathrm{UPD}_\varsigma) \quad \frac{\vdash t \rightsquigarrow v^n}{\vdash t.l_j \Leftarrow \varsigma(x)t' \rightsquigarrow [l_j = \varsigma(x)t',\ l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}\setminus\{j\}}]} \qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}]$$

Notice that when a variable is bound, the first rule represents the binding by using actual substitution (defined in the usual way, so that variable capture is avoided [1, page 61]). In other words, this semantic description implicitly assumes the existence of what Curien described as "a magic, metalevel, one-step operation" [4, Introduction] performing the substitution of the current object for the self parameter in the body of the method. In an actual implementation, this operation would of course not be one-step, since it requires a traversal of the entire method body term. We have more to say about the topic of substitution in Chapter 3.

Reduction semantics

Invocation and updating gives rise to the following contraction rules [1, Definition 6.2-1 (1)]:

$$\begin{aligned} v^n.l_j &\rightarrowtail t_j\{v^n/x_j\} \\ &\qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}] \\ v^n.l_j \Leftarrow \varsigma(x)t &\rightarrowtail [l_j = \varsigma(x)t,\ l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}\setminus\{j\}}] \\ &\qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}] \end{aligned}$$

Once again, we note that actual substitution is used when variables are bound. The contraction rules give us the following BNF of potential redexes:

$$pr \quad ::= \quad v.l \;\mid\; v.l \Leftarrow \varsigma(x)t$$

A potential redex is an actual one when its side conditions are satisfied, and contraction can take place. Otherwise, the potential redex is stuck.

Abadi and Cardelli do not define a grammar for reduction contexts. However, the following grammar plausibly reflects the 'evaluation strategy of the sort commonly used in programming languages' [1, Section 6.2.4, page 63]:

$$(\mathsf{Context}) \qquad C \quad ::= \quad [\,] \;\mid\; C[[\,].l] \;\mid\; C[[\,].l \Leftarrow \varsigma(x)t]$$

Lemma 1 (Unique decomposition)**.** *Any term in the ς-calculus which is not a value can be uniquely decomposed into a reduction context and a potential redex.*

Equivalence

The natural semantics is sound and complete with respect to the reduction semantics [1, Theorems 6.2-3 and 6.2-4, pages 64-65]. In other words, if evaluation with one semantics results in a value v, then evaluation with the other semantics will also result in v, and if evaluation diverges with one semantics, it also diverges with the other. Consequently, the two semantic descriptions are computationally equivalent.

We will give a proof of equivalence by derivation in Chapter 8, where we will also derive an (equivalent) abstract machine for the ς-calculus.

2.2 Summary and Conclusions

This chapter has presented the ς-calculus as defined by Abadi and Cardelli.

Along with an informal description, the chapter has presented the formal syntax, a natural semantics and a reduction semantics of the calculus. As shown by Abadi and Cardelli, the natural semantics and the reduction semantics are computationally equivalent.

Bindings in the calculus are defined using actual substitutions. In the following chapter, we will see how an equivalent calculus can be defined using explicit substitutions.

Chapter 3

The $\varsigma\rho$-calculus

This chapter explains the syntax, the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus, which is a calculus computationally equivalent to the ς-calculus, but which uses explicit substitutions instead of actual ones.

We define the calculus in terms of explicit substitutions, because actual substitutions are too impractical for concrete implementations. For the λ-calculus, this observation led to the study of *explicit substitutions* (which we also refer to as 'delayed substitutions') and their relationship to environment machines by Curien et al. [2,4], and more recently by Biernacka and Danvy [3,8]. Since one of the purposes of this dissertation is to derive abstract machines implementing the ς-calculus, it is relevant to study how the ς-calculus behaves when defined in terms of explicit substitutions.

The definition of the $\varsigma\rho$-calculus is based on the definition of the ς-calculus in Chapter 2. Delaying the substitution of bound variables means that the $\varsigma\rho$-calculus must be equipped with a means to represent the delayed substitutions. Furthermore, the semantics must reflect that evaluation now happens with respect to an environment.

Since values contain terms in their methods, and since free variables are no longer substituted at binding time, we also need to change the definition of values. Object values must now contain environments for each of their methods.

From a description of terms, environments and values, it is straightforward to define a natural semantics for the $\varsigma\rho$-calculus. However, defining a reduction semantics is impossible unless we extend the calculus to operate on a general notion of *closures* rather than on terms (a similar observation was made by Biernacka and Danvy in connection with Curien's $\lambda\rho$-calculus [3]).

3.1 Formal definitions

This section presents the formal definitions of the $\varsigma\rho$-calculus. The formal definitions consist of a syntactic definition for terms, values, substitutions and closures, a definition of a natural semantics and a definition of a reduction semantics.

3.1.1 Syntax

The terms of the $\varsigma\rho$-calculus are the same as those of the ς-calculus.

Environments are (possibly empty) lists of associations between identifiers and values. Choosing a list structure is purely a choice of syntax, as list syntax is easily written and understood. Lists may very well be an inefficient choice of representation in concrete implementations, but applying the functional and syntactic correspondences does not change the structure of the environments, so any representation with equivalent extend and lookup operators is valid.

A method body may contain other free variables than the method parameter, and hence it is necessary to associate each method with an environment. Methods equipped with an environment will be referred to as *method closures*.

The use of delayed substitutions induces a change in the category of values. Object values now contain method closures rather than methods (and so values are distinct from object literals). We refer to objects containing method closures rather than regular methods as *object closures*. Since method updates of a value may occur in a different scope than the one in which the object was created, the environments must be associated to individual methods rather than to objects.

These observations lead us to the following BNF of terms, values and environments:

$$\begin{array}{rrcl} \text{(Term)} & t & ::= & x \mid [l = \varsigma(x)t,\ \ldots,\ l = \varsigma(x)t] \mid t.l \mid t.l \Leftarrow \varsigma(x)t \\ \text{(Environment)} & e & ::= & \bullet \mid (x,\, v) \cdot e \\ \text{(Value)} & v & ::= & [l = (\varsigma(x)t)[e],\ \ldots,\ l = (\varsigma(x)t)[e]] \end{array}$$

As in the ς-calculus, we allow ourselves to group subterms using parentheses, and we occasionally index a value with the number of its methods, so that $v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i \in \{1 \ldots n\}}]$.

We could have chosen to use de Bruijn indices instead of variables. This choice would have corresponded more closely to array-like implementations of environments. However, it would have made the semantic descriptions more complex, without making a significant difference for the purpose of this dissertation.

3.1.2 Semantics

All evaluations must now occur with respect to an environment. The initial environment for the evaluation of a closed term is empty. The semantic descriptions must now include rules for evaluating variables. To this end, we need an auxiliary function *lookup* to look up identifiers in the current environment (the extension of an environment is handled syntactically, using the $(x,\, v) \cdot e$ notation).

Invoking a method closure no longer requires a substitution. Instead, the environment of the method closure is extended with a binding, and hence, substitution is 'delayed'. The method body is evaluated under the extended environment. Extending an environment with a new binding of a previously bound identifier shadows the previous binding. Method updates capture the current environment to form a method closure before updating the target object closure.

Just as for the ς-calculus, the evaluation of a term is stuck if it reaches an invocation or an update of a label that does not exist in the target object. Furthermore, evaluation is stuck if it reaches a variable that is not bound in the current environment (which never happens when evaluating closed terms).

Natural semantics

Since all evaluation must now take place with respect to an environment, the new evaluation judgement looks like this:

$$e \vdash t \rightsquigarrow v$$

The evaluation rules from Chapter 2 are straightforwardly adapted:

$$(\text{INV}_{\varsigma\rho}) \quad \frac{e \vdash t \rightsquigarrow v^n \quad (x_j,\, v^n) \cdot e_j \vdash t_j \rightsquigarrow v}{e \vdash t.l_j \rightsquigarrow v} \quad \begin{array}{l}\text{if } 1 \leq j \leq n,\\ \text{where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]\end{array}$$

$$(\text{UPD}_{\varsigma\rho}) \quad \frac{e \vdash t \rightsquigarrow v^n}{e \vdash t.l_j \Leftarrow \varsigma(x)t' \rightsquigarrow v} \quad \begin{array}{l}\text{if } 1 \leq j \leq n, \text{ where}\\ v = [l_j = (\varsigma(x)t')[e],\ l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}]\\ \text{and } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]\end{array}$$

We also need a rule to look up variables in the current environment

$$(\text{VAR-L}_{\varsigma\rho}) \quad \frac{}{e \vdash x \rightsquigarrow v} \quad \text{if } \textit{lookup}\ (x,\, e) = v$$

In addition, we need a rule to construct an object closure out of an object literal. The following rule captures the current environment and creates such an object closure:

$$(\text{CLO}_{\varsigma\rho}) \quad \frac{}{e \vdash [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}] \rightsquigarrow [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}]}$$

Reduction semantics

To properly define the reduction semantics for the $\varsigma\rho$-calculus, we need to define a notion of closures.

Originally, a closure was defined to be a term with an associated environment [22]. However, just as for the $\lambda\rho$-calculus [3], this notion is not general enough to allow a reduction semantics to be defined for the ς-calculus. Therefore, closures need to be allowed to contain subclosures rather than just subterms.

Keeping in mind the syntax for terms and environments, these observations lead us to the following BNF of closures:

$$(\textsf{Closure}) \qquad c \ ::= \ t[e] \mid [l = (\varsigma(x)t)[e],\ \ldots,\ l = (\varsigma(x)t)[e]] \mid c.l \mid c.l \Leftarrow (\varsigma(x)t)[e]$$

The initial closure of a closed term t is $t[\bullet]$.

Notice that there is no environment associated with invocations on closures (except for any environments associated with subclosures). The reason is that the current environment is no longer needed after an invocation, since the invoked method closure contains its own

environment. Notice also that the definition of values corresponds to the definition of object closures.

Contraction must now take place on closures rather than on terms. Adapting the contraction rules from Chapter 2 gives us the following new rules:

$$\begin{array}{rcl} v^n.l_j & \rightarrowtail & t_j[(x_j,\, v^n) \cdot e_j] \\ & & \text{if } 1 \leq j \leq n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}] \\ v^n.l_j \Leftarrow (\varsigma(x)t)[e] & \rightarrowtail & [l_j = (\varsigma(x)t)[e], l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}] \\ & & \text{if } 1 \leq j \leq n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}] \end{array}$$

We also need a contraction rule for looking up variables in the current environment:

$$\begin{array}{rcl} x[e] & \rightarrowtail & v \\ & & \text{if } \mathit{lookup}\ (x,\, e) = v \end{array}$$

In addition, we need rules to propagate environments into subterms:

$$\begin{array}{rcl} [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}][e] & \rightarrowtail & [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}] \\ (t.l)[e] & \rightarrowtail & t[e].l \\ (t.l \Leftarrow \varsigma(x)t')[e] & \rightarrowtail & t[e].l \Leftarrow (\varsigma(x)t')[e] \end{array}$$

The contraction give us the following BNF of potential redexes:

$$\begin{array}{rcl} pr & ::= & v.l \mid v.l \Leftarrow (\varsigma(x)t)[e] \mid \\ & & x[e] \mid [l = \varsigma(x)t,\ \ldots,\ l = \varsigma(x)t][e] \mid (t.l)[e] \mid (t.l \Leftarrow \varsigma(x)t')[e] \end{array}$$

As in the ς-calculus, a potential redex is an actual one when its side conditions are satisfied, and contraction can take place. Otherwise, the potential redex is stuck.

The definition of reduction contexts changes slightly, to account for the delayed substitutions. Update contexts must now capture the current environment, so that the updated method may be evaluated under it at a later time. As mentioned, the current environment is not needed for method invocations, so no environment needs to be captured in invocation contexts.

$$(\mathsf{Context}) \qquad C \ ::= \ [\,] \mid C[[\,].l] \mid C[[\,].l \Leftarrow (\varsigma(x)t)[e]]$$

Lemma 2 (Unique decomposition)**.** *Any term in the $\varsigma\rho$-calculus which is not a value can be uniquely decomposed into a reduction context and a potential redex.*

Equivalence

The soundness and completeness proofs for the ς-calculus apply *mutatis mutandis* to the $\varsigma\rho$-calculus [15]. Consequently, the two semantic descriptions are computationally equivalent. We give a proof of equivalence by derivation in Chapter 9, where we also derive an abstract machine for the $\varsigma\rho$-calculus.

3.2 Summary and Conclusions

This chapter has presented the $\varsigma\rho$-calculus, which is a version of the ς-calculus defined in terms of explicit substitutions rather than actual substitutions.

Along with an informal description, the chapter has presented the formal syntax, a natural semantics and a reduction semantics of the calculus. The natural semantics and the reduction semantics are computationally equivalent.

Chapter 4

Conclusion – Calculi

We have presented two calculi of objects, the ς-calculus and the $\varsigma\rho$-calculus. The ς-calculus was defined by Abadi and Cardelli [1, Chapter 6], whereas the $\varsigma\rho$-calculus is new. For each calculus, we have presented the syntax, a natural semantics and a reduction semantics.

The two calculi differ only in their handling of variable bindings. Whenever a variable is bound, the ς-calculus uses actual substitutions in the usual, capture-avoiding manner, whereas the $\varsigma\rho$-calculus uses explicit (or delayed) substitutions.

In Chapters 8 and 9, we derive abstract machines for the ς-calculus and the $\varsigma\rho$-calculus, respectively. Each of the chapters contains two derivations of the same abstract machine, one from the natural semantics and one from the reduction semantics.

The fact that for each calculus, the machines can be derived from both semantic descriptions, proves that the natural semantics and the reduction semantics of each of the calculi are computationally equivalent. Hence, we establish the equivalence by derivation, rather than by using pen and paper.

In Chapter 10, we show that the abstract machine for the ς-calculus and the abstract machine for the $\varsigma\rho$-calculus are strongly bisimilar. By doing so, we establish the computational equivalence between the two calculi.

Part II

Tools

Chapter 5

Functional Correspondence

This chapter introduces the functional correspondence, which connects functional interpreters and abstract machines by means of program transformations.

We begin by describing the *CPS transformation*, which maps programs in direct style (DS) to programs in continuation-passing style (CPS). The CPS transformation was discovered and rediscovered several times, so it is hard to point out one particular original purpose for it. In his account of the history of continuations [29], Reynolds mentions the elimination of labels and gotos as the first purpose for which the CPS transformation was applied, and Reynolds himself later used the transformation to avoid inter-dependence between the order of application for interpreted (defined) languages and the order of application for the language in which the interpreter is written (the defining language) [12].

We mainly exploit two inter-related properties of interpreters in CPS. First, all functions in CPS programs are tail-recursive, meaning that the value of a function application is either a simple value (requiring no further computation), or is given by the value of exactly one other function application (requiring only trivial intermediate computation). Second, the CPS transformation makes the call stack explicit in the interpreter, meaning that the interpreter gains access to (a higher-order representation of) the current evaluation context.

We then proceed to describe the process of *defunctionalisation*, which maps higher-order programs to first-order ones. Defunctionalisation was originally intended as a means for transforming higher-order interpreters into first-order ones, thereby providing a method for representing anonymous function abstractions in an interpreter written in a language that does not support higher-order functions. Furthermore, even if the defining language supports higher-order functions, representing anonymous functions of the defined language as first-order values eliminates inter-dependence between the scoping rules of the defined language and the scoping rules of the defining language [12].

We use defunctionalisation to transform the higher-order continuations into first-order ones, thereby getting a first-order representation of evaluation contexts. A first-order representation of the context will enable the interpreter to inspect the context (rather than just to apply it to a value), allowing for the definition of a state-transition system in which the context is part of the states.

The CPS transformation and the process of defunctionalisation are the building blocks of the *functional correspondence*, which links functional interpreters for a language to state-transition systems for the same language. Since functional interpreters implement language descriptions by natural semantics, and since a state-transition system for the evaluation of a

language defines an abstract machine, the functional correspondence provides a proof mechanism for equivalence of natural semantics and abstract machine. Furthermore, since the proof mechanism is based on mechanical program transformations, we may inter-derive these two types of semantic artifacts.

5.1 The CPS Transformation

Informally, a *continuation* is an explicit representation of 'the rest of the computation' at a given program point. Continuations are typically represented as anonymous functions, but may also be represented as first-order values, as we shall see later.

A program in *continuation-passing style* (CPS) passes continuations as extra arguments to all functions, possibly (pre-)composing the received continuation with extra instructions. When a function reaches a result, the continuation that the function received is applied to the intermediate result so that 'the rest of the computation' is performed.

The *CPS transformation* maps programs in direct style (that is, programs in which continuations are not represented explicitly), into programs in CPS. Informally, one may think of the transformation as changing a sequence of function applications into one application, which is passed an extra argument. The extra argument is the continuation, which takes the result of the application as its argument, and performs the remaining applications of the sequence (again CPS transformed).

We will perform the CPS transformation on programs written in SML. Therefore, we will only consider a version of the CPS transformation for a language similar to core SML, i.e., a call-by-value, left-to-right variant of the λ-calculus extended with conditionals and simple values v such as integers and booleans. The call-by-value CPS transformation is defined as follows:

$$\begin{aligned}
\mathcal{C}(v) &= \lambda c.c\ v \\
\mathcal{C}(x) &= \lambda c.c\ x \\
\mathcal{C}(\lambda x.t) &= \lambda c.c\ (\lambda x.\mathcal{C}(t)) \\
\mathcal{C}(t\ t') &= \lambda c.\mathcal{C}(t)\ (\lambda v.\mathcal{C}(t')\ (\lambda v'.v\ v'\ c)) \\
\mathcal{C}(\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'') &= \lambda c.\mathcal{C}(t)\ (\lambda v.\texttt{if}\ v\ \texttt{then}\ \mathcal{C}(t')\ c\ \texttt{else}\ \mathcal{C}(t'')\ c)
\end{aligned}$$

The initial continuation is the identity function $\lambda v.v$. Hence, a program p is transformed into $\mathcal{C}(p)\ (\lambda v.v)$.

When the original program uses simple terms (i.e., terms that do not require computation, such as variables and λ-abstractions) in applications, the CPS transformation generates so-called 'administrative redexes' of the form $(\lambda c.c\ s)\ (\lambda v.t)$, where s is a simple term. For instance, $s\ t$ is transformed into $\lambda c.(\lambda c.c\ \mathcal{C}(s))(\lambda v.\mathcal{C}(t)\ (\lambda v'.v\ v'\ c))$, which contains an administrative redex because $\mathcal{C}(s)$ is itself a simple term. Since it is possible to identify such simple terms during the transformation, we allow ourselves to shortcut the transformation by immediately performing two β-reductions, so that $s\ t$ is transformed into $\lambda c.\mathcal{C}(t)\ (\lambda v'.\mathcal{C}(s)\ v'\ c)$. We perform similar shortcuts when the argument of an application is a simple term, or when the condition of a conditional term is a simple term.

As an example of the CPS transformation, consider an SML implementation of the Ackermann function as seen in Figure 5.1. For the sake of simplicity, we regard applications of the functions +, - and = as simple terms, so we do not transform them.

```
1   fun A(m, n) =
2     if m = 0
3     then n + 1
4     else if n = 0
5          then A(m − 1, 1)
6          else A(m − 1, A(m, n − 1))
```

Figure 5.1: The Ackermann function

Line 3 contains the only branch in which a simple value is returned. Lines 5 and 6 both contain tail calls. In Line 6, there is also a recursive call which is not in tail position. The non-tail call is performed first, and the result is used as a parameter to the tail call.

CPS transforming this function yields the function in Figure 5.2.

```
1   fun A(m, n, c) =
2     if m = 0
3     then c (n + 1)
4     else if n = 0
5          then A(m − 1, 1, c)
6          else A(m, n − 1, fn v' => A(m − 1, v', c))
```

Figure 5.2: The Ackermann function in CPS

In Line 3, the received continuation `c` is applied to the result of the original function. Applying the continuation ensures that 'the rest of the computation' is in fact performed when a result is reached.

The tail calls of Lines 5 and 6 of the original program are both passed the received continuation. Since the calls were originally tail calls, no further computation occurs in that branch, and so 'the rest of the computation' is unchanged for those recursive calls.

Line 6 contains the same two calls as before, but they have 'switched places' syntactically. However, they will be performed in the same order as before, since the original tail call is passed as a continuation to the original non-tail call. This continuation expects to be applied to the result of the original non-tail call, and uses that result as the second parameter to the original tail call, thereby performing 'the rest of the computation' of that branch.

For CPS transformed programs, a continuation representing the rest of the computation is passed to any function call. In other words, if the body of a function in the original program contains instructions or applications to be performed after the first function application, these instructions and applications are passed as part of the continuation to the first function application in the transformed program. Since these new continuations are also CPS transformed, and since they are only ever applied in tail position, it follows that all calls in CPS transformed programs must be tail calls.

This fact can be verified in the example above where the non-tail call of Line 6 of the original program occurs in tail position in the transformed program. The tail call of Line 5 still occurs in tail position, as does the tail call of Line 6 in the original program, since all continuations are applied in tail position (Line 3).

In a tail-recursive program, there is no need for a call stack. We may therefore think of the continuations of a CPS program as an explicit, higher-order representation of the call stack. For interpreters, the call stack at a given point in the evaluation defines the evaluation context of that evaluation point, and hence, an interpreter in CPS has an explicit representation of the current evaluation context through its current continuation.

In the following section, we will see how to represent higher-order functions as first-order values. A first-order representation of functions allows us to use a first-order representation of continuations, and we will exploit this fact in Section 5.3 to derive an abstract machine.

Background: The CPS transformation was proved to be semantic-preserving by Plotkin [11].

As can be seen from the definition, the CPS transformation is mechanical, in the sense that it is possible to write a program which outputs a CPS transformed version of its input program. The same is true for the left inverse of the CPS transformation called the *DS transformation* [11].

5.2 Defunctionalisation

Defunctionalisation is a program transformation which replaces all syntactic occurrences of anonymous function abstractions with constructors of an algebraic sum type. Each constructor corresponds to a specific abstraction and holds a number of values corresponding to the free variables of the abstraction (i.e., each constructor represents an algebraic product type).

This transformation changes the abstractions from higher-order to first-order. Therefore, it is no longer possible to apply the abstractions to values directly, so the transformation furthermore introduces a new function (traditionally called `apply`). The `apply` function dispatches on the newly introduced data type and performs for each constructor the same computation as the constructor's higher-order counterpart in the original program. The new function is called whenever a higher-order function was called in the original program.

As an example, we consider an SML implementation of environments using higher-order functions. The implementation can be seen in Figure 5.3 (the example is adapted from Reynolds [12]).

```
1  structure Env
2  = struct
3      exception UNBOUND of string
4      val empty = fn x => raise UNBOUND x
5      fun lookup (x, env)
6          = env x
7      fun extend (x, v, env)
8          = fn x' => if x = x'
9                     then v
10                    else env x'
11   end
```

Figure 5.3: A higher-order implementation of an environment

The program contains two anonymous abstractions, one in Line 4 containing no free variables, and another in Lines 8-10 containing the free variables `x`, `v` and `env`.

To defunctionalise the program, we introduce a new SML data type containing two constructors, one for each of the abstractions. The abstraction in Line 4 will be replaced by the constructor `EMPTY` and the abstraction in Lines 8-10 will be replaced by the constructor `NON_EMPTY`, which itself contains the three values `x`, `v` and `env`. We also need to introduce an `apply` function, which dispatches on the introduced data type and performs the duties of each of the anonymous abstractions in the original program. The result of the defunctionalisation can be seen in Figure 5.4.

```
1  structure Env'
2  = struct
3      datatype 'a env = EMPTY
4                      | NON_EMPTY of string * 'a * 'a env
5      exception UNBOUND of string
6
7      fun apply (EMPTY, x')
8          = raise UNBOUND x'
9        | apply (NON_EMPTY (x, v, env), x')
10         = if x = x'
11           then v
12           else apply (env, x')
13
14     val empty = EMPTY
15     fun lookup (x, env)
16         = apply (env, x)
17     fun extend (x, v, env)
18         = NON_EMPTY (x, v, env)
19   end
```

Figure 5.4: The defunctionalised version of Figure 5.3

Notice how the introduced data type is isomorphic to the type `(string * 'a) list` with `EMPTY` and `NON_EMPTY` taking the places of `nil` and `::`, respectively. In other words, the result of defunctionalising an environment implemented using anonymous abstractions is in essence an environment implemented using association lists.

One might think of each individual data constructor as representing the environment associated with a flat closure. The program pointer of the closure is implicitly obtained by applying the `apply` function to the constructed value and a value of the defined language.

Background: Defunctionalisation was originally introduced by Reynolds as a means of converting higher-order interpreters into first-order ones, thereby allowing the definition of higher-order languages to be written in first-order languages, as well as making the scoping rules of the defined language independent of the scoping rules of the defining language [12]. To achieve these goals, Reynolds suggested the defunctionalisation of continuations, a topic which was further explored by Danvy and Nielsen [9].

Just as the CPS transformation, defunctionalisation is mechanical, in the sense that it is possible to write a program which outputs a defunctionalised version of its input program.

The same is true for the left inverse of defunctionalisation, *refunctionalisation* [7].

5.3 The Functional Correspondence

The CPS transformation and the process of defunctionalisation form the building blocks of the *functional correspondence*. The functional correspondence provides a link between functional interpreters and state-transition systems for the same language [2, 13].

Given a functional interpreter in DS, the functional correspondence consists in CPS transforming the interpreter and defunctionalising the continuations introduced by the CPS transformation.[1] The CPS transformation makes the evaluation context available to the interpreter as the current continuation at any point in the evaluation of a program, and hence, defunctionalising the introduced continuations provides the interpreter with a first-order representation of the evaluation context. Furthermore, since the interpreter is now tail-recursive, any application of the interpreter will result in exactly one recursive application (unless a final value is reached), so the interpreter performs exactly like a state-transition system, i.e., an abstract machine.

Since a natural semantics for a language can (usually) be implemented straightforwardly as a functional interpreter, and since a state-transition system defines an abstract machine, the functional correspondence provides a mechanism for proving equivalence between natural semantics and abstract machines for the same language.

$$\text{Natural semantics} \xrightarrow[\text{correspondence}]{\text{functional}} \text{Abstract machine}$$

Both the CPS transformation and the process of defunctionalisation are mechanical. Hence, it is also possible to derive an abstract machine from a natural semantics. Furthermore, since both transformations are reversible (in the sense that any program in the image of the CPS transformation or defunctionalisation can be transformed back to an equivalent program in the pre-image of the transformation), it is possible to derive a natural semantics from an abstract machine [5, 9, 10], as initially done with the SECD machine [13].[2]

The process of deriving an abstract machine from an interpreter is due to Reynolds [12]. Further work by Ager et al. subsequently identified this process as being applicable to functional interpreters and abstract machines in general, and prompted the coining of the phrase 'functional correspondence' [2]. The functional correspondence has been applied to various versions of the λ-calculus, e.g., the λ-calculus with monadic effects [5].

When the language in question is defined in terms of explicit substitutions, the functional interpreter contains an environment. The resulting abstract machine will therefore also be equipped with an environment. As we shall see in Chapter 9, this is also the case for the $\varsigma\rho$-calculus.

[1] Strictly speaking, if the interpreter is higher-order (e.g., uses a higher-order representation of environments, as in Figure 5.3), one needs to *closure convert* the interpreter before CPS transforming it. Closure conversion is the process of defunctionalising all anonymous abstractions, such that the DS interpreter becomes first-order. [9, 22]. Since we will not need to perform closure conversion for our interpreters, we will not go deeper into this process.

[2] Note, though, that some of the abstract machines that have been defined over the years are not in defunctionalised form, and that several of the ones that are have their apply function inlined (e.g., the Krivine machine), which makes it difficult to identify the machine as being in defunctionalised form [7].

5.4 Summary and Conclusion

This chapter has introduced the functional correspondence and its two building blocks, the CPS transformation and the process of defunctionalisation.

The CPS transformation and the process of defunctionalisation form the building blocks of the functional correspondence, which links functional interpreters to state-transition systems. Since a functional interpreter can be seen as a direct implementation of a natural semantics, and since a state-transition system can be seen as a direct implementation of an abstract machine, the functional correspondence provides a proof mechanism for equivalence between natural semantics and abstract machines. Furthermore, the two semantic descriptions can be inter-derived.

Chapter 6

Syntactic Correspondence

This chapter introduces the syntactic correspondence, which connects reduction-based evaluators and abstract machines by means of program transformations.

Reduction-based evaluation, or evaluation by iterated reduction, is the iterated application of functions that find the next reduction point (the next *redex*), contracts the redex into a *contractum* and finally rebuilds the term with the contractum replacing the redex.

We begin by describing the process of *refocusing*, which is a technique for optimising reduction-based evaluators. The technique was introduced by Danvy and Nielsen in 2001 [10]. Danvy and Nielsen observed that after a redex has been contracted, the term that is rebuilt is immediately decomposed again in order to find the next redex. They therefore proposed an optimisation technique for composing the rebuilding and decomposition functions such that the resulting function would go from redex to redex rather than rebuilding the entire term between each reduction. The resulting function was named the *refocusing* function.

We then proceed to describe the process of *fixed-point promotion*, which was proposed by Ohori and Sasano [26] as a technique for optimisation by fusion (i.e., the elimination of intermediate data structures). The technique is based on inlining, and in contrast to other fusion techniques, fixed-point promotion can be used to eliminate data structures produced and consumed by general recursive functions.

We use fixed-point promotion to fuse the iteration function with the function produced by the refocusing process. Although the fused evaluator still contains the intermediate data structures, we obtain an evaluator in the form of a transition system for the language.

Refocusing and fixed-point promotion form the building blocks of the *syntactic correspondence*, which links reduction-based evaluators for a language to state-transition systems for the same language. Since reduction-based evaluators implement language descriptions by reduction semantics, and since a state-transition system for the evaluation of a language defines an abstract machine, the syntactic correspondence provides a proof mechanism for equivalence of reduction semantics and abstract machine. Furthermore, since the proof mechanism is based on mechanical program transformations, we can derive one of these semantic artifacts from the other.

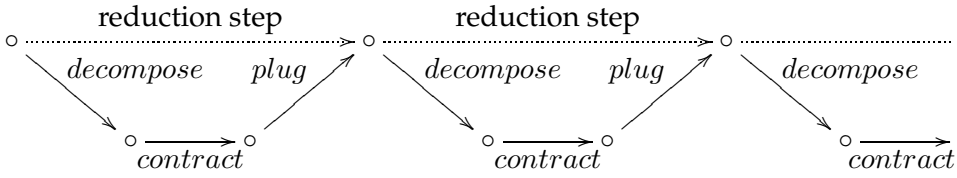
6.1 Refocusing

The process of *refocusing* was introduced by Danvy and Nielsen [10] as an optimisation of reduction-based evaluators. Furthermore, they observed that refocusing such an evaluator yields a small-step abstract machine (a one-step transition function and an iteration function computing the iteration of the transition function).

A reduction-based evaluator consists of four functions:

1. A decomposition function $decompose$, which traverses the source term in search for the next potential redex. The function returns the found redex and its context.
2. A contraction function $contract$, which takes a potential redex, reduces it to a contractum if possible and returns the contractum.
3. A plugging function $plug$, which takes a contractum and a context, and builds an intermediate term which is the copy of the decomposed term, except that the contractum takes the place of the redex.
4. An iterator, which applies the decomposition, contraction and plugging functions until a result is reached (if any).

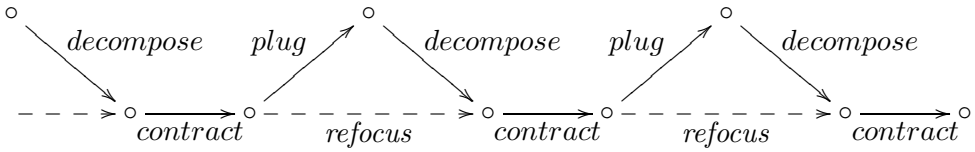
The evaluation of a term consists of a sequence of reduction steps, which can be depicted as follows:



From the diagram, we can see that each application of $decompose$ except the first one is preceded by an application of $plug$. Since the first application of $decompose$ can be preceded by a trivial application of $plug$ (plugging the entire term into the empty context), composing the two functions eliminates the need for either one.

Danvy and Nielsen observed that for a language satisfying the unique decomposition property (i.e., that any term can be decomposed uniquely into a context and a redex), the composition of $decompose$ and $plug$ could be made to go directly from redex point to redex point, thereby eliminating the overhead of constructing and decomposing the intermediate term. They call the resulting function the refocusing function.

As a result, the evaluation of a term consists of repeated applications of the refocusing and contraction functions, which can be depicted as follows:



The refocusing function is defined using two mutually recursive functions:

- $refocus$, which traverses subterms in search of the next redex. The function is defined by case over the term.
- $refocus_{aux}$, which plugs values into contexts until a redex is reached or another subterm needs to be traversed by $refocus$ (or until a final value is reached). The function is defined by case over the context.

Assume that $c(t_1, \ldots, t_n)$ is a syntactic construct where no subterms have been evaluated yet. If we assume left-to-right evaluation order on the subterms, $refocus$ is defined as follows:

1. If $c(t_1, \ldots, t_n)$ is a value, then we must plug the value back into the context, using $refocus_{aux}$. Intuitively, we need to "backtrack" to find another branch of the syntax tree in which a redex can be found:

$$refocus(c(t_1, \ldots, t_n),\, C) \;=\; refocus_{aux}(C,\, c(t_1, \ldots, t_n))$$

2. Otherwise, if $c(t_1, \ldots, t_n)$ is a potential redex, then we have found the next potential redex, and so we return:

$$refocus(c(t_1, \ldots, t_n),\, C) \;=\; (c(t_1, \ldots, t_n),\, C)$$

3. Otherwise, we must search for the next redex in the left-most subterm of $c(t_1, \ldots, t_n)$, using a recursive call to $refocus$:

$$refocus(c(t_1, \ldots, t_n),\, C) \;=\; refocus(t_1,\, C[[\,], t_2, \ldots, t_n])$$

Assume now that $C[c(v_1, \ldots, v_{m-1}, [\,], t_{m+1}, \ldots, t_n)]$ is a context which expects a value at the mth position. Then $refocus_{aux}$ is defined as follows:

1. If $c(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$ is a value, then we continue to plug values into the surrounding context, using a recursive call to $refocus_{aux}$:

$$\begin{aligned} &refocus_{aux}(C[c(v_1, \ldots, v_{m-1}, [\,], t_{m+1}, \ldots, t_n)],\, v_m) \\ &= refocus_{aux}(C,\, c(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)) \end{aligned}$$

2. Otherwise, if $c(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$ is a redex, then we have found the next redex, so we return it:

$$\begin{aligned} &refocus_{aux}(C[c(v_1, \ldots, v_{m-1}, [\,], t_{m+1}, \ldots, t_n)],\, v_m) \\ &= (c(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n),\, C) \end{aligned}$$

3. Otherwise, we must search for the next redex in the next unevaluated subterm of $c(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$. We perform the search by calling $refocus$:

$$\begin{aligned} &refocus_{aux}(C[c(v_1, \ldots, v_{m-1}, [\,], t_{m+1}, \ldots, t_n)],\, v_m) \\ &= refocus(t_{m+1},\, C[c(v_1, \ldots, v_m, [\,], t_{m+2}, \ldots, t_n)]) \end{aligned}$$

Note that for some constructs, not all subterms need to be evaluated before a redex or a value is found. A simple example of such a construct is a conditional branch, where evaluating the condition results in a redex, even if the consequent and the alternative have not been evaluated. Note also that our assumption of left-to-right evaluation order is not a restriction in practice. Any strict evaluation order on subterms will do, and since we require the language to be uniquely decomposable, such an evaluation order will always exist.

For some examples of the technique, we refer to Danvy's invited talk at WRS 2004 [5], which refocus an evaluator for arithmetic expressions and an evaluator for terms of the free monoid.

Background: Danvy and Nielsen proved refocusing to be semantic-preserving [10].

As can be seen, the construction of the *refocus* function is mechanical. However, as opposed to the transformations in the previous chapter, the operation is not known to be mechanically reversible.

Refocusing has no known applications apart from the optimisation of reduction-based evaluators.

6.2 Fixed-point Promotion

In compilers for functional languages, a common type of optimisation is the *fusing* of functions, that is, the composition of functions in order to eliminate intermediate data structures (also known as *deforestation*, since trees seem to be the most common data structure in functional programming). *Fixed-point promotion* is one such fusing technique, which deals with the fusing of recursive functions [26].

Suppose a program contains the application of a recursive function $f = \texttt{fix}\, f.\lambda x.E_f$ to the result of another recursive function $g = \texttt{fix}\, g.\lambda x.E_g$. The fixed-point promotion is used to construct a function $\texttt{fix}\, f\_g.\lambda x.E_{f\_g}$, which is equivalent to $f \circ g$. The intermediate data structure produced by g is thereby eliminated, and furthermore, the fixed-point of g is 'promoted' through f, hence the name of the technique. Intuitively, one may think of the process as an inlining of f in g.

The transformation is performed on $f \circ g$ as follows:

1. Inline the body of g. The result will be $f \circ \lambda x.E_g$, which can be rewritten as $\lambda x.f\ E_g$.
2. Distribute the function symbol f to all tail positions of E_g. This distribution will transform $\lambda x.f\ E_g$ into a term $\lambda x.E'_g$.
3. Inline the body of f, and simplify the resulting terms. The result will be a term $\lambda x.E_{f,g}$.
4. Replace all occurrences of $f \circ g$ in $\lambda x.E_{f,g}$ with a new name f_g. The result will be a term $\lambda x.E_{f\_g}$. Then bind the name f_g to the function $\texttt{fix}\, f\_g.\lambda x.E_{f\_g}$.

Going back to the refocused interpreter, we see that the iteration function calls itself on the result of the recursive refocusing function. We can therefore use the fixed-point promotion to fuse the two functions. The result is a recursive function which computes the iteration of the original contraction and refocusing functions without the use of an explicit iteration function [8].

For examples of the technique, we refer to Danvy et al.'s article on the one-pass CPS transformation [19, Appendix A], where fixed-point promotion is used on a program computing the run-length of a list, and to Danvy and Millikin's article on deriving big-step abstract machines from small-step ones, where fixed-point promotion is used on a recogniser for Dyck words [8].

Background: Ohori and Sasano in 2007 suggested fixed-point promotion as a deforestation technique, i.e., a technique for statically eliminating intermediate data structures in functional programs [26]. At that time, however, inlining into recursive functions had already been used extensively to simplify refocused evaluators [3, 5, 10]. Ohori and Sasano's contribution in our setting is to prove that the process is mechanical as well as correct, and that refocused evaluators can therefore be mechanically and correctly transformed into abstract machines. Ohori and Sasano also coined the term 'fixed-point promotion'.

As can be seen, fixed-point promotion is mechanical. However, just as for refocusing and contrary to the transformations in the previous chapter, fixed-point promotion is not known to be mechanically reversible.

6.3 The Syntactic Correspondence

The syntactic correspondence provides a link between reduction-based evaluators and state-transition systems for the same language.

Given a reduction-based evaluator for a language with unique decomposition, the syntactic correspondence consists in refocusing the decomposition and plugging functions, and fixed-point promoting the iteration and refocusing functions. Typically, the contraction function is also inlined, but since that function is not recursive, standard inlining suffices.

By the semantic preserving properties of refocusing and fixed-point promotion (and inlining), the resulting function implements the same language as the original reduction-based evaluator. Furthermore, both the iteration and refocusing functions are tail-recursive, and consequently, the result of fixed-point promoting them will also be tail-recursive. Also, the parameters of the function are the same as those of the reduction-based evaluator (terms and contexts), and so the resulting function performs exactly like a state-transition system, i.e., the implementation of an abstract machine.

Since a reduction semantics for a language can be implemented straightforwardly as a reduction-based evaluator, and since a state-transition system defines an abstract machine, the syntactic correspondence provides a mechanism for proving equivalence between reduction semantics and abstract machine for the same language.

$$\text{Reduction semantics} \xrightarrow[\text{correspondence}]{\text{syntactic}} \text{Abstract machine}$$

Refocusing and fixed-point promotion are both mechanical. Hence, it is also possible to derive an abstract machine from a reduction semantics. However, it is not known whether it is possible to mechanically derive a reduction semantics from an abstract machine semantics.

Background: The process of deriving an abstract machine from a reduction-based evaluator is due to Danvy and Nielsen [10], who also coined the phrase 'syntactic correspondence'.

The syntactic correspondence has been applied to various versions of the λ-calculus, including the λ-calculus with explicit substitutions [3], and with computational effects [8].

Biernacka and Danvy showed that when the language in question is defined in terms of explicit substitutions, the resulting abstract machine will be equipped with an environment [3]. As we shall see in Chapter 9, this is also the case for the $\varsigma\rho$-calculus.

6.4 Summary and Conclusion

This chapter the syntactic correspondence, and its two building blocks, the processes of refocusing and fixed-point promotion.

Refocusing and fixed-point promotion form the building blocks of the syntactic correspondence, which links reduction-based evaluators for languages with unique decomposition to state-transition systems. Since a reduction-based evaluator can be seen as a direct implementation of a reduction semantics, and since a state-transition system can be seen as a direct implementation of an abstract machine, the syntactic correspondence provides a proof mechanism for equivalence between reduction semantics and abstract machine. Furthermore, it is possible to derive an abstract machine from a reduction semantics, although it is not known whether the reverse derivation is possible. It is not known to be possible in general to derive a reduction semantics from an abstract machine, due to the lack of mechanical reversibility of the syntactic correspondence.

Chapter 7

Conclusion – Tools

We have presented two correspondences for semantic descriptions, the functional and syntactic correspondences. The functional correspondence connects natural semantics and abstract machines, whereas the syntactic correspondence connects reduction semantics and abstract machines. Both correspondences consist in semantic-preserving program transformations.

The functional correspondence arises from the CPS transformation and subsequent defunctionalisation of a functional interpreter. The result of this transformation is a direct implementation of a state-transition system, i.e., an abstract machine. Since functional interpreters are direct implementations of natural semantics, and since both the CPS transformation and the process of defunctionalisation are mechanical program transformations, the functional correspondence allows for the inter-derivation of natural semantics and abstract machines.

The syntactic correspondence arises from the refocusing and subsequent fixed-point promotion of a reduction-based evaluator. The result of this transformation is a direct implementation of a state-transition system, i.e., an abstract machine. Since reduction-based evaluators are direct implementations of reduction semantics, and since the both the process of refocusing and the process of fixed-point promotion are mechanical program transformations, the syntactic correspondence allows for the inter-derivation of reduction semantics and abstract machines.

Using the two correspondences together provides a connection between natural semantics and reduction semantics, via abstract machines. The connection is illustrated in the following diagram:

$$\text{Reduction semantics} \xrightarrow[\text{correspondence}]{\text{syntactic}} \text{Abstract machine} \xleftarrow[\text{correspondence}]{\text{functional}} \text{Natural semantics}$$

In Chapters 8 and 9 we use the functional correspondence to derive abstract machines from the natural semantics of the ς-calculus and the $\varsigma\rho$-calculus, respectively. We also use the syntactic correspondence to derive the same abstract machines from the reduction semantics of the two calculi, thereby proving the equivalence of the natural semantics and the reduction semantics for each of the calculi.

Part III

Derivation and Equivalence

Chapter 8

An Abstract Machine for the ς-calculus

Part II described two different ways of deriving abstract machines for a language; one starting from a reduction semantics of the language and one starting from the natural semantics of the language. In this chapter, we apply these two techniques to the ς-calculus.

Since our evaluators will be implemented in SML, we begin by defining terms and values of the ς-calculus as SML data types. The data types correspond to the grammars for terms and values given in Chapter 2. Furthermore, we give various utility functions, which all evaluators will use, and which are not touched by the transformations. These functions can therefore safely be abstracted away in a module.

We first derive an abstract machine from the natural semantics of the ς-calculus described in Chapter 2. We implement the natural semantics by building a direct-style interpreter, and use the functional correspondence to derive an abstract machine.

We then derive an abstract machine from the reduction semantics, as described in Chapter 2. We implement the reduction semantics by building a reduction-based evaluator, and use the syntactic correspondence to derive an abstract machine.

The two abstract machines turn out to be identical. The natural semantics and the reduction semantics of the ς-calculus therefore describe the same language.

8.1 Data Type Definitions and Utility Functions

Figure 8.1 shows the SML data type definition for ς-terms. Labels and variables are implemented as strings. Methods are pairs of variables and terms, and objects are lists of labelled methods. The `term` data type is a direct implementation of the grammar for terms given in Section 2.1.1.

Figures 8.2, 8.3 and 8.4 show three utility functions; `lookup_label`, which finds methods in objects (Figure 8.2), `update`, which updates methods in objects (Figure 8.3), and `substitute` which performs substitutions (Figure 8.4). Since the implementation of these functions is induced by the data type of terms rather than by a particular style of evaluation, they can be used by all ς-evaluators. We therefore abstract them away from the evaluators proper, and provide an SML structure `Util` containing these functions for all evaluators in this chapter.

```
type label = string
type id = string

datatype term = VAR of id
              | OBJECT of (label * id * term) list
              | INVOKE of term * label
              | UPDATE of term * label * (id * term)
```

Figure 8.1: Data type for terms of the ς-calculus

```
(* lookup_label :  (label * id * term) list * label -> (id * term) option *)
(* find the method labelled label in the object ob *)
fun lookup_label (ob, label)
    = let fun walk nil
              = NONE     (* label was not found *)
            | walk ((l, x, t) :: ms)
              = if l = label then SOME (x, t) else walk ms
      in walk ob
      end
```

Figure 8.2: Function for looking up methods in objects

```
(* update : (label * id * term) list * label * (id * term)
                                  -> (label * id * term) list option *)
(* update the method named by label in ob *)
fun update (ob, label, (x, t))
    = let fun walk nil
              = NONE     (* label was not found *)
            | walk ((m as (l, x', t')) :: ms)
              = if l = label
                then SOME ((l, x, t)::ms)
                else case walk ms of NONE => NONE
                                   | SOME ms => SOME (m :: ms)
      in walk ob
      end
```

Figure 8.3: Function for updating methods in objects

```
(* substitute : term * id * term -> term *)
(* substitute all free occurrences of id in v with a copy of t *)
fun substitute (v, x, t)
    = let fun walk (VAR y)
              = if x = y then v else VAR y
            | walk (OBJECT ob)
              = let fun visit nil
                        = nil
                      | visit ((m as (l, z, t)) :: ms)
                        = if x = z
                          then m :: (visit ms)    (* method parameter shadows
                                                     substituted parameter *)
                          else (l, z, (walk t)) :: (visit ms)
                in OBJECT (visit ob)
                end
            | walk (INVOKE (t, l))
              = INVOKE ((walk t), l)
            | walk (UPDATE (t, l, (y, t')))
              = UPDATE (walk t, l, (y, walk t'))
      in walk t
      end
```

Figure 8.4: Function for performing substitutions

```
datatype wrong = STUCK_VAR of id
               | STUCK_LABEL of label

datatype result = VALUE of (label * id * term) list
                | WRONG of wrong
```

Figure 8.5: Data type for values of the ς-calculus

```
signature Sigma_Evaluator =
sig
    val main : term -> result
end
```

Figure 8.6: Signature for evaluators of the ς-calculus

Figure 8.5 shows the SML data type definition for ς-values. As can be seen, we use a special value constructor `WRONG` for the result of a stuck evaluation. The `result` data type is a straightforward implementation of the grammar for values given in Section 2.1.1.

A ς-evaluator must map ς-terms to ς-values, Given the data types described above, the type signature for ς-evaluators is therefore the signature in Figure 8.6.

With these definitions in place, we are now ready to start the derivations.

8.2 Functional Correspondence

For the functional correspondence, the starting point of the derivation is a functional interpreter for the ς-calculus, written in direct style. A direct implementation of the natural semantics presented in Section 2.1.2 gives rise to the interpreter given in Figure 8.7. The interpreter also corresponds to the pseudo-code interpreter given by Abadi and Cardelli [1, Page 65].

```
1  structure Sigma_DS_Interpreter : Sigma_Evaluator =
2  struct
3      local open Util in
4
5      fun eval (VAR x)
6          = WRONG (STUCK_VAR x)
7        | eval (OBJECT v)
8          = VALUE v
9        | eval (INVOKE (t, l))
10         = (case eval t
11              of VALUE receiver
12                 => (case lookup_label (receiver, l)
13                       of SOME (var, body)
14                          => eval (substitute (OBJECT receiver, var, body))
15                        | NONE
16                          => WRONG (STUCK_LABEL l))
17               | error
18                 => error)
19       | eval (UPDATE (t, l, (x, t')))
20         = (case eval t
21              of VALUE receiver
22                 => (case update (receiver, l, (x, t'))
23                       of SOME v
24                          => VALUE v
25                        | NONE
26                          => WRONG (STUCK_LABEL l))
27               | error
28                 => error)
29
30     fun main t = eval t
31
32     end
33 end
```

Figure 8.7: DS interpreter for the ς-calculus

```
1  structure Sigma_CPS_Interpreter : Sigma_Evaluator =
2  struct
3      local open Util in
4
5      fun eval_cps (VAR x, c)
6          = c (WRONG (STUCK_VAR x))
7        | eval_cps (OBJECT v, c)
8          = c (VALUE v)
9        | eval_cps (INVOKE (t, l), c)
10         = eval_cps (t,
11               fn VALUE receiver
12                  => (case lookup_label (receiver, l)
13                        of SOME (var, body)
14                           => eval_cps (substitute (OBJECT receiver, var, body), c)
15                         | NONE
16                           => c (WRONG (STUCK_LABEL l)))
17                | error
18                  => c error)
19       | eval_cps (UPDATE (t, l, (x, t')), c)
20         = eval_cps (t,
21               fn VALUE receiver
22                  => (case update (receiver, l, (x, t'))
23                        of SOME v
24                           => c (VALUE v)
25                         | NONE
26                           => c (WRONG (STUCK_LABEL l)))
27                | error
28                  => c error)
29
30     fun main t = eval_cps (t, fn v => v)
31
32     end
33 end
```

Figure 8.8: CPS interpreter for the ς-calculus

CPS transforming the interpreter in Figure 8.7 yields the interpreter in Figure 8.8.

When an error occurs, all continuations will simply propagate the error result onwards. Hence, we can choose to not to apply the continuation when an error occurs.[1] Then, the continuations are only applied to non-error values `VALUE v`, so we may move the data type constructor `VALUE` to the initial continuation. The result is shown in Figure 8.9.

[1] When doing so, we in fact use CPS to emulate the effect of raising an exception. Had we used SML exceptions to signal errors in the DS interpreter, the CPS interpreter would also have used SML exceptions, and we could then have eliminated them in the same way as we eliminate error propagation here.

```
structure Sigma_CPS_Interpreter' : Sigma_Evaluator =
struct
    local open Util in

    fun eval_cps (VAR x, c)
        = WRONG (STUCK_VAR x)
      | eval_cps (OBJECT v, c)
        = c v
      | eval_cps (INVOKE (t, l), c)
        = eval_cps (t,
             fn receiver
                => (case lookup_label (receiver, l)
                      of SOME (var, body)
                         => eval_cps (substitute (OBJECT receiver, var, body), c)
                       | NONE
                         => WRONG (STUCK_LABEL l)))
      | eval_cps (UPDATE (t, l, (x, t')), c)
        = eval_cps (t,
             fn receiver
                => (case update (receiver, l, (x, t'))
                      of SOME v
                         => c v
                       | NONE
                         => WRONG (STUCK_LABEL l)))

    fun main t = eval_cps (t, fn v => VALUE v)

    end
end
```

Figure 8.9: CPS interpreter for the ς-calculus, with direct error propagation

We now defunctionalise the continuations introduced by the CPS transformation. The defunctionalisation introduces the new data type `abs` with constructors `ID`, `INV` and `UPD` representing the initial continuation, continuations introduced when evaluating invocation terms, and continuations introduced when evaluating update terms, respectively. Defunctionalisation also introduces the apply function `apply_abs`, which dispatches over values of type `abs`. The result of the defunctionalisation can be seen in Figure 8.10.

```
1  structure Sigma_Defunctionalised_Interpreter : Sigma_Evaluator =
2  struct
3      local open Util in
4
5      datatype abs = ID
6                   | INV of label * abs
7                   | UPD of label * (id * term) * abs
8
9      fun apply_abs (ID, v)
10         = VALUE v
11       | apply_abs (INV (l, c), v)
12         = (case lookup_label (v, l)
13              of SOME (var, body)
14                 => eval_defunct (substitute (OBJECT v, var, body), c)
15               | NONE
16                 => WRONG (STUCK_LABEL l))
17       | apply_abs (UPD (l, (x, t'), c), v)
18         = (case update (v, l, (x, t'))
19              of SOME v
20                 => apply_abs (c, v)
21               | NONE
22                 => WRONG (STUCK_LABEL l))
23
24     and eval_defunct (VAR x, c)
25         = WRONG (STUCK_VAR x)
26       | eval_defunct (OBJECT v, c)
27         = apply_abs (c, v)
28       | eval_defunct (INVOKE (t, l), c)
29         = eval_defunct (t, INV (l, c))
30       | eval_defunct (UPDATE (t, l, (x, t')), c)
31         = eval_defunct (t, UPD (l, (x, t'), c))
32
33     fun main t = eval_defunct (t, ID)
34
35     end
36 end
```

Figure 8.10: Defunctionalised CPS interpreter for the ς-calculus

As can be seen, the defunctionalised interpreter is a direct implementation of the following abstract machine:

$$
\begin{aligned}
\langle v,\, C\rangle &\Rightarrow_S \langle C,\, v\rangle \\
\langle t.l,\, C\rangle &\Rightarrow_S \langle t,\, C[[\,].l]\rangle \\
\langle t.l \Leftarrow \varsigma(x)t',\, C\rangle &\Rightarrow_S \langle t,\, C[[\,].l \Leftarrow \varsigma(x)t']\rangle \\
\\
\langle [\,],\, v\ \rangle &\Rightarrow_S v \\
\langle C[[\,].l_j],\, v^n\rangle &\Rightarrow_S \langle t_j\{v^n/x_j\},\, C\rangle \\
&\qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}] \\
\langle C[[\,].l_j \Leftarrow \varsigma(x)t],\, v^n\rangle &\Rightarrow_S \langle C,\, [l_j = \varsigma(x)t,\ l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle \\
&\qquad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}]
\end{aligned}
$$

For a closed term t, the machine starts in the state $\langle t,\, [\,]\rangle$, and continues until it is either stuck or reaches a result. The machine is stuck if it reaches either of the states $\langle C[[\,].l],\, v\rangle$ or $\langle C[[\,].l \Leftarrow \varsigma(x)t'],\, v\rangle$ and v does not contain a method labelled l. It is also stuck if it reaches a state $\langle x,\, C\rangle$ (which never happens when evaluating closed terms).

Since we have derived the abstract machine from the natural semantics using semantic-preserving program transformations, the natural semantics and the abstract machine specify the same language. In the following section, we derive the same abstract machine from the reduction semantics of the ς-calculus.

8.3 Syntactic Correspondence

For the syntactic correspondence, the starting point of the derivation is a reduction-based evaluator for the ς-calculus.

The evaluator contains four new data types, shown in Figure 8.11. The first two, called `potential_redex` and `reduction_context`, correspond to the grammars for potential redexes and reduction contexts, respectively. The grammars were given in Section 2.1.2. The last two, called `contractum` and `value_or_decomposition`, are data types for the results of contraction and decomposition, respectively. The iteration function uses values of the type `contractum` to determine whether evaluation can continue or is stuck, and uses values of the type `value_or_decomposition` to determine whether a final value is reached or whether further reduction must take place. The uses of the last two data types will be eliminated during the derivation.

```
1   datatype potential_redex =
2               INVOKE_REDEX of ((label * id * term) list) * label
3             | UPDATE_REDEX of ((label * id * term) list) * label * (id * term)
4
5   datatype reduction_context =
6               ID_CONTEXT
7             | INVOKE_CONTEXT of label * reduction_context
8             | UPDATE_CONTEXT of label * (id * term) * reduction_context
9
10  datatype contractum = ACTUAL_CONTRACTUM of term
11                      | STUCK_CONTRACTUM of wrong
12
13  datatype value_or_decomposition = VAL of result
14                                  | DEC of reduction_context * potential_redex
```

Figure 8.11: Data types of the reduction-based evaluator for the ς-calculus

Based on these data types, a direct implementation of the reduction semantics gives rise to the evaluator displayed in Figure 8.12. Two extra rules have been introduced. The first rule returns an error value when evaluation is stuck because an unsubstituted variable has been reached, which will never happen for the evaluation of closed terms. The rule has been added to make the SML compiler accept the evaluator without issuing a warning. The second rule simply says that $v \rightarrow v$, and handles the case where a final value is reached

Refocusing the evaluator in Figure 8.12 yields the evaluator displayed in Figure 8.13. The functions `decompose` and `plug` have now been eliminated, and the `refocus` function has taken their place. Given a term and a context, `refocus` now finds the next potential redex directly, rather than reconstructing the entire term and subsequently decomposing it.

Fusing the iteration and refocusing functions using fixed-point promotion now yields the evaluator displayed in Figure 8.14. The functions `iterate`, `refocus`, `refocus'` and `refocus' _aux` are no longer called, so their definitions have been removed. Similarly, there is no longer any need for the data type `value_or_decomposition`, since `iterate_refocus` now either returns the found value directly, or calls the contraction function on the found redex.

```
structure Sigma_Reductionbased : Sigma_Evaluator =
struct
    local open Util in

    fun contract (INVOKE_REDEX (v, l))
        = (case lookup_label (v, l)
             of SOME (var, body)
               => ACTUAL_CONTRACTUM (substitute (OBJECT v, var, body))
              | NONE
               => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (v, l, (id, t')))
        = (case update (v, l, (id, t'))
             of SOME v'
                => ACTUAL_CONTRACTUM (OBJECT v')
              | NONE
               => STUCK_CONTRACTUM (STUCK_LABEL l))

    fun plug (ID_CONTEXT, t)
        = t
      | plug (INVOKE_CONTEXT (l, rc), t)
        = plug (rc, INVOKE (t, l))
      | plug (UPDATE_CONTEXT (l, (id, t'), rc), t)
        = plug (rc, UPDATE (t, l, (id, t')))

    fun decompose' (VAR x, rc)      (* Extra rule. *)
        = VAL (WRONG (STUCK_VAR x))
      | decompose' (OBJECT v, rc)   (* Extra rule. *)
        = VAL (VALUE v)             (* Can only happen when rc = ID_CONTEXT *)
      | decompose' (INVOKE (OBJECT v, l), rc)
        = DEC (rc, INVOKE_REDEX (v, l))
      | decompose' (INVOKE (t, l), rc)
        = decompose' (t, INVOKE_CONTEXT (l, rc))
      | decompose' (UPDATE (OBJECT v, l, (id, t')), rc)
        = DEC (rc, UPDATE_REDEX (v, l, (id, t')))
      | decompose' (UPDATE (t, l, (id, t')), rc)
        = decompose' (t, UPDATE_CONTEXT (l, (id, t'), rc))

    fun decompose t
        = decompose' (t, ID_CONTEXT)

    fun iterate (VAL v)
        = v
      | iterate (DEC (rc, pr))
        = (case contract pr
             of ACTUAL_CONTRACTUM t
                => iterate (decompose (plug (rc, t)))
              | STUCK_CONTRACTUM w
                => WRONG w)

    fun main t = iterate (decompose (plug (ID_CONTEXT, t)))

    end
end
```

Figure 8.12: Reduction-based evaluator for the ς-calculus

```
structure Sigma_Refocused : Sigma_Evaluator =
struct
    local open Util in

    fun contract (INVOKE_REDEX (v, l))
        = (case lookup_label (v, l)
             of SOME (var, body)
                => ACTUAL_CONTRACTUM (substitute (OBJECT v, var, body))
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (v, l, (id, t')))
        = (case update (v, l, (id, t'))
             of SOME v'
                => ACTUAL_CONTRACTUM (OBJECT v')
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))

    fun refocus' (VAR x, rc)
        = VAL (WRONG (STUCK_VAR x))
      | refocus' (OBJECT v, rc)
        = refocus'_aux (v, rc)
      | refocus' (INVOKE (t, l), rc)
        = refocus' (t, INVOKE_CONTEXT (l, rc))
      | refocus' (UPDATE (t, l, (id, t')), rc)
        = refocus' (t, UPDATE_CONTEXT (l, (id, t'), rc))
    and refocus'_aux (v, ID_CONTEXT)
        = VAL (VALUE v)
      | refocus'_aux (v, INVOKE_CONTEXT (l, rc))
        = DEC (rc, INVOKE_REDEX (v, l))
      | refocus'_aux (v, UPDATE_CONTEXT (l, (id, t'), rc))
        = DEC (rc, UPDATE_REDEX (v, l, (id, t')))

    fun refocus (rc, t) = refocus' (t, rc)

    fun iterate (VAL v)
        = v
      | iterate (DEC (rc, pr))
        = (case contract pr
             of ACTUAL_CONTRACTUM t
                => iterate (refocus (rc, t))
              | STUCK_CONTRACTUM w
                => WRONG w)

    fun main t = iterate (refocus (ID_CONTEXT, t))

    end
end
```

Figure 8.13: Refocused evaluator for the ς-calculus

```
structure Sigma_Refocused_Fused : Sigma_Evaluator =
struct
    local open Util in

    fun contract (INVOKE_REDEX (v, l))
        = (case lookup_label (v, l)
             of SOME (var, body)
                => ACTUAL_CONTRACTUM (substitute (OBJECT v, var, body))
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (v, l, (id, t')))
        = (case update (v, l, (id, t'))
             of SOME v'
                => ACTUAL_CONTRACTUM (OBJECT v')
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))

    fun iterate_refocus (VAR x, rc)
        = WRONG (STUCK_VAR x)
      | iterate_refocus (OBJECT v, rc)
        = iterate_refocus_aux (v, rc)
      | iterate_refocus (INVOKE (t, l), rc)
        = iterate_refocus (t, INVOKE_CONTEXT (l, rc))
      | iterate_refocus (UPDATE (t, l, (id, t')), rc)
        = iterate_refocus (t, UPDATE_CONTEXT (l, (id, t'), rc))
    and iterate_refocus_aux (v, ID_CONTEXT)
        = VALUE v
      | iterate_refocus_aux (v, INVOKE_CONTEXT (l, rc))
        = (case contract (INVOKE_REDEX (v, l))
             of ACTUAL_CONTRACTUM t
                => iterate_refocus (t, rc)
              | STUCK_CONTRACTUM w
                => WRONG w)
      | iterate_refocus_aux (v, UPDATE_CONTEXT (l, (id, t'), rc))
        = (case contract (UPDATE_REDEX (v, l, (id, t')))
             of ACTUAL_CONTRACTUM t
                => iterate_refocus (t, rc)
              | STUCK_CONTRACTUM w
                => WRONG w)

    fun main t = iterate_refocus (t, ID_CONTEXT)

    end
end
```

Figure 8.14: Refocused and fused evaluator for the ς-calculus

Since the contraction function is now always called on known redex types, we can inline the body of `contract` and simplify the terms. In doing so, we eliminate the need for the `contractum` data type. The result can be seen in Figure 8.15.

```
structure Sigma_Refocused_Fused_Inlined : Sigma_Evaluator =
struct
    local open Util in

    fun iterate_refocus (VAR x, rc)
        = WRONG (STUCK_VAR x)
      | iterate_refocus (OBJECT v, rc)
        = iterate_refocus_aux (v, rc)
      | iterate_refocus (INVOKE (t, l), rc)
        = iterate_refocus (t, INVOKE_CONTEXT (l, rc))
      | iterate_refocus (UPDATE (t, l, (id, t')), rc)
        = iterate_refocus (t, UPDATE_CONTEXT (l, (id, t'), rc))
    and iterate_refocus_aux (v, ID_CONTEXT)
        = VALUE v
      | iterate_refocus_aux (v, INVOKE_CONTEXT (l, rc))
        = (case lookup_label (v, l)
             of SOME (var, body)
                => iterate_refocus (substitute (OBJECT v, var, body), rc)
              | NONE
                => WRONG (STUCK_LABEL l))
      | iterate_refocus_aux (v, UPDATE_CONTEXT (l, (id, t'), rc))
        = (case update (v, l, (id, t'))
             of SOME v'
                =>  iterate_refocus_aux (v', rc)
              | NONE
                => WRONG (STUCK_LABEL l))

    fun main t = iterate_refocus (t, ID_CONTEXT)

    end
end
```

Figure 8.15: Evaluator for the ς-calculus, with inlined contraction function

The evaluator directly implements a state-transition system, i.e., an abstract machine. Furthermore, the abstract machine is the same as the one derived by the functional correspondence, up to reordering of state components. This fact can be verified by observing that the types `abs` and `reduction_context` are isomorphic, and that `iterate_refocus` and `iterate_refocus_aux` are equivalent to `eval_defunct` and `apply_abs`, respectively.

Since the abstract machines are identical, the natural semantics and the reduction semantics of the ς-calculus (and, of course, the abstract machine) define the same language. This equivalence is illustrated in the following diagram:

$$\text{Natural semantics for the } \varsigma\text{-calculus} \xrightarrow[\text{correspondence}]{\text{functional}} \text{Abstract machine for the } \varsigma\text{-calculus} \xleftarrow[\text{correspondence}]{\text{syntactic}} \text{Reduction semantics for the } \varsigma\text{-calculus}$$

8.4 Summary and Conclusions

We have derived two abstract machines for the ς-calculus; one from the natural semantics using the functional correspondence, and one from the reduction semantics using the syntactic correspondence. The two machines are identical, and new.

Since the functional and syntactic correspondences are both semantic preserving, the interpreter, the evaluator and the abstract machine implement the same language. The fact that the two derived abstract machines are identical confirms the equivalence between the natural semantics and the reduction semantics of the ς-calculus as proved by Abadi and Cardelli. In contrast to their proof, however, we have obtained our proof by mechanical derivation rather than by using pen and paper.

Chapter 9

An Abstract Machine for the $\varsigma\rho$-calculus

In the previous chapter, we applied the functional and syntactic correspondences to the ς-calculus, and thereby derived an abstract machine for that calculus. In this chapter, we apply the same two techniques to the $\varsigma\rho$-calculus.

Since the $\varsigma\rho$-calculus is defined in terms of closures, we begin by defining closures as an SML data type. We also present a new data type for values, since the definition of values is different for the $\varsigma\rho$-calculus than for the ς-calculus. The data types correspond to the grammars for terms and values given in Chapter 3. Furthermore, we give various utility functions, which all evaluators will use, and which are not touched by the transformations. These functions can therefore safely be abstracted away.

We first derive an abstract machine from the natural semantics of the $\varsigma\rho$-calculus described in Chapter 3. We implement the natural semantics by building a direct-style interpreter, and use the functional correspondence to derive an abstract machine.

We then derive an abstract machine from the reduction semantics, also described in Chapter 3. We implement the reduction semantics by building a reduction-based evaluator, and use the syntactic correspondence to derive an abstract machine.

The two abstract machines turn out to be identical, proving that the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus describe the same language.

9.1 Data Type Definitions and Utility Functions

The terms of the $\varsigma\rho$-calculus are the same as those for the ς-calculus.

Figure 9.1 shows the SML data type definition for closures and environments of the $\varsigma\rho$-calculus. As can be seen, all closures are now equipped with environments. The only exception is invocation closures, which do not need them. The `closure` data type is a direct implementation of the grammar for closures given in Section 3.1.2, except that we have grouped all closures defined as a (term, environment) pair into one closure type, `GROUND_CLO`. Environments are implemented as association lists, as suggested by the syntax.

As mentioned, the definition of values changes for the $\varsigma\rho$-calculus, and hence, the `result` data type changes as well. Results are now either lists of method closures, a type which we have named `receiver_closure`, or they are values of the `wrong` data type, which is identical to the type from the ς-calculus. The new data type for values can be seen in Figure 9.2.

```
datatype closure = GROUND_CLO of term * environment
                 | OBJECT_CLO of receiver_closure
                 | INVOKE_CLO of closure * label
                 | UPDATE_CLO of closure * label * ((id * term) * environment)
and environment = ENV of (id * receiver_closure) list
withtype receiver_closure = (label * ((id * term) * environment)) list
```

Figure 9.1: Data type for closures of the $\varsigma\rho$-calculus

```
datatype result = VALUE of receiver_closure
                | WRONG of wrong
```

Figure 9.2: Data type for values of the $\varsigma\rho$-calculus

```
signature Sigmarho_Evaluator =
sig
    val main : term -> result
end
```

Figure 9.3: Signature for evaluators of the $\varsigma\rho$-calculus

The signature for $\varsigma\rho$-evaluators changes, but only because the definition of `result` changes. The definition can be seen in Figure 9.3.

```
(* lookup_label :
      receiver_closure * label -> ((id * term) * environment) option *)
fun lookup_label (rec_clo, l)
    = let fun walk nil
              = NONE     (* label was not found *)
            | walk ((l', mc) :: ms)
              = if l' = l
                then SOME mc
                else walk ms
      in walk rec_clo
      end
```

Figure 9.4: Function for looking up methods in objects

```
(* update :
      receiver_closure * label * ((id * term) * environment)
                                        -> receiver_closure option *)
(* update the method named by label in rec_clo *)
fun update (rec_clo, l, ((x, t), e))
    = let fun walk nil
              = NONE     (* label was not found *)
            | walk ((m as (l', ((x', t'), e'))) :: ms)
              = if l' = l
                then SOME ((l, ((x, t), e))::ms)
                else case walk ms of SOME ms'
                                     => SOME (m :: ms')
                                   | NONE
                                     => NONE
      in walk rec_clo
      end
```

Figure 9.5: Function for updating methods in objects

The `lookup_label` and `update` utility functions from the previous chapter change slightly, to account for the new definition of values. The `substitute` function is no longer needed, since the calculus is now defined in terms of explicit substitutions. The new function definitions can be seen in Figures 9.4 and 9.5.

```
(* The initial (empty) environment *)
(* empty_environment : environment *)
val empty_environment = ENV nil

(* assoc : id * environment -> receiver_closure option *)
(* Environment lookup function. *)
fun assoc (x, ENV es)
    = let fun walk nil
              = NONE
            | walk ((x', v) :: xvs')
              = if x = x'
                then SOME v
                else walk xvs'
      in walk es
      end

(* extend_environment : environment * id * receiver_closure -> environment *)
(* Extends an environment with a new (id, receiver_closure) pair *)
fun extend_environment (ENV es, x, v)
    = ENV ((x, v) :: es)
```

Figure 9.6: Functions implementing environments in the $\varsigma\rho$-calculus

```
(* close_object : receiver * environment -> receiver_closure *)
fun close_object (r, e)
    = let (* close : label * id * term -> label * ((id * term) * environment) *)
          (* converts members of object literals to closures *)
          fun close (l, x, t) = (l, ((x, t), e))
      in map close r
      end
```

Figure 9.7: Function mapping object literals and environments to `receiver_closures`

We also need a lookup and an extend function for environments, and a constant value for the empty environment. These function and value definitions can be seen in Figure 9.6.

Finally, we need a function for creating a `receiver_closure` from an object literal and an environment. This function can be seen in Figure 9.7.

With these definitions in place, we are now ready to start the derivations.

9.2 Functional Correspondence

For the functional correspondence, the starting point of the derivation is a functional interpreter for the $\varsigma\rho$-calculus, written in direct style. Since the $\varsigma\rho$-calculus is defined using explicit substitutions, the interpreter will be equipped with an environment. A direct implementation of the natural semantics presented in Section 3.1.2 gives rise to the interpreter given in Figure 9.8.

```
structure Sigmarho_Interpreter : Sigmarho_Evaluator =
struct
    local open Util in

    fun eval (VAR x, e)
        = (case assoc (x, e)
             of SOME rec_clo
                => VALUE rec_clo
              | NONE
                => WRONG (STUCK_VAR x))
      | eval (OBJECT v, e)
        = VALUE (close_object (v, e))
      | eval (INVOKE (t, l), e)
        = (case eval (t, e)
             of VALUE receiver
                => (case lookup_label (receiver, l)
                      of SOME ((var, body), e')
                         => eval (body, extend_environment (e', var, receiver))
                       | NONE
                         => WRONG (STUCK_LABEL l))
              | error
                => error)
      | eval (UPDATE (t, l, (x, t')), e)
        = (case eval (t, e)
             of VALUE receiver
                => (case update (receiver, l, ((x, t'), e))
                      of SOME rec_clo
                         => VALUE rec_clo
                       | NONE
                         => WRONG (STUCK_LABEL l))
              | error
                => error)

    fun main t = eval (t, empty_environment)

    end
end
```

Figure 9.8: DS interpreter for the $\varsigma\rho$-calculus

```
structure Sigmarho_CPS_Interpreter : Sigmarho_Evaluator =
struct
    local open Util in

    fun eval_cps (VAR x, e, c)
        = (case assoc (x, e)
             of SOME rec_clo
                => c (VALUE rec_clo)
              | NONE
                => c (WRONG (STUCK_VAR x)))
      | eval_cps (OBJECT v, e, c)
        = c (VALUE (close_object (v, e)))
      | eval_cps (INVOKE (t, l), e, c)
        = eval_cps (t, e,
             fn VALUE receiver
                => (case lookup_label (receiver, l)
                      of SOME ((var, body), e')
                         => eval_cps (body, extend_environment (e', var, receiver
                             ), c)
                       | NONE
                         => c (WRONG (STUCK_LABEL l)))
              | error
                => c error)
      | eval_cps (UPDATE (t, l, (x, t')), e, c)
        = eval_cps (t, e,
             fn VALUE receiver
                => (case update (receiver, l, ((x, t'), e))
                      of SOME rec_clo
                         => c (VALUE rec_clo)
                       | NONE
                         => c (WRONG (STUCK_LABEL l)))
              | error
                => c error)

    fun main t = eval_cps (t, empty_environment, fn v => v)

    end
end
```

Figure 9.9: CPS interpreter for the $\varsigma\rho$-calculus

CPS transforming the interpreter in Figure 9.8 yields the interpreter in Figure 9.9.

When an error occurs, all continuations will simply propagate the error result onwards. Hence, we can choose to not to apply the continuation when an error occurs, exactly like we did for the ς-calculus. Then, the continuations are only applied to non-error values `VALUE` v, so we may move the data type constructor `VALUE` to the initial continuation. The result is shown in Figure 9.10.

```
structure Sigmarho_CPS_Interpreter' : Sigmarho_Evaluator =
struct
    local open Util in

    fun eval_cps (VAR x, e, c)
        = (case assoc (x, e)
             of SOME rec_clo
                => c rec_clo
              | NONE
                => WRONG (STUCK_VAR x))
      | eval_cps (OBJECT v, e, c)
        = c (close_object (v, e))
      | eval_cps (INVOKE (t, l), e, c)
        = eval_cps (t, e,
             fn receiver
                => (case lookup_label (receiver, l)
                      of SOME ((var, body), e')
                         => eval_cps (body,
                                      extend_environment (e', var, receiver), c)
                       | NONE
                         => WRONG (STUCK_LABEL l)))
      | eval_cps (UPDATE (t, l, (x, t')), e, c)
        = eval_cps (t, e,
             fn receiver
                => (case update (receiver, l, ((x, t'), e))
                      of SOME rec_clo
                         => c rec_clo
                       | NONE
                         => WRONG (STUCK_LABEL l)))

    fun main t = eval_cps (t, empty_environment, fn v => VALUE v)

    end
end
```

Figure 9.10: CPS interpreter for the $\varsigma\rho$-calculus, with direct error propagation

```
structure Sigmarho_Defunctionalised_Interpreter : Sigmarho_Evaluator =
struct
    local open Util in

    datatype abs = ID
                 | INV of label * abs
                 | UPD of label * (id * term) * environment * abs

    fun apply_abs (ID, v)
        = VALUE v
      | apply_abs (INV (l, c), v)
        = (case lookup_label (v, l)
             of SOME ((var, body), e)
                => eval_defunct (body, extend_environment (e, var, v), c)
              | NONE
                => WRONG (STUCK_LABEL l))
      | apply_abs (UPD (l, (x, t), e, c), v)
        = (case update (v, l, ((x, t), e))
             of SOME rec_clo
                => apply_abs (c, rec_clo)
              | NONE
                => WRONG (STUCK_LABEL l))

    and eval_defunct (VAR x, e, c)
        = (case assoc (x, e)
             of SOME rec_clo
                => apply_abs (c, rec_clo)
              | NONE
                => WRONG (STUCK_VAR x))
      | eval_defunct (OBJECT v, e, c)
        = apply_abs (c, close_object (v, e))
      | eval_defunct (INVOKE (t, l), e, c)
        = eval_defunct (t, e, INV (l, c))
      | eval_defunct (UPDATE (t, l, (x, t')), e, c)
        = eval_defunct (t, e, UPD (l, (x, t'), e, c))

    fun main t = eval_defunct (t, empty_environment, ID)

    end
end
```

Figure 9.11: Defunctionalised CPS interpreter for the $\varsigma\rho$-calculus

We now defunctionalise the continuations introduced by the CPS transformation. The defunctionalisation introduces the new data type `abs` with constructors `ID`, `INV` and `UPD` representing the initial continuation, continuations introduced when evaluating invocation terms, and continuations introduced when evaluating update terms, respectively. Defunctionalisation also introduces the apply function `apply_abs`, which dispatches over values of type `abs`. The result of the defunctionalisation can be seen in Figure 9.11.

The defunctionalised interpreter is a direct implementation of the following abstract machine:

$$
\begin{array}{rcl}
\langle x,\, e,\, C\rangle & \Rightarrow_E & \langle C,\, v\rangle \\
 & & \quad \text{if } \textit{lookup}\ (x,\, e) = v \\
\langle [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}],\, e,\, C\rangle & \Rightarrow_E & \langle C,\, [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}]\rangle \\
\langle t.l,\, e,\, C\rangle & \Rightarrow_E & \langle t,\, e,\, C[[\,].l]\rangle \\
\langle t.l \Leftarrow \varsigma(x)t',\, e,\, C\rangle & \Rightarrow_E & \langle t,\, e,\, C[[\,].l \Leftarrow (\varsigma(x)t')[e]]\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle [\,],\, v\ \rangle & \Rightarrow_E & v \\
\langle C[[\,].l_j],\, v^n\rangle & \Rightarrow_E & \langle t_j,\, (x_j,\, v^n)\cdot e_j,\, C\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}] \\
\langle C[[\,].l_j \Leftarrow (\varsigma(x)t)[e]],\, v^n\rangle & \Rightarrow_E & \langle C,\, [l_j = (\varsigma(x)t)[e], l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]
\end{array}
$$

Notice that the machine is equipped with an environment. For a closed term t, the machine starts in the state $\langle t,\, \bullet,\, [\,]\rangle$, and continues until it is either stuck or reaches a result. The machine is stuck if it reaches either of the states $\langle C[[\,].l],\, v\rangle$ or $\langle C[[\,].l \Leftarrow (\varsigma(x)t')[e]],\, v\rangle$ and v does not contain a method labelled l. It is also stuck if it reaches a state $\langle x,\, e,\, C\rangle$, and x is not bound in e (which never happens when evaluating closed terms).

Since we have derived the abstract machine from the natural semantics using semantic-preserving program transformations, the natural semantics and the abstract machine specify the same language. In the following section, we derive the same abstract machine from the reduction semantics of the $\varsigma\rho$-calculus.

9.3 Syntactic Correspondence

For the syntactic correspondence, the starting point of the derivation is a reduction-based evaluator for the $\varsigma\rho$-calculus.

The evaluator contains four new data types, shown in Figure 9.12.

The first two, called `potential_redex` and `reduction_context`, correspond to the grammars for potential redexes and reduction contexts, respectively. The grammars were given in Section 3.1.2. The last two, called `contractum` and `value_or_decomposition`, are data types for the result of contraction and decomposition, respectively. The iteration function uses values of the type `contractum` to determine whether evaluation can continue or is stuck, and uses values of the type `value_or_decomposition` to determine whether a final value is reached or whether further reduction must take place. The uses of the last two data types will be eliminated during the derivation.

```
datatype potential_redex =
           INVOKE_REDEX of receiver_closure * label
         | UPDATE_REDEX of receiver_closure * label * ((id * term) * environment)
         | VAR_REDEX of id * environment
         | OBJECT_REDEX of receiver * environment
         | INVOKE_PROP of term * label * environment
         | UPDATE_PROP of term * label * ((id * term) * environment)

datatype reduction_context =
         ID_CONTEXT
       | INVOKE_CONTEXT of label * reduction_context
       | UPDATE_CONTEXT of label * ((id * term) * environment) *
           reduction_context

datatype contractum = ACTUAL_CONTRACTUM of closure
                    | STUCK_CONTRACTUM of wrong

datatype value_or_decomposition = VAL of result
                                | DEC of reduction_context * potential_redex
```

Figure 9.12: Data types of the reduction-based evaluator for the $\varsigma\rho$-calculus

Based on these data types, a direct implementation of the reduction semantics gives rise to the evaluator displayed in Figures 9.13 and 9.14. An extra rule has been introduced to handle the case where a final value is reached. The rule simply says that $v \to v$.

```
structure Sigmarho_Reductionbased : Sigmarho_Evaluator =
struct
    local open Util in

    fun contract (VAR_REDEX (x, e))
        = (case assoc (x, e)
             of SOME c
               => ACTUAL_CONTRACTUM (OBJECT_CLO c)
              | NONE
               => STUCK_CONTRACTUM (STUCK_VAR x))
      | contract (OBJECT_REDEX (r, e))
        = ACTUAL_CONTRACTUM (OBJECT_CLO (close_object (r, e)))
      | contract (INVOKE_REDEX (c, l))
        = (case lookup_label (c, l)
             of SOME ((var, body), e)
                => ACTUAL_CONTRACTUM
                        (GROUND_CLO (body, extend_environment (e, var, c)))
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (c, l, ((x, t), e)))
        = (case update (c, l, ((x, t), e))
             of SOME c'
                => ACTUAL_CONTRACTUM (OBJECT_CLO c')
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (INVOKE_PROP (t, l, e))
        = ACTUAL_CONTRACTUM (INVOKE_CLO (GROUND_CLO (t, e), l))
      | contract (UPDATE_PROP (t, l, ((x, t'), e)))
        = ACTUAL_CONTRACTUM (UPDATE_CLO (GROUND_CLO (t, e), l, ((x, t'), e)))

    fun plug (ID_CONTEXT, c)
        = c
      | plug (INVOKE_CONTEXT (l, rc), c)
        = plug (rc, INVOKE_CLO (c, l))
      | plug (UPDATE_CONTEXT (l, ((x, t), e), rc), c)
        = plug (rc, UPDATE_CLO (c, l, ((x, t), e)))
```

Figure 9.13: Reduction-based evaluator for the $\varsigma\rho$-calculus (continued in Figure 9.14)

```
37      fun decompose' (GROUND_CLO (VAR x, e), rc)
38          = DEC (rc, VAR_REDEX (x, e))
39        | decompose' (GROUND_CLO (OBJECT v, e), rc)
40          = DEC (rc, OBJECT_REDEX (v, e))
41        | decompose' (GROUND_CLO (INVOKE (t, l), e), rc)
42          = DEC (rc, INVOKE_PROP (t, l, e))
43        | decompose' (GROUND_CLO (UPDATE (t, l, (x, t')), e), rc)
44          = DEC (rc, UPDATE_PROP (t, l, ((x, t'), e)))
45        | decompose' (OBJECT_CLO c, rc)                    (* Extra rule *)
46          = VAL (VALUE c)               (* Can only happen when rc = ID_CONTEXT *)
47        | decompose' (INVOKE_CLO (OBJECT_CLO c, l), rc)
48          = DEC (rc, INVOKE_REDEX (c, l))
49        | decompose' (INVOKE_CLO (c, l), rc)
50          = decompose' (c, INVOKE_CONTEXT(l, rc))
51        | decompose' (UPDATE_CLO (OBJECT_CLO c, l, ((x, t), e)), rc)
52          = DEC (rc, UPDATE_REDEX (c, l, ((x, t), e)))
53        | decompose' (UPDATE_CLO (c, l, ((x, t), e)), rc)
54          = decompose' (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
55
56      fun decompose c
57          = decompose' (c, ID_CONTEXT)
58
59      fun iterate (VAL v)
60          = v
61        | iterate (DEC (rc, pr))
62          = (case contract pr
63               of ACTUAL_CONTRACTUM c
64                  => iterate (decompose (plug (rc, c)))
65                | STUCK_CONTRACTUM w
66                  => WRONG w)
67
68      fun main t = iterate (decompose (plug (ID_CONTEXT,
69                                             GROUND_CLO (t, empty_environment))))
70
71      end
72  end
```

Figure 9.14: Reduction-based evaluator for the $\varsigma\rho$-calculus (continued from Figure 9.13)

Refocusing the reduction-based evaluator yields the evaluator displayed in Figures 9.15 and 9.16. The functions `decompose` and `plug` have now been eliminated, and the `refocus` function has taken their place. Given a closure and a context, `refocus` will now find the next potential redex directly, rather than reconstructing the entire closure and subsequently decomposing it.

```
structure Sigmarho_Refocused : Sigmarho_Evaluator =
struct
    local open Util in

    fun contract (VAR_REDEX (x, e))
        = (case assoc (x, e)
             of SOME c
                => ACTUAL_CONTRACTUM (OBJECT_CLO c)
              | NONE
                => STUCK_CONTRACTUM (STUCK_VAR x))
      | contract (OBJECT_REDEX (r, e))
        = ACTUAL_CONTRACTUM (OBJECT_CLO (close_object (r, e)))
      | contract (INVOKE_REDEX (c, l))
        = (case lookup_label (c, l)
             of SOME ((var, body), e)
                => ACTUAL_CONTRACTUM
                       (GROUND_CLO (body, extend_environment (e, var, c)))
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (c, l, ((x, t), e)))
        = (case update (c, l, ((x, t), e))
             of SOME c'
                => ACTUAL_CONTRACTUM (OBJECT_CLO c')
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (INVOKE_PROP (t, l, e))
        = ACTUAL_CONTRACTUM (INVOKE_CLO (GROUND_CLO (t, e), l))
      | contract (UPDATE_PROP (t, l, ((x, t'), e)))
        = ACTUAL_CONTRACTUM (UPDATE_CLO (GROUND_CLO (t, e), l, ((x, t'), e)))

    fun refocus' (GROUND_CLO (VAR x, e), rc )
        = DEC (rc, VAR_REDEX (x, e))
      | refocus' (GROUND_CLO (OBJECT v, e), rc)
        = DEC (rc, OBJECT_REDEX (v, e))
      | refocus' (GROUND_CLO (INVOKE (t, l), e), rc)
        = DEC (rc, INVOKE_PROP (t, l, e))
      | refocus' (GROUND_CLO (UPDATE (t, l, (x, t')), e), rc)
        = DEC (rc, UPDATE_PROP (t, l, ((x, t'), e)))
      | refocus' (OBJECT_CLO c, rc)
        = refocus'_aux (c, rc)
      | refocus' (INVOKE_CLO (c, l), rc)
        = refocus' (c, INVOKE_CONTEXT(l, rc))
      | refocus' (UPDATE_CLO (c, l, ((x, t), e)), rc)
        = refocus' (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
    and refocus'_aux (c, ID_CONTEXT)
        = VAL (VALUE c)
      | refocus'_aux (c, INVOKE_CONTEXT (l, rc))
        = DEC (rc, INVOKE_REDEX (c, l))
      | refocus'_aux (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
        = DEC (rc, UPDATE_REDEX (c, l, ((x, t), e)))

    fun refocus (c, rc) = refocus' (c, rc)
```

Figure 9.15: Refocused evaluator for the $\varsigma\rho$-calculus (continued in Figure 9.16)

```
53      fun iterate (VAL v)
54          = v
55        | iterate (DEC (rc, pr))
56          = (case contract pr
57               of ACTUAL_CONTRACTUM c
58                  => iterate (refocus' (c, rc))
59                | STUCK_CONTRACTUM w
60                  => WRONG w)
61
62      fun main t = iterate (refocus
63                              (GROUND_CLO (t, empty_environment), ID_CONTEXT))
64
65      end
66  end
```

Figure 9.16: Refocused evaluator for the $\varsigma\rho$-calculus (continued from Figure 9.15)

Fusing `iterate` and `refocus` functions using fixed-point promotion yields the evaluator in Figures 9.17 and 9.18. The fixed-point promotion has eliminated all calls to functions `iterate`, `refocus`, `refocus'` and `refocus'_aux`, so their definitions have been removed. Similarly, there is no longer any need for the data type `value_or_decomposition`, since `iterate_refocus` now either returns the found value directly, or calls the contraction function on the found redex.

```
structure Sigmarho_Refocused_Fused : Sigmarho_Evaluator =
struct
    local open Util in

    fun contract (VAR_REDEX (x, e))
        = (case assoc (x, e)
             of SOME c
                => ACTUAL_CONTRACTUM (OBJECT_CLO c)
              | NONE
                => STUCK_CONTRACTUM (STUCK_VAR x))
      | contract (OBJECT_REDEX (r, e))
        = ACTUAL_CONTRACTUM (OBJECT_CLO (close_object (r, e)))
      | contract (INVOKE_REDEX (c, l))
        = (case lookup_label (c, l)
             of SOME ((var, body), e)
                => ACTUAL_CONTRACTUM
                        (GROUND_CLO (body, extend_environment (e, var, c)))
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (UPDATE_REDEX (c, l, ((x, t), e)))
        = (case update (c, l, ((x, t), e))
             of SOME c'
                => ACTUAL_CONTRACTUM (OBJECT_CLO c')
              | NONE
                => STUCK_CONTRACTUM (STUCK_LABEL l))
      | contract (INVOKE_PROP (t, l, e))
        = ACTUAL_CONTRACTUM (INVOKE_CLO (GROUND_CLO (t, e), l))
      | contract (UPDATE_PROP (t, l, ((x, t'), e)))
        = ACTUAL_CONTRACTUM (UPDATE_CLO (GROUND_CLO (t, e), l, ((x, t'), e)))
```

Figure 9.17: Refocused and fused evaluator for the $\varsigma\rho$-calculus (continued in Figure 9.18)

```
30
31      fun iterate_refocus (GROUND_CLO (VAR x, e), rc )
32          = (case contract (VAR_REDEX (x, e))
33               of ACTUAL_CONTRACTUM c
34                  => iterate_refocus (c, rc)
35                | STUCK_CONTRACTUM w
36                  => WRONG w)
37        | iterate_refocus (GROUND_CLO (OBJECT v, e), rc)
38          = (case contract (OBJECT_REDEX (v, e))
39               of ACTUAL_CONTRACTUM c
40                  => iterate_refocus (c, rc)
41                | STUCK_CONTRACTUM w
42                  => WRONG w)
43        | iterate_refocus (GROUND_CLO (INVOKE (t, l), e), rc)
44          = (case contract (INVOKE_PROP (t, l, e))
45               of ACTUAL_CONTRACTUM c
46                  => iterate_refocus (c, rc)
47                | STUCK_CONTRACTUM w
48                  => WRONG w)
49        | iterate_refocus (GROUND_CLO (UPDATE (t, l, (x, t')), e), rc)
50          = (case contract (UPDATE_PROP (t, l, ((x, t'), e)))
51               of ACTUAL_CONTRACTUM c
52                  => iterate_refocus (c, rc)
53                | STUCK_CONTRACTUM w
54                  => WRONG w)
55        | iterate_refocus (OBJECT_CLO c, rc)
56          = iterate_refocus_aux (c, rc)
57        | iterate_refocus (INVOKE_CLO (c, l), rc)
58          = iterate_refocus (c, INVOKE_CONTEXT(l, rc))
59        | iterate_refocus (UPDATE_CLO (c, l, ((x, t), e)), rc)
60          = iterate_refocus (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
61      and iterate_refocus_aux (c, ID_CONTEXT)
62          = VALUE c
63        | iterate_refocus_aux (c, INVOKE_CONTEXT (l, rc))
64          = (case contract (INVOKE_REDEX (c, l))
65               of ACTUAL_CONTRACTUM c
66                  => iterate_refocus (c, rc)
67                | STUCK_CONTRACTUM w
68                  => WRONG w)
69        | iterate_refocus_aux (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
70          = (case contract (UPDATE_REDEX (c, l, ((x, t), e)))
71               of ACTUAL_CONTRACTUM c
72                  => iterate_refocus (c, rc)
73                | STUCK_CONTRACTUM w
74                  => WRONG w)
75
76      fun main t = iterate_refocus (GROUND_CLO (t, empty_environment), ID_CONTEXT)
77
78      end
79  end
```

Figure 9.18: Refocused and fused evaluator for the $\varsigma\rho$-calculus (continued from Figure 9.17)

```
structure Sigmarho_Refocused_Fused_Inlined : Sigmarho_Evaluator =
struct
    local open Util in

    fun iterate_refocus (GROUND_CLO (VAR x, e), rc)
        = (case assoc (x, e)
             of SOME c
                => iterate_refocus (OBJECT_CLO c, rc)
              | NONE
                => WRONG (STUCK_VAR x))
      | iterate_refocus (GROUND_CLO (OBJECT v, e), rc)
        = iterate_refocus (OBJECT_CLO (close_object (v, e)), rc)
      | iterate_refocus (GROUND_CLO (INVOKE (t, l), e), rc)
        = iterate_refocus (INVOKE_CLO (GROUND_CLO (t, e), l), rc)
      | iterate_refocus (GROUND_CLO (UPDATE (t, l, (x, t')), e), rc)
        = iterate_refocus (UPDATE_CLO (GROUND_CLO (t, e), l, ((x, t'), e)), rc)
      | iterate_refocus (OBJECT_CLO c, rc)
        = iterate_refocus_aux (c, rc)
      | iterate_refocus (INVOKE_CLO (c, l), rc)
        = iterate_refocus (c, INVOKE_CONTEXT(l, rc))
      | iterate_refocus (UPDATE_CLO (c, l, ((x, t), e)), rc)
        = iterate_refocus (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
    and iterate_refocus_aux (c, ID_CONTEXT)
        = VALUE c
      | iterate_refocus_aux (c, INVOKE_CONTEXT (l, rc))
        = (case lookup_label (c, l)
             of SOME ((var, body), e)
                => iterate_refocus
                         (GROUND_CLO (body, extend_environment (e, var, c)), rc)
              | NONE
                => WRONG (STUCK_LABEL l))
      | iterate_refocus_aux (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
        = (case update (c, l, ((x, t), e))
             of SOME c'
                => iterate_refocus (OBJECT_CLO c', rc)
              | NONE
                => WRONG (STUCK_LABEL l))

    fun main t =  iterate_refocus (GROUND_CLO (t, empty_environment), ID_CONTEXT)

    end
end
```

Figure 9.19: Evaluator for the $\varsigma\rho$-calculus, with inlined contraction function

Since the contraction function is now always called on known redex types, we can inline the body of `contract` and simplify the terms. In doing so, we eliminate the need for the `contractum` data type. The result can be seen in Figure 9.19.

```
structure Sigmarho_Simplified : Sigmarho_Evaluator =
struct
  local open Util in

  fun iterate_refocus (GROUND_CLO (VAR x, e), rc)
      = (case assoc (x, e)
           of SOME c
              => iterate_refocus_aux (c, rc)
            | NONE
              => WRONG (STUCK_VAR x))
    | iterate_refocus (GROUND_CLO (OBJECT v, e), rc)
      = iterate_refocus_aux (close_object (v, e), rc)
    | iterate_refocus (GROUND_CLO (INVOKE (t, l), e), rc)
      = iterate_refocus (GROUND_CLO (t, e), INVOKE_CONTEXT(l, rc))
    | iterate_refocus (GROUND_CLO (UPDATE (t, l, (x, t')), e), rc)
      = iterate_refocus (GROUND_CLO (t, e),
                         UPDATE_CONTEXT (l, ((x, t'), e), rc))
    | iterate_refocus (OBJECT_CLO c, rc)                    (* No longer called *)
      = iterate_refocus_aux (c, rc)
    | iterate_refocus (INVOKE_CLO (c, l), rc)               (* No longer called *)
      = iterate_refocus (c, INVOKE_CONTEXT(l, rc))
    | iterate_refocus (UPDATE_CLO (c, l, ((x, t), e)), rc) (* No longer called *)
      = iterate_refocus (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
  and iterate_refocus_aux (c, ID_CONTEXT)
      = VALUE c
    | iterate_refocus_aux (c, INVOKE_CONTEXT (l, rc))
      = (case lookup_label (c, l)
           of SOME ((var, body), e)
              => iterate_refocus
                       (GROUND_CLO (body, extend_environment (e, var, c)), rc)
            | NONE
              => WRONG (STUCK_LABEL l))
    | iterate_refocus_aux (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
      = (case update (c, l, ((x, t), e))
           of SOME c'
              => iterate_refocus_aux (c', rc)
            | NONE
              => WRONG (STUCK_LABEL l))

  fun main t = iterate_refocus (GROUND_CLO (t, empty_environment), ID_CONTEXT)

  end
end
```

Figure 9.20: Simplified evaluator for the $\varsigma\rho$-calculus

The evaluator now calls itself recursively on known closure types. Inlining the function body of `iterate_refocus` whenever possible yields a simplified evaluator in Figure 9.20.

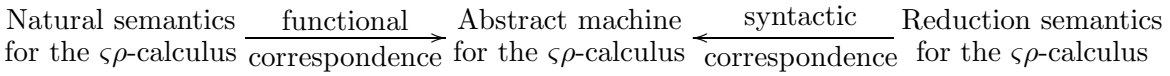
```
structure Sigmarho_Closure_Eliminated : Sigmarho_Evaluator =
struct
    local open Util in

    fun iterate_refocus (VAR x, e, rc)
        = (case assoc (x, e)
             of SOME c
                => iterate_refocus_aux (c, rc)
              | NONE
                => WRONG (STUCK_VAR x))
      | iterate_refocus (OBJECT v, e, rc)
        = iterate_refocus_aux (close_object (v, e), rc)
      | iterate_refocus (INVOKE (t, l), e, rc)
        = iterate_refocus (t, e, INVOKE_CONTEXT(l, rc))
      | iterate_refocus (UPDATE (t, l, (x, t')), e, rc)
        = iterate_refocus (t, e, UPDATE_CONTEXT (l, ((x, t'), e), rc))
    and iterate_refocus_aux (c, ID_CONTEXT)
        = VALUE c
      | iterate_refocus_aux (c, INVOKE_CONTEXT (l, rc))
        = (case lookup_label (c, l)
             of SOME ((var, body), e)
                => iterate_refocus (body, extend_environment (e, var, c), rc)
              | NONE
                => WRONG (STUCK_LABEL l))
      | iterate_refocus_aux (c, UPDATE_CONTEXT (l, ((x, t), e), rc))
        = (case update (c, l, ((x, t), e))
             of SOME c'
                => iterate_refocus_aux (c', rc)
              | NONE
                => WRONG (STUCK_LABEL l))

    fun main t = iterate_refocus (t, empty_environment, ID_CONTEXT)

    end
end
```

Figure 9.21: Evaluator for the $\varsigma\rho$-calculus after closure elimination

The evaluator now only creates ground closures, and so no other closure types are ever used in the evaluation of a term. We can therefore eliminate the `closure` data type altogether by 'lifting' the contents of the `GROUND_CLO`s in the entire program. In doing so, we also eliminate the now unused cases in `iterate_refocus`. The resulting evaluator is displayed in Figure 9.21.

The evaluator implements a state-transition system, i.e., an abstract machine. Furthermore, the abstract machine is the same as the one derived by the functional correspondence, up to reordering of state components. This fact can be verified by observing that the types `abs` and `reduction_context` are isomorphic, and that `iterate_refocus` and `iterate_refocus_aux` are equivalent to `eval_defunct` and `apply_abs`, respectively.

Since the abstract machines are identical, the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus (and, of course, the abstract machine) define the same language. This equivalence is illustrated in the following diagram:



9.4 Summary and Conclusions

We have derived two abstract machines for the $\varsigma\rho$-calculus; one from the natural semantics using the functional correspondence, and one from the reduction semantics using the syntactic correspondence. The two machines are identical, and provide yet another language description for the $\varsigma\rho$-calculus, along with the natural and reduction semantics given in Chapter 3.

Since the functional and syntactic correspondences are both semantic preserving, the interpreter, the evaluator and the abstract machine define the same language. The fact that the two derived abstract machines are identical proves the equivalence between the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus. Furthermore, the proof has been obtained by mechanical derivation rather than by using pen and paper.

Chapter 10

Formal Connection

The two previous chapters have presented two abstract machines, one for the ς-calculus and one for the $\varsigma\rho$-calculus. Since for each language, the abstract machine can be derived from both the natural semantics and the reduction semantics using semantics-preserving transformations, the three language specifications are equivalent.

We now turn to the question of equivalence between the two calculi. This chapter presents the formal connection between the ς-calculus and the $\varsigma\rho$-calculus by proving that the abstract machines for the two languages are strongly bisimilar.

We first consider the relation between values of the ς-calculus and values of the $\varsigma\rho$-calculus. $\varsigma\rho$-values are object closures, and hence, they contain environments. These environments contain the substitutions that were 'delayed' during the evaluation that resulted in the closure, so by 'forcing' these substitutions to be carried out, we obtain a ς-value. Forcing the delayed substitutions induces a mapping from $\varsigma\rho$-values to ς-values, and we present this mapping formally, as well as similar mappings for terms, methods and contexts.

We then present a relation between states of the two abstract machines. This relation is defined in terms of the mappings above, and can informally be thought of as forcing all substitutions that have been captured in the current environment and context, and in the object closures and method closures created during evaluation.

Finally, we prove that the relation is a strong bisimulation, thereby establishing the computational equivalence of the two calculi.

10.1 From Closures to Terms

Since the category of values of the ς-calculus is different from the category of values of the $\varsigma\rho$-calculus, we must define a mapping from $\varsigma\rho$-values to ς-values in order to prove computational equivalence between the two calculi. Since $\varsigma\rho$-values contain method closures, a similar mapping must be defined for method closures. We let the set $FV(t)$ denotes the set of free variables of t (as defined by Abadi and Cardelli [1, page 61]). The mappings are then defined by simultaneous induction as follows:

$$sub_{\mathbf{V}}([l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]) \;=\; [l_i = sub_{\mathbf{M}}(\varsigma(x_i)t_i,\, e_i)^{i\in\{1\ldots n\}}]$$

$$sub_{\mathbf{M}}(\varsigma(x)t,\, e) \;=\; \varsigma(x)t\{sub_{\mathbf{V}}(v_i)/x_i\} \qquad \begin{array}{l}\forall x_i \in FV(t) \setminus \{x\},\\ \text{where } \textit{lookup}\ (x_i,\, e) = v_i\end{array}$$

Here we have extended the notation for substitutions to account for the simultaneous substitution of multiple values for multiple variables. $sub_{\mathbf{V}}$ maps values of the $\varsigma\rho$-calculus to values of the ς-calculus, and $sub_{\mathbf{M}}$ maps method closures to methods.

Since the mappings force the substitution of all free variables in a value or method, the mappings are only well-defined provided that all free variables of the value or method are bound in the corresponding environment (including free variables of the closures bound in those environments, which must be bound in the environments contained in those closures, and so on). For closures generated by the $\varsigma\rho$ abstract machine when evaluation starts with a closed term, such bindings will always exist.

We also need a mapping from intermediate terms of the $\varsigma\rho$ abstract machine to intermediate terms in the ς abstract machine, and a mapping from $\varsigma\rho$-contexts to ς-contexts. The intermediate terms of the $\varsigma\rho$ abstract machine in general contain free variables, so the mapping must also take an environment into account. The mappings are defined as follows, using $sub_{\mathbf{V}}$ and $sub_{\mathbf{M}}$ as defined above:

$$sub_{\mathbf{T}}(t,\, e) \;=\; t\{sub_{\mathbf{V}}(v_i)/x_i\} \qquad \begin{array}{l}\forall x_i \in FV(t),\\ \text{where } \textit{lookup}\ (x_i,\, e) = v_i\end{array}$$

$$\begin{aligned} sub_{\mathbf{C}}([\,]) &= [\,]\\ sub_{\mathbf{C}}(C[[\,].l]) &= (sub_{\mathbf{C}}(C))[[\,].l]\\ sub_{\mathbf{C}}(C[[\,].l \Leftarrow (\varsigma(x)t)[e]]) &= (sub_{\mathbf{C}}(C))[[\,].l \Leftarrow sub_{\mathbf{M}}(\varsigma(x)t,\, e)] \end{aligned}$$

$sub_{\mathbf{T}}$ maps (term, environment) pairs to terms, and $sub_{\mathbf{C}}$ maps contexts of the $\varsigma\rho$-calculus to contexts of the ς-calculus. Again, these mappings are only well-defined if all free variables are bound in the corresponding environment.

In the following section, we use these mappings to define a bisimulation relation on the two abstract machines.

10.2 Bisimilarity

To prove bisimilarity between the two abstract machines, we must define a relation between states of the ς abstract machine and the states of the $\varsigma\rho$ abstract machine. The relation is defined using the mappings from the previous sections, as follows:

Definition 1. *Let $ST_{\varsigma\rho}$ denote the set of states that the $\varsigma\rho$ abstract machine may enter when evaluating a closed term, and let ST_{ς} denote the set of states that the ς abstract machine may enter when evaluating a closed term. The substitution relation $\simeq_S$: $ST_{\varsigma\rho} \times ST_{\varsigma}$ is defined as follows:*

$$\begin{aligned} \langle t,\, e,\, C\rangle &\simeq_S \langle sub_{\mathbf{T}}(t,e),\, sub_{\mathbf{C}}(C)\rangle\\ \langle C,\, v\rangle &\simeq_S \langle sub_{\mathbf{C}}(C),\, sub_{\mathbf{V}}(v)\rangle\\ v &\simeq_S sub_{\mathbf{V}}(v) \end{aligned}$$

$$
\begin{array}{rcl}
\langle v,\, C\rangle & \Rightarrow_S & \langle C,\, v\rangle \\
\langle t.l,\, C\rangle & \Rightarrow_S & \langle t,\, C[[\,].l]\rangle \\
\langle t.l \Leftarrow \varsigma(x)t',\, C\rangle & \Rightarrow_S & \langle t,\, C[[\,].l \Leftarrow \varsigma(x)t']\rangle \\
 & & \\
\langle [\,],\, v\ \rangle & \Rightarrow_S & v \\
\langle C[[\,].l_j],\, v^n\rangle & \Rightarrow_S & \langle t_j\{v^n/x_j\},\, C\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}] \\
\langle C[[\,].l_j \Leftarrow \varsigma(x)t],\, v^n\rangle & \Rightarrow_S & \langle C,\, [l_j = \varsigma(x)t,\ l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}]
\end{array}
$$

Figure 10.1: The abstract machine for the ς-calculus, from page 46

$$
\begin{array}{rcl}
\langle x,\, e,\, C\rangle & \Rightarrow_E & \langle C,\, v\rangle \\
 & & \quad \text{if } \textit{lookup}\ (x,\, e) = v \\
\langle [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}],\, e,\, C\rangle & \Rightarrow_E & \langle C,\, [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}]\rangle \\
\langle t.l,\, e,\, C\rangle & \Rightarrow_E & \langle t,\, e,\, C[[\,].l]\rangle \\
\langle t.l \Leftarrow \varsigma(x)t',\, e,\, C\rangle & \Rightarrow_E & \langle t,\, e,\, C[[\,].l \Leftarrow (\varsigma(x)t')[e]]\rangle \\
 & & \\
\langle [\,],\, v\ \rangle & \Rightarrow_E & v \\
\langle C[[\,].l_j],\, v^n\rangle & \Rightarrow_E & \langle t_j,\, (x_j,\, v^n)\cdot e_j,\, C\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}] \\
\langle C[[\,].l_j \Leftarrow (\varsigma(x)t)[e]],\, v^n\rangle & \Rightarrow_E & \langle C,\, [l_j = (\varsigma(x)t)[e], l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle \\
 & & \quad \text{if } 1 \le j \le n, \text{ where } v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]
\end{array}
$$

Figure 10.2: The abstract machine for the $\varsigma\rho$-calculus, from page 61

For ease of reference, the abstract machines are shown again in Figures 10.1 and 10.2. We are now ready to prove the main result of this chapter, namely that the relation in Definition 1 is a bisimulation:

Theorem 1. *For the evaluation of closed terms, the abstract machines from Sections 8.2 and 9.2 are strongly bisimilar with respect to $\simeq_S$.*

Proof. By induction on the execution of the abstract machine for the $\varsigma\rho$-calculus:

Base case: Let t be a closed term. The starting state in the evaluation of t in the $\varsigma\rho$ abstract machine is $\langle t,\, \bullet,\, [\,]\rangle$, which is in relation to $\langle sub_{\mathbf{T}}(t, \bullet),\, sub_{\mathbf{C}}([\,])\rangle$. By the definitions of $sub_{\mathbf{T}}$ and $sub_{\mathbf{C}}$, $\langle sub_{\mathbf{T}}(t, \bullet),\, sub_{\mathbf{C}}([\,])\rangle = \langle t,\, [\,]\rangle$, which is the starting state in the evaluation of t in the ς abstract machine, so the two machines start in related states.

Induction step: Let $s_e \simeq_S s_s$, and let $s_s \Rightarrow_S s'_s$ and $s_e \Rightarrow_E s'_e$. We must show that $s'_e \simeq_S s'_s$. This is done by case on s_e:

- $s_e = \langle x, e, C\rangle$:
  Then $s'_e = \langle C, v\rangle$ and $s_s = \langle sub_{\mathbf{T}}(x, e), sub_{\mathbf{C}}(C)\rangle = \langle sub_{\mathbf{V}}(v), sub_{\mathbf{C}}(C)\rangle$, where $v = lookup\ (x, e)$, and by the definition of $\Rightarrow_S$, $s'_s = \langle sub_{\mathbf{C}}(C), sub_{\mathbf{V}}(v)\rangle$.

  By the definition of $\simeq_S$, $\langle C, v\rangle \simeq_S \langle sub_{\mathbf{C}}(C), sub_{\mathbf{V}}(v)\rangle$, and so $s'_e \simeq_S s'_s$.

  If x is not bound in e, then the original term was not closed.

- $s_e = \langle [l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}], e, C\rangle$:
  Then $s'_e = \langle C, [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}]\rangle$ and $s_s = \langle sub_{\mathbf{T}}([l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}], e), C\rangle$. $sub_{\mathbf{T}}([l_i = \varsigma(x_i)t_i{}^{i\in\{1\ldots n\}}], e) = sub_{\mathbf{V}}([l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}])$ since x cannot occur free in $\varsigma(x)t$, and hence by the definition of $\Rightarrow_S$, $s'_s = \langle C, sub_{\mathbf{V}}([l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}])\rangle$.

  By definition, $\langle C, [l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}]\rangle \simeq_S \langle C, sub_{\mathbf{V}}([l_i = (\varsigma(x_i)t_i)[e]^{i\in\{1\ldots n\}}])\rangle$, and so $s'_e \simeq_S s'_s$.

- $s_e = \langle t.l, e, C\rangle$:
  Then $s'_e = \langle t, e, C[[\,].l]\rangle$ and $s_s = \langle sub_{\mathbf{T}}(t.l, e), sub_{\mathbf{C}}(C)\rangle = \langle sub_{\mathbf{T}}(t, e).l, sub_{\mathbf{C}}(C)\rangle$, and so by the definition of $\Rightarrow_S$, $s'_s = \langle sub_{\mathbf{T}}(t, e), sub_{\mathbf{C}}(C)[[\,].l]\rangle$.

  By the definition of $sub_{\mathbf{C}}$, $sub_{\mathbf{C}}(C)[[\,].l] = sub_{\mathbf{C}}(C[[\,].l])$, and by the definition of $\simeq_S$, $\langle t, e, C[[\,].l]\rangle \simeq_S \langle sub_{\mathbf{T}}(t, e), sub_{\mathbf{C}}(C[[\,].l])\rangle$ and so $s'_e \simeq_S s'_s$.

- $s_e = \langle t.l \Leftarrow \varsigma(x)t', e, C\rangle$:
  Then $s'_e = \langle t, e, C[[\,].l \Leftarrow (\varsigma(x)t')[e]]\rangle$ and $s_s = \langle sub_{\mathbf{T}}(t.l \Leftarrow \varsigma(x)t', e), sub_{\mathbf{C}}(C)\rangle$. Since x cannot occur free in $\varsigma(x)t'$, $sub_{\mathbf{T}}(t.l \Leftarrow \varsigma(x)t', e) = sub_{\mathbf{T}}(t, e).l \Leftarrow sub_{\mathbf{M}}(\varsigma(x)t', e)$, and so by the definition of $\Rightarrow_S$, $s'_s = \langle sub_{\mathbf{T}}(t, e), sub_{\mathbf{C}}(C)[[\,].l \Leftarrow sub_{\mathbf{M}}(\varsigma(x)t', e)]\rangle$.

  By definition, $sub_{\mathbf{C}}(C)[[\,].l \Leftarrow sub_{\mathbf{M}}(\varsigma(x)t', e)] = sub_{\mathbf{C}}(C[[\,].l \Leftarrow (\varsigma(x)t')[e]])$, and by the definition of $\simeq_S$, $\langle t, e, C[[\,].l \Leftarrow (\varsigma(x)t')[e]]\rangle \simeq_S \langle sub_{\mathbf{T}}(t, e), sub_{\mathbf{C}}(C[[\,].l \Leftarrow (\varsigma(x)t')[e]])\rangle$ and so $s'_e \simeq_S s'_s$.

- $s_e = \langle [\,], v\rangle$:
  Then $s'_e = v$ and $s_s = \langle sub_{\mathbf{C}}([\,]), sub_{\mathbf{V}}(v)\rangle$, and since $sub_{\mathbf{C}}([\,]) = [\,]$, then by the definition of $\Rightarrow_S$, $s'_s = sub_{\mathbf{V}}(v)$.

  By the definition $\simeq_S$, $v \simeq_S sub_{\mathbf{V}}(v)$, and so $s'_e \simeq_S s'_s$.

- $s_e = \langle C[[\,].l_j], v^n\rangle$, where $v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]$ and $1 \leq j \leq n$:
  Then $s'_e = \langle t_j, (x_j, v^n) \cdot e_j, C\rangle$ and $s_s = \langle sub_{\mathbf{C}}(C[[\,].l_j]), sub_{\mathbf{V}}(v^n)\rangle$.
  Let $sub_{\mathbf{M}}(\varsigma(x_j)t_j, e_j) = \varsigma(x_j)t'_j$, for some t'_j. Then, since $sub_{\mathbf{C}}(C[[\,].l_j]) = sub_{\mathbf{C}}(C)[[\,].l_j]$ we get by the definition of $\Rightarrow_S$ that $s'_s = \langle t'_j\{sub_{\mathbf{V}}(v^n)/x_j\}, sub_{\mathbf{C}}(C)\rangle$.

  Since $sub_{\mathbf{M}}(\varsigma(x_j)t_j, e_j)$ substitutes all free variables in t_j except occurrences of x_j to obtain t'_j, we have that $sub_{\mathbf{T}}(t_j, (x_j, v^n) \cdot e_j) = t'_j\{sub_{\mathbf{V}}(v^n)/x_j\}$. By the definition of $\simeq_S$, we then get that $\langle t_j, (x_j, v^n) \cdot e_j, C\rangle \simeq_S \langle t'_j\{sub_{\mathbf{V}}(v^n)/x_j\}, sub_{\mathbf{C}}(C)\rangle$, and so $s'_e \simeq_S s'_s$.

  If $j < 1$ or $j > n$, then both machines are stuck.

- $s_e = \langle C[[\,].l_j \Leftarrow (\varsigma(x)t)[e]],\, v^n\rangle$, where $v^n = [l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}}]$ and $1 \le j \le n$: Then $s'_e = \langle C,\, [l_j = (\varsigma(x)t)[e], l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle$ and by the definition of $\simeq_S$ and $sub_{\mathbf{V}}$, $s_s = \langle sub_{\mathbf{C}}(C[[\,].l_j \Leftarrow (\varsigma(x)t)[e]]),\, [l_i = sub_{\mathbf{M}}(\varsigma(x_i)t_i,\, e_i)^{i\in\{1\ldots n\}}]\rangle$.

  Since $sub_{\mathbf{C}}(C[[\,].l_j \Leftarrow (\varsigma(x)t)[e]]) = sub_{\mathbf{C}}(C)[[\,].l_j \Leftarrow sub_{\mathbf{M}}(\varsigma(x)t,\, e)]$, then by the definition of $\Rightarrow_S$, $s'_s = \langle sub_{\mathbf{C}}(C),\, [l_j = sub_{\mathbf{M}}(\varsigma(x)t,\, e),\, l_i = sub_{\mathbf{M}}(\varsigma(x_i)t_i,\, e_i)^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle$.

  By the definition $\simeq_S$ and $sub_{\mathbf{V}}$, $\langle C,\, [l_j = (\varsigma(x)t)[e], l_i = (\varsigma(x_i)t_i)[e_i]^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle \simeq_S$ $\langle sub_{\mathbf{C}}(C),\, [l_j = sub_{\mathbf{M}}(\varsigma(x)t,\, e),\, l_i = sub_{\mathbf{M}}(\varsigma(x_i)t_i,\, e_i)^{i\in\{1\ldots n\}\setminus\{j\}}]\rangle$, and so $s'_e \simeq_S s'_s$.

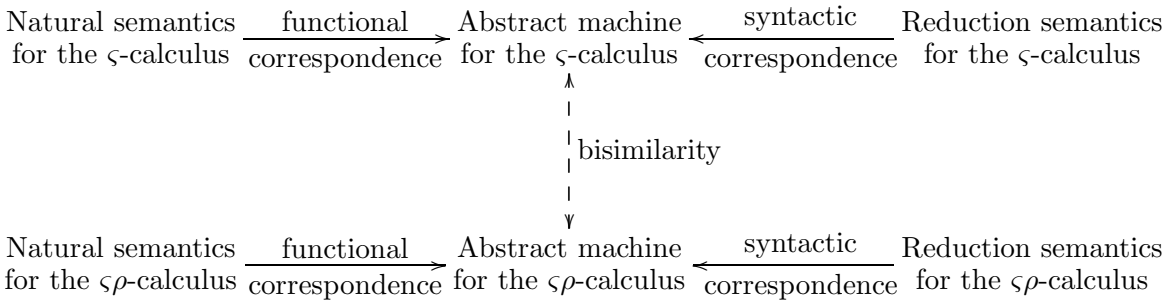
  If $j < 1$ or $j > n$, then both machines are stuck.

□

Corollary 1 (Full correctness of the $\varsigma\rho$-calculus)**.** *If the $\varsigma\rho$ abstract machine computes a value v as the result of a closed term t, then the ς abstract machine computes the value $sub_{\mathbf{V}}(v)$ for the same term.*

10.3 Computational Equivalence

Since the abstract machine for the ς-calculus and the abstract machine for the $\varsigma\rho$-calculus are bisimilar, we conclude that the ς-calculus and the $\varsigma\rho$-calculus are computationally equivalent. Hence, we can complete our diagram by connecting the two abstract machines:



Since the mappings from Section 10.1 are only defined when all free variables of the terms, methods and values in question are bound in the corresponding environments, the proof of bisimulation is only valid when the machines evaluate closed terms. However, if we extend the mappings so that unbound variables are mapped to themselves, bisimulation still holds. Of course, if the machines ever enter a state in which such a variable must be evaluated, they are both stuck (one because the variable is not bound in the current environment and the other because no transition exists to handle variables).

10.4 Summary and Conclusions

This chapter has presented a formal connection between the ς-calculus and the $\varsigma\rho$-calculus.

To connect the categories of values of the two calculi, we have presented mappings from $\varsigma\rho$-values to ς-values. We have also presented similar mappings for methods, terms and

contexts. Using these mappings, we have defined a relation between states of the abstract machine for the $\varsigma\rho$-calculus to states of the abstract machine for the ς-calculus. We have proved that this relation is a strong bisimulation.

The bisimilarity result implies that the two calculi are computationally equivalent. In other words, the semantic descriptions given in Chapters 2 and 3, as well as the two abstract machines derived in Chapters 8 and 9, all specify the same language.

Chapter 11

Conclusion – Derivation and Equivalence

We have presented two abstract machines, one for the ς-calculus and one for the $\varsigma\rho$-calculus. The two machines have been obtained by derivation from the semantic descriptions of the calculi, using the functional and syntactic correspondences.

Since both the abstract machines can be derived from both the natural semantics and the reduction semantics of their respective calculi, they prove the relative computational equivalence between those two semantic descriptions. Furthermore, the proof has been established by derivation, rather than by pen and paper.

We have also presented a mapping from values of the $\varsigma\rho$-calculus to values of the ς-calculus. Using this mapping, we have proved that the abstract machine for the ς-calculus and the abstract machine for the $\varsigma\rho$-calculus are strongly bisimilar, which establishes the computational equivalence between the two calculi.

Part IV

Conclusion

Chapter 12

Conclusion and Perspectives

In this final chapter of the dissertation we summarise our results, and we present possible directions for further study.

12.1 Summary

In this dissertation, we have seen how natural semantics, abstract machines and reduction semantics for Abadi and Cardelli's untyped ς-calculus can be inter-derived using Danvy et al.'s functional and syntactic correspondences.

Using the functional and syntactic correspondences, we have derived an abstract machine from both the natural semantics and the reduction semantics of the ς-calculus. The machine is new, and the fact that it can be derived from both the natural semantics and the reduction semantics by means of semantic-preserving program transformations establishes the computational equivalence of the two semantic specifications.

We have also presented a new version of the ς-calculus. This new calculus, called the $\varsigma\rho$-calculus, is defined using explicit substitutions. The use of explicit substitutions, as opposed to the actual substitutions of the ς-calculus, allows for a calculus which is closer to realistic language implementations. Indeed, source terms are now invariant through execution (instead of being modified by actual substitution), and therefore, their representation can be changed, e.g., by a compiler.

Using the functional and syntactic correspondences, we have derived an abstract machine from both the natural semantics and the reduction semantics of the $\varsigma\rho$-calculus, similar to how it was done for the ς-calculus. Again, identical machines can be derived from both the natural semantics and the reduction semantics by means of semantic-preserving program transformations, which establishes the computational equivalence of the two semantic descriptions.

Finally, we have shown that the abstract machine for the ς-calculus and the abstract machine for the $\varsigma\rho$-calculus are strongly bisimilar. The bisimilarity result formally connects the two calculi, by establishing their relative computational equivalence. Since the two calculi are equivalent, the $\varsigma\rho$-calculus provides a definition of an object-oriented computational model in terms of explicit substitutions.

12.2 Perspectives

We wrap up the treatment of the ς-calculus by suggesting a few possible directions for further work.

Object references: The ς-calculus captures a number of traditional object-oriented features, such as method updates (a generalisation of field update) and a notion of self. Further, the calculus can be used to model object-oriented phenomena such as classes, traits, and inheritance [1, Section 6.6], and can be extended with a type system with support for a notion of subtyping [1, Chapter 7]. However, the ς-calculus does not capture a notion of object identity, which is a feature found in practically any object-oriented programming language (at least in all the ones known to the present author).

To be able to capture or model such a notion, the calculus could be extended with a store, and with syntactic constructs to manipulate the contents of the store. A calculus thus extended would correspond more closely to actual object-oriented languages, and it would be interesting to see the result of the functional and syntactic correspondences applied to such a calculus.

Connection with other calculi: Since the purpose of the ς-calculus is to provide an object-oriented model of computation, it is necessary to establish that the calculus can indeed be used to describe any computable function. Abadi and Cardelli flesh out this claim by providing a translation of terms of the λ-calculus to terms of the ς-calculus [1, Page 66].

However, they do not address the question of what reduction strategy this encoding induces, nor do they establish any type of connection between the result of evaluating the λ-term and the result of evaluating the ς-encoded λ-term. These topics are the subject of ongoing joint work with Olivier Danvy and Zaynah Dargaye.

Chapter 13

Bibliographic References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. A preliminary version was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 1990).

[3] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[6] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[7] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.

[8] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[9] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[10] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[11] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[12] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[13] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33.

[14] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006.

[15] Olivier Danvy. Personal communication, Aarhus, Denmark, December 2007.

[16] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In Wilfrid Hodges and Ruy de Queiroz, editors, *Proceedings of the 15th Workshop on Logic, Language, Information and Computation (WoLLIC 2008)*, number 5110 in Lecture Notes in Artificial Intelligence, pages 1–16, Edinburgh, Scotland, July 2008. Springer-Verlag.

[17] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 200? In press. A preliminary version is available as the research report BRICS RS-07-7.

[18] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

[19] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *Journal of Functional Programming*, 17(6):793–812, 2007.

[20] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[21] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal

proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[22] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[23] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007.

[24] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, May 2007.

[25] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[26] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, New York, NY, USA, January 2007. ACM Press.

[27] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[28] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [13].

[29] John C. Reynolds. The discoveries of continuations. pages 233–247, 1993.

[30] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

Chapter 14

Pensum

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. Chapter 6.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[3] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.

[4] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[5] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[6] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In Wilfrid Hodges and Ruy de Queiroz, editors, *Proceedings of the 15th Workshop on Logic, Language, Information and Computation (WoLLIC 2008)*, number 5110 in Lecture Notes in Artificial Intelligence, pages 1–16, Edinburgh, Scotland, July 2008. Springer-Verlag.

[7] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 200? In press. A preliminary version is available as the research report BRICS RS-07-7.

[8] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

[9] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[10] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[11] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[12] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [13].

[13] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.