Tag Elimination and Jones-Optimality

Taha, Makholm and Hughes

Yale, DIKU, and Chalmers
`taha@cs.yale.edu`

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].

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 language is to simply write an interpreter for the DSL in some meta-language, and then to stage this interpreter either manually adding explicit staging annotations (multi-stage programming [16, 11, 14]) or by applying an automatic binding-time analysis (offline 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[2]. 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.

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 object 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 [10] this weakness and generates programs that contain *superfluous tagging and untagging operations*. To give

---

[1] Proofs in a long version at `http://www.cs.yale.edu/homes/taha/publications.html`.

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

an idea of the cost of these extra tags, here is the cost of running two sample programs with and without the tags in them[3]:

Term	fact 12	fib 10
Speedup	2.6x	1.9x

How 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 system). However, it is not clear that self-interpretation can be achieved in such languages [12]. A more pressing practical concern is that such systems lose decidabile 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 to 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].

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, formally specifying which is non-trivial. 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.

---

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

2

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, 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 interpreterations. The tag elimination analysis 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 the "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 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:

$$t ::= D \mid V \mid t \to t$$

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:

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

But we do not need case analysis on V so we omit it for simplicity. We assume an infinite set of **names** $\mathbb{X}$ ranged over by x, and that this set includes the special

variables "nil, true, false". The set of **expressions** $\mathbb{E}$ are defined as follows:

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

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. One should note that substitution "does not do anything" with ${}`s$, or rather, it is the identity. We use a standard **type system** for this language.

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

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

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

The type system enjoys weakening and substitution properties. A standard **big-step operational semantics** $\hookrightarrow : \mathbb{E} \to \mathbb{E}$ for this language is used:

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

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

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

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

The set of **values** $\mathbb{V}$ is defined as follows:

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

We can refine the type of evaluation to $\hookrightarrow : \mathbb{E} \to \mathbb{V}$. The semantics also enjoys **type-preservation** for closed terms. We will write $\equiv$ for **syntactic equality** of terms, up to α conversion. For semantic equality we will use the largest con-

gruence when termination is observed. 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 write the interpreter in a language of annotated terms. **Annotated terms** are terms where each occurrence of $\mathsf{E}\ \_, \mathsf{E}^{-1}\ \_, \mathsf{F}\ \_$, and $\mathsf{F}^{-1}\ \_$ are annotated 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 **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 making any changes to the annotations.

   The **subject interpretation** on terms $\lfloor \_ \rfloor$ simply forgets the annotations, and the **target interpretation** $\lVert \_ \rVert$ eliminates constructs annotated with e but keeps all others (and also forgets the k annotation on the remaining ones). For example, $\lVert \mathsf{F}_{\mathsf{e}}\ (\mathsf{car}\ x) \rVert \equiv \mathsf{car}\ x$. Note that $\lfloor \lceil e \rceil \rfloor \equiv \lVert \lceil e \rceil \rVert \equiv e$, 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:

$$e_1 \approx e_2 \iff |e_1| \approx |e_2|$$

This means that, from the user 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.

3   Tag Elimination Analysis

In this section, we present 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"*. A notion of an *annotated type* will be used to capture what can be described as a *"specialization path"* from a subject type (type before specialization) to a target type (type after specialization). **Annotated types** $(\mathbb{C}, \mathbb{A})$ are defined by the following two syntactic categories (see [15] for a fuller treatment.):

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

An annotated type c corresponds to legitimate specializations of a (subject) value of type V, and an annotated type a corresponds to legitimate specializations of a (subject) value of any type t. An annotated type $\mathsf{E}\ D$ describes a specialization path from a value of type V to a value of type D. The additional tag information in the annotated type tell us that we achieve this specialization (semantically) by eliminating an E tag. The **subject** $|a|$ and **target** $||a||$ interpretations of annotated types are:

$$|D| = D, \quad |c| = V, \quad |a_1 \to a_2| = |a_1| \to |a_2|,$$

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

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

$$\frac{\Gamma(x) = a}{\Gamma \vdash x/a} \qquad \frac{\Gamma \vdash e_1/a_1 \to a_2 \quad \Gamma \vdash e_2/a_1}{\Gamma \vdash e_1\ e_2/a_2} \qquad \frac{\Gamma, x/a_1 \vdash e/a_2}{\Gamma \vdash \lambda x.e/a_1 \to a_2} \qquad \frac{\Gamma, x/a \vdash e/a}{\Gamma \vdash \mathsf{fix}\ x.e/a}$$

$$\frac{}{\Gamma \vdash {}^\backprime s/D} \qquad \frac{\Gamma \vdash e/D}{\Gamma \vdash u\ e/D} \qquad \frac{\Gamma \vdash e_1/D \quad \Gamma \vdash e_2/D}{\Gamma \vdash o\ e_1\ e_2/D} \qquad \frac{\Gamma \vdash e_1/D \quad \Gamma \vdash e_2/a \quad \Gamma \vdash e_2/a}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_2/a}$$

$$\frac{\Gamma \vdash e/D}{\Gamma \vdash \mathsf{E_k}\ e/V} \qquad \frac{\Gamma \vdash e/V}{\Gamma \vdash \mathsf{E_k^{-1}}\ e/D} \qquad \frac{\Gamma \vdash e/V \to V}{\Gamma \vdash \mathsf{F_k}\ e/V} \qquad \frac{\Gamma \vdash e/V}{\Gamma \vdash \mathsf{F_k^{-1}}\ e/V \to V}$$

$$\frac{\Gamma \vdash e/D}{\Gamma \vdash \mathsf{E_e}\ e/\mathsf{E}\ D} \qquad \frac{\Gamma \vdash e/\mathsf{E}\ D}{\Gamma \vdash \mathsf{E_e^{-1}}\ e/D} \qquad \frac{\Gamma \vdash e/c_1 \to c_2}{\Gamma \vdash \mathsf{F_e}\ e/\mathsf{F}\ (c_1 \to c_2)} \qquad \frac{\Gamma \vdash e/\mathsf{F}\ (c_1 \to c_2)}{\Gamma \vdash \mathsf{F_e^{-1}}\ e/c_1 \to c_2}$$

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 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 $\vdash \lceil e \rceil /t$. Lifting is substitutive, as $\lceil e_1 \rceil [x := \lceil e_2 \rceil] \equiv \lceil e_1[x := e_2] \rceil$. We can also establish stronger properties of the analysis:

Lemma 1 (Double Typing).

$$\vdash |e| : |a| \impliedby \vdash e/a \implies \vdash ||e|| : ||a||$$

This lemma captures the fact the analysis performs 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 instead of a. Next we prove stronger, semantic properties about the analysis.

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 $W, U : \mathbb{B} \times \mathbb{A} \to \mathbb{E}$ (which are a bit more general than the classic embedding/projection pair discussed in the next section):

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

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 the **specialization path** mentioned earlier.

Lemma 2 (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.

Lemma 3 (Simulating Erasure). *For all* $\vdash e/a$ *and* $\vdash v/a$ *we have*

$$
|e| \hookrightarrow |v| \iff ||e|| \hookrightarrow ||v||
$$

Corollary 1. *Lemma 3 has a number of useful consequences:*

1. *For* $\vdash e_1/a$ *and* $\vdash e_2/a$, *we have*

$$
|e_1| \approx |e_2| \iff ||e_1|| \approx ||e_2||
$$

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

$$
||e|| \approx |e| \equiv |\lceil e \rceil| \equiv ||\lceil e \rceil||
$$

7

Lemma 4 (Projecting Embeddings). *For all* $\vdash v : \|a\|$

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

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

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

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 these two inputs, 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* $\vdash |e| : |a|$

$$TE(e,a)\ \approx\ U_a\ |e|$$

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

5   Interpreters

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 issue of encodings 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

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

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 $\hookrightarrow_t \colon \mathbb{E}_t \to \mathbb{V}_t$.

A **first-order datatype** is 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* [9, 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 first-order datatype.

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

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

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

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

and,

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

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 [13]. 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 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: $\ulcorner \_ \urcorner_t : \mathbb{E}_t \to \mathbb{V}_D$ and $\llcorner \_ \lrcorner_t : \mathbb{V}_D \to \mathbb{E}_t$, 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[4] the following function:

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

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 [18] 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

---

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

there is *one term* that we can call the interpreter and that would work for all types.

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*. A **universal datatype** is 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. Given a *wrapper* (or *embedding*) function $w_t : \mathbb{V}_t \to \mathbb{V}_V$ we have:

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

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:

$$
\begin{aligned}
E_D &\equiv \lambda x.\mathsf{E}\ x & P_D &\equiv \lambda x.\mathsf{E}^{-1}\ x \\
E_V &\equiv \lambda x.x & P_V &\equiv \lambda x.x \\
E_{t_1 \to t_2} &\equiv \lambda f.\mathsf{F}\ \lambda x.E_{t_2}(f(P_{t_1}\ x)) & P_{t_1 \to t_2} &\equiv \lambda f.\lambda x.P_{t_2}(\mathsf{F}^{-1}\ f(E_{t_1}\ x))
\end{aligned}
$$

And it is easy to show that $P_t(E_t\ v) \approx v$. 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 a program (call it indirect) expressing the function:

$$(\llcorner\lrcorner t; \hookrightarrow_t; w_t) : \mathbb{V}_D \to \mathbb{V}_V$$

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[5]:

```
(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 curried application) to the empty environment.

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

---

[5] Unfortunately, space does not allow us to give all the details here. A longer version

6 Staged Interpreters and Translation

We mentioned in the introduction that a staged interpreter can be viewed simply as a translation. However, this shift in perspective is not trivial. 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 involve repeatedly traverse the expression being interpreted. Ideally, one would like this traversal to be done once and for all. In generally, 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* in a general sense of being semantically reasonable.

Pragmatic experience with staged interpreters suggests that their input-output behavior can be modeled rather straight-forwardly as a *translator*. On terms, the translation is verbose but simple:

$$\mathcal{E}(x) \equiv x \qquad\qquad \mathcal{E}(u\ e_1 e_2) \equiv \mathsf{E_e}\ u\ (\mathsf{E_e^{-1}}\ \mathcal{E}(e_1))(\mathsf{E_e^{-1}}\ \mathcal{E}(e_1))$$
$$\mathcal{E}(e\ e) \equiv (\mathsf{F_e^{-1}}\ \mathcal{E}(e))\ \mathcal{E}(e) \qquad \mathcal{E}(\mathsf{if}\ e_1\ e_2\ e_3) \equiv \mathsf{if}\ (\mathsf{E_e^{-1}}\ \mathcal{E}(e_1))\ \mathcal{E}(e_2)\ \mathcal{E}(e_3)$$
$$\mathcal{E}(\lambda x.e) \equiv \mathsf{F_e}\ \lambda x.\mathcal{E}(e) \qquad\qquad \mathcal{E}(\mathsf{E}\ e) \equiv \mathsf{E_k}\ \mathsf{E_e^{-1}}\ \mathcal{E}(e)$$
$$\mathcal{E}(\mathsf{fix}\ x.e) \equiv \mathsf{fix}\ x.\mathcal{E}(e) \qquad\qquad \mathcal{E}(\mathsf{E^{-1}}\ e) \equiv \mathsf{E_e}\ \mathsf{E_k^{-1}}\ \mathcal{E}(e)$$
$$\mathcal{E}('s) \equiv \mathsf{E_e}\ 's \qquad\qquad\qquad \mathcal{E}(\mathsf{F}\ e) \equiv \mathsf{F_k}\ \mathsf{F_e^{-1}}\ \mathcal{E}(e)$$
$$\mathcal{E}(o\ e) \equiv \mathsf{E_e}\ o\ (\mathsf{E_e^{-1}}\ \mathcal{E}(e)) \qquad\qquad \mathcal{E}(\mathsf{F^{-1}}\ e) \equiv \mathsf{F_e}\ \mathsf{F_k^{-1}}\ \mathcal{E}(e)$$

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[6]:

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

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 5 (Soundness (and Full-Abstraction) of Translation).

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

---

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

Lemma 6 (Well-Typed Terms "Go Through").

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

The first part of this lemma means that running the staged interpreter on a well-typed subject program yields a term that passed 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.

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 old-standing open problem of Jones-optimality, formally. A function PE is a **partial evaluator** if:

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

A partial evaluator is *partially-correct* if it is a partial function satisfying the above equation, when defined. A function tPE is a A **typed** partial evaluator if

$$\mathsf{tPE}(e_1, e_2, a) \approx U_a \; (e_1 \; e_2)$$

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:

$$\mathsf{tsi} \; \ulcorner e \urcorner \approx w_t \; e$$

where w_t is some type-indexed embedding function. Now we can recap 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* $\mathsf{PE}(e_1, e_2)$ *is a (partially) correct partial evaluator,* $\mathsf{TE}(\mathsf{PE}(e_1, e_2), a)$ *is a (partially) correct* **typed** *partial evaluator, furthermore*
2. *Whenever* $\mathsf{PE}(\mathsf{tsi}, \ulcorner e \urcorner) \equiv \mathcal{E}(e)$, *then* $\mathsf{TE}(\mathsf{PE}(e_1, e_2), a)$ *is* **Jones-optimal**.

We have built an implementation that 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. The analysis we presented here contains technical improvements over the original proposal [15] in that it uses a simpler judgement. The main reason for this simplicity is that we exploit information about well-formedness of annotated types in the judgement[7]. 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 that requires only some simple annotations by the user. There is also 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 embedded in the staged interpreter. Whether or not this will scale to the higher-order setting is not known. The work on dynamic type may help establish such a result formally [2]. Finally, we hope to generalize this work to richer settings with state and polymorphism.

---

[7] 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 here.

References

1. Olivier Danvy. A simple solution to type specialization. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *LNCS*, Aalborg, July 1998.

2. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California*. ACM Press, June 1995.

3. John Hughes. Type specialization. *Computing Surveys*, 30(3es), September 1998.

4. John Hughes. The correctness of type specialisation. In *European Symposium on Programming (ESOP)*, 2000. To appear. Available online from author's home page.

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 types languages. In *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *LNCS*, pages 129–148, Montréal, Canada, 20 September 2000. Springer-Verlag.

8. The MetaML Home Page, 2000. Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.

9. J. 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 *LNCS*, pages 290–310. Springer-Verlag, September 1991.

10. Torben Mogensen. Inherited limits. In *Partial Evaluation: Practice and Theory*, volume 1706 of *LNCS*, pages 189–202. Springer-Verlag, 1999.

11. 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 *LNCS*, pages 193–207. Springer-Verlag, 1999.

12. 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 *LNCS*, pages 345–359. Springer-Verlag, 1989.

13. Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982.

14. Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, January 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, January 2000.

16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

17. Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.

18. Zhe Yang. Encoding types in ML-like languages. *SIGPLAN Notices*, 34(1):289–300, January 1999.