Tag Elimination
– or –
Type Specialisation is a Type-Indexed Effect

Walid Taha*, Henning Makholm**

taha@cs.chalmers.se, henning@makholm.net

Abstract. In response to a challenge posed by Jones (over thirteen years ago), Hughes and Danvy each proposed a different flavour of what has been called "type specialisation". Until recently, however, there were no technical results regarding either proposal. This paper proposes a simple transformation called "Tag Elimination", and proves that it eliminates a large class of superfluous tags from programs. Furthermore, we give the first semantic characterisation of a "type specialising" transformation as a *type-indexed effect*. Our work can be viewed as a unifying synthesis of the previous works by Hughes and Danvy that has enabled new technical results and insights. Our work is novel in that it emphasises that tag elimination can be performed as an independent post-processing phase after traditional partial evaluation.

1 Introduction

Over thirteen years ago, Jones identified the challenge of "optimal specialisation". Hughes and Danvy each proposed a different approach to solving this problem. Both approaches involve using a partial evaluator in an essential way. This may have contributed to the absence of theoretical results for a long time, because operational reasoning about partial evaluators in general, and multi-level languages in particular, can be quite involved. To avoid the complexities of reasoning about the correctness of a transformation which involves (partial) evaluation, we propose *tag elimination*, a novel optimisation simple enough to allow us to *prove* quite easily that an important class of tags are indeed removed (Theorem 6). This first result is in itself rather simple, and could also be achieved for Hughes' original type specialisation system. This result rigorously demonstrates that we have identified a *small subset* of the language studied in the original type specialisation work is enough for tag elimination[1]. A subset very similar to the one we identified was also used by Hughes [Hug00] to arrive at the first interesting theoretical results about type specialisation. What requires more effort to develop and validate is our second result, which is to demonstrate that tag elimination

* Supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403. The TOPPS groups at DIKU, University of Copenhagen has kindly provided financial support for a one week visit to the department during the last week of April.

** DIKU, University of Copenhagen. Supported by the Danish Natural Sciences Research Council (project PLT).

[1] In the language of Hughes, the language we study can be viewed as having only dynamic terms and static sums. Our shadow datatypes seem to be analogous to dynamic sums in Hughes' development. While shadows are not part of the transformation, they do simplify the formal development. We omit establishing a formal correspondence with the system of Hughes for reasons of space.

is *semantically simple and well-behaved* (Theorem 15). No comparable characterisation for Hughes' original formulation is known, and it seems that such a comparable result would be very hard for anything substantially different from the subset that we have identified. In particular, tag elimination (and therefore possibly type specialisation) can be viewed as a rather novel form of type dependency, namely, as an effect determined by a type, or a *type-indexed effect*.

Finally, an important conceptual contribution of our work is that tag elimination is possible for a large and interesting class of terms without the need for performing it as an integral part of a partial evaluation process. Rather, it can be done as a post-processing phase. This was not the case in the work of Hughes or Danvy.

1.1 Datatypes, Tags, Interpreters, and Partial Evaluation

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 of getting around this limitation: they allow the programmer to introduce additional run-time tags into the system, thus, in a sense, *relaxing* safety. For example, using an appropriate datatype we can express an interpreter for an untyped language in a typed language.

Unfortunately, using tags introduced by the use of datatypes comes at a cost to runtime performance. In many important applications such as implementations of domain-specific languages through staged interpreters [SBP99], the runtime overhead of manipulating and checking these tagged values can dominate. In this paper we will demonstrate how, when only one-tag case is used, and under some simple consistency conditions, tag checks are not necessary, and can be simply erased.

1.2 Tags in Well-Typed, Staged Interpreters:

Launchbury [Lau91] identified an important instance of the general problem of superfluous tags in the context of off-line partial evaluation of well-typed interpreters. During off-line partial evaluation, the interpreter is automatically staged. The specialised programs generated by such a staged (or "binding-time annotated") interpreter is where superfluous tags arise. The presence of these tags implies that Jones' "optimality" [JGS93] has not yet been established for typed languages, as the superfluous tags have a non-zero run-time cost. In this paper, we present a simple approach to eliminating a large class of tags (Theorem 6). Based on this result, we conjecture that our technique can be used to achieve optimal specialisation for a well-typed language.

A staged interpreter for a simply-typed lambda calculus can be modelled by a total map from terms to what is essentially a higher-order abstract syntax [PE88] encoding. We clarify this analogy with an object-language having the following syntax:

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

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

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

assuming $y \in \mathbb{Y}$ and $\mathbb{Y}$ 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 a datatype that can be described as follows[2]:

$$\mathsf{data\ Value} = \mathsf{I\ nat} |\ \mathsf{F\ (Value} \to \mathsf{Value)}$$

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

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

This encoding function is an abstract model of what is referred to in the partial evaluation community as a *generating extension* [JGS93]. It can be *implemented* by a two-level or staged interpreter. By focusing only on the output of this function, we show that tag elimination can be used *in conjunction* with a traditional partial evaluator. Furthermore, tag elimination is *independent* of the *implementation details* of the partial evaluator that is being used: The partial evaluator could be online or off-line, could use a two-level (or multi-level) intermediate language, or not. Multi-level languages are certainly interesting and important, but they are an *implementation detail*: They are a *means* to efficient computation, not an *end* in their own right.

The encoding function $\mathcal{E}$ produces terms of type Value. Without the datatype Value a staged or two-level interpreter based on 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

$$(\lambda(\mathsf{F\ f}).\ \lambda \mathsf{v}.\ \mathsf{f\ v})\ (\mathsf{F}\ (\lambda(\mathsf{I\ i}).\ \mathsf{I\ (i{+}1)}))\ (\mathsf{I\ x})$$

The key observation to be made here is that *not all the tags appearing in the encoding produced by the staged interpreter are needed for well-typedness.* For example, if we know that the only use of the abstraction in the term above 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:

$$\mathsf{I}\ ((\lambda \mathsf{f}.\ \lambda \mathsf{v}.\ \lambda\ \mathsf{f\ v})\ (\lambda \mathsf{i}.\ \mathsf{i{+}1})\ \mathsf{x})$$

which has two fewer tagging and two fewer 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 doing tag-elimination **at runtime** (Section 6).

1.3 Type Specialisation:

Partly to address the problem of tags in staged interpreters, Hughes proposed a new paradigm called **type specialisation** [Hug98]. The scope of Hughes' type specialisation system is much wider than the problem of eliminating tags, and combines forms of

[2] Note that the encoding produces *terms* of type Value, and *not values* of type Value.

term (or "traditional") specialisation [JGS93], closure conversion (or "firstification"), constructor specialisation [Mog93], dead code elimination, and program point specialisation. A number of technical subtleties in the definition of type specialisation makes reasoning about its semantics challenging (see the Contributions section below). The results reported here are part of a study into the semantics of Hughes' type specialisation. 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, in addition to two simple type-indexed expansions which can be generated at specialisation time. 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 specialisation, such as the notion of a "principal specialisation".

Danvy [Dan98] proposed what he called **a simple solution to type specialisation** based on the use of **type-directed partial evaluation** [Dan96] and the encoding of projection/embedding pairs in SML [Yan99]. However, the formal characterisation 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 of *annotated types* that appears in Hughes' work[3], and that does not appear in Danvy's work. In particular, using these annotated types in our development helped us

1. explicate basic subtleties in tag elimination. For example, type specialisations have a *different meaning* when they are done on the co- and contra-variant positions in types (see **interpretations**, Section 3), and are therefore unlike subtyping, 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 captured precisely by an annotated type. For example, our results are not formulated to say interesting things about a particular datatype that we want to eliminate (such as Value), but rather, about all possible interesting *specialisations* of that datatype (such as nat and $\mathsf{nat}\to\ \mathsf{nat}$ for which the annotated types are respectively $\mathsf{I\ nat}$ and $\mathsf{F(I\ nat}\to\ \mathsf{I\ nat))}$. Indeed, this is manifest in the statement of our characterisation of the extensional semantics of tag elimination (Theorem 15).

Furthermore, by taking as input to our problem the output of the encoding function, we *avoid* the need for dealing with two-level languages, and are able to focus on the essence of the problem of tag elimination. At the same time, we do expect two-level or multi-level languages to provide a good setting for *implementing* this transformation.

[3] Hughes calls the counterpart of annotated types "residual types". The word "residual" suggest analogy with "residual terms" in traditional partial evaluation, which are terms that have not been reduced away during specialisation. This is an unnatural analogy, because what matters is that they contain types *annotated* with additional information, not types containing pending reductions. A more potentially appropriate candidate for the term "residual" is the result of void erasure in Hughes' system.

1.4 Contribution:

This paper advances our understanding of type specialisation in the following ways:

1. **Determinism:** The original work by Hughes specified type specialisation by a set of inference rules. Because the rules defining Hughes' type specialisation system are not syntax-directed, we can have more than one specialisation derivation. Thus, the rules for type specialisation allow deriving more than one "residual type" for any input. This means that type specialisation can change the termination behaviour of a program in different (unpredictable) ways. While there are promising proposals for addressing this problem by characterising a notion of "principal specialisation" (in analogy to "principal typing"), the problem is still open. An important conceptual contribution of our present work takes a view that the "residual type" is part of the (user-specified) input to what we view as "tag elimination", addressing both of these two caveats at their root.
2. **Abstract semantics specification:** Hughes' early work on type specialisation did not present a simple relation between the source and target types or terms for type specialisation. Thus, there was no known simple characterisation of what the type specialisation is *intended to achieve*. Such a simple specification is necessary for understanding what type specialisation means in a general setting, and how it can be generalised to richer settings. Furthermore, even though new results by Hughes [Hug00] do establish some "one-to-many" (i.e., relational) properties of type specialisation, and pragmatic argument is made that these results are "good enough", these properties still allow type specialisation to make arbitrary "improvement" to the termination of a program, and there is no characterisation of when or how much "improvement" is introduced by type specialisation. This paper presents an attempt to provide a thorough remedy to this caveat, and propose *two orthogonal, functional characterisations*: one *extensional*, and one *intensional*. The first one tells us precisely how the transformation affects termination, and the second tells us that the transformation eliminates an interesting and large class of tags.
3. **Separation from (partial) evaluation and multi-level languages:** The early papers about type specialisation were set in the context of a very rich two-level language with advance partial evaluation features (realised by two constructs called poly and spec). Since the beginning of our work, we have insisted on focusing on subsets of this language. One of our important conceptual contributions in this respect is identifying that a powerful form of "specialising types", namely tag elimination, is *independent* of traditional partial evaluation and the details of the particular partial evaluation technology, such as multi-level languages or polyvariant specialisation. Understanding the result of combining these features into one language is still an interesting but *different* question.

1.5 Organisation of this Paper:

After introducing a small language for the purpose of this study (Section 2), we present a simple formal specification of the erasure of tags, and point out its interesting features (Section 3). Just writing out this definition 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 an important instance of the tag elimination problem identified by Launchbury (Section 5, Theorem 6). Next, we point out a potential danger with the use of such an 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 8 and 9).

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

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*) $\{C_i\colon s_i^0 \to D \mid 1 \leq i \leq N\}$. 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 role of the shadow datatype D' is purely technical, that is, it does not get involved in the tag elimination as such, but rather, is introduced primarily to simplify the technical development (and in particular, Lemma 15). 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', and we will write these types as $\{C_i'\colon s_i' \to D' \mid 1 \leq i \leq N\}$ where $s_i' = s_i^0[D\colon= D']$. Type environments have the syntax

$$\Gamma \in \mathbb{G} := [\,] \mid x\colon s; \Gamma$$

where $[\,]$ is the empty environment, and $x\colon s;\Gamma$ is an environment containing the binding of a variable name x to a type term s. We write $\Gamma(x) = s$ when $x\colon s;\Gamma'$ is a sub-term of Γ.

Notation 1 *For reasons of layout, when there is need to indicate a point-wise correspondence between a set $\{a_1, a_2, ...a_n\}$ and another set $\{b_1, b_2, ...b_n\}$ we will occasionally abbreviate the notation for the two sets down to $\{a_i\}$ and $\{b_i\}$.*

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

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

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

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

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

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

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

Fig. 1. Type System.

matching (carrying up to $size(L)$ different cases), and is analogous to the Haskell notation λ(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 arguments are a variable of type s and an expression of type s.

The last four rules associate constructions of data with a constructor C_k a type term D when 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 s when the appropriate assumption about the local variable x_i is made. The rules for shadows are similar.

Lemma 2 (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

In the introduction we noted that the tags appearing in the result of specialising a well-typed interpreter to a given program are being used primarily to allow static typing of *the interpreter*. A natural question to ask is therefore: "given a particular result of the specialisation process, can't we just throw all the tags away?". Writing the tag erasure $||\_|| : \mathbb{E} \to \mathbb{E}$ function down immediately shows that it is in fact a partial operation[4]:

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

Erasure does nothing interesting except on the constructs for D, where it simply eliminates data construction and pattern matching. The function is partial because it 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 (as defined here) cannot be performed. By simply writing down the definition of erasure (which we don't see anywhere

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

else in the literature) we explicate some of the intrinsic partiality in the operation we wish 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"[5].

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** $|\_| \colon \mathbb{A} \to \mathbb{S}$ and **target** $||\_|| \colon \mathbb{A} \to \mathbb{T}$ **interpretations** capture the type of the source terms and the type of the erased terms that are the input and 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 was not accounted for in the earlier developments of type specialisation, and results in some superfluous anomalies in the behaviour of the type specialisation system. For example, if the datatype D has exactly one constructor $C_k = \mathsf{I}$, and $t_k = \mathsf{nat}$, then the annotated type $a = \mathsf{I}\ \mathsf{String}$ is useful because it has a source interpretation $|a| = D$ and a target interpretation $||a|| = \mathsf{String}$, and it is not clear how we can convert an expression $e = \mathsf{I}\ 5 \colon D$ to an expression of type $||e|| \colon \mathsf{String}$ 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. This question of well-formedness was in fact the starting point of our investigation, and seems to be an essential aspect of investigating "principal specialisations" that was absent from Hughes' original work. Ongoing work by Martínez López and Hughes tries to use Mark Jones' qualified types systems to enforce this notion of well-formedness [MLH00].

Annotated type environments are defined as:

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

[5] 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 safety of an operational semantics.

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

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

$$\frac{x \colon s_1 :> a_1; \Delta \vdash e \colon s_2 :> a_2}{\Delta \vdash \lambda x.e \colon s_1 \to s_2 :> a_1 \to a_2} \qquad \frac{\begin{array}{l}\Delta \vdash e_1 \colon s_1 \to s_2 :> a_1 \to a_2 \\ \Delta \vdash e_2 \colon s_1 :> a_1\end{array}}{\Delta \vdash e_1\ e_2 \colon s_2 :> a_2}$$

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

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

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

Fig. 2. Tag Elimination Analysis.

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 analysis judgement $\Delta \vdash e \colon s :> a$ is read as *"under the environment Δ, we can tag-erase the term E of type S with annotated type A"*. 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 both to recover the original type, and to produce an appropriate wrapper in the final result of tag elimination. It should also be noted that the annotated type can carry more information than D, depending on what is discovered by the rest of the analysis. In the rules for the shadow 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 which handles exactly one tag. While this may seem a non-trivial restriction on the analysis, we will see in the next section that, as is, the analysis nevertheless has useful applications.

Lemma 3 (Weakening and Substitution) *The analysis is sensible in that*

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

Furthermore, passing the analysis means that erasing D tags is sensible in the sense that it is both well-defined and well-typed:

Lemma 4 (Well-Typed Erasure) $\Delta \vdash e \colon t :> a \Longrightarrow ||\Delta|| \vdash ||e|| \colon ||a||$

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

$$\frac{}{\Omega \vdash n \colon \mathsf{i}} \qquad \frac{}{\Omega \vdash x \colon u}\, \Omega(x) = u \qquad \frac{x \colon u_1; \Omega \vdash o \colon u_2}{\Omega \vdash \lambda x.o \colon u_1 \to u_2}$$

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

Fig. 3. Object Language Type System.

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 5 (Encodings of (Untyped) Terms are Well-Typed)

$$\{x_i\} = FV(o) \Longrightarrow + \colon \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}; y_i \colon \mathsf{Value} \vdash \mathcal{E}(o)(x_i \colon y_i) \colon \mathsf{Value}$$

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 *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

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

Figure 3 presents the type system for the object language.

Theorem 6 (Encodings of Well-Typed Terms are Analysable)

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

where $\mathcal{A}(\mathsf{i}) = \mathsf{I}\ \mathsf{nat}$ *and* $\mathcal{A}(u_1 \to u_2) = \mathsf{F}\ (\mathcal{A}(u_1) \to \mathcal{A}(u_2))$

Combining this result with "Well-Typed Erasure", we have demonstrated that our simple analysis is strong enough to allow us to safely remove all tags from the result of specialising an interpreter for a higher-order language.

Note that this result establishes that the simple analysis we presented here achieves tag elimination for *a particular interpreter*, albeit it an interpreter for a non-trivial language. We expect that there are other interesting interpreters for which the same analysis works, and that extensions to the analysis would allow tag elimination for a bigger class of interpreters.

Note also that an important feature of the interpreter that we employ here is that it produces lambda terms with *partial* pattern matches. In fact, they are matches on *exactly one case*. Exploiting this property of interpreters is an integral part of our approach. At this point in time, we speculate that the use of such singleton case statements is pervasive in interpreters where tags are used primarily to "keep the interpreter typable". An in-depth treatment of this facet our development is planned as future work.

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

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

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

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

6 Tag Elimination is an Extensional Analysis

In general, a program transformation does not take place in a vacuum: it is usual applied to a program in the context of a larger software system. It is well-known that it is generally desirable that a transformation is semantics preserving. But sometimes, as is the case for tag elimination, this transformation can alter the types. Then, it is impossible that the transformation be semantics preserving.

What, then, can we do?

This brings us to the second important result in this paper: altering the types does not mean that we have to give up on giving a simple characterisation of the semantics of the transformation that can be used to describe how the transformation affects the interaction of the transformed program with the outside world. We believe that it is highly significant, *from the software engineering point of view*, that a program transformation have such a characterisation, if nothing else, for the sake of modularity. This section, we will show that the extensional effect of tag elimination can be achieved exactly using syntactic analogues of projection/embedding pairs.

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

Theorem 8 *Evaluation preserves typability and analysability:*

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

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

Lemma 9 (Simulating Erasure) *Erasure commutes with evaluation: If* $\Delta \vdash e\colon t :> a$ *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 10 (Well-Typed Shadows) *For every well-typed term, there is a well-typed shadow:* $\Gamma \vdash e : s \Longrightarrow \lceil \Gamma \rceil \vdash \lceil e \rceil : \lceil s \rceil$

Lemma 11 (Simulating Shadows) *Shadowing commutes with evaluation: 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 (There are standard references such as Pitts [Pit95]. More recently, a theory was developed for a superset of the language studies here [PST00]). Instead, we simply use the following characterisation:

Definition 12 *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. $e \hookrightarrow v \Longrightarrow e \approx v$, *and*
5. $(\forall c.\ FV(c[e_1], c[e_2]) = \{\} \wedge c[e_1] \hookrightarrow v \Longleftrightarrow c[e_2] \hookrightarrow v') \Longleftrightarrow e_1 \approx e_2$.

For an extension of this language, it has been shown by Pašalić, Sheard, and Taha [PST00] that the first four properties follow from the last one, which is a standard definition of observational equivalence.

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[6]) type we have:

Lemma 13 (Target Shadows) *If* $[\,] \vdash e : t$ *then* $[\,] \vdash \lceil e \rceil : t :> t$, $||\lceil e \rceil|| \equiv \lceil e \rceil$ *and* $\lceil e \rceil \approx e$.

[6] 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.

Note that the proof of the last part of this lemma is not trivial, because a term which happens to have a target type is not necessarily free of tagging and untagging operations on D. This property is also the key property of shadows that will be needed for being able to prove our main results by induction on the structure of annotated types.

The **wrap** function $W_{\_}\colon \mathbb{A} \to \mathbb{E}$ 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 $U_{\_}\colon \mathbb{A} \to \mathbb{E}$. These two functions, together with the identity generator $I_{\_}\colon \mathbb{A} \to \mathbb{E}$ are defined as

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

where $C^{-1} \equiv \lambda(C\ x).x$. Danvy has used operations similar to the wrapper and unwrapper function, except that his functions are simply indexed by types, where as our functions are indexed by *annotated types*.

The operators above have a number of useful properties:

Lemma 14 (Wrappers) *For all* $\vdash a$ *we have*

1. $[\,] \vdash W_a\colon ||a|| \to |a|$ *and* $[\,] \vdash U_a\colon |a| \to ||a||$
2. $[\,] \vdash W_a\colon ||a|| \to |a| :> ||a|| \to a$ *and* $[\,] \vdash U_a\colon |a| \to ||a|| :> a \to ||a||$
3. $||W_a|| = ||U_a|| = ||I_a|| = I_a$

There are two properties that suggest a way for achieving extensionality for tag elimination:

Theorem 15 (Main) *If* $[\,] \vdash e\colon |a| :> a$ *then*

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

This result is the formal sense in which we view tag elimination as a type-indexed effect: Partiality is introduced by the wrapper or unwrapper function in a manner that is prescribed precisely by the annotated types. Pursuing this result was partly motivated by Danvy's *simple approach to type specialisation* where he proposed the use of an analogy of U_a in combination with type directed partial evaluation (TDPE) to achieve $||e||$. Thus, our result also gives some formal justification to Danvy's approach, even though it is still an open question can in fact perform erasure for a language that is not strongly normalising.

Remark 16 *Note that we could not have included* $W_a||e|| \approx e$ *in the above list of properties, because it does not hold. Take* $e = \lambda x.x$ *and* $a = \mathsf{I}\ \mathsf{nat} \to \mathsf{I}\ \mathsf{nat}$*. Then,* $W_a||e||$ *is essentially* $\lambda(\mathsf{I}\ x).\mathsf{I}\ x$ *which is not the same as* $\lambda x.x$*.*

This result also provide two different systematic ways for re-integrating the result of the transformation in the context of a bigger, well-type software system. The first equivalence in the Main Theorem has the advantage of introducing fewer 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 a type system for tag elimination as a first-class programming language construct (see next section).

7 Eliminating Tags at Runtime

Until now we have presented tag elimination *before* runtime, and as a post-processing phase to a traditional partial evaluator. But what if we want to do tag elimination *at runtime*? Would that have any undesirable effects on the semantics of our programming language? In some sense, we have already given the answer to this question: the results derived in the last section show that the tag elimination transformation, as presented here, is so semantically well-behaved that it is, extensionally, *already* in the CBV lambda calculus. In particular, using either one of the two equalities presented in the last section, we are justified in internalise tag elimination into a runtime[7] construct $!_a$ as either having

1. the type rule $\dfrac{\Gamma \vdash e\colon D}{\Gamma \vdash !_a\ e\colon ||a||}$, and operational semantics

$$\frac{e \hookrightarrow e' \quad (\text{if } [\,] \vdash e'\colon |a| :> a \text{ then } ||e'|| \text{ else } U_a e') \hookrightarrow e''}{!_a\ e \hookrightarrow e''},$$

or

2. the type rule $\dfrac{\Gamma \vdash e\colon D}{\Gamma \vdash !_a\ e\colon D}$ and operational semantics

$$\frac{e \hookrightarrow e' \quad (\text{if } [\,] \vdash e'\colon |a| :> a \text{ then } W_a||e'|| \text{ else } W_a(U_a e')) \hookrightarrow e''}{!_a\ e \hookrightarrow e''}$$

respectively, and where $|a| = D$. In either case, we provide a sound transformation that performs tag elimination, without injuring the operational theory of the meta-language. This is not to say that we are advocating that it is *necessary* to perform tag elimination at runtime to address problems such as optimal specialisation – not at all: it is enough to use tag elimination at partial evaluation time after specialisation. The fact that tag elimination *can* be made into a semantically well-behaved construct is significant for a number of reasons:

1. We see tag elimination (and in turn, idealised type specialisation) as *depending* on a, and not computing a. This is a significant difference between our view and that of Hughes: Hughes advocated the use of type inference to *compute* types. We advocate the use of type inference only for determining whether tags can be safely erased or not. Hughes' – admittedly more ambitious – goal seems to have give rise to many unsolved difficulties in implementing his original type specialisation proposal.
2. To contrast it to all other formulations of type specialisation known today, which probably cannot be treated as semantically well-behaved constructs because the are not known to have an extensional semantics characterisation.
3. Internalising an optimisation as a language construct in a semantically reasonable way (that is, without injuring the notion of observational equivalence) reflects the fact that the optimisation is well-behaved even in a computationally rich environment. This is concern that can be easily overlooked. To clarify, we will say that a runtime operation suffers **intensionality** if adding it into the language allows us

[7] This would feed naturally in the context of an implementation that uses a two- or multi-level language to implement a staged interpreter. While such languages do not "need" to maintain a high-level representation at runtime, as this is not dictated by their high-level semantics. Keeping the source code, however, can be viewed as the simplest way of implementing them [Tah99]. (Thiemann, for example, has explored a number of alternative implementations of the "future-stage" code [Thi99]).

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*. Our Main Theorem has established this for tag elimination. In the normal terminology of programming language semantics, adding a tag elimination construct would therefore be a **conservative extension** to our meta-language.
4. Last, but not least, to demonstrate that there are no fundamental theoretical barriers in the way of integrating tag elimination into a multi-stage language. In particular, our experience ([Tah99,Tah00]) suggests that having a simple extensional semantics is strong evidence that a particular (operational defined) optimisation will be well-behaved in a multi-stage setting.

8 Related Works

Tag elimination is related to many other analyses and optimisations. If nothing else, the absence of references to the following works in the type specialisation literature and their evident relation to tag elimination is clear evidence that our work provides a novel perspective on the problem:

- 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 *performance* 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).
- Refinement types: Each annotated type can be seen as specifying a *refinement* of a source type. While this is somewhat speculative, there maybe some connections with Freeman and Pfenning's notion of a refinement type [FP91].

We intend to further investigate connections with these works.

9 Conclusions and Future Work

Capitalising on Hughes' notion of annotated types and Danvy's attempt to use an operational analogy of projection/embedding pairs to realise tag elimination, we have exhibited a simple, novel, and semantically well-behaved runtime transformation that solves an interesting instance of the problem of eliminating tags from the result of a staged interpreter.

The primary goal of this work has been providing a simple conceptual basis for type specialisation with a thoroughly developed theory. In the future, we wish to consider

1. Implementing the analysis to get some practical validation of the theoretical results presented here, and to develop a quantitative understanding of how much improvement can be achieved in practice by eliminating tags,
2. Extending the object-language with various features, such as polymorphism and effects.
3. Investigating the operational details of type inference in the analysis. Note that (unlike the case of Hughes full type specialisation system) 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 specialisation" 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. If this invariant can be maintained or generalised in a uniform way, we believe that it will benefit aid in characterising the notion of "principal specialisation".
4. 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 15). Having identified that the partiality introduced by type tag elimination is characterised precisely by the annotated type, we believe that an accurate denotational characterisation using Moggi's computational monads [Mog91] is closer than before.

Revision Note The formal development presented here is based on an earlier unpublished manuscript entitled "Runtime Tag Elimination", January 24th, 2000. While visiting DIKU (February 28th to March 3rd, 2000), the first author learnt of independent yet very similar, implementation oriented, ongoing work by the second author. Since then, a collaboration was initiated that lead to improving the present manuscript, and a more extensive report on a unified account of tag elimination as a solution to optimal specialisation is in preparation.

Acknowledgements: We are especially grateful to John Hughes and David Sands for their encouragement and technical suggestions during this work. Jörgen Gustavsson carefully read an early draft and pointed out various technical slips. We would like to thank him for being critical of the initial (and incomplete) proof of the Target Shadows lemma. Eugenio Moggi, Thierry Coquand, Makoto Takayama, and John Launchbury have also given me helpful suggestions about this work. Last, but not least, we are also indebted to Fidel for his interest and his careful comments on all 200 revisions of this paper.

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.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 268–277, Toronto, June 1991.

[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.

[Lau91] John Launchbury. A strongly-typed self-applicable partial evaluator. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 145–164, August 1991.

[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.

[MLH00] Pablo E. Martínez López and John Hughes. Towards principal type specialisation, March 2000. Unpublished manuscript. Available from `http://www-lifia.info.unlp.edu.ar/~fidel/Works/TowardsPTS.dvi`.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 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.

[Ore] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.

[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.

[PST00] Emir Pašalić, Tim Sheard, and Walid Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report CSE-00-007, OGI, March 2000. Available from [Ore].

[SBP99] T. Sheard, Z. Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, October 1999. USEUNIX.

[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.

[Thi99] Peter Thiemann. Higher-order code splicing. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[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 2). By induction on e and e_2, respectively. □

Proof (Lemma 3). Same as for type system (Lemma 2). □

Proof (Lemma 4). By a induction over the height of the first derivation. □

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

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

  $$+ \colon \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}; y_i \colon \mathsf{Value}; y \colon \mathsf{Value} \vdash \mathcal{E}(o')(x_i \colon y_i; x \colon y) \colon \mathsf{Value}$$

  then by the type rule for lambda

  $$+ \colon \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}; y_i \colon \mathsf{Value} \vdash \lambda y.\mathcal{E}(o')(x_i \colon y_i; x \colon y) \colon \mathsf{Value} \to \mathsf{Value}$$

  then finally by the type rule for constructors we have

  $$+ \colon \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}; y_i \colon \mathsf{Value} \vdash \mathsf{F}\ \lambda y.\mathcal{E}(o')(x_i \colon y_i; x \colon y) \colon \mathsf{Value}.$$

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

□

Proof (Theorem 6). By a simple induction over the structure of the term o.

- $o = n$.
  - $x_i\colon u_i \vdash n\colon \mathsf{nat}$ implies (trivially)
  - $\Delta \vdash n\colon \mathsf{nat}| :> \mathsf{nat}$ implies (by the analysis judgement)
  - $\Delta \vdash \mathsf{I}\ n\colon \mathsf{Value} :> \mathsf{I}\ \mathsf{nat} = \mathcal{A}(\mathsf{nat})$
- $o = +$.
  - $x_i\colon u_i \vdash +\colon \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$. We ignore the environment part, and note that $\mathcal{A}(u) = \mathsf{F}\ (\mathsf{I}\ \mathsf{nat} \to \mathsf{F}\ (\mathsf{I}\ \mathsf{nat} \to \mathsf{I}\ \mathsf{nat}))$. By a lengthy but direct analysis judgement derivation we show
  - $\Delta \vdash \mathsf{F}\ (\lambda(\mathsf{I}\ a).\mathsf{F}\ (\lambda(\mathsf{I}\ b).\mathsf{I}\ (+\ a\ b)))\colon \mathsf{Value} :> \mathsf{F}\ (\mathsf{I}\ \mathsf{nat} \to \mathsf{F}\ (\mathsf{I}\ \mathsf{nat} \to \mathsf{I}\ \mathsf{nat}))$
- $o = x_k \in x_i$.
  - $x_i\colon u_i \vdash x_k\colon u_k$ implies
  - $y_i\colon \mathsf{Value} :> \mathcal{A}(u_i) \vdash y_k\colon \mathsf{Value} :> \mathcal{A}(u_k)$.
- $o = \lambda x.o'$.
  - $x_i\colon u_i \vdash \lambda x.o\colon u \to u'$ implies by object typing rules
  - $x_i\colon u_i, x\colon s \vdash o\colon u'$ implies by IH
  - $\Delta, y\colon \mathsf{Value} :> \mathcal{A}(u) \vdash \mathcal{E}(o)(x_i\colon y_i; x\colon y)\colon \mathsf{Value} :> \mathcal{A}(u')$ implies by analysis rules
  - $\Delta \vdash \lambda y.\mathcal{E}(o)(x_i\colon y_i; x\colon y)\colon \mathsf{Value} \to \mathsf{Value} :> \mathcal{A}(u) \to \mathcal{A}(u')$ implies by analysis rules
  - $\Delta \vdash \mathsf{F}\ (\lambda y.\mathcal{E}(o)(x_i\colon y_i; x\colon y))\colon \mathsf{Value} :> \mathcal{A}(u \to u')$
- $o = o_1\ o_2$.
  - $x_i\colon u_i \vdash o_1\ o_2\colon u$ implies by object typing rules
  -
    * $x_i\colon u_i \vdash o_1\colon u' \to u$ and
    * $x_i\colon u_i \vdash o_2\colon u'$

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

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

□

Proof (Lemma 7). By induction on the height of the derivation $e \hookrightarrow e'$. □

Proof (Theorem 8). Both parts are by a simple induction on the height of the derivation of $e \hookrightarrow v$. □

Proof (Lemma 9). First we establish that for analysable terms e_1, e_2 we have $||e_1||[x\colon= ||e_2||] \equiv ||e_1[x\colon= e_2]||$. 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 10). By a simple induction over the height of the first derivation. □

Proof (Lemma 11). 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 13). 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 14). 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 15). 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 9
- $a \equiv a_1 \to a_2$. 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 13
  $\approx ||e(W_{a_1}\lceil v\rceil)||$ by IH
  $\equiv ||e||(||W_{a_1}||\ ||\lceil v\rceil||)$ by definition of $||\_||$
  $\equiv ||e||(||W_{a_1}||\ \lceil v\rceil)$ by Lemma 13
  $\equiv ||e||(I_{a_1}\ \lceil v\rceil)$ by Lemma 14 (part 3)
  $\approx ||e||\lceil v\rceil$
  $\approx ||e||v$ by Lemma 13
  and we are done.
- $a \equiv C\ a$. $U_{(C\ a)}e \approx U_a(C^{-1}e)$ by IH $\approx ||C^{-1}e|| \approx ||e||$
- $a \equiv D$. $U_{D'}e \equiv I_{D'}e \approx e \approx \lceil e\rceil \equiv ||\lceil e\rceil|| \approx ||e||$

□