Tag Elimination and Jones-Optimality
(Preliminary Report)

Walid Taha[1]*, Henning Makholm[2]** and John Hughes[3]

[1] Department of Computer Science, Yale University, New Haven, CT, USA
taha@cs.yale.edu

[2] DIKU, University of Copenhagen, Copenhagen, Denmark
henning@makholm.net

[3] Department of Computing Sciences, Chalmers, Göteborg, Sweden
rjmh@cs.chalmers.se

Abstract. Tag elimination is a program transformation for removing unnecessary tagging and untagging operations from automatically generated programs. Tag elimination was recently proposed as having immediate applications in implementations of domain specific languages (where it can give a two-fold speedup), and may provide a solution to the long standing problem of Jones-optimal specialization in the typed setting. This paper explains in more detail the role of tag elimination in the implementation of domain-specific languages, presents a number of significant simplifications and a high-level, higher-order, typed self-applicable interpreter. We show how tag elimination achieves Jones-optimality.

1 Introduction

In recent years, substantial effort has been invested in the development of both theory and tools for the rapid implementation of domain specific languages (DSLs). DSLs are formalisms that provide their users with notation appropriate for a specific family of tasks at hand. A popular and viable strategy for implementing domain specific languages is to simply write an interpreter for the DSL in some meta-language, and then to stage this interpreter either manually by adding explicit staging annotations (multi-stage programming [16, 10, 14]) or by applying an automatic binding-time analysis (off-line partial evaluation [6]). The result of either of these steps is a *staged interpreter*. A staged interpreter is essentially a *translation* from a subject-language (the DSL) to a target-language[1]. If there is already a (native code) compiler for the target-language, the approach yields *a simple (native code) compiler* for the DSL at hand.

* Funded by a Postdoctoral Fellowship from the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403, and by subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

** Funded by the University of Copenhagen.

[1] Staging also turns the meta-language to be (additionally) the target-language.

This paper is concerned with a costly problem which can arise when *both* the subject- and the meta-language are statically typed. In particular, when the meta-language is typed, there is generally a need to introduce a "universal datatype" to represent values. At runtime, having such a universal datatype means that we have to perform tagging and untagging operations. When the subject-language is untyped, we really do need these checks (e.g. in an ML interpreter for Scheme). But when the subject-language is also statically typed (e.g. an ML interpreter for ML), we do not really need the extra tags: they are just there because we need them to *statically type check the interpreter*. When such an interpreter is staged, it inherits [9] this weakness and generates programs that contain *superfluous tagging and untagging operations.* To give an idea of the cost of these extra tags, here is the cost of running two sample programs (the factorial function applied to 12 and the Fibonacci function applied to 10) with and without the tags in them[2]:

Term (fully inlined)	fact 12	fib 10
Speedup (after tag elimination)	2.6x	1.9x

The table shows that removing the superfluous tags from these two programs speeds up their execution by a factor of 2.6 and 1.9 times, respectively.

How, then, can we ensure that programs produced by the staged interpreter do not contain superfluous uses of the universal datatype?

One possibility is to look for more expressive type systems that alleviate the need for a universal datatype (such as dependent type systems). But it is not clear that self-interpretation can be achieved in such languages [11]. A more pressing practical concern is that such systems lose decidable type *inference*, which is a highly-valued feature of many typed functional programming languages.

Tag elimination [15, 7] is a recently proposed transformation that was designed to remove the superfluous tags in a post-processing phase. Thus our approach is to stage the interpreter into *three* distinct stages (rather than the traditional two). The new extra stage, called *tag elimination*, is distinctly different from the traditional partial evaluation (or specialization) stage. In essence, tag elimination allows us to type check the subject program after it has been interpreted. If it checks, superfluous tags are simply erased from the interpretation. If not, a "semantically equivalent" interface is added around the interpretation.

1.1 Jones-Optimality

The problem of the superfluous tags is tightly coupled with the problem of *Jones-optimal self-interpretation in a statically-typed language*. The significance of Jones-optimality lies both in its relevance to the effective application of the above strategy when a statically-typed meta-language is used, and the fact that the problem has remained open for over thirteen years, eluding numerous significant efforts [3, 4, 1].

[2] Data based on 100,000 runs of each in SML/NJ.

Intuitively, Jones-optimality tries to address the problem of whether for a given meta-language there exists a partial evaluator strong enough to remove an entire level of "interpretive overhead" [6, Section 6.4]. A key difficulty is in formalizing the notion of interpretive overhead. To this end, Jones chose to formulate this in the special case where the program being specialized is an interpreter. This restriction makes the question more specific, but there is still the question of what removing a layer of interpretive overhead means, even when we are specializing an interpreter. One choice is to say that the cost of specializing the interpreter to a particular program produces a term that is no more expensive than the original program. This however, introduces the need for a notion of cost, which is non-trivial to formalize. Another approach which we take here is to say that the generated program must be syntactically the same as the original one. While this requires prohibiting additional reductions, we accept that, as it still captures the essence of what we are trying to formalize.

The relevance of Jones-optimality lies in that, if we *cannot* achieve it, then staging/partial evaluation will no-doubt produce sub-optimal programs for a large variety of languages. Thus, resolving this problem for statically-typed programming languages means that we have established that statically-typed languages can be used to efficiently implement compilers for a large class of domain specific languages (including, for example, all languages that can be easily mapped into any subset of the language that we consider).

1.2 Contribution and summary of the rest of the paper

This paper shows how tag elimination achieves Jones-optimality, and reports (very briefly) on an implementation that supports our theoretical results. In doing so, this paper extends previous theoretical work [15] by presenting 1) a typed, high-level language together with a self-interpreter for it (needed for Jones-optimality), and 2) a substantially simplified version of tag-elimination. Previous implementation work [7] was in a first-order language.

Section 2 presents a simply-typed programming language that will be used as the main vehicle for presenting the self-interpreter and the proposed transformation. The language has first-order data and higher-order values. We define the new annotations and their interpretations. A specification of a tag elimination analysis[3] is presented in Section 3 as a set of inference rules defined by induction on the structure of raw terms. In Section 4 we summarize the basic semantic properties of tag elimination. In this section, we define the wrapper and unwrapper functions that are needed to define a "fall-back" path for the tag-elimination transformation.

Section 5 reviews interpreters. Section 6 addresses the relation between an interpreter and a staged interpreter, emphasizing the utility of the notion of a

[3] In this paper, we present and focus on a specification of the analysis, not an algorithm for carrying it out. We expect that the analysis can be implemented by a total function that yields a result that can be validated by our specification, but at this time, we have not yet established this formally. In this paper, we will often refer to this specification as "the analysis".

translation in this setting. In this section, we show how tag elimination analysis is sufficient to allow us to eliminate superfluous tags from typed staged self-interpreters. In Section 7 we demonstrate the relevance of this result to the problem of Jones-optimal specialization.

2 A Typed Language for Self-Interpretation

First we present a programming language with first-order datatypes and with higher-order values. The **types** $\mathbb{T}$ in this language are simply:

$$t ::= \mathsf{D} \mid \mathsf{V} \mid t \to t$$

The type D is for first-order data (like LISP S-expressions), the type V is for higher-order values (a universal datatype), and the last production is for function types. We can think of V as being generated by the following ML declaration:

$$\mathsf{datatype\ V = F\ of\ V \to V \mid E\ of\ D}$$

But we do not need case analysis on V: We will only need value contructors (tagging) and simple descructors (untagging) for our purposes here (writing interpreters). We assume an infinite set of **names** $\mathbb{X}$ ranged over by x, and that this set includes the special variables "$\mathsf{nil}, \mathsf{true}, \mathsf{false}$". The set of **expressions** $\mathbb{E}$ is defined as follows[4]:

$$\begin{array}{l} s ::= x \mid (s.s) \qquad u ::= \mathsf{car} \mid \mathsf{cdr} \mid \mathsf{atom?} \qquad o ::= \mathsf{cons} \mid \mathsf{equal?} \\ e ::= x \mid e\ e \mid \lambda x.e \mid \mathsf{fix}\ x.e \mid `s \mid u\ e \mid o\ e\ e \mid \mathsf{if}\ e\ e\ e \mid \mathsf{E}\ e \mid \mathsf{E}^{-1}\ e \mid \mathsf{F}\ e \mid \mathsf{F}^{-1}\ e \end{array}$$

The type D will be inhabited by S-expressions represented by dotted-pairs s. One can use a distinct set for names of atoms, but it causes no confusion to simply use variables here. Note that substitution "does not do anything" with $`s$, or rather, it is the identity. We use a standard **type system** for this language.

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2} \quad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \quad \frac{\Gamma, x : t \vdash e : t}{\Gamma \vdash \mathsf{fix}\ x.e : t}$$

$$\frac{}{\Gamma \vdash `s : \mathsf{D}} \quad \frac{\Gamma \vdash e : \mathsf{D}}{\Gamma \vdash u\ e : \mathsf{D}} \quad \frac{\Gamma \vdash e_1 : \mathsf{D} \quad \Gamma \vdash e_2 : \mathsf{D}}{\Gamma \vdash o\ e_1\ e_2 : \mathsf{D}} \quad \frac{\Gamma \vdash e_1 : \mathsf{D} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : t}$$

$$\frac{\Gamma \vdash e : \mathsf{D}}{\Gamma \vdash \mathsf{E}\ e : \mathsf{V}} \quad \frac{\Gamma \vdash e : \mathsf{V}}{\Gamma \vdash \mathsf{E}^{-1}\ e : \mathsf{D}} \quad \frac{\Gamma \vdash e : \mathsf{V} \to \mathsf{V}}{\Gamma \vdash \mathsf{F}\ e : \mathsf{V}} \quad \frac{\Gamma \vdash e : \mathsf{V}}{\Gamma \vdash \mathsf{F}^{-1}\ e : \mathsf{V} \to \mathsf{V}}$$

[4] The fixed-point construct used here seems to worry some readers. In particular, they expect such a construct to either take an additional parameter, or have a type restricted to function types, or both. We choose this form primarily because it keeps the various type-preserving translations simple. It may help the reader to note that any term written in such a language as ours can be easily translated into a language that uses a fixed-point operator restricted to function types. Note, however, that in a language with a fixed-point operator such as ours, variables are *not* values.

The type system enjoys weakening and substitution properties.

Lemma 1 (Weakening, substitution).

1. $\Gamma \vdash e : t_1 \land x \notin FV(e) \cup dom(\Gamma) \Longrightarrow x : t_2; \Gamma \vdash e : t_1$
2. $\Gamma \vdash e_1 : t_1 \land x : t_1; \Gamma \vdash e_2 : t_2 \Longrightarrow \Gamma \vdash e_2[x := e_1] : t_2$.

Proof. All by easy inductions. □

A standard **big-step operational semantics** $\hookrightarrow : \mathbb{E} \to \mathbb{E}$ for this language is used:

$$\frac{\begin{array}{l} e_1 \hookrightarrow \lambda x.e_3 \\ e_2 \hookrightarrow v_1 \\ e_3[x := v_1] \hookrightarrow v_2 \end{array}}{e_1\ e_2 \hookrightarrow v_2} \quad \frac{}{\lambda x.e \hookrightarrow \lambda x.e} \quad \frac{e[x := \mathsf{fix}\ x.e] \hookrightarrow v}{\mathsf{fix}\ x.e \hookrightarrow v} \quad \frac{}{`s \hookrightarrow `s} \quad \frac{e_1 \hookrightarrow `(s_1.s_2)}{\mathsf{car}\ e_1 \hookrightarrow `s_1}$$

$$\frac{e_1 \hookrightarrow `(s_1.s_2)}{\mathsf{cdr}\ e_1 \hookrightarrow `s_2} \quad \frac{e \hookrightarrow `x}{\mathsf{atom?}\ e \hookrightarrow `\mathsf{true}} \quad \frac{e \hookrightarrow `(s_1.s_2)}{\mathsf{atom?}\ e \hookrightarrow `\mathsf{false}} \quad \frac{\begin{array}{c} e_1 \hookrightarrow `s_1 \\ e_2 \hookrightarrow `s_2 \end{array}}{\mathsf{cons}\ e_1\ e_2 \hookrightarrow `(s_1.s_2)}$$

$$\frac{\begin{array}{c} e_1 \hookrightarrow `s \\ e_2 \hookrightarrow `s \end{array}}{\mathsf{equal?}\ e_1\ e_2 \hookrightarrow `\mathsf{true}} \quad \frac{\begin{array}{c} e_1 \hookrightarrow `s_1 \\ e_2 \hookrightarrow `s_2 \\ s_1 \not\equiv s_2 \end{array}}{\mathsf{equal?}\ e_1\ e_2 \hookrightarrow `\mathsf{false}} \quad \frac{\begin{array}{c} e_1 \hookrightarrow v_1 \\ e_2 \hookrightarrow v_2 \\ v_1 \not\equiv `\mathsf{false} \end{array}}{\mathsf{if}\ e_1\ e_2\ e_3 \hookrightarrow v_2} \quad \frac{\begin{array}{c} e_1 \hookrightarrow `\mathsf{false} \\ e_3 \hookrightarrow v_3 \end{array}}{\mathsf{if}\ e_1\ e_2\ e_3 \hookrightarrow v_3}$$

$$\frac{e \hookrightarrow v}{\mathsf{E}\ e \hookrightarrow \mathsf{E}\ v} \quad \frac{e \hookrightarrow \mathsf{E}\ v}{\mathsf{E}^{-1}\ e \hookrightarrow v} \quad \frac{e \hookrightarrow v}{\mathsf{F}\ e \hookrightarrow \mathsf{F}\ v} \quad \frac{e \hookrightarrow \mathsf{F}\ v}{\mathsf{F}^{-1}\ e \hookrightarrow v}$$

This semantics induces a set of values, namely, the largest set of terms on which the semantics is idempotent.The set of **values** $\mathbb{V}$ is defined as follows[5]:

$$v ::= \lambda x.e \mid `s \mid \mathsf{E}\ v \mid \mathsf{F}\ v$$

Note that $\mathbb{V} \subset \mathbb{E}$. This containment is one of the reasons why our treatment is often considered "syntactic" (as opposed to "denotational"). We find the containment useful because it allows us to avoid having two similar but still slightly different notions of various concepts, such as typing.

We can refine the type of evaluation to $\hookrightarrow : \mathbb{E} \to \mathbb{V}$. The three basic properties of values are the following:

Lemma 2 (Values).

1. $e_1 \hookrightarrow e_2 \ \land e_1 \hookrightarrow e_3 \Longrightarrow e_2 = e_3$*, and*
2. $e_1 \hookrightarrow e_2 \Longrightarrow e_2 \in \mathbb{V}$*, and*

[5] We could have written $\mathsf{E}\ `s$ and $\mathsf{F}\ \lambda x.e$ for the last two cases, but that puts unnecessary restrictions on the untyped language. It also fails to give us "the largest" set on which the semantics is idempotent. In the typed setting, the type system will ensure that typed values will necessarily have the more restricted form.

3. $v \hookrightarrow v$.

Proof. All proofs are by simple inductions over the height of the derivation.

The semantics also enjoys **type-preservation**.

Lemma 3 (Type Preservation). $\Gamma \vdash e : t \wedge e \hookrightarrow v \Longrightarrow \Gamma \vdash v : t$

Note, however, that it is still possible for some terms to "get stuck" [18] in our language, such as in trying to take the tail (car) of an empty list (nil). We will write $\equiv$ for **syntactic equality** of terms, up to α conversion. For semantic equality we will use the largest congruence when termination is observed[6]. A **context** C is a term with exactly one hole []. We will write $C[e]$ for the variable capture filling of the hole in C with the term e. Two terms $e_1, e_2 \in \mathbb{E}$ are **observationally equivalent**, written $e_1 \approx e_2$, when for every context C it is the case that:

$$(\exists v.C[e_1] \hookrightarrow v) \iff (\exists v.C[e_2] \hookrightarrow v)$$

2.1 Semantics-Preserving Annotations

The key idea behind the proposed approach to dealing with the interpretive overhead is that the user writes an interpreter which includes some additional annotations that have no effect on the semantics of the program but that do have an effect on what happens to the superfluous tags. Thus, the programmer writes the interpreter in a language of annotated terms. An **annotated term** $\hat{e} \in \hat{\mathbb{E}}$ is a term where each occurrence of E _,E^{-1} _,F _, and F^{-1} _ is annotated with one of two **annotations** $\mathbb{B}$:

$$b ::= \mathsf{k} \mid \mathsf{e}$$

Where k stands for "keep" and e stands for "eliminate". **Substitution** is defined on annotated terms in the standard manner. Any term can be **lifted** into an annotated term. Lifting, written, $\lceil \_ \rceil$ simply annotates every constructor and destructor with the tag k. Lifting is substitutive: $\lceil e_1 \rceil [x := \lceil e_2 \rceil] \equiv \lceil e_1[x := e_2] \rceil$. Lifting **types** and **environments** is simply the identity embedding.

Annotated contexts are defined similarly to terms. Lifting on contexts $\lceil C \rceil$ is defined similarly to terms. The **evaluation function** on terms can be lifted to annotated terms where all the constructs are propagated during a computation without inspecting or making any changes to the annotations (Thus, it's OK to use a k-untag operation to remove an e-tag in this semantics.)

[6] Note that, because we have datatypes and some datatype operations can get stuck, one should ideally use a notion of equivalence which distinguishes between getting stuck and diverging. In this paper, we avoid this distinction for the sake of simplicity. Because nowhere in our treatment do we exchange a possibly-stuck term with a possibly-non-terminating term (or the other way around), we expect our results to generalize.

The **subject interpretation** on annotated terms $|\_| : \hat{\mathbb{E}} \to \mathbb{E}$ simply forgets the annotations, and the **target interpretation** $||\_|| : \hat{\mathbb{E}} \to \mathbb{E}$ eliminates constructs annotated with e and just drops the k annotation from the others. For example, $||\mathsf{F}_{\mathsf{e}}\ (\mathsf{car}\ x)|| \equiv \mathsf{car}\ x$. Note that $|\lceil e \rceil| \equiv ||\lceil e \rceil|| \equiv e$, and that both notions of erasure are substitutive. Both notions of erasure are also onto. These facts will allow us to keep reasoning with observational equivalence simple.

The subject interpretation allows us to lift **observational equivalence** to annotated terms, that is, we will define equivalence on annotated terms as follows:

$$\hat{e}_1 \approx \hat{e}_2 \overset{def}{\iff} |\hat{e}_1| \approx |\hat{e}_2|$$

This means that, from the user's point of view, tagging and untagging operations annotated with e or k are semantically the same. Thus, we can really think of these annotations as being purely *hints* to the tag-elimination analysis that can affect only performance. In this paper, if one "hint" is wrong, no tag elimination will be performed at all. The goal of this paper is not, however, to demonstrate that there is a *robust* analysis that solves optimality, but rather that there is an analysis at all. (See Makholm [7] for some ideas to alleviate this practical problem.)

3 A Specification of a Tag Elimination Analysis

In this section, we present a specification of a new tag-elimination analysis. The analysis will be presented as a type system defining a judgment $\Gamma \vdash e/a$. Intuitively, the judgment says *"a describes the type of e before and after the extra tags are eliminated"*. **Annotated types** $\hat{\mathbb{T}}$ are basically types carrying names of tags (either E or F) in certain positions, and are defined as follows:

$$\hat{t} ::= \mathsf{D} \mid \mathsf{V} \mid \hat{t} \to \hat{t} \mid \mathsf{E}\,\hat{t} \mid \mathsf{F}\,\hat{t}$$

We will use two strict subsets ($\mathbb{C} \subset \hat{\mathbb{T}}$ and $\mathbb{A} \subset \hat{\mathbb{T}}$) that can identify two special *families of refinements*:

$$\begin{aligned} c &::= \mathsf{V} \mid \mathsf{E}\ \mathsf{D} \mid \mathsf{F}\ (c \to c) \\ a &::= \mathsf{D} \mid c \mid a \to a \end{aligned}$$

An annotated type c identifies a subset of values (and terms) of type V, and an annotated type a corresponds to legitimate type-specializations of a (subject) value of any type. In the case of the first production for c, the subset is the whole set. In the second case, the subset values that have E tags as the outermost (or top most) construct. In the next case, values that have a F tag. For terms, the annotated type identifies certain terms that can either diverge or evaluate to values identified by the annotated type.

In the context of the work of Hughes and Danvy, an annotated type $\mathsf{E}\ \mathsf{D}$ can be seen as describing a "type-specialization path" from a value of type V

to a value of type D. The additional tag information in the annotated type tells us that we achieve this "type specialization" (semantically) by eliminating an E tag. The **subject** $|a|$ and **target** $||a||$ interpretations of annotated types are:

$$|\mathsf{D}| = \mathsf{D}, \quad |c| = \mathsf{V}, \quad |a_1 \to a_2| = |a_1| \to |a_2|,$$
$$||\mathsf{D}|| = ||\mathsf{E}\ \mathsf{D}|| = \mathsf{D}, \quad ||\mathsf{V}|| = \mathsf{V}, \quad ||\mathsf{F}\ a|| = ||a||, \quad ||a_1 \to a_2|| = ||a_1|| \to ||a_2||.$$

The **tag elimination analysis** is defined as follows:

$$\frac{\Gamma(x) = a}{\Gamma \vdash x/a} \quad \frac{\Gamma \vdash \hat{e}_1/a_1 \to a_2 \quad \Gamma \vdash \hat{e}_2/a_1}{\Gamma \vdash \hat{e}_1\ \hat{e}_2/a_2} \quad \frac{\Gamma, x/a_1 \vdash \hat{e}/a_2}{\Gamma \vdash \lambda x.\hat{e}/a_1 \to a_2} \quad \frac{\Gamma, x/a \vdash \hat{e}/a}{\Gamma \vdash \mathsf{fix}\ x.\hat{e}/a}$$

$$\frac{}{\Gamma \vdash `s/\mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{D}}{\Gamma \vdash u\ \hat{e}/\mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}_1/\mathsf{D} \quad \Gamma \vdash \hat{e}_2/\mathsf{D}}{\Gamma \vdash o\ \hat{e}_1\ \hat{e}_2/\mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}_1/\mathsf{D} \quad \Gamma \vdash \hat{e}_2/a \quad \Gamma \vdash \hat{e}_2/a}{\Gamma \vdash \mathsf{if}\ \hat{e}_1\ \hat{e}_2\ \hat{e}_2/a}$$

$$\frac{\Gamma \vdash \hat{e}/\mathsf{D}}{\Gamma \vdash \mathsf{E}_\mathsf{k}\ \hat{e}/\mathsf{V}} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{V}}{\Gamma \vdash \mathsf{E}_\mathsf{k}^{-1}\ \hat{e}/\mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{V} \to \mathsf{V}}{\Gamma \vdash \mathsf{F}_\mathsf{k}\ \hat{e}/\mathsf{V}} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{V}}{\Gamma \vdash \mathsf{F}_\mathsf{k}^{-1}\ \hat{e}/\mathsf{V} \to \mathsf{V}}$$

$$\frac{\Gamma \vdash \hat{e}/\mathsf{D}}{\Gamma \vdash \mathsf{E}_\mathsf{e}\ \hat{e}/\mathsf{E}\ \mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{E}\ \mathsf{D}}{\Gamma \vdash \mathsf{E}_\mathsf{e}^{-1}\ \hat{e}/\mathsf{D}} \quad \frac{\Gamma \vdash \hat{e}/c_1 \to c_2}{\Gamma \vdash \mathsf{F}_\mathsf{e}\ \hat{e}/\mathsf{F}\ (c_1 \to c_2)} \quad \frac{\Gamma \vdash \hat{e}/\mathsf{F}\ (c_1 \to c_2)}{\Gamma \vdash \mathsf{F}_\mathsf{e}^{-1}\ \hat{e}/c_1 \to c_2}$$

The first two lines of the type system are completely standard. The last line introduces new rules that assign special annotated types for tagging and untagging operations annotated with "eliminate" annotations.

This type system also enjoys weakening and substitution properties, and the semantics also enjoys an **analog of type-preservation** on closed terms. The analysis *includes* the type system, in that any type judgment $\vdash e : t$ has a **canonical** corresponding analysis judgment $\vdash \lceil e \rceil / t$. We can also establish stronger properties of the analysis:

Lemma 4 (Double Typing).

$$\vdash |\hat{e}| : |a| \Longleftarrow \vdash \hat{e}/a \Longrightarrow \vdash ||\hat{e}|| : ||a||$$

This lemma captures the fact the analysis performs (at least) two things implicitly: First, typing the term without the annotations, and second, typing the term after the e-marked tagging and untagging operations have been eliminated. Note that we would not be able to prove this property if we used a instead of c in the last rules. Next we prove stronger, semantic properties about the analysis.

Proof. Both directions by simple inductions. □

4 Semantic Properties of Tag Elimination

Tag elimination changes the type of a term, and so, necessarily, changes the semantics of the term. Fortunately, it is possible to give a simple and accurate account of this change in semantics using so called wrapper/unwrapper functions $W, U : \mathbb{B} \times \mathbb{A} \to \hat{\mathbb{E}}$ (which are a bit more general than the classic embedding/projection pair discussed in the next section):

$$\begin{array}{rlrl}
W_{b,\mathsf{D}} & \equiv \lambda x.x & U_{b,\mathsf{D}} & \equiv \lambda x.x \\
W_{b,\mathsf{V}} & \equiv \lambda x.x & U_{b,\mathsf{V}} & \equiv \lambda x.x \\
W_{b,\mathsf{E}\ \mathsf{D}} & \equiv \lambda x.\mathsf{E}_b\ x & U_{b,\mathsf{E}\ \mathsf{D}} & \equiv \lambda x.\mathsf{E}_b^{-1}\ x \\
W_{b,\mathsf{F}\ a} & \equiv \lambda x.\mathsf{F}_b\ (W_{b,a}\ x) & U_{b,\mathsf{F}\ a} & \equiv \lambda x.U_{b,a}(\mathsf{F}_b^{-1}\ x) \\
W_{b,a_1\to a_2} & \equiv \lambda f.\lambda x.W_{b,a_2}(f(U_{b,a_1}x)) & U_{b,a_1\to a_2} & \equiv \lambda f.\lambda x.U_{b,a_2}(f(W_{b,a_1}x))
\end{array}$$

For simplicity, **we will write** W_a (and similarly U_a) for $|W_{b,a}| \equiv ||W_{\mathsf{k},a}||$. The wrapper and unwrapper functions at a given type a can be seen as completely determining a "type-specialization path".

Lemma 5 (Wrapper/Unwrapper Types and Annotated Types).

1. $\vdash W_{\mathsf{k},a}/||a|| \to |a|$
2. $\vdash W_{\mathsf{e},a}/||a|| \to a$
3. $\vdash |W_{b,a}| : ||a|| \to |a|$
4. $\vdash ||W_{\mathsf{k},a}|| : ||a|| \to |a|$
5. $\vdash ||W_{\mathsf{e},a}|| : ||a|| \to ||a||$

The unwrapper function has the dual types.

Proof. The first two are by simple inductions. The last two come from the basic properties of erasure, and the first two properties. In all cases, we have to establish the properties of the unwrapper function simultaneously. □

Lemma 6 (Simulating Erasure). *For all* $\vdash \hat{e}/a$ *and* $\vdash \hat{v}/a$ *we have*

$$|\hat{e}| \hookrightarrow |\hat{v}| \iff ||\hat{e}|| \hookrightarrow ||\hat{v}||$$

Proof. The forward direction is by a simple induction on the height of the derivation. The backward direction is by an induction on the lexicographic order generated by the height of the derivation and then the size of the term. □

Corollary 1. *Lemma 6 has a number of useful consequences:*

1. *For* $\vdash \hat{e}_1/a$ *and* $\vdash \hat{e}_2/a$*, we have*

$$|\hat{e}_1| \approx |\hat{e}_2| \iff ||\hat{e}_1|| \approx ||\hat{e}_2||$$

2. *For* $\vdash \hat{e}/t$ *we have*

$$||\hat{e}|| \approx |\hat{e}| \equiv |\ulcorner|\hat{e}|\urcorner| \equiv ||\ulcorner|\hat{e}|\urcorner||$$

Lemma 7 (Projecting Embeddings). *For all* $\vdash v : ||a||$

$$U_a(W_a\ v) \approx v$$

Proof. By induction on the structure of the annotated types. □

For any $\hat{e}$ such that $\vdash |\hat{e}| : |a|$, **the tag elimination transformation** $\mathsf{TE}(\hat{e}, a)$ is defined as:

$$\mathsf{TE}(\hat{e}, a) \equiv \begin{cases} ||\hat{e}|| & \vdash \hat{e}/a \\ U_a\ |\hat{e}| & \text{o.w.} \end{cases}$$

Note that the input to the tag-elimination transformation is an annotated term e and an annotated type a. Both the annotations and the annotated type are used to ensure that the transformation is functional. The study of inference techniques for the annotations and the annotated type can alleviate the need for the annotations and the annotated type, but we leave this for future work. Leaving out inference is pragmatically well-motivated, because it is easy for the programmer to provide these inputs.

Theorem 1 (Extensional Semantics of Tag Elimination). *For all* $\vdash |\hat{e}| : |a|$

$$\mathsf{TE}(\hat{e}, a) \ \approx \ U_a\ |\hat{e}|$$

The proof technique used in a previous study on tag elimination [15] works here.

Proof. We only need to prove that for all $\vdash \hat{e}/a$ we have

$$||\hat{e}|| \approx U_a\ |\hat{e}|$$

This proof proceeds by induction the structure of the annotated type a. In the case of D and V, the proof comes from the fact that $||\hat{e}|| \approx |\hat{e}|$ when the annotated type a is simply a type. In the case of $\mathsf{E}\ \mathsf{D}$ and $\mathsf{F}\ a$, the proof uses the definition of erasure, and the induction hypothesis. The case of arrows is the most interesting. It is done using simulation, extensionality, lifting, the ontoness of both erasure functions, and the second part of Corollary 1. □

5 Interpreters

"To explain what interpreters do it is worthwhile to start by discussing the differences between interpreting and translation."

Introduction on web-page of
Russian Interpreters Co-op (RIC).

In order to be able to address the issue of Jones-optimality formally and to establish that a certain program is indeed an *interpreter*, we will need to review some basic issues of encoding and expressibility. Because we are interested in *typed interpreters*, we will begin by refining our notation and define the sets of typed terms and values as

$$e \in \mathbb{E}_{\Gamma \vdash t} \overset{def}{\iff} \Gamma \vdash e : t \quad \text{and} \quad v \in \mathbb{V}_{\Gamma \vdash t} \overset{def}{\iff} \Gamma \vdash v : t$$

And we will write $\mathbb{E}_t$ and $\mathbb{V}_t$ when Γ is empty. By proving type preservation on closed terms, we now can give evaluation a finer type $\hookrightarrow_t : \mathbb{E}_t \to \mathbb{V}_t$.

We define a **first-order datatype** as a type D whose *values* can be tested for meta-level syntactic equality within the language. That is, for all $v_1, v_2 \in \mathbb{V}_D$,

$$v_1 \equiv v_2 \iff v_1 \approx v_2$$

Note that, in general, it is desirable that a language have types which *do not* have this property. In meta-programming settings, it is *dangerously easy* to thusly "trivialize" observational equivalence for *all types* [8, 17, 14]. In particular, if observational equality is the same syntactic equality, many interesting local optimizations such as β reduction become semantically unsound. The type D in the language presented above is a first-order datatype.

A programming language has **syntactic self-representation** if[7] 1) it has an *first-order data type* D, and 2) there exists a full embedding $\ulcorner\_\urcorner : \mathbb{E} \to \mathbb{V}_{\mathsf{D}}$, meaning that:

- $\ulcorner e \urcorner$ is defined for all e, and
- $\ulcorner\_\urcorner$ has a left inverse called $\llcorner\_\lrcorner : \mathbb{V}_{\mathsf{D}} \to \mathbb{E}$, that is, $\llcorner\ulcorner e\urcorner\lrcorner \equiv e$.

The left inverse does not have to be a total function: it just needs to be defined on all elements of the range (image) of the embedding.

For our language, we can define the function and its left-inverse as follows:

$$\begin{array}{rlrl}
\ulcorner x\urcorner & \equiv x & \ulcorner \mathsf{cons}\ e_1\ e_2\urcorner & \equiv (\mathsf{cons}.(\ulcorner e_1\urcorner.\ulcorner e_2\urcorner)) \\
\ulcorner e_1\ e_2\urcorner & \equiv (\mathsf{apply}.(\ulcorner e_1\urcorner.\ulcorner e_2\urcorner)) & \ulcorner \mathsf{equal?}\ e_1\ e_2\urcorner & \equiv (\mathsf{equal?}.(\ulcorner e_1\urcorner.\ulcorner e_2\urcorner)) \\
\ulcorner \lambda x.e\urcorner & \equiv (\mathsf{lambda}.(\ulcorner x\urcorner.\ulcorner e\urcorner)) & \ulcorner \mathsf{if}\ e_1\ e_2\ e_3\urcorner & \equiv (\mathsf{if}.(\ulcorner e_1\urcorner.(\ulcorner e_2\urcorner.\ulcorner e_3\urcorner))) \\
\ulcorner \mathsf{fix}\ x.e\urcorner & \equiv (\mathsf{fix}.(\ulcorner x\urcorner.\ulcorner e\urcorner)) & \ulcorner \mathsf{E}\ e\urcorner & \equiv (\mathsf{tagE}.\ulcorner e\urcorner) \\
\ulcorner `s\urcorner & \equiv (\mathsf{quote}.s) & \ulcorner \mathsf{E}^{-1}\ e\urcorner & \equiv (\mathsf{untagE}.\ulcorner e\urcorner) \\
\ulcorner \mathsf{car}\ e\urcorner & \equiv (\mathsf{car}.\ulcorner e\urcorner) & \ulcorner \mathsf{F}\ e\urcorner & \equiv (\mathsf{tagF}.\ulcorner e\urcorner) \\
\ulcorner \mathsf{cdr}\ e\urcorner & \equiv (\mathsf{cdr}.\ulcorner e\urcorner) & \ulcorner \mathsf{F}^{-1}\ e\urcorner & \equiv (\mathsf{untagF}.\ulcorner e\urcorner)
\end{array}$$

and,

$$\begin{array}{rlrl}
\llcorner x\lrcorner & \equiv x & \llcorner(\mathsf{cons}.(e_1.e_2))\lrcorner & \equiv \mathsf{cons}\ \llcorner e_1\lrcorner\ \llcorner e_2\lrcorner \\
\llcorner(\mathsf{apply}.(e_1.e_2))\lrcorner & \equiv \llcorner e_1\lrcorner\ \llcorner e_2\lrcorner & \llcorner(\mathsf{equal?}.(e_1.e_2))\lrcorner & \equiv \mathsf{equal?}\ \llcorner e\lrcorner\ \llcorner e\lrcorner \\
\llcorner(\mathsf{lambda}.(x.e))\lrcorner & \equiv \lambda\llcorner x\lrcorner.\llcorner e\lrcorner & \llcorner(\mathsf{if}.(e_1.(e_2.e_3)))\lrcorner & \equiv \mathsf{if}\ \llcorner e_1\lrcorner\ \llcorner e_2\lrcorner\ \llcorner e_3\lrcorner \\
\llcorner(\mathsf{fix}.(x.e))\lrcorner & \equiv \mathsf{fix}\ \llcorner x\lrcorner.\llcorner e\lrcorner & \llcorner(\mathsf{tagE}.e)\lrcorner & \equiv \mathsf{E}\ \llcorner e\lrcorner \\
\llcorner(\mathsf{quote}.s)\lrcorner & \equiv s & \llcorner(\mathsf{untagE}.e)\lrcorner & \equiv \mathsf{E}^{-1}\ \llcorner e\lrcorner \\
\llcorner(\mathsf{car}.e)\lrcorner & \equiv \mathsf{car}\ \llcorner e\lrcorner & \llcorner(\mathsf{tagF}.e)\lrcorner & \equiv \mathsf{F}\ \llcorner e\lrcorner \\
\llcorner(\mathsf{cdr}.e)\lrcorner & \equiv \mathsf{cdr}\ \llcorner e\lrcorner & \llcorner(\mathsf{untagF}.e)\lrcorner & \equiv \mathsf{F}^{-1}\ \llcorner e\lrcorner
\end{array}$$

It is easy to see that $\llcorner\ulcorner e\urcorner\lrcorner \equiv e$. Such encoding/decoding pairs have sometimes been called **reify** and **reflect**. This terminology was promoted by Brian Cantwell Smith [12]. Reify provides us with a way of "materializing" or "representing" terms within the language, and reflect provides us with a way of interpreting an internal representation back into a (meta-level) term. Note that all these

[7] The second requirement is not hard: it is enough to know that D can represent the natural numbers. The detailed treatment here is primarily expository.

functions exist at the meta-level, and that expressing them within the language requires first defining them at the meta-level.

As with evaluation, we will be more interested in the "subject-typed" versions of these functions: $\ulcorner\_\urcorner_t : \mathbb{E}_t \to \mathbb{V}_{\mathsf{D}}$ and $\llcorner\_\lrcorner_t : \mathbb{V}_{\mathsf{D}} \to \mathbb{E}_t$, where the first one is achieved by restricting the input to be well-typed, and the second by restricting the output to being well-typed.

With syntactic representation in hand, it is tempting to view interpreters as a program (call it direct) expressing[8] the following function:

$$(\llcorner\_\lrcorner_t ; \hookrightarrow_t) : \mathbb{V}_{\mathsf{D}} \to \mathbb{V}_t$$

Because such a function produces a value of the same (subject) type as the (subject) term being interpreted, we will call them *direct interpreters*. But it turns out that expressing such an interpreter in a statically typed programming language (such as the one at hand) is a rather subtle matter. In fact, it is only recently that some work on programming type-indexed values in ML [19] has given a hint of how such a function can be expressed. But even then, it is known that we can express such an "interpreter" for each type, but it is not known that there is *one term* that we can call the interpreter and that would work for all types.

5.1 Expressibility and Admissibility of Encoding/Decoding

While the encoding and decoding function presented above would generally be enough for expressing an interpreter in an untyped setting, they are generally not enough in a typed setting. To clarify this point, we will analyze the expressibility of these functions and of interpreters.

A partial (meta-level) function $f : \mathbb{V}_{t_1} \to \mathbb{V}_{t_2}$ is **expressed** by a term $e_f \in \mathbb{V}_{t_1 \to t_2}$ when for all $v \in \mathbb{V}_{t_1}$

$$e_f\ v \approx f(v).$$

As a simple example, for any t, the function $id : \mathbb{V}_t \to \mathbb{V}_t$ is expressed by the term $e_{id} \equiv \lambda x.x \in \mathbb{V}_{t \to t}$. In contrast, any function that distinguishes between the terms $\lambda x.(\lambda y.y)x$ and $\lambda x.x$ would not be expressible. A partial meta-level function $f : \mathbb{V}_{t_1} \to \mathbb{V}_{t_2}$ is **admissible** when for all $v_1, v_2 \in \mathbb{V}_{t_1}$ such that $v_1 \approx v_2$ if $f(v_1)$ is defined then 1) $f(v_2)$ is defined, and 2) $f(v_1) \approx f(v_2)$. Expressible functions are admissible, but not necessarily the converse. Thus, admissibility helps in establishing negative statements on what can be expressed.

As is, the two untyped functions described above *cannot* be expressed in our language: in both cases, they don't have the right type: at least they need to be restricted to values in both the domain and the co-domain.

If we restrict the decoding function to values and its result to values of type $\mathsf{V} \to \mathsf{V}$, we get a function of type $\mathbb{V}_{\mathsf{D}} \to \mathbb{V}_{\mathsf{V}\to\mathsf{V}}$. This encoding function is admissible (in fact, even though we don't prove it, we expect that it is expressible).

[8] Here we omit the formal definition of "expresses" for reasons of space.

Because one of the main things that we generally want to do with expressions is to evaluate them after decoding them, a decoding function restricted to values almost (but not quite) models an interpreter.

However, if we restrict the encoding function to values, and then further restrict it to values of some type, say, type $\mathsf{V} \to \mathsf{V}$, we get a function of type $\mathbb{V}_{\mathsf{V}\to\mathsf{V}} \to \mathbb{V}_{\mathsf{D}}$. This function distinguishes between operationally equivalent terms, therefore, it is not even *admissible*.

Because of the subtleties involved in expressing a direct interpreter, a more commonly used technique for implementing interpreters involves the use of a *universal datatype*. We define a **universal datatype** as a type V that allows us to *simulate* values of any type by a value of one (universal) type. We can formalize the notion of simulation concisely as follows: There must exist a *universal embedding function* $w_t : \mathbb{V}_t \to \mathbb{V}_V$ such that:

$$v_1 \approx v_2 \iff w_t(v_1) \approx w_t(v_2)$$

We can establish that the datatype V in our programming language is a universal datatype by using a family of terms E_t and P_t and showing that the latter is a left-inverse of the former:

$$\begin{array}{llll} E_{\mathsf{D}} & \equiv \lambda x.\mathsf{E}\ x & P_{\mathsf{D}} & \equiv \lambda x.\mathsf{E}^{-1}\ x \\ E_{\mathsf{V}} & \equiv \lambda x.x & P_{\mathsf{V}} & \equiv \lambda x.x \\ E_{t_1\to t_2} & \equiv \lambda f.\mathsf{F}\ \lambda x.E_{t_2}(f(P_{t_1}\ x)) & P_{t_1\to t_2} & \equiv \lambda f.\lambda x.P_{t_2}(\mathsf{F}^{-1}\ f(E_{t_1}\ x)) \end{array}$$

And it is easy to show that $P_t(E_t\ v) \approx v$.

Lemma 8 (Projecting Embeddings). $P_t(E_t\ v) \approx v$

Proof. By induction on the structure of the types. □

Remark 1. Note that the fact that we don't need to apply the induction hypothesis in the case of V is essential for the ability to do the proof by induction on the structure of types.

Such a universal datatype plays a crucial role in allowing us to express simple interpreters in non-dependently typed programming languages. In particular, they allow us to implement *typed interpreters* by what we will call an **indirect interpreter**. A term is an indirect interpreter (call it $\mathsf{indirect}$) if it expresses the function:

$$(\llcorner\lrcorner_t; \hookrightarrow_t; w_t) : \mathbb{V}_{\mathsf{D}} \to \mathbb{V}_{\mathsf{V}}$$

While this shift from direct to indirect interpreters makes writing interpreters easier, it also introduces the very overhead that the tag elimination transformation will need to remove. In our Scheme-based implementation the self-interpreter is essentially as follows[9]:

[9] Unfortunately, space does not allow us to give all the details here.

```
(fix newenv-eval (lambda env (fix myeval (lambda e
 (if (atom? e) (app env e)
 (if (equal? (car e) (quote lambda)) (.tagF. (lambda x (app (app newenv-eval
   (lambda y (if (equal? y (car (cdr e))) x (app env y)))) (car (cdr (cdr e))))))
 (if (equal? (car e) (quote app)) (app (.untagF.
   (app myeval (car (cdr e)))) (app myeval (car (cdr (cdr e)))))
 (if (equal? (car e) (quote tagF)) (tagF (.untagF. (app myeval (car (cdr e)))))
...
```

Where, for example, tagF is $\mathsf{F_k}$ and .tagF. is $\mathsf{F_e}$. We will define our **typed self-interpreter** tsi to be the term above specialized (by simple application) to the empty environment.

It is folklore that tsi is an indirect interpreter and we do not prove it here.

6 Staged Interpreters and Translation

We mentioned in the introduction that a staged interpreter can be viewed simply as a translation. This is a subtle shift in perspective. In particular, the only requirement on interpreters is that they yield "the right value" from a program. Often, the straight-forward implementation of interpreters (in both CBN and CBV programming languages) tends to have a pragmatic disadvantage: They simply do not ensure a clean separation between the various "stages" of computing "the right value" of an expression. In particular, straight-forward implementations of interpreters tend to repeatedly traverse the expression being interpreted. Ideally, one would like this traversal to be done once and for all. In general, achieving this kind of separation gives rise to the need for using two- and multi-level languages.

"Staged interpreters", therefore, are not any composition of the functions described above, but rather, a particular *implementation* of this composition that behaves in a certain manner. Because CBN and CBV functional languages cannot force evaluation under lambda, they are thought to be insufficient for expressing staging. Nevertheless, the *result* of a staged interpreter (which is a term in the target language corresponding to the given term in the subject language) is expressible in the language. Furthermore, the result of a staged interpreter is also observationally equivalent to the result of an interpreter. These facts imply that, while staged interpreters are not known to be expressible in a language such as the one we are studying in this paper, they are still *admissible*.

Pragmatic experience with staged interpreters suggests that their input-output behavior can be modeled rather straight-forwardly as a *translator*. For simplicity, it is enough to focus on a **self-interpreter**: an interpreter written in the same language it interprets. When staged, a self-interpreter *translates* terms in one language into terms in the same language. Note however, that for typing reasons, the resulting translation is not the identity. Rather, it is the following function:

$$\begin{array}{rlrl}
\mathcal{E}(x) &\equiv x & \mathcal{E}(u\ e_1 e_2) &\equiv \mathsf{E_e}\ u\ (\mathsf{E}^{-1}_{\mathsf{e}}\ \mathcal{E}(e_1))(\mathsf{E}^{-1}_{\mathsf{e}}\ \mathcal{E}(e_2)) \\
\mathcal{E}(e_1\ e_2) &\equiv (\mathsf{F}^{-1}_{\mathsf{e}}\ \mathcal{E}(e_1))\ \mathcal{E}(e_2) & \mathcal{E}(\mathsf{if}\ e_1\ e_2\ e_3) &\equiv \mathsf{if}\ (\mathsf{E}^{-1}_{\mathsf{e}}\ \mathcal{E}(e_1))\ \mathcal{E}(e_2)\ \mathcal{E}(e_3) \\
\mathcal{E}(\lambda x.e) &\equiv \mathsf{F_e}\ \lambda x.\mathcal{E}(e) & \mathcal{E}(\mathsf{E}\ e) &\equiv \mathsf{E_k}\ \mathsf{E}^{-1}_{\mathsf{e}}\ \mathcal{E}(e) \\
\mathcal{E}(\mathsf{fix}\ x.e) &\equiv \mathsf{fix}\ x.\mathcal{E}(e) & \mathcal{E}(\mathsf{E}^{-1}\ e) &\equiv \mathsf{E_e}\ \mathsf{E}^{-1}_{\mathsf{k}}\ \mathcal{E}(e) \\
\mathcal{E}(`s) &\equiv \mathsf{E_e}\ `s & \mathcal{E}(\mathsf{F}\ e) &\equiv \mathsf{F_k}\ \mathsf{F}^{-1}_{\mathsf{e}}\ \mathcal{E}(e) \\
\mathcal{E}(o\ e) &\equiv \mathsf{E_e}\ o\ (\mathsf{E}^{-1}_{\mathsf{e}}\ \mathcal{E}(e)) & \mathcal{E}(\mathsf{F}^{-1}\ e) &\equiv \mathsf{F_e}\ \mathsf{F}^{-1}_{\mathsf{k}}\ \mathcal{E}(e)
\end{array}$$

One can show that $|\mathcal{E}(e)| \approx (\mathsf{tsi}\ \ulcorner e \urcorner)$. It is reasonable to expect that a staged tsi produces $|\mathcal{E}(e)|$ when applied to $\ulcorner e \urcorner$, and our implementation confirms it.

The annotated type of the result of translating a term of type t is defined as follows[10]:

$$\mathcal{E}(\mathsf{D}) \equiv \mathsf{E\ D}, \quad \mathcal{E}(\mathsf{V}) \equiv \mathsf{V}, \quad \mathcal{E}(t_1 \to t_2) \equiv \mathsf{F}\ (\mathcal{E}(t_1) \to \mathcal{E}(t_2)),$$

The idempotence of the translation on V is essential for being able to do the various proofs that are carried out by induction on the structure of the types.

Lemma 9 (Soundness (and Full-Abstraction) of Translation).

$$e_1 \approx e_2 \iff |\mathcal{E}(e_1)| \approx |\mathcal{E}(e_2)|$$

Proof. Proved by showing that $\mathcal{E}(e) \approx W_t\ e$, and Projecting Embeddings lemma. □

Lemma 10 (Well-Typed Terms "Go Through").

1. $\Gamma \vdash e : t \implies \mathcal{E}(\Gamma) \vdash \mathcal{E}(e)/\mathcal{E}(t)$
2. $||\mathcal{E}(e)|| \equiv e$

Proof. Both by a simple induction on the structure of e. □

The first part of this lemma means that running the staged interpreter on a well-typed subject program yields a term that passes the tag elimination analysis. The second part means that erasing the operations marked by e yields back precisely the term that we started with.

Note further that, using the second part, we can strengthen the first part of this lemma to be:

$$\Gamma \vdash e : t \iff \mathcal{E}(\Gamma) \vdash \mathcal{E}(e)/\mathcal{E}(t)$$

This statement means that applying the tag elimination analysis to the result of a staged self-interpreter is exactly the same as type-checking the term being interpreted. This is probably the most accurate characterization of the strength of the idea of tag-elimination.

[10] Now we can see how U and W generalize P and E. For example, $W_{\mathcal{E}(t)} = E_t$.

7 Jones-Optimal Specialization

At this point, we have presented a variety of results that indicate that tag elimination has a useful application in the context of self-application of a specific typed programming language, and would therefore be useful in improving the effectiveness of traditional partial evaluators in staging many interesting interpreters written in this language. Now we turn to addressing the long-standing open problem of Jones-optimality, formally. A function PE is a **partial evaluator** if, for all closed e_1 and e_2:

$$\mathsf{PE}(e_1, e_2) \approx e_1\ e_2$$

A partial evaluator is *partially-correct* if it is a partial function satisfying the above equation, when defined. Note that we require e_1 and e_2 to be closed only for simplicity, as partial evaluators are syntactic operations and therefore must deal with free variables anyway. A function tPE is a A **typed** partial evaluator if

$$\vdash e_1\ e_2 : |a| \implies \mathsf{tPE}(e_1, e_2, a) \approx U_a\ (e_1\ e_2)$$

A partial evaluator is *partially-correct* if it is a partial function satisfying the above equation, when defined. This definition of a typed partial evaluator is motivated by the definition of an self-interpreter in a typed programming language. A **self-interpreter** si is a term such that:

$$\mathsf{si}\ \ulcorner e \urcorner\ \approx\ e$$

A **typed self-interpreter** tsi is a term such that:

$$\vdash e : t \implies \mathsf{tsi}\ \ulcorner e \urcorner\ \approx\ w_t\ e$$

where w_t is a universal embedding function. Now we can recapitulate the definition of Jones-optimality [5]. A partial evaluator PE is **Jones-optimal** with respect to a an untyped self-interpreter si when for all $\vdash e : t$ we have

$$\mathsf{PE}(\mathsf{si}, \ulcorner e \urcorner) \equiv e$$

Again motivated by the role that a universal datatype plays in typed interpreters, we generalize the definition of Jones-optimality to the typed setting as follows: A typed partial evaluator tPE is said to be **Jones-optimal** with respect to a typed self-interpreter tsi when for all $\vdash e : t$:

$$\mathsf{tPE}(\mathsf{tsi}, \ulcorner e \urcorner, \mathcal{E}(t)) \equiv e$$

Theorem 2 (Main).

1. *Whenever* $\mathsf{PE}(\_, \_)$ *is a (partially) correct partial evaluator,* $\mathsf{TE}(\mathsf{PE}(\_, \_), \_)$ *is a (partially) correct* **typed** *partial evaluator, furthermore*

2. *Whenever, for all e it is the case that* $\mathsf{PE}(\mathsf{tsi}, \ulcorner e \urcorner) \equiv \mathcal{E}(e)$*, then* $\mathsf{TE}(\mathsf{PE}(\_,\_),\_)$ *is* **Jones-optimal**.

Proof. For the first part, all we need is to follow a simple sequence of semantic equalities:

$$\begin{aligned} & \mathsf{TE}(\mathsf{PE}(e_1, e_2), a) \text{ by extensional semantics of } \mathsf{TE} \\ \approx\ & U_a(\mathsf{PE}(e_1, e_2)) \text{ by definition of a } \mathsf{PE} \\ \approx\ & U_a(e_1\ e_2) \end{aligned}$$

and we have satisfied the definition of a tPE.

For the second part, we only have to follow a simple sequence of syntactic equalities:

$$\begin{aligned} & \mathsf{TE}(\mathsf{PE}(\mathsf{tsi}, \ulcorner e \urcorner), \mathcal{E}(t)) \text{ by assumption} \\ \equiv\ & \mathsf{TE}(\mathcal{E}(e), \mathcal{E}(t)) \text{ by Lemma 10.1, } \vdash \mathcal{E}(e)/\mathcal{E}(t) \text{ so } \mathsf{TE} \text{ ``succeeds''} \\ \equiv\ & ||\mathcal{E}(e)|| \text{ and by Lemma 10.2} \\ \equiv\ & e \end{aligned}$$

and we have satisfied the definition of typed Jones-optimality. □

We have built an implementation that supports this result.

8 Conclusions and Future Work

In this paper, we have presented the theoretical results showing how Jones-optimality is achieved using tag elimination. We have also implemented a system based on the analysis presented here (in Scheme), and it has validated our theoretical results[11]. The analysis we presented here contains technical improvements over the original proposal [15] in that it uses a simpler judgment. The main reason for this simplicity is that we exploit information about well-formedness of annotated types in the judgment[12]. However, it is also more specialized than the original analysis, which is parametric over an arbitrary datatype that we might want to eliminate.

The moral of the present work is that there is a practical solution to the problem of Jones-optimality which can be attained through some simple annotations by the user. There is evidence that the annotations may not be necessary in practice. Makholm [7] implemented a variant of tag elimination for a first-order language whose type structure is different than that of the language we use here. In this implementation, the analog of our e and k annotations are inferred automatically by the tag eliminator instead of being embedded in the staged interpreter. In principle, it seems that such inference in the setting presented in this paper should be decidable although not necessarily efficient. Whether or

[11] The system can be downloaded from http://www.diku.dk/~makholm/teal.tar.gz.

[12] In fact, in this paper, we have only talked about well-formed annotated types. There is a general way to go from the original definition of well-formedness to the kind of presentation given here, but this is beyond the limits of space available here.

not efficient and practical inference will scale to the higher-order setting is not known. The work on dynamic typing may help establish such a result formally [2]. Combinations of wrapper and unwrapper functions provide natural mechanisms for a notion of subsumption or subtyping that can be used to provide an analog of soft-typing. Finally, we hope to generalize this work to richer settings with state and polymorphism.

Acknowledgements: We are very grateful to Neil Jones, who played an active part in the early stages of this work, and offered valuable comments thereafter. We also enjoyed many useful discussions with Makato Takayama and Peter Thiemann. We recieved many useful comment on earlier drafts from Christopher League, Henrik Nilsson, Stefan Monnier, Carsten Schürmann, Valery Trifonov, and an anonymous referee.

References

1. Olivier Danvy. A simple solution to type specialization. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, Aalborg, 1998.
2. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California.* ACM, ACM Press, 1995.
3. John Hughes. Type specialization. *ACM Computing Surveys*, 30(3es), 1998.
4. John Hughes. The correctness of type specialisation. In *European Symposium on Programming (ESOP)*, 2000.
5. Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14, North-Holland, 1988. IFIP World Congress Proceedings, Elsevier Science Publishers B.V.
6. Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
7. Henning Makholm. On Jones-optimal specialization for strongly typed languages. In *[13]*, pages 129–148, 2000.
8. John C. Mitchell. On abstraction and the expressive power of programming languages. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 290–310. Springer-Verlag, 1991.
9. Torben Mogensen. Inherited limits. In *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 189–202. Springer-Verlag, 1999.
10. Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
11. Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development,*,

volume 352 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 1989.

12. Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, 1982.
13. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
14. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Boston, 2000.
15. Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.
16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM.
17. Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.
18. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
19. Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 289–300, Baltimore, 1998. ACM Press.