Finally Tagless, Partially Evaluated*
Tagless Staged Interpreters for Simpler Typed Languages

Jacques Carette¹, Oleg Kiselyov², and Chung-chieh Shan³

¹ McMaster University
carette@mcmaster.edu
² FNMOC
oleg@pobox.com
³ Rutgers University
ccshan@rutgers.edu

Abstract. We have built the first family of tagless interpretations for a higher-order typed object language in a typed metalanguage (Haskell or ML) that require no dependent types, generalized algebraic data types, or postprocessing to eliminate tags. The statically type-preserving interpretations include an evaluator, a compiler (or staged evaluator), a partial evaluator, and call-by-name and call-by-value CPS transformers.

Our main idea is to encode HOAS using cogen functions rather than data constructors. In other words, we represent object terms not in an initial algebra but using the coalgebraic structure of the \(\lambda\)-calculus. Our representation also simulates inductive maps from types to types, which are required for typed partial evaluation and CPS transformations.

Our encoding of an object term abstracts over the various ways to interpret it, yet statically assures that the interpreters never get stuck. To achieve self-interpretation and show Jones-optimality, we relate this exemplar of higher-rank and higher-kind polymorphism to plugging a term into a context of let-polymorphic bindings.

It should also be possible to define languages with a highly refined syntactic type structure. Ideally, such a treatment should be metacircular, in the sense that the type structure used in the defined language should be adequate for the defining language.

1 Introduction

A popular way to define and implement a language is to embed it in another. Embedding means to represent terms and values of the object language as terms and values in the metalanguage. Embedding is especially appropriate for domain-specific object languages because it supports rapid prototyping and integration with the host environment. If the metalanguage supports staging, then the

* We thank Martin Sulzmann and Walid Taha for helpful discussions. Sam Staton, Pieter Hofstra, and Bart Jacobs kindly provided some useful references. We thank anonymous reviewers for pointers to related work.
embedding can compile object programs to the metalanguage and avoid the overhead of interpreting them on the fly \cite{23}. A staged definitional interpreter is thus a promising way to build a domain-specific language (DSL).

We focus on embedding a typed object language into a typed metalanguage. The benefit of types in this setting is to rule out meaningless object terms, thus enabling faster interpretation and assuring that our interpreters do not get stuck. To be concrete, we use the typed object language in Figure 1 throughout this paper. We aim not just for evaluation of object programs but also for compilation, partial evaluation, and other processing.

Pašalić et al. \cite{23} and Xi et al. \cite{37} motivated interpreting a typed object language in a typed metalanguage as an interesting problem. The common solutions to this problem store object terms and values in the metalanguage in a universal type, a generalized algebraic data type (GADT), or a dependent type. In the remainder of this section, we discuss these solutions, identify their drawbacks, then summarize our proposal and contributions. We leave aside the solved problem of writing a parser/type-checker, for embedding object language objects into the metalanguage (whether using dependent types \cite{23} or not \cite{2}), and just enter them by hand.

1.1 The Tag Problem

It is straightforward to create an algebraic data type, say in OCaml, Fig. 2(a), to represent object terms such as those in Figure 1. For brevity, we elide treating integers, conditionals, and fixpoint in this section. We represent each variable using a unary de Bruijn index. For example, we represent the object term \((\lambda x. x)\) \texttt{true} as \texttt{let test1 = A (L (V VZ), B true)}.

Following \cite{23}, we try to implement an interpreter function \texttt{eval0}, Fig. 2(b). It takes an object term such as \texttt{test1} above and gives us its value. The first argument to \texttt{eval0} is the environment, initially empty, which is the list of values bound to free variables in the interpreted code. If our OCaml-like metalanguage were untyped, the code above would be acceptable. The \texttt{Le} line exhibits interpretive overhead: \texttt{eval0} traverses the function body \texttt{e} every time (the result of evaluating) \texttt{Le} is applied. Staging can be used to remove this interpretive overhead \cite{23, §1.1–2}.

\footnote{We use de Bruijn indices to simplify the comparison with Pašalić et al.’s work \cite{23}.}