Runtime Tag Elimination. (Extended Abstract)

Walid Taha*

taha@cs.chalmers.se

Abstract. Over ten years ago, Jones identified the problem of eliminating tags from the result of staged interpreters. Recently, Hughes and Danvy each proposed a different approach to solving this problem. Both approaches involve partial evaluation. To avoid the complexities of reasoning about the correctness of a transformation which involves partial evaluation, we propose *runtime tag elimination*, a novel optimisation simple enough to allow us to *prove* that we do solve the problem posed by Jones, and that we solve the problem in a *semantically well-behaved manner*. After explaining the transformation as a compile-time operation, we show how it can be incorporated into the language as a semantically sound *runtime* operation. Our development is operational, and avoids multi-level languages.

1 Introduction

Typed programming languages provide a *guarantee* to the programmer: If a program is well-typed, we know that certain kinds of run-time errors cannot occur. Providing this guarantee requires a sacrifice in expressivity: Some useful programs, even ones that can never lead to run-time errors, are no longer acceptable programs. Datatypes provide a way for getting around this limitation: They allow the programmer to introduce additional run-time tags into the system, thus, in a sense, relaxing safety. Unfortunately, the use of tags also comes at a cost to runtime performance. In some interesting applications, the runtime overhead of manipulating and checking these tagged values can dominate. In this paper, we present **runtime tag elimination**, a simple and sound runtime transformation aimed at removing tagging and untagging operations from the body of well-typed programs.

The Problem in Well-Typed, Staged Interpreters: Jones [Jon88] identified an important instance of the general problem of superfluous tags in the context of off-line partial evaluation of well-typed interpreters [JGS93]. During off-line partial evaluation, the interpreter is automatically staged. The specialised programs generated by such a staged (or "binding-time annotated") interpreter

* Supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

is where superfluous tags arise. A staged interpreter for a simply-type lambda calculus [JGS93] can be viewed as a total map from terms to what is essentially a higher-order syntax [PE88] encoding. We clarify this analogy with a(n object-)language having the following syntax:

$$o \in \mathbb{O} ::= n \mid \texttt{+} \mid x \mid \lambda x.o \mid o\ o \mid \mathbf{fix}\ x.o$$

assuming $x \in \mathbb{X}$ and $\mathbb{X}$ is some given infinite set of (object-language) variable names. To define the encoding map will need to a translation environment:

$$\rho \in \mathbb{R} ::= [] \mid x \mapsto y; \rho$$

assuming $y \in \mathbb{Y}$ and is some given infinite set of (meta-language) variable names. The encoding function takes a term in this language together with a translation environment and produces a term of the datatype (declared in Haskell notation)

```
data Value = I Natural | F (Value -> Value)
```

The encoding function $\mathcal{E}\colon \mathbb{O} \times \mathbb{R} \to \mathbb{E}$ into some "meta-language" $\mathbb{E}$ (formally introduced in Section 2) is defined as:

$$\begin{aligned}
\mathcal{E}(n)\rho &= \texttt{I}\ n \\
\mathcal{E}(\texttt{+})\rho &= \texttt{F}\ (\lambda(\texttt{I}\ a).\texttt{F}\ (\lambda(\texttt{I}\ b).\texttt{I}\ (+\ a\ b))) \\
\mathcal{E}(x)\rho &= \rho(x) \\
\mathcal{E}(\lambda x.o)\rho &= \texttt{F}\ \lambda y.\mathcal{E}(o)(x \mapsto y; \rho) \\
\mathcal{E}(o_1\ o_2)\rho &= (\lambda(\texttt{F}\ f).\lambda x.f\ x)\ (\mathcal{E}(o_1)\rho)\ (\mathcal{E}(o_2)\rho) \\
\mathcal{E}(\mathbf{fix}\ x.o)\rho &= (\lambda(\texttt{F}\ f).\mathbf{fix}\ x.f\ x)\ (\mathcal{E}(o)\rho)
\end{aligned}$$

Without the datatype `Value` the encoding $\mathcal{E}$ cannot be expressed in a simply typed (or even a Hindley-Milner polymorphic) meta-language $\mathbb{E}$. Expressing this encoding function in a formal meta-language is necessary because partial evaluation computes such encodings *mechanically*. Encoding the term $(\lambda i.i + 1)\ x$ yields

```
(\(F f) -> \v -> f v) (F (\(I i) -> I (i+1))) (I x)
```

The key observation to be made here is that *not all the tags appearing in encoding produced by the staged interpreter are need for well-typedness*. For example, if we know that the only use of the above term is in an application to a tagged value, and the tag is `I`, then we would like to statically remove some of the tag-checks before this term is actually used. Most of the tagging and untagging operations in this term can be removed, resulting in the term:

```
I ((\f -> \v -> f v) (\i -> i+1) x)
```

which has two less tagging and two less untagging operations. The new tag has been added to illustrate that *the typing of the whole term can be kept unchanged*. This possibility is one of two interesting and sound options for **re-integration** (Section 6).

Type Specialization: Partly to address the problem of tags in staged interpreters, Hughes proposed a new paradigm called **type specialization** [Hug98]. The scope of Hughes' type specialization system is much wider than the problem of eliminating tags, and combines forms of term (or "traditional") specialization [JGS93], closure conversion (or "firstification"), constructor specialization [Mog93], dead code elimination, and program point specialization. A number of technical subtleties in the definition of type specialization makes reasoning about its semantics challenging [Hug00]. The results reported here are part of a study into the semantics of Hughes' type specialization. In particular, we show that the sub-problem of tag elimination can in fact be solved in a simple and well-behaved manner that involves neither evaluation or partial evaluation. Rather, it is enough to use a simple and decidable analysis *at runtime*, in addition to two simple type-indexed expansions which can be generated before runtime. Because our development uses more standard notions than have been employed in the past, we are able to suggest new and simple interpretations of some technical questions that arose in the context of Hughes' original formulation of type specialization, such as the notion of a "principal specialization".

Danvy [Dan98] proposed *a simple solution to type specialization* based on the use of **type-directed partial evaluation** [Dan96] and the encoding of projection/embedding pairs in SML [Yan99]. However, the formal characterization of the correctness of this approach, especially in terms ensuring that tags are indeed eliminated, was not addressed. For a language with recursion, such a proof is not obvious, as the semantic foundation of a type-directed partial evaluation relies the existence of $\beta\eta$-normal forms, which is generally not the case in programming languages that allow non-termination and other effects. This paper proposes an elementary approach that *does not* involve partial evaluation (or the need for a "gensym" renaming operator), and has allowed us to formally establish strong correctness properties in the minimal setting of a simply-typed CBV language with recursion. Our development also explicates the importance of the notion *annotated types* that appears in Hughes' work, and that does not appear in Danvy's work. In particular, using these annotated type in our development helped us

1. explicate basic subtleties in tag elimination. For example, type specializations have a *different meanings* on the co- and contra-variant positions in types, and
2. avoid the need to "do induction over recursive types", and instead, we use them to prove correctness by induction over the structure of *all possible unfoldings* of the recursive type, each of which is capture precisely by an annotated type.

Simplifying Assumption We make two non-trivial assumptions in this study. First, we assume that the source code is available at run-time. This assumption is made by the other studies either by mixing type inference and evaluation (as in type specialization) or by requiring the program to execute with a slightly non-standard semantics (as in the need to annotate the source program in TDPE).

Second, we assume that the staged interpreter can be treated as a meta-level operation. This assumption follows from correctness of the staged interpreter. *It is by explicating and exploiting these assumptions we are able to avoid the need for two-level languages, and are able to focus on the essence of the problem of tag-elimination.* We also expect two-level or multi-level languages to provide a good setting for *implementing* this transformation, because multi-level languages typically[1] maintain the source code for "higher-level" terms.

Organization of this Paper: After introducing a small language for purpose of this study (Section 2), we present a simple formal specification of the erasure of tags, and point its interesting features (Section 3). It is surprising that such a specification is not already in the literature, especially that it almost *dictates* the solution we propose. We present an analysis (Section 4) that ensures that erasure "doesn't go wrong". Having established this kind of safety property, we demonstrate that the analysis is non-trivial by showing that it solves the problem posed by Jones (Section 5). Next, we point out a potential danger with the use of such a run-time analysis ("intensionality"), suggest a simple and natural recipe for avoiding this problem in general ("extensionality"), and demonstrate it in our setting (Section 6). Finally, we point out some related works and future works (Sections 7 and 8).

Selected proofs are presented in detail in the Appendix, including the main results. Intermediate lemmas are summarized.

2 A Simply-Typed Language with Recursion

The **core source types** for the (meta-)language we use through this paper are

$$s^0 \in \mathbb{S}^0 ::= \mathsf{nat} \mid s^0 \to s^0 \mid D$$

where nat is the type for natural numbers, $s^0 \to s^0$ is a function type, and D is a name for a particular recursive datatype. The reader can interpret D as the datatype we want to "eliminate". Without any loss of generality, we assume the datatype D has exactly N unique **tags** (or *value constructors*) $\{C_i : s_i^0 \to D \mid 1 \leq i \leq N\}$. While these types are enough for explaining the analysis, *it is not clear how a proof of correctness can be constructed without additional technical machinery.* For this reason (which we expand upon in Section 6), we introduce the **shadow datatype** D' and take **source types** to be

$$s \in \mathbb{S} ::= \mathsf{nat} \mid s \to s \mid D \mid D'.$$

The datatype D' also has N different tags $\{C'_i \mid 1 \leq i \leq N\}$. Further, we require that the types for the tags C'_i be the same as for C_i, but with D replaced by D',

[1] Two- and multi-level languages do not "need" to maintain a high-level representation at run-time, as this is not dictate by their high-level semantics. Keeping the source code, however, is generally viewed as the simplest way of implementing them [Tah99].

$\boxed{\_ \vdash \_ : \_ \subseteq \mathbb{G} \times \mathbb{E} \times \mathbb{S}}$

$$\frac{}{\Gamma \vdash n : \mathsf{nat}} \qquad \frac{}{\Gamma \vdash x : s}\,\Gamma(x) = s \qquad \frac{x : s_1; \Gamma \vdash e : s_2}{\Gamma \vdash \lambda x.e : s_1 \to s_2} \qquad \frac{\Gamma \vdash e_1 : s_1 \to s_2 \quad \Gamma \vdash e_2 : s_1}{\Gamma \vdash e_1\ e_2 : s_2}$$

$$\frac{x : s; \Gamma \vdash e : s}{\Gamma \vdash \mathbf{fix}\ x.e : s} \qquad \frac{\Gamma \vdash e : s_k}{\Gamma \vdash C_k\ e : D} \qquad \frac{\forall i \in L.\ x_i : s_i; \Gamma \vdash e_i : s}{\Gamma \vdash \lambda^{i \in L}(C_i\ x_i).e_i : D \to s}$$

$$\frac{\Gamma \vdash e : s'_k}{\Gamma \vdash C'_k\ e : D'} \qquad \frac{\forall i \in L.\ x_i : s'_i; \Gamma \vdash e_i : s}{\Gamma \vdash \lambda^{i \in L}(C'_i\ x_i).e_i : D' \to s}.$$

Fig. 1. Type System.

and we will write these types as $\{C'_i : s'_i \to D \mid 1 \leq i \leq N\}$. Type environments have the syntax $\Gamma \in \mathbb{G} := [] \mid x : s; \Gamma$ where $[]$ is the empty environment, and $x : s; \Gamma$ is an environment containing the binding of a variable name x to a type term s. We write $\Gamma(x) = s$ when $x : s; \Gamma'$ is a sub-term of Γ.

Because our goal is eliminating the tags of type D, it will be useful to distinguish the notion of a **target type**

$$t \in \mathbb{T} := \mathsf{nat} \mid t \to t \mid D'.$$

Note that $s'_i \in \mathbb{T}$. Expression terms of our language are

$$e \in \mathbb{E} := n \mid x \mid \lambda x.e \mid e\ e \mid \mathbf{fix}\ x.e \mid C\ e \mid \lambda^{i \in L}(C_i\ x_i).e_i \mid C'\ e' \mid \lambda^{i \in L}(C'_i\ x_i).e_i$$
$$\text{where } L \subseteq \{1...N\},$$

where n ranges over natural numbers, x is a variable, $\lambda x.e$ is a lambda abstraction, $e_1\ e_2$ is an application of a term e_1 to a term e_2, $\mathbf{fix}\ x.e$ is a fixed-point construction, $C\ e$ is a formation of an element of the datatype with tag C drawn from the set of names of constructors, and $\lambda^{i \in L}(C_i\ x_i).e_i$ is a data de-constructor term with pattern matching (carrying up to $size(L)$ different cases), and is analogous to the Haskell notation `\(Succ x) -> x`. The type system is presented in Figure 1. The first five rules are standard. Naturals are associated with naturals. Variables are associated with the type term they are associated with in the environment. Applications are associated with a type term s_2 as long as the argument can be associated to a type term s_1 and the operand to a type term $s_1 \to s_2$. Fixed point constructions are associated with a type s as long as their argument is associated with a type term $s \to s$.

The last two rules associate constructions of data with a constructor C_k a type term D as long as the argument is associated with the type term s_k. A de-construction is associated with type $D \to s$ when every "branch" of the de-construction is associated with the type term t when the appropriate assumption about the local variable x_i is made. The rules for shadows are similar.

Lemma 1 (Weakening and Substitution) *The type system is sensible in that*

1. $\Gamma \vdash e : s_1 \wedge x \notin FV(e) \cup dom(\Gamma) \Longrightarrow x : s_2; \Gamma \vdash e : s_1$
2. $\Gamma \vdash e_1 : s_1 \wedge x : s_1; \Gamma \vdash e_2 : s_2 \Longrightarrow \Gamma \vdash e_2[x := e_1] : s_2$.

3 A Tag Erasure Function

Given that we view the tags used in the staged interpreter as being used only to allow static typing, the natural question to ask is "can't we just throw them all away?". Writing the tag erasure $||\_|| : \mathbb{E} \to \mathbb{E}$ function down immediately shows that it is a partial operation[2]:

$$||n|| = n, \quad ||x|| = x, \quad ||\lambda x.e|| = \lambda x.||e||, \quad ||e_1\ e_2|| = ||e_1||\ ||e_2||,$$
$$||\mathbf{fix}\ x.e|| = \mathbf{fix}\ x.||e||, \quad ||C_k\ e|| = ||e||, \quad ||\lambda^{i \in \{k\}}(C_i\ x_i).e_i|| = \lambda x_k.||e_i||,$$
$$||C'_k\ e|| = C'_k\ ||e||, \quad ||\lambda^{i \in L}(C'_i\ x_i).e_i|| = \lambda^{i \in L}(C'_i\ x_i).||e_i||.$$

Erasure does nothing interesting except on the constructs for D, and simply eliminates data construction and pattern matching. Note that this function is not defined on terms where there is more than one case in the pattern being matched. If such a term occurs in the source program, tag erasure simply fails. By simply writing down the definition of erasure (which we don't see anywhere else in the literature) we explicate some of the intrinsic partiality in the operation we which to perform. A basic contribution of this paper is showing that there is a simple, decidable, and useful analysis that tells us when "erasure can't go wrong"[3].

4 A Basic Tag Elimination Analysis

We will characterise **analysable terms** by an analysis judgement defined by induction over the structure of the term. **Annotated types** are defined as

$$a \in \mathbb{A} ::= \mathsf{nat} \mid a \to a \mid C_k\ a \mid D'.$$

The third production should *not* be confused with the traditional notation of applying a type constructor to a type, rather, C_k is a name for the value constructor and is simply **annotating** the (annotated) type a.

The **source** $|\_| : \mathbb{A} \to \mathbb{S}$ and **target** $||\_|| : \mathbb{A} \to \mathbb{T}$ **interpretations** capture the type of the source terms and the type of the erased terms that are the input and

[2] We would have preferred to define erasure explicitly on well-typed terms. That definition, however, is too verbose.

[3] We identify going wrong with partiality, because the analysis will ensure that an analysable term has an erasure, and that the erasure is well-typed. Thus, there is no need to introduce a syntactic term wrong and manipulate it formally. But our treatment of erasure is essentially the same as ensuring type safety for an operational semantics.

output to tag elimination:

$$\begin{array}{llll} |\mathsf{nat}| = \mathsf{nat}, & |a_1 \to a_2| = |a_1| \to |a_2|, & |C_k\ a| = D, & |D'| = D' \\ ||\mathsf{nat}|| = \mathsf{nat}, & ||a_1 \to a_2|| = ||a_1|| \to ||a_2||, & ||C_k\ a|| = ||a||, & ||D'|| = D'. \end{array}$$

The source function suggests that both the tag C and the annotated type term a in the case $C\ a$ are simply additional information that the analysis "should" compute about a term of type D. Note, however, that a specification of this form can, in general, admit more than one possible D.

Given these interpretations, it immediately becomes clear that not all annotated types are meaningful. This observation was not made in previous work by Hughes, and results in some superfluous anomalies in the behaviour of the type specialisation system. For example, if the datatype D has exactly one constructor $C_k = \texttt{I}$, and $t_k = \texttt{Natural}$, then the annotated type $a = \texttt{I String}$ is meaningless because it has a source interpretation $|a| = D$ and a target interpretation $||a|| = \texttt{String}$, and it is not clear how we can convert an expression $e = \texttt{I 5}\colon D$ to an expression of type $||e||\colon \texttt{String}$ in a uniform (or "sensible") way. Thus we define **well-formed annotated types** $\vdash \_ \subseteq \mathbb{A}$ as

$$\frac{}{\vdash \mathsf{nat}} \quad \frac{\vdash a_1 \quad \vdash a_2}{\vdash a_1 \to a_2} \quad \frac{\vdash a \quad |a| = s_k}{\vdash C_k\ a} \quad \frac{}{\vdash D'}.$$

That is, all we require for an annotated type to be well-formed is that the source of an annotated type annotated with C_k must have exactly the same type t_k as that of the argument for the the value constructor C_k.

Annotated type environments are defined as:

$$\frac{}{[] \in \mathbb{L}} \quad \frac{\vdash a \quad |a| = s \quad \Delta \in \mathbb{L}}{x\colon s :> a; \Delta \in \mathbb{L}}.$$

Thus, the empty environment is allowed, but non-empty environments are required to satisfy two conditions: First, the annotated terms must be well-formed according to the rules presented above. Second, it must always be the case that the *source* interpretation of the annotated term must match the type exactly. We write $\Delta(x) = (t :> a)$ when $x\colon t :> a; \Delta'$ is a sub-term of Δ. Both source and target functions extend naturally to annotated type environments. We overload our notation and write $|\_|\colon \mathbb{L} \to \mathbb{G}$ and $||\_||\colon \mathbb{L} \to \mathbb{G}$ for the extensions of the two functions on types to type environments. From now on, **we will omit writing the condition** $\vdash a$ as we will only be concerned with well-formed as.

Figure 2 defines the tag elimination analysis. The first five constructs erase to constructs of the same "shape", thus, the resulting terms should also be type-checked in the exactly the same way as before erasure. The rule for tagging requires that the name of the tag be "registered" in the annotated type. This allows us to both the recover the original type, and to produce an appropriate wrapper in the final result of runtime tag elimination. It should also be noted that the annotated type can carry more information in the a part, depending on what is discovered by the rest of the analysis. In the rules for the shadow

$$\boxed{\_ \vdash \_ : \_ :> \_ \subseteq \mathbb{L} \times \mathbb{E} \times \mathbb{S} \times \mathbb{A}}$$

$$\frac{}{\Delta \vdash n : \mathsf{nat} :> \mathsf{nat}} \qquad \frac{}{\Delta \vdash x : s :> a}\ \Delta(x) = (s :> a)$$

$$\frac{x : s_1 :> a_1; \Delta \vdash e : s_2 :> a_2}{\Delta \vdash \lambda x.e : s_1 \to s_2 :> a_1 \to a_2} \qquad \frac{\Delta \vdash e_1 : s_1 \to s_2 :> a_1 \to a_2 \quad \Delta \vdash e_2 : s_1 :> a_1}{\Delta \vdash e_1\ e_2 : s_2 :> a_2}$$

$$\frac{x : s :> a; \Delta \vdash e : s :> a}{\Delta \vdash \mathbf{fix}\ x.e : s :> a}$$

$$\frac{\Delta \vdash e : s_k :> a}{\Delta \vdash C_k\ e : D :> C_k\ a} \qquad \frac{x_k : s_k :> a_1; \Delta \vdash e_k : s :> a_2}{\Delta \vdash \lambda^{i \in \{k\}}(C_i\ x_i).e_i : D \to s :> C_k\ a_1 \to a_2}$$

$$\frac{\Delta \vdash e : s'_k :> s'_k}{\Delta \vdash C'_k\ e : D' :> D'} \qquad \frac{\forall i \in L.\ x_i : s'_i :> s'_i; \Delta \vdash e_i : s :> a}{\Delta \vdash \lambda^{i \in L}(C'_i\ x_i).e_i : D' \to s :> D' \to a}.$$

Fig. 2. Tag Elimination Analysis.

datatype, we make use of the fact that if $|a| = t$ then $a = t$ to avoid introducing a seemingly "unused" variable a in the antecedents.

The rule for de-constructors considers only the case that which handles exactly one tag. While this is may seem a non-trivial restriction on the analysis, we will see in the next section that, as is, the analysis still has useful applications.

Lemma 2 (Weakening and Substitution) *The analysis is sensible in that*

1. $\Delta \vdash e : s_1 :> a_1 \wedge x \notin FV(e) \cup dom(\Delta) \Longrightarrow x : s_2 :> a_2; \Delta \vdash e : s_1 :> a_1$
2. $\Delta \vdash e_1 : s_1 :> a_1 \wedge x : s_1 :> a_1; \Delta \vdash e_2 : s_2 :> a_2 \Longrightarrow \Delta \vdash e_2[x := e_1] : s_2 :> a_2.$

Furthermore, passing the analysis means that erasing D tags is sensible:

Lemma 3 (Well-Typed Erasure) $\Delta \vdash e : t :> a \Longrightarrow ||\Delta|| \vdash ||e|| : ||a||$

5 Application to Well-Typed Interpreters

All encodings produced by the encoding function presented in the Introduction are well-typed, even for untyped object terms:

Theorem 4 (Encodings of (Untyped) Terms are Well-Typed)

$$\{x_i\} = FV(o) \Longrightarrow +: \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value} \vdash \mathcal{E}(o)(x_i : y_i) : \mathtt{Value}$$

Where $\mathcal{E}$ is the encoding function presented in the introduction.

Tag elimination is *not* possible for the encoding of untyped terms: Consider the untyped object term $\lambda x.x\,x$. In fact, this example demonstrates that it is

$$\boxed{\_ \vdash \_ : \_ \subseteq \mathbb{W} \times \mathbb{O} \times \mathbb{U}}$$

$$\frac{}{\Omega \vdash i : \mathtt{i}} \qquad \frac{}{\Omega \vdash x : u}\ \Omega(x) = u \qquad \frac{x : u_1; \Omega \vdash o : u_2}{\Omega \vdash \lambda x.o : u_1 \to u_2}$$

$$\frac{\Omega \vdash o_1 : u_1 \to u_2 \quad \Omega \vdash o_2 : u_1}{\Omega \vdash o_1\ o_2 : u_2} \qquad \frac{x : u; \Omega \vdash o : u}{\Omega \vdash \mathbf{fix}\ x.o : u} \qquad \frac{}{\Omega \vdash + : \mathtt{i} \to \mathtt{i} \to \mathtt{i}}.$$

Fig. 3. Object Language Type System.

impossible to "optimise" the interpreter so that it only produces tag-free terms (possibly surrounded by a wrapper). Tag elimination is, however, possible for all *well-typed* object terms. To demonstrate this, we introduce a type system for the object language. The types and type environments are

$$u \in \mathbb{U} := \mathtt{i} \mid u \to u \text{ and } \Omega \in \mathbb{W} := [] \mid x : u; \Omega.$$

Figure 3 presents the type system for the object language.

Theorem 5 (Encodings of Well-Typed Terms are Analysable)

$$x_i : u_i \vdash o : u \Longrightarrow \begin{cases} \Delta = + : \mathtt{i} \to \mathtt{i} \to \mathtt{i} :> \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value} :> \mathcal{A}(u_i) \\ \wedge\ \Delta \vdash \mathcal{E}(o)(x_i : y_i) : \mathtt{Value} :> \mathcal{A}(u) \end{cases}$$

where $\mathcal{A}(\mathtt{i}) = \mathtt{I\ i}$ *and* $\mathcal{A}(s_1 \to s_2) = \mathtt{F}\ \mathcal{A}(s_1) \to \mathcal{A}(s_2)$

6 Runtime Tag Elimination is an Extensional Analysis

We will say that a runtime operation suffers **intensionality** if adding it into the language allows us to distinguish any (otherwise) observationally equivalent terms. This is highly undesirable, because it can invalidate some previously valid optimisations (that may also be in use after the new construct is introduce). Inspecting representations of programs at runtime is known in many cases to trivialise observational equivalence to syntactic identity [Mit91,Wan98,Tah00]. Dynamic type systems (see for example [SSP98]) can easily introduce this problem. The approach we proposed here to addressing this subtle problem, that is, establishing the **extensionality** of a runtime operation, is simple: *Show transformation is already "extensionally expressible" in the language.* In this section, we will apply this principle to runtime tag elimination.

Figure 4 presents the definition of the operational semantics for the language with the syntax $\mathbb{E}$ presented earlier on. The set of values is defined as

$$v \in \mathbb{V} := \lambda x.e \mid C\ v \mid \lambda^{i \in L}(C_i\ x_i).e_i \mid C'\ v \mid \lambda^{i \in L}(C'_i\ x_i).e_i \text{ where } L \subseteq \{1...n\}.$$

Lemma 6 (Values) $e \hookrightarrow e' \Longrightarrow e' \in \mathbb{V}$

$\boxed{\_ \hookrightarrow \_ : \mathbb{E} \to \mathbb{E}}$

$$\frac{}{\lambda x.e \hookrightarrow \lambda x.e} \qquad \frac{}{\lambda^{i \in L}(C_i\ x_i).e_i \hookrightarrow \lambda^{i \in L}(C_i\ x_i).e_i} \qquad \frac{e_1 \hookrightarrow e_2}{C_k\ e_1 \hookrightarrow C_k\ e_2}$$

$$\frac{\begin{array}{l} e_1 \hookrightarrow \lambda x.e_3 \\ e_2 \hookrightarrow e_4 \\ e_3[x := e_4] \hookrightarrow e_5 \end{array}}{e_1\ e_2 \hookrightarrow e_5} \qquad \frac{\begin{array}{l} e_1 \hookrightarrow \lambda^{i \in L \cup \{k\}}(C_i\ x_i).e_i \\ e_2 \hookrightarrow C_k\ e_4 \\ e_k[x := e_4] \hookrightarrow e_5 \end{array}}{e_1\ e_2 \hookrightarrow e_5} \qquad \frac{e_1[x := \mathbf{fix}\ x.e_1] \hookrightarrow e_2}{\mathbf{fix}\ x.e_1 \hookrightarrow e_2}$$

Fig. 4. Untyped Operational Semantics (C' rules are the same as C rules).

Theorem 7 *Evaluation preserves typability and analysability:*

1. $\Gamma \vdash e : t \wedge e \hookrightarrow v \Longrightarrow \Gamma \vdash v : t$
2. $\Delta \vdash e : t :> a \wedge e \hookrightarrow v \Longrightarrow \Delta \vdash v : t :> a$

Note that Type Preservation *does not* follow from Analysis Preservation.

Lemma 8 (Simulating Erasure) *If* $\Delta \vdash e : t :> a$ *then*

1. $e \hookrightarrow v \Longrightarrow ||e|| \hookrightarrow ||v||$
2. $||e|| \hookrightarrow v' \Longrightarrow e \hookrightarrow v \wedge v' \equiv ||v||$

We are now ready to explain how the shadow datatype will be used to establish the *correctness* of tag elimination. In particular, it allows us to provide a trivial way of mapping types (and then terms) that use D into terms that don't use D. The presence of the shadow datatype allows us to achieve this in a rather trivial way, which we will write as $\lceil \_ \rceil : \mathbb{S} \to \mathbb{T}$ and define as

$$\lceil \mathsf{nat} \rceil = \mathsf{nat}, \quad \lceil s_1 \to s_2 \rceil = \lceil s_1 \rceil \to \lceil s_2 \rceil, \quad \lceil D \rceil = D', \quad \lceil D' \rceil = D'.$$

We also extend this to type environments as before. Now, we can define the corresponding operation $\lceil \_ \rceil : \mathbb{E} \to \mathbb{E}$ as

$$\lceil n \rceil = n, \quad \lceil x \rceil = x, \quad \lceil \lambda x.e \rceil = \lambda x.\lceil e \rceil, \quad \lceil e_1\ e_2 \rceil = \lceil e_1 \rceil\ \lceil e_2 \rceil,$$
$$\lceil \mathbf{fix}\ x.e \rceil = \mathbf{fix}\ x.\lceil e \rceil, \quad \lceil C_k\ e \rceil = C'_k\ \lceil e \rceil, \quad \lceil \lambda^{i \in L}(C_i\ x_i).e_i \rceil = \lambda^{i \in L}(C'_i\ x_i).\lceil e_i \rceil,$$
$$\lceil C'_k\ e \rceil = C'_k\ \lceil e \rceil, \quad \lceil \lambda^{i \in L}(C'_i\ x_i).e_i \rceil = \lambda^{i \in L}(C'_i\ x_i).\lceil e_i \rceil$$

Lemma 9 (Well-Typed Shadows) $\Gamma \vdash e : s \Longrightarrow \lceil \Gamma \rceil \vdash \lceil e \rceil : \lceil s \rceil$

Lemma 10 (Simulating Shadows) *If* $[] \vdash e : s$ *then*

1. $e \hookrightarrow v \Longrightarrow \lceil e \rceil \hookrightarrow \lceil v \rceil$
2. $\lceil e \rceil \hookrightarrow v' \Longrightarrow e \hookrightarrow v \wedge v' \equiv \lceil v \rceil$

To begin describing the semantic properties of tag elimination, we need a notion of contextual equivalence where termination of the big-step semantics and agreement on based values (naturals in our case) are the only observable, and where context are defined as:

$$c \in \mathbb{C} ::= [] \mid \lambda x.c \mid c\ e \mid e\ c \mid \mathbf{fix}\ x.c \mid C\ c \mid \lambda^{i \in L}(C_i\ x_i).d_i \mid C'\ c \mid \lambda^{i \in L}(C'_i\ x_i).d_i$$
$$\text{where } L \subseteq \{1...N\}, \text{ and } d_k \in \mathbb{C} \text{ for exactly one } k \in L.$$

Developing the theory of this notion from scratch is a non-trivial matter and is beyond the scope of this paper (see for example [Pit95]). Instead, we simply use the following characterisation:

Definition 11 *Let* $\approx \subseteq \mathbb{E} \times \mathbb{E}$ *be any equivalence relation such that*

1. $(\lambda x.e)\ v \approx e[x := v]$
2. $(\lambda^{i \in L \cup \{k\}}(C_i\ x_i).e_i)\ (C_k\ v) \approx e_k[x := v]$
3. $e_1 \approx e_2 \Longrightarrow c[e_1] \approx c[e_2]$,
4. $e_1 \approx e_2 \wedge e_1 \hookrightarrow v_1 \wedge e_2 \hookrightarrow v_2 \Longrightarrow v_1 \approx v_2$,
5. $v_1 \approx v_2 \wedge e_1 \hookrightarrow v_1 \wedge e_2 \hookrightarrow v_2 \Longrightarrow e_1 \approx e_2$, *and*
6. $FV(e_1, e_2) = \{\} \wedge e_1 \not\hookrightarrow \_ \wedge e_2 \not\hookrightarrow \_ \Longrightarrow e_1 \approx e_2$.

Now we can present the key property of the shadowing function, which, in essence, is that it ensures that for every value at a particular (target[4]) type we have:

Lemma 12 (Target Shadows) *If* $[] \vdash e : t$ *then* $[] \vdash \lceil e \rceil : t :> t$, $||\lceil e \rceil|| \equiv \lceil e \rceil$ *and* $\lceil e \rceil \approx e$.

The **wrap** function $W\_ : \mathbb{A} \to \mathbb{E}$ takes an annotated type and produces a term that allows us to "package" the target term of tag elimination as a term that has the same type as the source term. The wrap function is defined simultaneously with the **unwrap** function $U\_ : \mathbb{A} \to \mathbb{E}$. These two functions, together with the identify generator $I\_ : \mathbb{A} \to \mathbb{E}$ are defined as

$$\begin{array}{rrrrr}
W_{\mathsf{nat}} = \lambda x.x, & W_{a_1 \to a_2} = \lambda f.(W_{a_2} \circ f \circ U_{a_1}), & W_{C\ a} = \lambda x.C(W_a\ x), & W_{D'} = \lambda x.x, \\
U_{\mathsf{nat}} = \lambda x.x, & U_{a_1 \to a_2} = \lambda f.(U_{a_2} \circ f \circ W_{a_1}), & U_{C\ a} = \lambda x.U_a\ (C^{-1}x), & U_{D'} = \lambda x.x, \\
I_{\mathsf{nat}} = \lambda x.x, & I_{a_1 \to a_2} = \lambda f.(I_{a_2} \circ f \circ I_{a_1}), & I_{C\ a} = \lambda x.I_a\ x, & I_{D'} = \lambda x.x.
\end{array}$$

where $C^{-1} \equiv \lambda(C\ x).x$. These operators have a number of useful properties:

Lemma 13 (Wrappers) 1. $\vdash a \Longrightarrow [] \vdash W_a : ||a|| \to |a| \wedge [] \vdash U_a : |a| \to ||a||$
2. $\vdash a \Longrightarrow [] \vdash W_a : ||a|| \to |a| :> ||a|| \to a \wedge [] \vdash U_a : |a| \to ||a|| :> a \to ||a||$
3. $\vdash a \Longrightarrow ||W_a|| = ||U_a|| = ||I_a|| = I_a$

There are two ways of achieving extensionality for runtime tag elimination:

[4] Our lemmas state this property for only target types, and we have only proved it for target types, because that's all we need. We expect it to generalise.

Theorem 14 (Main) *If* $[] \vdash e\colon |a| :> a$ *then*

1. $U_a e \approx ||e||$
2. $W_a(U_a e) \approx W_a||e||$

The first equivalence has the advantage of introducing less wrapper/unwrapper tags than the other, but the second leaves the type of the term e the same, and may therefore be considered to introduce less complexity to the type system. To further clarify, using either one of these two equalities, we can now internalise tag elimination into a runtime construct $!_a$ as either

1. $\dfrac{\Gamma \vdash e\colon D}{\Gamma \vdash !_a\ e\colon ||a||}$ and $\dfrac{e \hookrightarrow e' \quad (\text{if } [] \vdash e'\colon |a| :> a \text{ then } ||e'|| \text{ else } U_a e') \hookrightarrow e''}{!_a\ e \hookrightarrow e''}$, or
2. $\dfrac{\Gamma \vdash e\colon D}{\Gamma \vdash !_a\ e\colon D}$ and $\dfrac{e \hookrightarrow e' \quad (\text{if } [] \vdash e'\colon |a| :> a \text{ then } W_a||e'|| \text{ else } W_a(U_a e')) \hookrightarrow e''}{!_a\ e \hookrightarrow e''}$,

respectively. In either case, we provide a sound runtime mechanism for solving the problem posed by Jones, without injuring the operational theory of the (meta-)language.

7 Related Works

Runtime tag elimination is related to many other analysis and optimisations:

- Dynamic typing (in a statically typed language [SSP98,ACPP91] or a dynamically typed language [Hen92a]): Runtime tag elimination is intended to be *semantically transparent* in that
  1. It is developed on top of a language with a standard types system and semantics,
  2. If the analysis fails or succeeds, this cannot be observed by the programmer within the language, and only affects the *behaviour* of the program. This means that performing this runtime transformation does not injure the notion of equivalence for our programming language, thereby providing semantics justification for internalising this meta-level operation into the language.
- "Tagging optimisation": Our notion of a tag is different, in particular, we are concerned with tags introduced by user-defined datatypes, not the "tag per type (or type constructor)" tags treated for example by Henglein [Hen92b], or the machine level problem addressed by Peterson [Pet89].
- Boxing/unboxing [HJ94,PL91]: Our concern is with datatypes that have an arbitrary number of constructors, whereas the boxing/unboxing problem can be viewed (loosely) as an instance of a datatype with one variant. Further, the type of a boxed value is parameterised by the type of the value it carries. This is not the case in our setting (and we conjecture cannot be made the case without moving to type systems richer than Hindley-Milner).

8 Conclusions Future Work

Capitalising on Hughes' notion of annotated types, we have exhibited a simple, novel, and semantically well-behaved runtime transformation that solves the problem of eliminating tags from the result of a staged interpreter.

We have made some strong simplifying assumptions in this study (such as the presence of source code at runtime) explicitly to keep the formal treatment simple (such as avoiding the need to deal with the details of multi-level languages). It is still too early, in our view, to consider lifting these assumptions. In particular, assessing the pragmatic merits of a new analysis can be prohibitively difficult. And even though runtime elimination is less aggressive than type specialisation, it is still a radical technique. Thus, we would like to first ensure that there is a reasonable and promising theory underlying tag elimination before starting to address implementation issues. To this end, we wish to consider

1. Extending the object-language with various features, such as datatypes and effects.
2. Extending of meta-language to polymorphism and investigating the operational details of type inference. Note, however, that we have established that the operational behaviour of the analysis does not depend on the details of the what types are chosen for the sub-terms. There is, therefore, no concern about the coherence of the transformation, and establishing the existence of a notion of principal type will simply mean that there is a notion "best search" that is sound and complete with respect to deciding the typability of a given term.
3. Relating our treatment to the denotational and categorical treatments of datatypes. For example, we did try to use logical relations for the main theorem, but we ended up with a construction that only seems marginally related to logical relations (Theorem 14).

References

[ACPP91] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268., April 1991.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 242–257, Florida, January 1996. ACM Press.

[Dan98] 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, July 1998.

[Hen92a] Fritz Henglein. Dynamic typing: syntax and proof theory. *Lecture Notes in Computer Science*, 582:197–230, 1992.

[Hen92b] Fritz Henglein. Global tagging optimization by type inference. In *1992 ACM Conferenc on Lisp and Functional Programming*, pages 205–215. ACM, ACM, August 1992.

[HJ94] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Portland, Oregon*, pages 213–226, January 1994.

[Hug98] John Hughes. Type specialization. *ACM Computing Surveys*, 30(3es), September 1998.

[Hug00] John Hughes. The correctness of type specialisation. In *European Symposium on Programming (ESOP)*, 2000. To appear. Available online from author's home page.

[JGS93] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jon88] 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.

[Mit91] 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 *Lecture Notes in Computer Science*, pages 290–310. Springer-Verlag, September 1991.

[Mog93] Torben Æ. Mogensen. Constructor specialization. In David Schmidt, editor, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, June 1993.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, June 1988.

[Pet89] J. Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 89–99, New York, NY, 1989. ACM.

[Pit95] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation.* Cambridge University Press, 1995. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, September 1991.

[SSP98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM'00)*, January 2000.

[Wan98] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.

[Yan99] Zhe Yang. Encoding types in ML-like languages. *ACM SIGPLAN Notices*, 34(1):289–300, January 1999.

A Notes and Details on Selected Proofs

Proof (Lemma 1). By induction on e and e_2, respectively. □

Proof (Lemma 2). Same as for type system (Lemma 1). □

Proof (Lemma 3). By a induction over the height of the first derivation. □

Proof (Lemma 4). By a simple induction over the structure of the term o.

- $o = n$. Then $\Gamma \vdash \mathtt{I}\ n : \mathtt{Value}$.
- $o = \mathtt{+}$. Then the term $\mathtt{F}\ (\lambda(\mathtt{I}\ a).\mathtt{F}\ (\lambda(\mathtt{I}\ b).\mathtt{I}\ (+\ a\ b)))$ has type $\mathtt{Value}$ under the given environment.
- $o = x_k \in x_i$. Then $(+ : \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value})(y_k) = \mathtt{Value}$ and we have $(+ : \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value})(y_k) \vdash y_k : \mathtt{Value}$.
- $o = \lambda x.o'$. We have $x_i, x \vdash o'$, and so by induction we also have

  $$+ : \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value}; y : \mathtt{Value} \vdash \mathcal{E}(o')(x_i : y_i; x : y) : \mathtt{Value}$$

  then by the type rule for lambda

  $$+ : \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value} \vdash \lambda y.\mathcal{E}(o')(x_i : y_i; x : y) : \mathtt{Value} \to \mathtt{Value}$$

  then finally by the type rule for constructors we have

  $$+ : \mathtt{i} \to \mathtt{i} \to \mathtt{i}; y_i : \mathtt{Value} \vdash \mathtt{F}\ \lambda y.\mathcal{E}(o')(x_i : y_i; x : y) : \mathtt{Value}.$$

- $o = o_1\ o_2$. By induction we have it that both $\mathcal{E}(o_1)(x_i : y : i)$ and $\mathcal{E}(o_2)(x_i : y_i)$ have type $\mathtt{Value}$ under the given environment. The term $\lambda(\mathtt{F}\ f).\lambda x.f\ x$ has type $\mathtt{Value} \to \mathtt{Value} \to \mathtt{Value}$ under any environment. Thus, the application of the latter term to the former two terms has type $\mathtt{Value}$ under the given environment.
- $o = \mathbf{fix}\ x.o'$. By induction we have it that $\mathcal{E}(o')(x_i : y_i)$ has type $\mathtt{Value}$ under the given environment. The term $\lambda(\mathtt{F}\ f).\mathbf{fix}\ x.f\ x$ has type $\mathtt{Value} \to \mathtt{Value}$ under any environment. Thus, the application of the latter term to the former has type $\mathtt{Value}$ under the given environment.

□

Proof (Lemma 5). By a simple induction over the structure of the term o.

- $o = i$.
  - $x_i : s_i \vdash i : \mathtt{i}$ implies (trivially)
  - $\Delta \vdash i : \mathtt{i} :> \mathtt{i}$ implies (by the analysis judgement)
  - $\Delta \vdash \mathtt{I}\ i : \mathtt{Value} :> \mathtt{I}\ \mathtt{i} = \mathcal{A}(\mathtt{i})$
- $o = \mathtt{+}$.
  - $x_i : s_i \vdash \mathtt{+} : \mathtt{i} \to \mathtt{i} \to \mathtt{i}$. We ignore the environment part, and note that $\mathcal{A}(s) = \mathtt{F}\ (\mathtt{I}\ \mathtt{i} \to \mathtt{F}\ (\mathtt{I}\ \mathtt{i} \to \mathtt{I}\ \mathtt{i}))$. By a lengthy but direct analysis judgement derivation we show

  - $\Delta \vdash \mathtt{F}\ (\lambda(\mathtt{I}\ a).\mathtt{F}\ (\lambda(\mathtt{I}\ b).\mathtt{I}\ (+\ a\ b))) : \mathtt{Value} :> \mathtt{F}\ (\mathtt{I}\ \mathtt{i} \to \mathtt{F}\ (\mathtt{I}\ \mathtt{i} \to \mathtt{I}\ \mathtt{i}))$

- $o = x_k \in x_i$.
  - $x_i : s_i \vdash x_k : s_k$ implies
  - $y_i : \mathtt{Value} :> \mathcal{A}(s_i) \vdash y_k : \mathtt{Value} :> \mathcal{A}(s_k)$.
- $o = \lambda x.o'$.
  - $x_i : s_i \vdash \lambda x.o : s \to s'$ implies by object typing rules
  - $x_i : s_i, x : s \vdash o : s'$ implies by IH
  - $\Delta, y : \mathtt{Value} :> \mathcal{A}(s) \vdash \mathcal{E}(o)(x_i : y_i; x : y) : \mathtt{Value} :> \mathcal{A}(s')$ implies by analysis rules
  - $\Delta \vdash \lambda y.\mathcal{E}(o)(x_i : y_i; x : y) : \mathtt{Value} \to \mathtt{Value} :> \mathcal{A}(s) \to \mathcal{A}(s')$ implies by analysis rules
  - $\Delta \vdash \mathtt{F}\ (\lambda y.\mathcal{E}(o)(x_i : y_i; x : y)) : \mathtt{Value} :> \mathcal{A}(s \to s')$
- $o = o_1\ o_2$.
  - $x_i : s_i \vdash o_1\ o_2 : s$ implies by object typing rules
  - * $x_i : s_i \vdash o_1 : s' \to s$ and
    * $x_i : s_i \vdash o_2 : s'$

    implies by IH
  - * $\Delta \vdash A : \mathtt{Value} :> \mathcal{A}(s' \to s)$ where $A = \mathcal{E}(o_1)(x_i : y_i)$
    * $\Delta \vdash B : \mathtt{Value} :> \mathcal{A}(s')$ where $B = \mathcal{E}(o_2)(x_i : y_i)$

    Independently, it is verbose but simple to show
  - $\Delta \vdash C : \mathtt{Value} \to \mathtt{Value} \to \mathtt{Value} :> \mathtt{F}\ (\mathcal{A}(s') \to \mathcal{A}(s)) \to \mathcal{A}(s') \to \mathcal{A}(s)$ where $C = \lambda(\mathtt{F}\ f).\lambda x.f\ x$. It is then direct to show that
  - $\Delta \vdash C\ A\ B : \mathtt{Value} :> \mathcal{A}(s)$.
- $o = \mathbf{fix}\ x.o'$.
  - $x_i : s_i \vdash \mathbf{fix}\ x.o : s$ implies by object type rules
  - $x_i : s_i \vdash o : s \to s$ implies by IH
  - $\Delta \vdash A : \mathcal{A}(s \to s)$ where $A = \mathcal{E}(o)(x_i : y_i)$.

    Independently, we show by the analysis rules that
  - $\Delta \vdash B : \mathtt{Value} \to \mathtt{Value} :> F(\mathcal{A}(s) \to \mathcal{A}(s)) \to \mathcal{A}(s)$

    where $B = \lambda(\mathtt{F}\ f).\mathbf{fix}\ x.f\ x$. It is then immediate by analysis rules that
  - $\Delta \vdash B\ A : \mathtt{Value} :> \mathcal{A}(s)$.

□

Proof (Lemma 6). By induction on the height of the derivation $e \hookrightarrow e'$. □

Proof (Theorem 7). Both parts are by a simple induction on the height of the derivation of $e \hookrightarrow v$. □

Proof (Lemma 8). First we establish that for analysable terms e_1, e_2 we have $||e_1||[x := ||e_2||] \equiv ||e_1[x := e_2]||$. Then, the first part is proved by induction over the height of the evaluation derivation, and the second part is by induction over the lexicographic order made from the height of the evaluation derivation, and then the size of the term. A lexicographic ordering is needed in the second case because two terms of different size can have the same size after tagging operations have been eliminated by erasure. □

Proof (Lemma 9). By a simple induction over the height of the first derivation. □

Proof (Lemma 10). First we establish that $\lceil e_1[x := e_2] \rceil \equiv \lceil e_1 \rceil [x := \lceil e_2 \rceil]$. Then the proofs for each of the two parts proceed as follows:

1. By induction over the derivation of $e \hookrightarrow v$, and a case analysis on e.
2. By induction over the derivation of $\lceil e \rceil \hookrightarrow v'$, and a cases analysis over e.

□

Proof (Lemma 12). We prove each part separately. The first part is proved by induction over the structure of terms that do not contain D' operations (As is the case for the co-domain of shadows). The second part is trivial (shadows have no D tags). The third part comes from the compatibility of the equivalence, and the shadow simulation lemma. □

Proof (Lemma 13). The first part is by a simple structural induction over the height of the derivation $\vdash a$. The interesting case is when $a \equiv C_k\ a'$. The second is similar. The third part is by a simple induction on the derivation of $\vdash a$. □

Proof (Theorem 14). We only need to prove the first part, and the second part follows directly. By induction over the structure of the annotated type a.

- $a \equiv \mathsf{nat}$. Wrappers and unwrappers are identity, and we get $e \approx ||e||$ from Lemma 8
- $a \equiv a_1 \to a_2$. This is the most interesting case, and is in fact the main reason why it is useful to have the shadow datatype D' in the language. Using the extensionality principle, we will only prove that both sides are equal when applied to every possible value they can be applied to.
  $(U_{a_1 \to a_2} e)\ v$
  $\approx U_{a_2}(e(W_{a_1} v))$ by definition of U, and simplification
  $\approx U_{a_2}(e(W_{a_1} \lceil v \rceil))$ by Lemma 12
  $\approx ||e(W_{a_1} \lceil v \rceil)||$ by IH
  $\equiv ||e||(||W_{a_1}||\ ||\lceil v \rceil||)$ by definition of $||\_||$
  $\equiv ||e||(||W_{a_1}||\ \lceil v \rceil)$ by Lemma 12
  $\equiv ||e||(I_{a_1}\ \lceil v \rceil)$ by Lemma 13 (part 3)
  $\approx ||e||\lceil v \rceil$
  $\approx ||e||v$ by Lemma 12
  and we are done.
- $a \equiv C\ a$. $U_{(C\ a)} e \approx U_a(C^{-1} e)$ by IH $\approx ||C^{-1} e|| \approx ||e||$
- $a \equiv D$. $U_{D'} e \equiv I_{D'} e \approx e \approx \lceil e \rceil \equiv ||\lceil e \rceil|| \approx ||e||$

□