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 continuationised 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 $D_\infty$ 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 σ₁,σ₂,... or functional types α → β. 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 λv:α. 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 α → β, 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:

\[ σ' = σ \]
\[ (α → β)' = α' → (β' → o) → o \]

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

\[ \overline{x} = \lambda κ. κx \]
\[ \overline{λx.M} = \lambda κ. κ(λx. \overline{M}) \]
\[ \overline{MN} = \lambda κ. \overline{M}(\lambda m. \overline{N}(\lambda n. mnκ)) \]

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 α → β, then m must be of type \( (α → β)' = α' → (β' → o) → o \), n of type α', and κ of type β' → o.