Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Retractile Continuations

by

Roberto da Silva Bigonha

RT 01/96

Caixa Postal, 702
30.161 - Belo Horizonte - MG

January 4, 1996

Abstract

We introduce the notion of **retractile continuation**, which is the basis of a technique for accommodating, in a same denotational semantics definition, both the continuation and direct semantics styles.

Contents

1 Introduction

The kind of semantics used in most denotational definitions is **continuation** semantics[**?**, **?**, **?**, **?**, **?**]. The continuation approach is generally chosen for its convenient way of dealing with error conditions. However, the sequential nature of continuations presents some difficulties. Take a list of mutually recursive equation definitions as an example. In order to manufacture each one of the definitions, it is required that all others have already been defined so that the types of free variables in each one are available in a common environment.

In the **direct** approach to semantics, a situation like this is easily modelled by defining a system of mutually recursive equations, each defining a partial environment that results from the evaluation of the associated definition. Each equation is then defined in the same global environment, which should be recursively defined in terms of those partial environments produced by the individual equations.

On the other hand, in the continuation approach, each semantic function is supposed to pass the intermediate value it produces, for example an environment, to the rest of the definition, i.e., to the normal continuation, which generally maps intermediate results to final answers. Thus, strictly speaking, intermediate values in the continuation approach usually are not available locally to construct the desired system of recursive equations.

Clearly, the continuation approach makes it easier to cope with error conditions and the direct approach facilitates the modelling of non-sequential evaluations. Thus, it would be very convenient to have a mechanism that allows a harmonious coexistence of both semantics styles, so that we may use the *right* kind of semantics where it works best.

2 The Style Switching Mechanism

The above problem is solved here by defining a method of switching from continuation to direct semantics, which allows the establishment of systems of mutually recursive equations, and then, to move back to continuation semantics to formulate the rest of the definition.

The switching mechanism is implemented by passing to the semantic functions involved in the process not the normal continuation but a conveniently manufactured special continuation that just indicates the value to be returned by the function. For example, let f be a semantic function of type:

$$f : X \mapsto C \mapsto Ans$$

where $C = V \mapsto Ans$ is a domain of continuations, Ans is that of final answers, and V is the domain of intermediate values that f passes to its continuation.

The first step is to make f return a value, say $v \in V$, rather than the final answer. The idea is to make arrangements to force f to return the intermediate value v it produces instead of passing it to the normal continuation. And this must be achieved without modifying f, because this function may be also applied in different contexts, which certainly assume that f still deals properly and systematically with its continuation.

The solution we have devised consists of passing to f a special type of continuation, named **retractile continuation**, with the purpose of hoisting the intermediate value produced by f to the passing point. A retractile continuation works like a boomerang which when correctly thrown (passed as parameter) glides back to a point near the thrower (the calling point).

A **retractile continuation** τ has type:

$$\tau : V \mapsto A$$

and must be always defined as an identity function such as:

$$\tau \;=\; \lambda v \,.\, v$$

where the new domain of final answers A is the domain Ans extended to incorporate the domain V. The type of f must be changed accordingly.

Hence, the value of $a \in A$ in $a = f(x)(\tau)$ is the intermediate value which f passes to the continuation τ if it succeeds; otherwise a denotes some other value in the domain of final answers, such as an error message, for example.

Later, when comes the time to switch back to continuation semantics, the intermediate value a produced locally can be explicitly passed to the normal continuation as would have been done under normal conditions.

3 An Example

In order to illustrate the application of the proposed mechanism, the denotational semantics of a toy language Ω is presented in the sequel using a combination of continuation and direct semantics.

A program in the language Ω is simply the constant 0, an identifier *id*, a function application, a λ-abstraction or a sequence of **let**-clauses ended by an **in**-expression. The **let**-clauses serve to bind identifiers to Ω-expressions. Bound identifiers can be freely used in any of Ω-expressions of the **let**-clauses and in the corresponding **in**-expression. In order to capture this semantics, all **let**-clauses shall be evaluated in a environment containing all bindings they introduce, possibly in a mutually recursive fashion. The direct semantic approach is more indicate to model the meaning of this feature.

The initial environment in which Ω is defined has all identifiers bound to a special value **unbound**. Thus, an error should be indicated whenever, in the evaluation of an Ω-expression, a reference to identifier not bound by any of the **let**-clauses is encountered. Errors should also be indicated when non-functional values are applied to any other values. Since continuation semantics works best in this situation, it shall be used here.

The syntactic domains of Ω are:

$exp \in Exp$	$=$	$Def+$ **in** Exp	Ω-expressions
	$\mid$	Id (Exp)	function application
	$\mid$	$\lambda Id.\ Exp$	λ-abstraction
	$\mid$	Id	bound identifier
	$\mid$	0	constant zero
$def \in Def$	$=$	**let** $Id = Exp$	**let**-clauses
$id \in Id$	$=$	*left unspecified*	Ω-identifiers

To formulate the continuation semantics of Ω, the following semantic domains are defined:

Ans	$=$	$N +$ **error**	Traditional final answers
$a_1, a_2 \in A$	$=$	$Ans + Env$	Extended final answers
$\rho, \rho' \in Env$	$=$	$Id \mapsto D_v$	Environments
$v \in D_v$	$=$	$N + F_0 + F_1 +$ **unbound**	Denotable values
$\delta, \tau \in D_c$	$=$	$Env \mapsto A$	Decl continuations
$\kappa \in E_c$	$=$	$D_v \mapsto A$	Expr continuations
F_0	$=$	$E_c \mapsto A$	Function without args
$f_1 \in F_1$	$=$	$D_v \mapsto E_c \mapsto A$	One arg function
N	$=$	$\{ 0 \}$	A natural number

The domain Env of environments is part of the domain A of extended final answers in order to implement the proposed switching mechanism.

The initial environment $\rho_0 \in Id \mapsto D_v$ is defined as:

$$\rho_0 = \lambda id.\, \textbf{unbound}$$

The **retractile continuation** $\tau \in D_c$ needed by the switching mechanism is defined as:

$$\tau = \lambda\rho\,.\,\rho$$

The semantic function $\mathcal{E}$, which defines the denotation of Ω-expression, has type:

$$\mathcal{E} \;:\; Exp \mapsto Env \mapsto E_c \mapsto A$$

Hence, the continuation semantics of an Ω-constant `0` is given by:

$$\mathcal{E}[\![\mathtt{0}]\!](\rho)(\kappa) \;:\mathrm{A} = \; \kappa(0)$$

The continuation semantics of an Ω-idenfifiers *id* is given by:

$$\begin{array}{l}\mathcal{E}[\![id]\!](\rho)(\kappa) \;:\mathrm{A} = \\ \qquad \rho(id) = \textbf{unbound} \;\; \rightarrow \;\textbf{error},\; \kappa(\rho(id))\end{array}$$

The continuation semantics of $\lambda id\,.\,exp$ is given by:

$$\begin{array}{l}\mathcal{E}[\![\lambda id\,.\,exp]\!](\rho)(\kappa) \;:\mathrm{A} = \\ \qquad \textbf{let}\; f_1 = \lambda v\;\kappa'\,.\,\mathcal{E}[\![exp]\!]\;\rho\{id \leftarrow v\}\;\kappa' \\ \qquad \textbf{in}\; \kappa(f_1)\end{array}$$

The notation $\rho = \rho_1\{z \leftarrow \; y\}$ defines an environment ρ such that:

$$\rho(x) = \begin{cases} y & \text{if } x \;=\; z \\ \rho_1(x) & \text{otherwise} \end{cases}$$

The continuation semantics of functional applications *id* (*exp*) is given by:

$$\begin{array}{l}\mathcal{E}[\![id\;(exp)]\!](\rho)(\kappa) \;:\mathrm{A} = \\ \qquad \rho(id) \in F_1 \rightarrow \textbf{let}\; \kappa_1 = \lambda v\,.\,\rho(id)\;v\;\kappa \\ \qquad\qquad\qquad\qquad \textbf{in}\; \mathcal{E}[\![exp]\!]\rho\;\kappa_1,\; \textbf{error}\end{array}$$

The continuation semantics of **let**-expressions of the form *def*+ **in** *exp* is:

$$\begin{array}{l}\mathcal{E}[\![def+\;\textbf{in}\;exp]\!](\rho)(\kappa) \;:\mathrm{A} = \\ \qquad \textbf{let}\; \delta = \lambda\rho'\,.\,\mathcal{E}[\![exp]\!]\;\rho'\;\kappa \\ \qquad \textbf{in}\; \mathcal{L}[\![def+]\!]\;\rho\;\delta\end{array}$$

In our notation, *def*$*$ denotes a list of zero or more *def*, and *def*+ is a list with at least one element. The symbol <> denotes an empty list.

The function $\mathcal{L}$, which defines the semantics of a list of **let**-clauses, has the type:

$$\mathcal{L} : \mathit{Def}* \mapsto \mathit{Env} \mapsto D_c \mapsto A$$

The semantics of the empty list of **let**-clauses is:

$$\mathcal{L}(<>)(\rho)(\delta) \ : \mathrm{A} = \ \delta(\rho)$$

The meaning of non-empty lists of **let**-clauses is given the equation below which is defines $\mathcal{L}$ in terms of a system of recursive equations in a pure **direct semantics** style:

$$\begin{array}{l}\mathcal{L}(\mathit{def} \ \mathbf{::} \ \mathit{def}*)(\rho)(\delta) \ : \mathrm{A} = \\ \quad \mathbf{let} \ \tau = \lambda\rho \,.\, \rho \\ \quad \mathbf{let} \ a_1 = \mathcal{F}[\![\mathit{def}]\!] \ \rho' \ \tau \\ \quad \mathbf{let} \ a_2 = \mathcal{L}[\![\mathit{def}*]\!] \ \rho' \ \tau \\ \quad \mathbf{let} \ \rho' = a_1, a_2 \in \mathit{Env} \to \rho\{a_1\}\{a_2\}, \ \rho \\ \quad \mathbf{in} \ a_1 \in \mathit{Env} \to (a_2 \in \mathit{Env} \to \delta(\rho'), \ a_2), \ a_1\end{array}$$

The occurrence of a term like def **::** $\mathit{def}*$ in the binding context above indicates that def is to get bound to the first element of the given list, and $\mathit{def}*$ is to be bound to the remaining elements. Note that the functions $\mathcal{F}$ and $\mathcal{L}$ are called to evaluate local environments, and that at the end of the body of $\mathcal{L}$ the computed environment ρ', that is, the new environment containing all the bindings in def and $\mathit{def}*$, is passed to the continuation δ in order to comply with the rest of the definition.

The notation $\rho' = \rho\{\rho_1\}\{\rho_2\}$, for $\rho_1, \rho_2 \in \mathit{Env}$, defines a new environment ρ' such that:

$$\rho'(x) = \begin{cases} \rho_2(x) & \text{if } \rho_2(x) \neq \mathbf{unbound} \\ \rho_1(x) & \text{if } \rho_2(x) = \mathbf{unbound} \ \wedge \ \rho_1(x) \neq \mathbf{unbound} \\ \rho(x) & \text{otherwise} \end{cases}$$

To conclude the example, the function $\mathcal{F}$, which defines the semantics of a **let**-clause, has the type:

$$\mathcal{F} : \mathit{Def} \mapsto \mathit{Env} \mapsto D_c \mapsto A$$

And the equation for $\mathcal{F}$ is:

$$\begin{array}{l}\mathcal{F}[\![\mathbf{let} \ \mathit{id} = \mathit{exp}]\!](\rho)(\delta) \ : \mathrm{A}= \\ \quad \mathbf{let} \ \rho_1 = \rho\{\mathit{id} \leftarrow (\lambda\kappa \,.\, \mathcal{E}[\![\mathit{exp}]\!] \ \rho_1 \ \kappa\} \\ \quad \mathbf{in} \ \delta(\rho_1)\end{array}$$

4 Conclusion

We have proposed a method for accommodating in a same denotational semantics definition both the continuation and direct semantics styles and illustrated the application of the method through an example.

We claim that the advantages of this method is that it permits the *right* kind of semantics to be used where it works best. The continuation approach makes it easier to cope with error conditions and the direct approach facilitates the modelling of non-sequential evaluations.

We successfully used **retractile continuation** for the first time in 1981 in the formulation of the formal definition of a functional language of realistic size and complexity[**?**].

References

[1] Roberto S. Bigonha. *A Denotational Semantics Implementation System.* PhD thesis, University of California, Los Angeles, 1981. 428 pages.

[2] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, Massachusetts, 1992.

[3] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics, Parts a and b.* Chapman and Hall, London, 1976.

[4] Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[5] C. Wadsworth and C. Strachey. Continuations - a mathematical semantics for handling full jumps. Technical monograph prg-11, Oxford University Computing Lab, 1974. Programming Research Group.

[6] G. Winskel. *Semantics of Programming Languages.* MIT Press, 1993.