On the Rôle of Minimal Typing Derivations in Type-driven Program Transformation

Stefan Holdermans
Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven
The Netherlands
stefan@vectorfabrics.com

ABSTRACT

Standard inference algorithms for type systems involving ML-style polymorphism aim at reconstructing most general types for all let-bound identifiers. Using such algorithms to implement modular program optimisations by means of type-driven transformation techniques generally yields sub-optimal results. We demonstrate how this defect can be made up for by using algorithms that target at obtaining so-called minimal typing derivations instead. The resulting approach retains modularity and is applicable to a large class of polyvariant program transformations.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; F.3.3 [**Logics and Mean-**

ings of Programs]: Studies of Program Constructs—*Type structure*

General Terms
Languages, Theory

Keywords
annotated type systems, type-driven program transformation, minimal typing derivations

1. INTRODUCTION
Type-driven program transformation typically proceeds in two logical phases:

1. an *analysis* phase in which the program under transformation is annotated in accordance with a (nonstandard) type system capable of expressing certain properties of interest; and

LDTA '10, March 28-29, 2010, Paphos, Cyprus
© ACM 2010 ISBN: 978-1-4503-0063-6/10/03...$10.00

Jurriaan Hage
Dept. of Information and Computing Sciences
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands
jur@cs.uu.nl

2. a *synthesis* phase in which the annotations from the previous phase are used to drive the actual transformation of the source program into a target program.

Often such a transformation establishes some form of program optimisation.

A manifest advantage of using types in the analysis phase is that a wide range of techniques and idioms from type systems can be adopted in the design and implementation of transformations. Of particular interest is the use of parametric polymorphism—as found in modern functional programming languages like ML [18] and Haskell [20]—to boost the precision of type-based analyses and to yield transformations that naturally support separate compilation. However, when incorporating ML-style polymorphism into the analysis phase of a type-driven transformation, carefulness is in order: although it seems natural to base implementations of such analyses on standard inference algorithms for reconstructing types in the Hindley-Milner discipline, in practice the use of such algorithms easily leads to suboptimal transformations. This paper offers a closer look at the problem:

- We demonstrate where adaptations of standard type-reconstruction algorithms for analysis in optimising type-driven program transformations fall short. In particular, we argue that the incentive of such algorithms to associate each let-bound identifier with the principal type scheme of its definiens is at odds with the objective to deliver transformations that are as good as possible.

- To be able to substantiate this claim, we formalise, in the context of modular program optimisation, the notion of "best" transformations with respect to a given polymorphic type system. Concretely, we require such transformations, foremost, to guarantee full correctness in the presence of separate compilation and, next, to subject isolated compilation units to as agressive as possible intramodular optimisation.

- In the process, we articulate the connection between our notion of best transformations and *minimal typing derivations* [6], which aim at circumventing "unnecessary" polymorphic type assignments.

Throughout the paper we consider, as an example, a simple type-driven transformation for removing dead code from

Evaluation $\boxed{t \Downarrow w}$

$$\frac{}{c \Downarrow c}\ [\textit{e-const}] \qquad \frac{}{\lambda x.\, t_1 \Downarrow \lambda x.\, t_1}\ [\textit{e-abs}]$$

$$\frac{t_1 \Downarrow \lambda x.\, t_{11} \quad [x \mapsto t_2]t_{11} \Downarrow w}{t_1\ t_2 \Downarrow w}\ [\textit{e-app}]$$

$$\frac{[x \mapsto t_1]t_2 \Downarrow w}{\textbf{let}\ x = t_1\ \textbf{in}\ t_2 \Downarrow w}\ [\textit{e-let}]$$

Figure 1: Natural semantics.

programs written in what is essentially an extended version of the call-by-name lambda-calculus. We stress, however, that the concepts under discussion apply to a whole class of type-driven program transformations, including, for instance, parallelisation [12], dethunkification [2], and update avoidance [21]. Indeed, we consider the most important contribution of this paper its depiction of a general type-based methodology for modular optimising program transformations.

2. ELEMENTARY DEAD-CODE ELIMINATION

Let c range over an abstract set of constant symbols and x over a countable infinite set of variable symbols. Then, consider the set of terms given by

$$t \quad ::= \quad c \mid x \mid \lambda x.\, t_1 \mid t_1\ t_2 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \mid \bot.$$

That is, terms are built from constants, variables, lambda-abstractions, function applications, (nonrecursive) local definitions, and the special constant $\bot$ representing a failing computation. As usual, function application associates to the left and lambda-abstractions extend as far to the right as possible.

Terms are evaluated under a call-by-name strategy. Successful evaluation of a closed term t yields a weak-head normal form w, which is either a constant or a lambda-abstraction:

$$w \quad ::= \quad c \mid \lambda x.\, t_1.$$

Formally, t evaluates to w if the judgement $t \Downarrow w$ can be produced from the set of inference rules in Figure 1. Note that, under this nonstrict semantics, the evaluation of the program $(\lambda x.\, \lambda y.\, x)\ t_1\ t_2$ does not require the second argument term t_2 to be reduced to weak-head normal form. It is the goal of *dead-code elimination* to, within a given program, identify as many of such nonrequired terms as possible and subsequently remove them from the program.

In the sequel, we consider a type-driven approach to dead-code elimination for our term language that breaks down into a type-based *liveness analysis* and a translation that replaces dead terms by the special constant $\bot$.

In the analysis phase we make use of types τ, annotated with liveness properties D and L, ranged over by φ. The idea is

Transformation $\boxed{\Gamma \vdash t \rhd t' : \tau^{\varphi}}$

$$\frac{}{\Gamma \vdash c \rhd c : \mathsf{base}^{\varphi}}\ [\textit{t-const}] \qquad \frac{\Gamma(x) = \tau^{\varphi}}{\Gamma \vdash x \rhd x : \tau^{\varphi}}\ [\textit{t-var}]$$

$$\frac{\Gamma[x \mapsto {\tau_1}^{\varphi_1}] \vdash t_1 \rhd t_1' : {\tau_2}^{\varphi_2}}{\Gamma \vdash \lambda x.\, t_1 \rhd \lambda x.\, t_1' : {\tau_1}^{\varphi_1} \xrightarrow{\varphi} {\tau_2}^{\varphi_2}}\ [\textit{t-lam}]$$

$$\frac{\Gamma \vdash t_1 \rhd t_1' : {\tau_2}^{\varphi_2} \xrightarrow{\varphi} \tau^{\varphi} \quad \Gamma \vdash t_2 \rhd t_2' : {\tau_2}^{\varphi_2}}{\Gamma \vdash t_1\ t_2 \rhd t_1'\ t_2' : \tau^{\varphi}}\ [\textit{t-app}]$$

$$\frac{\Gamma \vdash t_1 \rhd t_1' : {\tau_1}^{\varphi_1} \quad \Gamma[x \mapsto {\tau_1}^{\varphi_1}] \vdash t_2 \rhd t_2' : \tau^{\varphi}}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \rhd \mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2' : \tau^{\varphi}}\ [\textit{t-let}]$$

$$\frac{\Gamma \vdash t \rhd t' : \tau^{\mathsf{L}}}{\Gamma \vdash t \rhd t' : \tau^{\mathsf{D}}}\ [\textit{t-sub}] \qquad \frac{\Gamma \vdash t \rhd t' : \tau^{\mathsf{D}}}{\Gamma \vdash t \rhd \bot : \tau^{\mathsf{D}}}\ [\textit{t-elim}]$$

Figure 2: Monovariant dead-code elimination.

to associate the property D with dead code, i.e., code that is guaranteed not to be evaluated, and the property L with live code, i.e., code that may be evaluated. Types are then constructed from a base type base and annotated function types ${\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2}$:

$$\begin{array}{rcl}\varphi & ::= & \mathsf{D} \mid \mathsf{L} \\ \tau & ::= & \mathsf{base} \mid {\tau_1}^{\varphi_1} \rightarrow {\tau_2}^{\varphi_2}.\end{array}$$

Initially, having type environments Γ map from variables x to pairs τ^{φ} consisting of a liveness type τ and an annotation φ, the transformation is expressed through judgements of the form

$$\Gamma \vdash t \triangleright t' : \tau^{\varphi},$$

indicating that, in the type environment Γ, the source term t can be safely transformed into the target term t' as its liveness properties are captured by the type τ and the annotation φ. As a notational convenience, pairs $({\tau_1}^{\varphi_1} \rightarrow {\tau_2}^{\varphi_2})^{\varphi}$, of which the first component denotes a function type, will be written as ${\tau_1}^{\varphi_1} \xrightarrow{\varphi} {\tau_2}^{\varphi_2}$ and the now annotated function-space constructor $\dot{\rightarrow}$ associates to the right.

The rules for deriving transformations are given in Figure 2. The axiom [*t-const*] expresses that constants are considered to be of base type and, depending on the context in which they appear, can be either dead or live. The rule [*t-var*] states that the type and annotation assigned to a variable have to agree with the corresponding entry in the type environment. In the rule [*t-lam*], assumptions are made for the type and annotation for the formal parameter of a lambda-abstraction and the body of the abstraction is analysed and transformed in a type environment that reflects these assumptions; note that the body is analysed independent from the liveness of the abstraction itself. The rule for applications, [*t-app*], requires that the type and annotation for the argument have to match the type and annotation for the formal parameter of the function; moreover, if the application is live (i.e., if its result may be evaluated), then so is the function. In the rule [*t-let*] for transforming local definitions, the type and annotation obtained for the definiens are added to the type environment and the extended type environment is used to analyse and transform the body of the definition. Rule [*t-sub*] introduces *subeffecting*: it allows for variables that are live at their binding sites to be considered dead at some of their use sites, effectively allowing for more subterms to be identified as dead. As far as the transformation from source to target terms is concerned, all of the aforementioned rules simply carry out the identity transformation; hence, crucial to the intended optimisation is the rule [*t-elim*], which states that a dead term may be eliminated and replaced by the special constant $\perp$. Note that we have not included any rule that deals with occurrences of $\perp$ in source terms; such terms are simply considered ill-typed.

Assuming that a given program as a whole may be evaluated (and thus has to receive the annotation L), transformation proceeds by identifying and eliminating as many D-annotated terms as possible.

Example 1. Consider again the program $(\lambda x.\, \lambda y.\, x)\ t_1\ t_2$ and assume that t_1 and t_2 are closed subterms of arbitrary types τ_1 and τ_2, respectively. Then, as the derivation in Figure 3 demonstrates, the second argument term t_2 is in fact dead and can be safely eliminated, yielding the target program $(\lambda x.\, \lambda y.\, x)\ t_1\ \perp$. $\square$

Here, "safely" means that the transformation preserves the semantics of the source program. For instance, in the example above, we have that for any weak-head normal form w with $t_1 \Downarrow w$, both the source and the target program evaluate to w.

3. POLYVARIANT LIVENESS ANALYSIS

Liveness, as determined in the analysis phase of the transformation from the previous section, is not an intrinsic property: whether a term is live or dead depends on the context in which it appears. As the following two examples illustrate, this is a concern especially for higher-order functions.

Example 2. Let t_1 and t_2 be closed terms of base type in the program

$$\textbf{let } \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x) \textbf{ in } \mathit{twice}\ (\lambda y.\, t_1)\ t_2,$$

in which *twice* is applied to a function that never evaluates its argument. Then, *twice* is assigned the liveness type $(\mathsf{base}^{\mathsf{D}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{D}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$ and dead-code elimination results in $\textbf{let } \mathit{twice} = \lambda f.\, \lambda x.\, f\ \bot \textbf{ in } \mathit{twice}\ (\lambda y.\, t_1)\ \bot$. □

Example 3. But in the program

$$\textbf{let } \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x) \textbf{ in } \mathit{twice}\ (\lambda z.\, z)\ t,$$

with t a closed term of base type, *twice* is applied to a function of type $\mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$ and, so, analysis of *twice* yields $(\mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$, leaving no terms to be identified as dead. □

These examples show that what liveness type to assign to a higher-order function depends on the functions to which it is applied. However, in scenarios that require separate compilation, the arguments to which a function is applied are, in general, not known at compile-time. So, if a higher-order function like *twice* in the previous examples is exported by a separately transformed module, its liveness analysis becomes a delicate matter.

Since our aim is to facilitate safe, i.e., semantics-preserving, transformations, a straightforward approach to analysing exported or open-scope functions is to subject them to what Wansbrough [22] calls *pessimisation.* That is, we simply assume that any formal parameters of functional type are to be bound to functions that may use all of their arguments in order to produce a result. For instance, if the function *twice* from Examples 2 and 3 above were to be analysed pessimistically, it would receive the liveness type $(\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ (cf. Example 3). Obviously, this strategy leads to a safe transformation as there can be no harm in binding a live argument to a dead parameter: it will just not be used. The other way around, i.e., binding a dead argument to a live parameter, would, however, be unsafe as dead arguments are to be replaced by $\perp$.

Unfortunately, the effects of pessimisation propagate to the use sites of higher-order functions, causing fewer subterms to be identified as dead:

Example 4. Assume that

$$\Gamma \vdash \mathit{twice} \triangleright \mathit{twice} : (\mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$$

and that t_1 and t_2 are closed subterms of base type. Then, in the program $\mathit{twice}\ (\lambda y.\, t_1)\ t_2$ (cf. Example 2), the second argument term t_2 is to be assumed live and cannot be eliminated during dead-code elimination. □

A better but more involved solution to the problem of dealing with open-scope higher-order functions is to opt for a transformation that is *polyvariant* or *context-sensitive*. In type-driven transformation, this is typically achieved by allowing abstraction over the properties of interest in the analysis phase. The resulting type system makes essential use of polymorphic types, much like those of ML and Haskell, but with the important difference that terms are polymorphic in their annotations rather than their types.

To make our dead-code elimination polyvariant, we extend the annotation language with annotation variables drawn from a countable infinite set ranged over by β. Moreover, the set of concrete annotations is thought of as a two-point join-semilattice with $\mathsf{D} \sqsubset \mathsf{L}$ and least upper bounds $\varphi_1 \sqcup \varphi_2$:

$$\varphi \quad ::= \quad \mathsf{D} \mid \mathsf{L} \mid \beta \mid \varphi_1 \sqcup \varphi_2.$$

The type language is stratified into monomorphic types τ

and possibly polymorphic type schemes σ:

$$
\begin{array}{rcl}
\tau & ::= & \mathtt{base} \mid {\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2} \\
\sigma & ::= & \tau \mid \forall \beta .\, \sigma_1 .
\end{array}
$$

The annotation variable β is bound in $\forall \beta .\, \sigma_1$; we write $\mathit{fav}(\Gamma)$ for the set of annotation variables that appear free

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{[x \mapsto {\tau_1}^{\mathsf{L}}, y \mapsto {\tau_2}^{\mathsf{D}}](x) = {\tau_1}^{\mathsf{L}}}{[x \mapsto {\tau_1}^{\mathsf{L}}, y \mapsto {\tau_2}^{\mathsf{D}}] \vdash x \triangleright x : {\tau_1}^{\mathsf{L}}}
}{[x \mapsto {\tau_1}^{\mathsf{L}}] \vdash \lambda y .\, x \triangleright \lambda y .\, x : {\tau_2}^{\mathsf{D}} \xrightarrow{\mathsf{L}} {\tau_1}^{\mathsf{L}}}
}{[] \vdash \lambda x .\, \lambda y .\, x \triangleright \lambda x .\, \lambda y .\, x : {\tau_1}^{\mathsf{L}} \xrightarrow{\mathsf{L}} {\tau_2}^{\mathsf{D}} \xrightarrow{\mathsf{L}} {\tau_1}^{\mathsf{L}}}
\qquad
\dfrac{\vdots}{[] \vdash t_1 \triangleright t_1 : {\tau_1}^{\mathsf{L}}}
}{[] \vdash (\lambda x .\, \lambda y .\, x)\ t_1 \triangleright (\lambda x .\, \lambda y .\, x)\ t_1 : {\tau_2}^{\mathsf{D}} \xrightarrow{\mathsf{L}} {\tau_1}^{\mathsf{L}}}
\qquad
\dfrac{\dfrac{\vdots}{[] \vdash t_2 \triangleright t_2 : {\tau_2}^{\mathsf{D}}}}{[] \vdash t_2 \triangleright \bot : {\tau_2}^{\mathsf{D}}}
}{[] \vdash (\lambda x .\, \lambda y .\, x)\ t_1\ t_2 \triangleright (\lambda x .\, \lambda y .\, x)\ t_1\ \bot : {\tau_1}^{\mathsf{L}}}
$$

Figure 3: Example derivation.

Transformation $\boxed{\Gamma \vdash t \triangleright t' : \sigma^{\varphi}}$

$$\frac{}{\Gamma \vdash c \triangleright c : \mathsf{base}^{\varphi}}\ [\textit{t-const}] \qquad \frac{\Gamma(x) = \sigma^{\varphi}}{\Gamma \vdash x \triangleright x : \sigma^{\varphi}}\ [\textit{t-var}]$$

$$\frac{\Gamma[x \mapsto {\tau_1}^{\varphi_1}] \vdash t_1 \triangleright t_1' : {\tau_2}^{\varphi_2}}{\Gamma \vdash \lambda x.\, t_1 \triangleright \lambda x.\, t_1' : {\tau_1}^{\varphi_1} \xrightarrow{\varphi} {\tau_2}^{\varphi_2}}\ [\textit{t-lam}]$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' : {\tau_2}^{\varphi_2} \xrightarrow{\varphi} \tau^{\varphi} \quad \Gamma \vdash t_2 \triangleright t_2' : {\tau_2}^{\varphi_2}}{\Gamma \vdash t_1\ t_2 \triangleright t_1'\ t_2' : \tau^{\varphi}}\ [\textit{t-app}]$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' : {\sigma_1}^{\varphi_1} \quad \Gamma[x \mapsto {\sigma_1}^{\varphi_1}] \vdash t_2 \triangleright t_2' : \tau^{\varphi}}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \triangleright \mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2' : \tau^{\varphi}}\ [\textit{t-let}]$$

$$\frac{\Gamma \vdash t \triangleright t' : {\sigma_1}^{\varphi} \quad \beta \notin \mathsf{fav}(\Gamma) \cup \{\varphi\}}{\Gamma \vdash t \triangleright t' : \forall\beta.\, {\sigma_1}^{\varphi}}\ [\textit{t-gen}]$$

$$\frac{\Gamma \vdash t \triangleright t' : \forall\beta.\, {\sigma_1}^{\varphi}}{\Gamma \vdash t \triangleright t' : ([\beta \mapsto \varphi_0]\sigma_1)^{\varphi}}\ [\textit{t-inst}]$$

$$\frac{\Gamma \vdash t \triangleright t' : \sigma^{\varphi \sqcup \varphi_0}}{\Gamma \vdash t \triangleright t' : \sigma^{\varphi}}\ [\textit{t-sub}] \qquad \frac{\Gamma \vdash t \triangleright t' : \sigma^{\mathsf{D}}}{\Gamma \vdash t \triangleright \bot : \sigma^{\mathsf{D}}}\ [\textit{t-elim}]$$

$$\frac{\Gamma \vdash t \triangleright t' : \sigma^{\varphi'} \quad \varphi \equiv \varphi'}{\Gamma \vdash t \triangleright t' : \sigma^{\varphi}}\ [\textit{t-eq}]$$

Figure 4: Polyvariant dead-code elimination.

in Γ and $\forall(\beta_1, \cdots, \beta_n).\, {\tau_1}^{\varphi}$ for the pair consisting of the type scheme $\forall\beta_1.\,(\cdots(\forall\beta_n.\,\tau_1)\cdots)$ and the annotation φ. Transformations are now expressed through judgements of the form

$$\Gamma \vdash t \triangleright t' : \sigma^{\varphi},$$

with type environments Γ mapping from variables x to pairs σ^{φ}.

The rules that constitute the polyvariant transformation are given in Figure 4. Rules [*t-const*], [*t-lam*], and [*t-app*] are identical to their monovariant counterparts, while, in comparison to Figure 2, rules [*t-var*] and [*t-elim*] just make mention of type schemes rather than types. Rule [*t-let*] indicates that let-bound identifiers can have polymorphic types. In rule [*t-sub*], subeffecting is expressed in terms of the least-upper bound operator $\sqcup$. Rule [*t-eq*] expresses that definitional equivalent annotations are interchangeable; here, equivalence, formally defined in Figure 5, simply conveys that annotations are indeed interpreted as elements of a join-semilattice.

Example 5. Using the rules from Figure 4, the function

Annotation Equivalence $\boxed{\varphi \equiv \varphi'}$

$$\frac{}{\varphi \equiv \varphi}\ [\textit{q-refl}] \qquad \frac{\varphi' \equiv \varphi}{\varphi \equiv \varphi'}\ [\textit{q-symm}]$$

$$\frac{\varphi \equiv \varphi'' \quad \varphi'' \equiv \varphi'}{\varphi \equiv \varphi'}\ [\textit{q-trans}]$$

$$\frac{\varphi_1 \equiv \varphi_1' \quad \varphi_2 \equiv \varphi_2'}{\varphi_1 \sqcup \varphi_2 \equiv \varphi_1' \sqcup \varphi_2'}\ [\textit{q-join}] \qquad \frac{}{\mathsf{D} \sqcup \varphi \equiv \varphi}\ [\textit{q-bot}]$$

$$\frac{}{\mathsf{L} \sqcup \varphi \equiv \mathsf{L}}\ [\textit{q-top}]$$

$$\frac{}{\varphi \sqcup \varphi \equiv \varphi}\ [\textit{q-idem}] \qquad \frac{}{\varphi_1 \sqcup \varphi_2 \equiv \varphi_2 \sqcup \varphi_1}\ [\textit{q-comm}]$$

$$\frac{}{\varphi_1 \sqcup (\varphi_2 \sqcup \varphi_3) \equiv (\varphi_1 \sqcup \varphi_2) \sqcup \varphi_3}\ [\textit{q-ass}]$$

Figure 5: Definitional equivalence of annotations.

twice, defined as

$$\lambda f.\, \lambda x.\, f\ (f\ x),$$

can now be assigned the polymorphic liveness type

$$\forall \beta.\, (\mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}},$$

indicating that the liveness of its second argument depends on the liveness properties of its first argument. $\square$

Example 6. Assume that *twice* has the polymorphic liveness type

$$\forall\beta.\, (\mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$$

and that t_1 and t_2 are closed subterms of base type. Then, in the program *twice* $(\lambda y.\, t_1)\ t_2$, the liveness variable β can be instantiated with D. Consequently, the argument t_2 is annotated with $\mathtt{base}^{\mathsf{D}}$ as well, yielding the target program *twice* $(\lambda y.\, t_1)\ \bot$. $\square$

A crucial observation with respect to polymorphically driven transformations is that, although pessimisation is no longer propagated to the use sites of open-scope higher-order functions (cf. Example 4), these functions are themselves still transformed pessimistically. For instance, having associated the polymorpic type $\forall\beta.\, (\mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ with the function $\lambda f.\, \lambda x.\, f\ (f\ x)$, we need to consider all possible instantiations of the liveness variable β. In particular, we need to prepare for β being instantiated with L, meaning that the function bound to f requires its argument in order to produce a result. Hence, to keep the transformation safe, no terms can be eliminated from the definition of the higher-order function.

Now, let us formalise our notion of safety. To this end, we first make precise what it means for two terms to have the same semantics.

DEFINITION 1. *Two weak-head normal forms w_1 and w_2 are* extensionally equal, *written $w_1 \sim w_2$, if*

1. $w_1 = w_2$*, or*
2. *for all terms t_0 and weak-head normal forms w_1', it is implied from $w_1\ t_0 \Downarrow w_1'$ that there exists a weak-head normal form w_2' such that $w_2\ t_0 \Downarrow w_2'$ and $w_1' \sim w_2'$.* □

We say that two terms have the same semantics if they evaluate to extensionally equal normal forms. Safety of the transformation then follows from the following correctness theorem:

THEOREM 2 (SEMANTIC CORRECTNESS). *If $\Gamma \vdash t \rhd t' : \sigma^{\mathsf{L}}$ and $t \Downarrow w$, then there exists a w', such that $t' \Downarrow w'$ and $w \sim w'$.* □

4. MINIMAL TYPING DERIVATIONS

A clear advantage of a type-driven approach to program transformation is that a wide range of techniques and results from type systems can be readily adapted to a transformational setting. An example was given in the previous section, where an adaptation of ML-style polymorphism was used to render type-driven dead-code elimination more context-sensitive. Now, when implementing the resulting transformation, it may seem natural to consider an analogous adaptation of the standard Algorithm W [7] or any other off-the-shelf algorithm for reconstructing types in ML-like languages. However, as it turns out, carefulness is in order as straightforward adaptations of Algorithm W and other standard inference algorithms, in general, result in transformations that are suboptimal in a sense that will be made precise below.

As we will demonstrate shortly, the main defect of standard inference algorithms in a transformational setting is their incentive to associate *all* let-bound identifiers with their *principal types* [11].

For the polymorphic type system from Section 3, principal types may be defined in terms of a partial order on annotated type schemes, presented in Figure 6[1].

THEOREM 3 (PRINCIPAL TYPES). *If* $\Gamma \vdash t \triangleright t' : \sigma^{\mathsf{L}}$, *then there exists a type scheme* $\sigma_\star$ *such that* $\Gamma \vdash t \triangleright t'_\circ : {\sigma_\star}^{\mathsf{L}}$ *for some* $t'_\circ$ *and* $\sigma_\star \leqslant \sigma''$ *for all* σ'' *and* t'' *with* $\Gamma \vdash t \triangleright t'' : {\sigma''}^{\mathsf{L}}$. $\square$

[1]Note that, in addition to giving priority to the more polymorphic of any two equally shaped types, the order in Figure 6 favours base types over function types (rule [*s-bot*]) and, hence, is covariant in *both* type arguments of the function-space constructor (rule [*s-arr*]).

Type-scheme ordering $\boxed{\sigma \leqslant \sigma'}$

$$\frac{}{\sigma \leqslant \sigma}\ [\textit{s-refl}] \qquad \frac{\sigma \leqslant \sigma'' \quad \sigma'' \leqslant \sigma'}{\sigma \leqslant \sigma'}\ [\textit{s-trans}]$$

$$\frac{\tau_1 \leqslant \tau_1' \quad \varphi_1 \equiv \varphi_1' \quad \tau_2 \leqslant \tau_2' \quad \varphi_2 \equiv \varphi_2'}{{\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2} \leqslant {\tau_1'}^{\varphi_1'} \to {\tau_2'}^{\varphi_2'}}\ [\textit{s-arr}]$$

$$\frac{}{\mathsf{base} \leqslant \sigma'}\ [\textit{s-bot}] \qquad \frac{[\beta \mapsto \varphi]\sigma_1 \leqslant \sigma'}{\forall \beta.\, \sigma_1 \leqslant \sigma'}\ [\textit{s-inst}]$$

$$\frac{\sigma \leqslant \sigma_1' \quad \beta \notin \mathsf{fav}(\sigma)}{\sigma \leqslant \forall \beta.\, \sigma_1'}\ [\textit{s-skol}]$$

Figure 6: Partial order on annotated type schemes.

Intuitively, a principal type $\sigma_\star$ of a term t is the most polymorphic type assignable to t, guaranteeing the highest degree of context-sensitivity. However, assigning principal types to all let-bound identifiers, as Algorithm W and the like aim at, typically results in analyses that are "too polyvariant":

Example 7. Consider again the program

$$\textbf{let}\ \mathit{twice} = \lambda f.\,\lambda x.\, f\ (f\ x)\ \textbf{in}\ \mathit{twice}\ (\lambda y.\, t_1)\ t_2$$

from Example 2, in which t_1 and t_2 are closed subterms of base type. Assigning *twice* its principal type

$$\begin{aligned}&\forall(\beta_1, \cdots, \beta_6).\\ &\quad(\mathsf{base}^{\beta_1} \xrightarrow{\beta_1 \sqcup \beta_2 \sqcup \beta_3 \sqcup \beta_4} \mathsf{base}^{\beta_1 \sqcup \beta_2 \sqcup \beta_3}) \xrightarrow{\mathsf{L}}\\ &\quad\mathsf{base}^{\beta_1 \sqcup \beta_5} \xrightarrow{\beta_6} \mathsf{base}^{\beta_2}\end{aligned}$$

results in a conservative transformation that accounts for the possibility that β_1 will be instantiated with L. Consequently, the given program is transformed into $\textbf{let}\ \mathit{twice} = \lambda f.\,\lambda x.\, f\ (f\ x)\ \textbf{in}\ \mathit{twice}\ (\lambda y.\, t_1)\ \bot$, missing out on the opportunity to eliminate the subterm $(f\ x)$ from the definiens of *twice* (cf. Example 2). □

As the example demonstrates, context-sensitivity here comes at the expense of conservativeness. Hence, when assigning a possibly redundant polyvariant liveness type to a locally defined function (or, more general, a function with a closed scope), one risks reducing the opportunities for dead-code elimination unnecessarily.

Note, however, that that is not to say that polyvariance is to be avoided for closed-scope functions altogether. Indeed, polyvariance still plays a valuable rôle in keeping pessimisation from propagating to the use sites of higher-order functions.

Example 8. Let t_0 be a binary operation on terms of base type, and t_1 and t_2 closed subterms of base type. Then, assigning a polyvariant liveness type to the locally defined higher-order function *twice* in the program

$$\begin{array}{l}\textbf{let } \mathit{twice} = \lambda f.\,\lambda x.\,f\ (f\ x)\ \textbf{in}\\ t_0\ (\mathit{twice}\ (\lambda y.\,t_1)\ t_2)\ (\mathit{twice}\ (\lambda z.\,z)\ t_2)\end{array}$$

allows for the elimination of the dead argument term t_2 in the first application of *twice*, yielding

$$\begin{array}{l}\textbf{let } \mathit{twice} = \lambda f.\,\lambda x.\,f\ (f\ x)\ \textbf{in}\\ t_0\ (\mathit{twice}\ (\lambda y.\,t_1)\ \bot)\ (\mathit{twice}\ (\lambda z.\,z)\ t_2).\end{array}$$

If we were to assign a monomorphic type to *twice* instead, the only safe choice would be $(\mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$, which forces the analysis to identify both occurrences of t_2 as live, preventing the elimination of the first occurrence. □

The problem that a single application of a monomorphically typed higher-order function to a live argument forces all arguments to that function to be live, is known as the *poisoning problem* [23] and the example above shows how it is solved by allowing close-scoped functions to have polymorphic types.

In summary, the problem of standard inference algorithms is not so much that they assign polymorphic types to local functions, but rather that they associate local functions with their most polymorphic type. A better approach would be to assign local functions types that are only as polymorphic as needed:

- If a closed-scope higher-order function is only applied to dead arguments, the relevant formal parameter is to receive the monomorphic annotation D, so that the body of the function can be optimised as agressively as possible.
- If a closed-scope higher-order function is only applied to live arguments, its formal parameter could just as well receive the monomorphic annotation L as nothing can be gained by making it context-sensitive.
- If a closed-scope higher-order function may be applied to both dead and live arguments, its formal parameter should be annotated with a polymorphic annotation variable in order to avoid the poisoning problem.

At the same time, to ensure the highest degree of safety and flexibility, exported (i.e., open-scope) functions should be assigned their principal types.

Now, the approach outlined above is suggestive of adapting Bjørner's notion of *minimal typing derivations* [6] to our transformational setting. A typing derivation for a given term and type is minimal if no other typing derivation for the same term and type would avoid type abstractions where the derivation under consideration could not. In our situation, we are interested in derivations for the principal types of separately compiled terms; these derivations then need to be minimal with respect to abstraction over liveness properties.

By definition, the minimality of a typing derivation can not be read from the type that is assigned to a term. Instead, in order to state that a given transformation is not only correct, but also the "best" of all possible transformations of a term, we also need to take into account, as an abstraction of the derivation, the target term that is produced by the transformation. In this target term, unnecessary liveness abstractions that trigger suboptimal transformations, show up as uneliminated terms that could have otherwise been replaced by $\perp$.

Term ordering $\boxed{t \leqslant t'}$

$$\frac{}{t \leqslant t}\,[\textit{u-refl}] \qquad \frac{t \leqslant t'' \quad t'' \leqslant t'}{t \leqslant t'}\,[\textit{u-trans}]$$

$$\frac{t_1 \leqslant t_1'}{\lambda x.\, t_1 \leqslant \lambda x.\, t_1'}\,[\textit{u-abs}] \qquad \frac{t_1 \leqslant t_1' \quad t_2 \leqslant t_2'}{t_1\ t_2 \leqslant t_1'\ t_2'}\,[\textit{u-app}]$$

$$\frac{t_1 \leqslant t_1' \quad t_2 \leqslant t_2'}{\textbf{let}\ x = t_1\ \textbf{in}\ t_2 \leqslant \textbf{let}\ x = t_1'\ \textbf{in}\ t_2'}\,[\textit{u-let}] \qquad \frac{}{\perp \leqslant t'}\,[\textit{u-bot}]$$

Figure 7: Partial order on terms.

To make our notion of best transformations precise, we define a partial order on terms, given in Figure 7. Note that the ordering is congruent and has $\bot$ as its least element. Intuitively, we have that $t \leqslant t'$ if t has more code eliminated than t'.

Now, in addition to the correctness of the transformation (Theorem 2), we have that dead-code elimination yields a target term that is at least as "good" as the corresponding source term:

PROPOSITION 4. *If* $\Gamma \vdash t \rhd t' : \sigma^{\varphi}$, *then* $t' \leqslant t$. □

However, we actually wish for a stronger result: in general, a term admits multiple transformations and we are interested in the best of these transformations. But what constitutes the best transformation? First of all, to support separate transformation, we demand that a transformation is as context-sensitive as possible. Therefore, it seems natural to require that the best transformation for an exported term corresponds to the term's principal type (cf. Theorem 3). But still, there may be many derivations that result in a principal type for a given term. From these derivations, we will favour the one that maximises the number of eliminated subterms. Paramountly, the following theorem guarantees the existence of such derivations:

THEOREM 5 (PRINCIPAL SOLUTIONS). *If* $\Gamma \vdash t \rhd t' : \sigma^{\mathsf{L}}$, *then there exists a type scheme* $\sigma_\star$ *and a term* $t'_\star$ *such that*

1. $\Gamma \vdash t \triangleright t'_\star : {\sigma_\star}^{\mathsf{L}}$,

2. $\sigma_\star \leqslant \sigma''$ *for all* σ'' *and* t'' *for which* $\Gamma \vdash t \triangleright t'' : {\sigma''}^{\mathsf{L}}$, *and*

3. $t'_\star \leqslant t''$ *for all* t'' *for which* $\Gamma \vdash t \triangleright t'' : {\sigma_\star}^{\mathsf{L}}$. $\square$

It then remains to come up with an algorithm that computes such principal solutions. As argued, straightforward adaptions of Algorithm W and the like will not do as these are primarily concerned with computing principal types. Instead, one needs an algorithm that computes minimal derivations for principal types. An example of such an algorithm is an adaptation of the two-pass algorithm of Bjørner [6]. A one-pass algorithm appears in the first author's forthcoming PhD thesis.

5. RELATED WORK

Minimal typing derivations were considered in the context of ML by Bjørner [6] as an alternative to the typing derivations produced by standard inference algorithms such as Algorithm W. Applications that benefit from minimal typing derivations for traditional (i.e., nonannotated) type systems, include unboxing analysis and resolution of overloading. Bjørner gives an algorithm for computing minimal typing derivations, that postprocesses suboptimal derivations produced by conventional algorithms such as Algorithm W.

In this paper, we have focussed on enhancing transformation systems that exploit ML-style let-polymorphism as a means to support separate compilation. Orthogonally, others have made efforts toward increasing the modularity of type-based analyses, most notably by considering type systems that admit, in addition or in lieu of mere principal types, so-called *principal typings* [14] and that allow for genuine compositional analysis. Such systems have been succesfully developed on top of rank-2 intersection types [10, 3, 13, 4] and, at the cost of increased implementation effort, approaches based on the work of Kfoury and others [16, 15] seem to allow for analyses that involve intersection types of arbitrary finite rank. We believe that such systems are amendable to notions of intramodular optimality that are similar to our notion of prinicipal solutions.

Although dead code does not occur often in hand-written code, it does arise frequently as a result from optimising program transformations such as inline expansion and constant propagation (see, for instance, Aho et al. [1]). Also, programs extracted from proofs conducted in logical frameworks typically carry a significant share of dead code [19].

Dead-code elimination is a special instance of useless-code elimination which intends to avoid computations that have no effect on the outcome of a computation, thus reducing execution time. Dead-code elimination only aims to identify expressions that never need to be evaluated and is mainly intended to reduce program size. A related analysis is useless-variable elimination, which intends to discover variables and arguments to functions that are not relevant to the outcome. Damiani and Giannini [9] suggest that an effective approach to useless-code elimination is to first replace unneeded computations by $\perp$ (dead-code elimination) and then apply useless-variable elimination to further optimise the program. Many authors have contributed to the investigation of type-driven useless-variable elimination. Kobayashi [17], for example, defines a type and effect system for useless-variable elimination and presents a reconstruction algorithm that closely follows Algorithm W, enjoying similar properties in terms of ease of implementation and efficiency. Kobayashi is the first to provide examples of the interaction between useless-variable elimination and polymorphism: useless-variable elimination can make functions more polymorphic and polymorphism allows for more useless-variable elimination. For a more comprehensive overview of the field, the reader is referred to Berardi et al. [5] and Daminani [8].

6. CONCLUSION

Within the context of modular type-driven program transformation, we have considered how minimal typing derivations may amplify program optimisation.

In the interest of separate compilation, selecting the principal type for an exported function implies that no assumption is made about the contexts in which the function will be used, and, thus, ensures that full flexibility is maintained. The choice for a minimal typing derivation, on the other hand, ensures that, local to a module, the number of opportunities for optimisation is maximised. Importantly, this is done without endangering safety and without changing the principal types of any exported functions.

We have illustrated our approach by means of an intentionally easy polyvariant transformation for dead-code analysis. We stress, however, once more that our ideas also apply to other, more involved, type-based transformations such as parallelisation, dethunkification, and update avoidance.

7. ACKNOWLEDGEMENTS

This work was supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and carried out while the first author was employed at Utrecht University. The authors wish to express their gratitude to the anonymous reviewers for their helpful and insightful comments.

8. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Boston, Massachusetts, 2nd edition, 2006.

[2] T. Amtoft. Minimal thunkification. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis, Third International Workshop, WSA '93, Padova, Italy, September 22–24, 1993, Proceedings*, volume 724 of *Lecture Notes of Computer Science*, pages 218–229. Springer-Verlag, 1993.

[3] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9–11, 1997*, pages 1–10. ACM Press, 1997.

[4] A. Banerjee and T. P. Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathemathical Structures in Computer Science*, 13(1):87–124, 2003.

[5] S. Berardi, M. Coppo, F. Damiani, and P. Giannini. Type-based useless-code elimination for functional programs. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation, International Workshop SAIG 2000, Montreal, Canada, September 20, 2000, Proceedings*, volume 1924 of *Lecture Notes in Computer Science*, pages 172–189. Springer-Verlag, 2000.

[6] N. Bjørner. Minimal typing derivations. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida (USA), June 25–26, 1994*, pages 120–126, 1994. The proceedings of the workshop have been published as a technical report (2265) at the Institute National Recherche en Informatique et Automatique.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.

[8] F. Damiani. A conjunctive type system for useless-code elimination. *Mathematical Structures in Computer Science*, 13(1):157—197, 2003.

[9] F. Damiani and P. Giannini. Automatic useless-code elimination for HOT functional programs. *Journal of Functional Programming*, 10(6):509–559, 2000.

[10] F. Damiani and F. Prost. Detecting and removing dead-code using rank 2 intersection. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop, TYPES 1996, Aussois, France, December 15–19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 66–87. Springer-Verlag, 1998.

[11] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the Americal Mathematical Society*, 146:29–60, 1969.

[12] G. Hogen, A. Kindler, and R. Loogen. Automatic parallelization of lazy functional programs. In

B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26–28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 1992.

[13] T. P. Jensen. Inference of polymorphic and conditional strictness properties. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 209–221. ACM Press, 1998.

[14] T. Jim. What are principal typings and what are they good for? In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, Papers Presented at Symposium, St. Petersburg Beach, Florida, 21–24 January 1996*, pages 42–53. ACM Press, 1996.

[15] A. J. Kfoury, H. G. Mairson, F. A. Turbak, and J. B. Wells. Relating typabililty and expressiveness in finte-rank intersection type systems (Extended abstract). In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27–29, 1999*, pages 90–101. ACM Press, 1999.

[16] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finte-rank intersection types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 161–174. ACM Press, 1999.

[17] N. Kobayashi. Type-based useless variable elimination. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22–23, 2000*, pages 84–93, 2000.

[18] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[19] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Langauges, Austin, Texas, January 1989*, pages 89–104. ACM Press, 1989.

[20] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

[21] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25–28 June 1995*, pages 1–11. ACM Press, 1995.

[22] K. Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, University of Cambridge, 2002.

[23] K. Wansbrough and S. Peyton Jones. Once upon a polymorphic type. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 15–28. ACM Press, 1999.