Continuation Semantics in Typed Lambda-Calculi

(summary)

Albert R. Meyer
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Mitchell Wand
Computer Science Department
Brandeis University
Waltham, MA 02254

From *Logics of Programs (Brooklyn, June, 1985)* (R. Parikh, ed.) Springer Lecture Notes in Computer Science, Vol. 193 (1985), 219–224.

1. Abstract. This paper reports preliminary work on the semantics of the continuation transform. Previous work on the semantics of continuations has concentrated on untyped lambda-calculi and has used primarily the mechanism of inclusive predicates. Such predicates are easy to understand on atomic values, but they become obscure on functional values. In the case of the typed lambda-calculus, we show that such predicates can be replaced by retractions. The main theorem states that the meaning of a closed term is a retraction of the meaning of the corresponding continuationized term.

2. Introduction. The method of continuations was introduced in [Strachey & Wadsworth 74] as a device for formalizing the notion of control flow in programming languages. In this method, a term is evaluated in a context which represents "the rest of the computation". If the term involves the evaluation of a subterm, then the subterm is evaluated in a new context which evaluates the rest of the term and then proceeds to the old context. If the term can be evaluated immediately, then the value is passed to the context. Such a context is called a *continuation.*

In many cases, one may write either a direct or a continuation semantics for a language. Then one is faced with the problem of formulating the relationship between the two semantics. This relationship has been studied by [Reynolds 74], [Stoy 81], and [Sethi & Tang 80]. All of these papers discussed essentially an untyped lambda calculus with atoms, interpreted in Scott's D_∞ model or something like it. The result in each case was that the two semantics for a given term were connected by a relation which was the identity relation on atoms. The key was the construction of a suitable relation. This was done by the method of *inclusive predicates,* which depended on the details of the models. As a result, the significance of these predicates on values other than atoms was obscure.

In this paper, we show that for the case of the *typed* lambda-calculus, one may replace these inclusive predicates by retractions, which are far easier to understand. Rather than have the direct and continuation meanings of a term connected by a relation, we show that the direct meaning may be recovered from the continuation meaning by a retraction. In this way, the continuation semantics appears as a representation of the direct semantics in the sense of [Hoare 72], with the retraction as Hoare's "abstraction mapping". Hoare's notion of a "concrete invariant" appears as a crucial part of the proof.

Furthermore, the reasoning in the proof lies entirely in the λ-calculus, and hence the theorem holds in *any* model. Thus we avoid the detailed model-theoretic manipulation characteristic of the inclusive predicate approach.

3. Language. We consider the simply-typed lambda-calculus. The types are either ground types $\sigma_1, \sigma_2, \ldots$ or functional types $\alpha \to \beta$. Among the types is a distinguished type o (not necessarily a ground type) of *answers*. These are the only types. Terms are either variables, combinations, or abstractions $\lambda v{:}\, \alpha.\, M$, where v is a variable, α is a type, and M is a term. We assume that the semantics is given using a many-sorted environment model [Meyer 82, Wand 84].

4. Interpretation of Types. The continuation semantics will manipulate representations of the objects that appear in the direct semantics. We assign to each type α a type α' of representations of objects of type α. Ground types are represented as themselves. Corresponding to a function of type $\alpha \to \beta$, we will have in the continuation semantics a function that takes two arguments: a representation of an α and a continuation that expects a representation of a β. With this information, the function computes an answer. Thus we have:

$$\sigma' = \sigma$$

$$(\alpha \to \beta)' = \alpha' \to (\beta' \to o) \to o$$

5. The Transformation. For each term M of type α, we construct a term $\overline{M}$ of type $(\alpha' \to o) \to o$ as follows:

$$\overline{x} = \lambda\kappa.\, \kappa x$$

$$\overline{\lambda x.\, M} = \lambda\kappa.\, \kappa(\lambda x.\, \overline{M})$$

$$\overline{MN} = \lambda\kappa.\, \overline{M}(\lambda m.\, \overline{N}(\lambda n.\, mn\kappa))$$

Here we have deleted the type annotations for clarity. If we have a variable, we send the result to the continuation κ. If we have an abstraction, we return an appropriate function to the continuation. If we have a combination, we evaluate the operator in a continuation which in turn evaluates the operand in a continuation which applies the value of the operator to the value of the operand and the current continuation. This fits nicely with the definition of $(-)'$: if M is of type $\alpha \to \beta$, then m must be of type $(\alpha \to \beta)' = \alpha' \to (\beta' \to o) \to o$, n of type α', and κ of type $\beta' \to o$.

Operationally, the pleasant fact about $\overline{M}$ is that it is *tail-recursive*: no operand is a combination. (This property, furthermore, is preserved under beta-reduction). Thus there is at most one redex which is not inside the scope of a lambda, and thus call-by-value reduction coincides with outermost or call-by-name reduction [Plotkin 75]. This also allows simpler implementation on standard machines [Abelson & Sussman 85].

6. Retractions. We say α is a *retract* of β (we write $\alpha \triangleleft \beta$) iff there exist lambda-definable maps $i: \alpha \to \beta$ and $j: \beta \to \alpha$ such that $j \circ i$ is the identity function on α. We can formulate retractions between the various domains used in the semantics, as follows:

7. Theorem. (Statman) If $i \triangleleft o$ for every ground type ι, then $\alpha \triangleleft (\alpha \to o) \to o$.

Proof: Define $I: \alpha \to (\alpha \to o) \to o$ by $I = \lambda x \kappa.\, \kappa x$. To define the inverse mapping, observe that α must be of the form $\alpha_1 \to \alpha_2 \to \ldots \to \gamma$, where γ is a ground type. Let $r: \gamma \to o$ be the injection of the retraction from γ to o, with left inverse l. Then we define $J = \lambda u: (\alpha \to o) \to o.\, \lambda x_1: \alpha_1 \ldots \lambda x_n: \alpha_n.\, l(u(\lambda a: \alpha.\, r(a x_1 \ldots x_n)))$.

We will use I_α and J_α to denote these retractions. Note that I_α does *not* denote the usual combinator $\lambda x: \alpha.\, x$.

We can "apply" an element m of type $(\alpha \to \beta)'$ to an element n of type α' and get an element of type β' by writing $J(mn)$. We use $m \bullet n$ to denote this combination, which we call *pseudo-application*.

8. Theorem. For any type α, $\alpha \triangleleft \alpha'$.

Proof: For basic types, the retraction is the identity. At functional types, define

$$i_{\alpha \to \beta} = \lambda f: (\alpha \to \beta).\, I_{\beta'} \circ i_\beta \circ f \circ j_\alpha$$

$$j_{\alpha \to \beta} = \lambda f': (\alpha \to \beta)'.\, j_\beta \circ J_{\beta'} \circ f' \circ i_\alpha$$

We may now state the main theorem:

9. Main Theorem. Let M be any closed term of type α. Then $M =$

$j_\alpha(J_{\alpha'}\overline{M})$.

10. Concrete Invariants. To prove the theorem, we need to consider terms with free variables as well. To do this, we must consider which elements of α' are legal representations of elements of α, that is, what constitutes an appropriate concrete invariant for this representation scheme. For each type α, we need a predicate P_α on the elements of type α' in any environment model.

Define a set of such predicates P_α to be *acceptable* iff it has the following properties:

(1) For all types α, β, and γ, and all values x of type α, the following hold:

$$P_\alpha(i_\alpha x)$$

$$P_{\alpha\to(\beta\to\alpha)}(J\overline{K})$$

$$P_{(\alpha\to\beta\to\gamma)\to(\beta\to\gamma)\to(\alpha\to\gamma)}(J\overline{S})$$

(2) If $P_{\alpha\to\beta}(m)$ and $P_\alpha(n)$, then

$$P(m \bullet n)$$

$$(j_{\alpha\to\beta}m)(j_\alpha n) = j_\beta(m \bullet n)$$

$$mn = I_{\beta'}(m \bullet n)$$

Property (1) says that the canonical representations and the images of the standard combinators S and K are legal. Property (2) says that (a) the legal representations are closed under pseudo-application, (b) conventional application is a homomorphic image of pseudo-application, and (c) that real application of representations is well-behaved: that is, it sends a value to its continuation.

With this definition we can state the key result:

11. Theorem. Let P be any acceptable predicate. For any term M and any environment ρ' such that for all x, $P(\rho'[\![x]\!])$ holds, we have $[\![M]\!](j \circ \rho') = j(J([\![\overline{M}]\!]\rho'))$.

Proof (Sketch): We first show that $\overline{\lambda x.\, M} = \overline{[x]M}$, where $[x]M$ denotes the usual bracket-abstraction (lambda-elimination) algorithm. This allows us to concentrate on combination. The theorem then follows by algebraic manipulation, using the additional induction hypotheses that $[\![\overline{M}]\!]\rho' = I(J([\![\overline{M}]\!]\rho'))$ and $P(J([\![\overline{M}]\!]\rho'))$.

12. Existence of acceptable predicates. To finish the main theorem, we need to show that in any environment model, there exists an acceptable predicate. We prove the existence of two acceptable predicates. The first uses induction on types; the second depends on the strong normalization theorem.

13. Proposition. Define Q as follows: Let $Q_\sigma(m) = \mathit{true}$ for ground types. At higher types, define $Q_{\alpha\to\beta}(m)$ to be true iff for all n such that $Q_\alpha(n)$ holds, we have

$$Q_\beta(m \bullet n)$$

$$(jm)(jn) = j(m \bullet n)$$

$$\text{and } mn = I(m \bullet n)$$

Then Q is an acceptable predicate.

Proof: The definition proceeded by induction on types. Property (1) in the definition of acceptability follows by induction on types as well. Property (2) follows *a fortiori.*

14. Theorem. Let R be the smallest predicate containing ix for all x, containing $J\overline{S}$ and $J\overline{K}$ of every type, and closed under $\bullet$. Then R is acceptable.

Proof (Sketch): Any element of R can be expressed as a term built up from combinations of ix, $J\overline{S}$, and $J\overline{K}$ under $\bullet$. We turn these terms into a rewriting system using the usual rules for S and K. We first show that R is acceptable when the range of the quantifiers in the definition of acceptability is restricted to the denotations of terms in normal form in the system. It follows that the usual rules for S and K are sound when applied applicatively, that is, when the subterms are in normal form. Therefore, by the Strong Normalization Theorem, any term, and hence any element of R, codesignates with a term in normal form and therefore the restriction of the quantifiers has no effect.

15. Conclusions. We have shown that at least in the case of the typed lambda calculus, we can explain the continuation transform using retractions and depending only on the principle of beta-conversion. Our development goes through as well if we redefine the continuation transformation to set

$$\overline{MN} = \lambda\kappa.\,\overline{N}(\lambda n.\,\overline{M}(\lambda m.\,mn\kappa))$$

where the operand is evaluated before the operator, or even to allow the algorithm to choose nondeterministically between the two. In that case we need to ensure that both versions of $\overline{S}$ satisfy the concrete invariant. The case of the call-by-name transformation, as given in [Plotkin 75], can also be treated by these methods.

The situation becomes more complicated if we study calculi in which strong normalization fails, such as the typed calculus with fixed-point operators. We are currently studying both this and the untyped case, but it is not clear as of this writing whether this approach extends to these cases. The retractions themselves are of some interest, and add another dimension to our understanding.

16. References

[Abelson & Sussman 84]
Abelson, H., and Sussman, G.J. *The Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[Hoare 72]
Hoare, C.A.R., "Proving Correctness of Data Representations," *Acta Informatica 1* (1972), 271–281.

[Meyer 82]

Meyer, A.R. "What Is a Model of the Lambda Calculus?" *Information and Control 52* (1982), 87–122.

[Plotkin 75]
Plotkin, G.D. "Call-by-Name, Call-by-Value and the λ-Calculus," *Theoret. Comp. Sci. 1* (1975) 125–159.

[Reynolds 74]
Reynolds, J.C. "On the Relation between Direct and Continuation Semantics," *Proc. 2nd Colloq. on Automata, Languages, and Programming* (Saarbrucken, 1974) Springer Lecture Notes in Computer Science, Vol. 14 (Berlin: Springer, 1974) 141–156.

[Sethi & Tang 80]
Sethi, R. and Tang, A. "Constructing Call-by-value Continuation Semantics," *J. ACM 27* (1980), 580–597.

[Stoy 81]
Stoy, J.E. "The Congruence of Two Programming Language Definitions," *Theoret. Comp. Sci. 13* (1981), 151–174.

[Strachey & Wadsworth 74]
Strachey, C. and Wadsworth, C.P. "Continuations: A Mathematical Semantics for Handling Full Jumps," Oxford University Computing Laboratory Technical Monograph PRG-11 (January, 1974).

[Wand 84]
Wand, M. "A Types-as-Sets Semantics for Milner-style Polymorphism," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 158–164.