Ruy J. G .B. de Queiroz & Walter A. Carnielli, eds., 6^{th} *Workshop on Logic, Language, Information and Computation (WoLLIC'99)*, Rio de Janeiro, Brazil, May 1999, pp. 127–138

Dependence Analysis Through Type Inference

O. Hafızoğulları and C. Kreitz, *Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*
E-mail: {ozan,kreitz}@cs.cornell.edu

Abstract

We introduce a method for dependence analysis in typed functional programming languages. Our approach relies on a type system with simple subtypes for specifying dead code and a type inference algorithm for it. Through a careful seperation of the type system and the problem-specific assumptions we avoid *ad hoc* rules in the type system. This, combined with the fact that our approach makes the flow information in a program explicit and is based on well-understood concepts makes our approach a good candidate for a general framework for program analysis. Our technique can be used in optimizing compilers, optimization of programs extracted from theorem provers, optimization of modular systems, and software engineering.

Keywords: dependence analysis, dead code elimination, program analysis, subtyping, type inference

1 Introduction

Dependence analysis is concerned with the detection of dependencies between different parts of a program. This problem arises naturally in dead code elimination [1], program slicing [24, 19], binding time analysis for program specialization [6], strictness analysis for the optimization of lazy functional languages [22]. In particular, *dead code* in a given program are those parts that do not affect the final result of its execution. Dead code elimination is a symbolic program optimization technique, in which a program is analyzed and parts of it that are irrelevant to the final result of the computation are detected and deleted. Dead code can arise during the compilation process due to program transformations and during the evolution of programs. Program extracts from proofs also tend to contain a lot of code irrelevant to the result of the computation [2, 7, 3, 4, 17, 20]. In this paper, we investigate dependence analysis in the context of dead code elimination, bearing in mind it is readily applicable to the problems mentioned above. We give a method for dead code elimination for typed λ-calculus based languages, such as ML and typed intermediate languages for compilers. The central idea of our approach is to introduce a type system based on simple subtypes [16], which functions as a logic capable of specifying dead code. We then introduce a type inference algorithm for this type system, which in effect, locates dead code. Apart from solving the specific problem of dead code elimination, our method makes the flow information about a program explicit, which makes it a good candidate for a general framework for program analysis. The power and extensibility of the approach arises from its clean conceptual basis, relying on well-understood concepts like type system, typing, type inference, simple subtypes,

[1]The authors are supported by ONR and DARPA under the respective contract numbers N00014-92-J-1764 and F30602-95-1-0047.

which makes it blend naturally with typed functional languages like *ML*. One point to note is that our approach does not transform the program being analyzed, but instead pinpoints its *dead* parts first. This is especially crucial in software engineering applications.

Using a type based approach for program analysis offers several benefits. Type theory is the most natural and widely used methodology to make specifications about higher order functions. Our emphasis on types leads to a conceptually clearer program analysis methodology for higher-order functions in comparison to other methods. This is esp. important, when several different analysis problems must be solved simultaneously, which is the case in compilers and software engineering systems. Esp. in compiler research, there is a current trend to use types throughout the compilation process [21, 18, 5].

Proofs and a more detailed discussion with examples that could not be presented here due to space limitations can be found in our technical report [8].

2 A Simple Type System for Dependence Types

Before presenting our formal approach we briefly discuss the basic ideas. To analyze dependencies, we build a system of *dependence types*, on top of the ordinary type system of the typed λ-calculus. We use this system to express which subexpressions of a given expression are relevant for its value.

In $e = (\lambda x.y)z$, for instance, the subexpression z is irrelevant to the result of the computation. In our system e would be as follows: $\varepsilon = ((\lambda x^D.y^L)^{D\to L} z^D)^L$. Instead of giving a type to the whole expression, we "decorate" its subexpressions with their *dependence types*. L is used to mark possibly "useful" pieces of code and D for "useless" code. Hence, the decoration $(\lambda x^D.y^L)^{D\to L}$ intuitively means that y has a value that can affect the result of the overall computation and that $\lambda x.y$ is a function whose input does not affect the computation, but its result may. Given a "decorated" ε, we can "sieve out" dead code and get $\epsilon = (\lambda x.y)_{\Box_\tau}$, where $\Box_\tau$ stands for a subexpression of (standard) type τ that does not affect the result of the computation. Therefore it does not matter what is inside $\Box_\tau$, which is z in this case.

So, our general dependency analysis proceeds by taking a typed λ-term e, decorating it with dependence types and finally locating dead code according to these types. We will use the notation ε_i for *decorated expressions* and ϵ_i for *sieved expressions*. As usual, an environment Γ will be used to store assumptions about the (dependence) type of variables in an expression, and we will use the notation $\Gamma \triangleright \varepsilon$ for this purpose.

Example 2.1
The expression $e = (\lambda y.((\lambda f.f)(\lambda x.gu))y)((\lambda z.z)w)$ *contains two irrelevant subexpressions* y *and* $(\lambda z.z)w$. *In our type system,* e *would be decorated as follows:* $\varepsilon = ((\lambda y^D.((\lambda f^{D\to L}.f^{D\to L})^{(D\to L)\to(D\to L)}(\lambda x^D.(g^{L\to L}u^L)^L)^{D\to L})^{D\to L} y^D)^L$ $\ldots((\lambda z^D.z^D)^{D\to D}w^D)^D)^L$. *The decoration* y^D *identifies* y *as term whose value does not affect the result of the overall computation.* $(g^{L\to L}u^L)^L$ *expresses that the value of* gu *will be used. Thus the result of* $\lambda x.gu$ *may affect the computation, although its input does not.*

To sieve dead code, all expressions decorated with D *are replaced by a term* $\Box_\tau$, *where* τ *is the original type of such an expression in typed* λ*-calculus. Eliminating the dead code from* ε *results in* $\epsilon = (\lambda y.((\lambda f.f)(\lambda x.gu))\,\Box_t)\,\Box_t$ *where* t *is a type variable.*

$$\frac{}{L \leq D} \quad \text{(CONST)} \qquad \frac{}{\sigma \leq \sigma} \quad \text{(REF)}$$

$$\frac{\sigma_1 \leq \sigma_2 \quad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3} \quad \text{(TRANS)} \qquad \frac{\sigma_3 \leq \sigma_1 \quad \sigma_2 \leq \sigma_4}{\sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_4} \quad \text{(ARROW)}$$

$$\frac{x^{\sigma} \epsilon \Gamma}{\Gamma \rhd x^{\sigma}} \quad \text{(VAR)} \qquad \frac{\Gamma, x^{\sigma_1} \rhd e^{\sigma_2}}{\Gamma \rhd (\lambda x^{\sigma_1}.e^{\sigma_2})^{\sigma_1 \to \sigma_2}} \quad \text{(LAM)}$$

$$\frac{\Gamma \rhd e_1^{\sigma_1 \to \sigma_2} \quad \Gamma \rhd e_2^{\sigma_1}}{\Gamma \rhd (e_1^{\sigma_1 \to \sigma_2} e_2^{\sigma_1})^{\sigma_2}} \quad \text{(APP)} \qquad \frac{\Gamma \rhd e^{\sigma_1} \quad \vdash \sigma_1 \leq \sigma_2}{\Gamma \rhd e^{\sigma_2}} \quad \text{(SUB)}$$

FIG. 1. Type system for the typed λ-calculus with dependence types

In the rest of this section we formally define a system of dependence types which refines the usual simply typed λ-calculus. Several issues that arise in the extension to a full-scale programming language, like *let-polymorphism*, recursion and implementation are given in [8]. The extensions to other programming analysis problems like program slicing, binding time analysis, strictness analysis, automatic debugging will be discussed briefly in Section 5. More details can be found in [8].

DEFINITION 2.2
The *simply typed λ-calculus* is defined by:

(Types) $\tau := t \mid \tau_1 \to \tau_2$

(Expressions) $e := x \mid \lambda x : \tau.e \mid e_1 e_2$

In the following we discard the typing assertion in λ-abstractions if it has no direct bearing on the argument. We also assume that λ-expressions are α-converted, i.e. that there are no name clashes.

DEFINITION 2.3
Dependence types are defined by the grammar $\sigma := L \mid D \mid \sigma_1 \to \sigma_2$.
A dependence type σ *refines* a type τ of the simply typed λ-calculus iff $\tau = t$ and ($\sigma = L$ or $\sigma = D$) or $\tau = \tau_1 \to \tau_2$, $\sigma = \sigma_1 \to \sigma_2$, σ_1 refines τ_1, and σ_2 refines τ_2.

Hence, a dependence type refines the syntactical structure of an ordinary type. It does not distinguish between different type variables but includes information about the usefulness of the expressions under consideration. The type L is used to specify "*relevant to the result of the computation*" and D is for "*irrelevant to the result of the computation*". These types are used in addition to the standard type system of the typed λ-calculus.

The type system given in Figure 1 is an instance of Mitchell's type system with simple subtypes [16]. However, since dependence types contain no variables, there is no need for a constraint environment. Instead of giving only an overall type to a λ-expression e, we *decorate* it by annotating all subexpressions with their corresponding dependence types, because the types of all these subexpressions are utilized for dead code elimination. A *decorated expression* is denoted by e^{σ}.

As usual the type system is based on proof rules for checking that a given decoration σ for an expression e is correct in the context $\Gamma = x_1^{\sigma_1}, ..x_n^{\sigma_n}$ of *typings* for the free variables x_i of e. We denote this statement by $\Gamma \rhd e^{\sigma}$, while $\leq$ denotes the subtype relation between types. There are just two type constants, D and L.

The dependence type σ of a decorated expression e^{σ} determines whether e is dead or alive. A subexpression e may be useful for the overall computation if its dependence

type σ has L as final result type. Only this *tail* of σ is relevant, since the usefulness of a function is determined by the usefulness of its result.

DEFINITION 2.4
The *tail* of a dependence type σ is defined recursively by $tail(L) = L, tail(D) = D, tail(\sigma_1 \to \sigma_2) = tail(\sigma_2)$. A dependence type σ is *useful* if $tail(\sigma) = L$ and *useless* if $tail(\sigma) = D$. We denote this by $useful(\sigma)$ and $useless(\sigma)$.

A subexpression of a given expression e whose dependence type σ is useless does not contribute to the evaluation of e and can safely be eliminated. Formally we replace it by a placeholder $\Box_\tau$ where τ is the original typed λ-calculus type of the subexpression. This indicates that the value of the expression is irrelevant while its type may still be needed for a correct typing. The resulting *sieved expression* ϵ makes all the dead code in e explicit. Sieved expression are evaluated using the ordinary λ-reduction. For proof purposes, we denote this reduction by $\to_c$.

DEFINITION 2.5
Sieved expressions are defined by the grammar $\epsilon = x \mid \Box_\tau \mid \lambda x.e \mid e_1 e_2$.
The reduction $\to_c$ for sieved expressions is defined by $(\lambda x.e_1)e_2 \to_c e_1\{e_2/x\}$.

Sieved expressions are generated from decorated expressions by *sieve* which makes dead code explicit and removes the decorations from the other subexpression.

DEFINITION 2.6
The function *sieve* is defined recursively as follows

$$
\begin{array}{lll}
sieve({e_\tau}^\sigma) = \mathsf{if}\ \mathit{useless}(\sigma)\ \mathsf{then}\ \Box_\tau & & \\
\quad \mathsf{else\ if} & e^\sigma = x^\sigma & \mathsf{then}\ x \\
\quad \mathsf{else\ if} & e^\sigma = (\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3 \to \sigma_4} & \mathsf{then}\ \lambda x.sieve(e_1^{\sigma_4}) \\
\quad \mathsf{else\ if} & e^\sigma = (e_1^{\sigma_1 \to \sigma_2} e_2^{\sigma_1})^{\sigma_3} & \mathsf{then}\ sieve(e_1^{\sigma_1 \to \sigma_3}) sieve(e_2^{\sigma_1})
\end{array}
$$

Here, ${e_\tau}^\sigma$ is an expression e, whose λ-calculus type is τ and whose dependence type is σ.

We show in the next section that $sieve(e^\sigma)$ and e have the same value provided σ is a correct decoration of e. In Section 4 we show how to find such a correct decoration through type inference.

3 Properties of the Type System

In this section, we investigate the properties of the type system. Our aim is to show that if we eliminate *useless* parts of a given piece of code, as specified by the type system, the resulting code reduces to the same normal form as the original code and thus is in fact dead code. In more specific terms, if ε_1 is a "decorated" version of e_1, then under certain assumptions, $sieve(\varepsilon_1)$ and e_1 reduce to the same normal form (Theorem 3.11). First, let us explain what we mean by "normal form".

DEFINITION 3.1
An expression e is in *β-normal form* (*c-normal form*) if it is not possible to reduce any of its subexpressions, w.r.t. $\to_\beta$ ($\to_c$ respectively).

Normalization $\Downarrow$ and $\Downarrow_c$ is defined by

- $e_1 \Downarrow e_n$ if $e_1 \to_\beta \ldots \to_\beta e_n$ and e_n is in β-normal form.
- $e_1 \Downarrow_c e_n$ if $e_1 \to_c \ldots \to_c e_n$ and e_n is in c-normal form.

In the following, we will use *normal form*, if the meaning can be inferred from the context.

Here is a rough discussion of what is to follow. We want to prove Theorem 3.11, which roughly states that if ε_1 is a decoration of e_1 and $sieve(\varepsilon_1) = \epsilon_1$, then $e_1 \Downarrow v$ iff $\epsilon_1 \Downarrow_c v$.

$$\begin{array}{ccccccccc} e_1 & \rightarrow_\beta & e_2 & \rightarrow_\beta & \dots & \dots & \rightarrow_\beta & e_n \\ \downarrow & & \downarrow & & & & & \downarrow \\ \varepsilon_1 & \rightarrow_b & \varepsilon_2 & \rightarrow_b & \dots & \dots & \rightarrow_b & \varepsilon_n \\ \downarrow & \swarrow & \dots & & & & & \swarrow \\ \epsilon_1 & \rightarrow_c & \epsilon_2 & \rightarrow_c & \dots & \rightarrow_c & \epsilon_k & \end{array}$$

The method of the proof can be summarized by the diagram above. The main idea is to investigate the correspondence between the reduction steps e_1 and ϵ_1. For this purpose we define $\rightarrow_b$, a reduction of typed terms. In the figure above, the reduction steps of e_1, and the corresponding steps of ε_1 and ϵ_1 are explicitly given. In Theorem 3.8, we prove that if ε_1 is a *proper* decoration of e_1, $e_1 \rightarrow_\beta e_2$ and $\varepsilon_1 \rightarrow_b \varepsilon_2$, then ε_2 is a *proper* decoration of e_2. By induction, this means that the first and second rows of the figure correspond to each other, as depicted in the diagram.

Similarly, there is a correspondence between the second and third rows, as proven in Theorem 3.9: if $sieve(\varepsilon_1) = \epsilon_1$ and $\varepsilon_1 \rightarrow_b \varepsilon_2$, then either $sieve(\varepsilon_2) = \epsilon_1$, if there is no corresponding step in ϵ_1 (i.e., the reduction in ε_1 occurs inside a $\Box_\tau$ in ϵ_1) or $\epsilon_1 \rightarrow_c \epsilon_2$ and $sieve(\varepsilon_2) = \epsilon_2$. Finally, Theorem 3.6 states that under certain conditions, an expression in normal form has no dead code, i.e. $e_n = \epsilon_k$. To this aim, we first introduce the concept of *IO-correctness*, which we will discuss now.

Consider the typing $f^{D\rightarrow L} \triangleright (f^{D\rightarrow L} 3^D)^L$. Although, the typing is a correct one according to the rules in Figure 1, it assumes that the input to f (i.e. 3) is useless. This is not a conservative statement, because it is not possible to instantiate f with $\lambda x.x$ in any way that respects the typing rules. Intuitively, although we assume in the above typing that the input to f is irrelevant to the result of the computation, whereas its output is, the input of $\lambda x.x$ is relevant to its output. Hence, f cannot be instantiated by $\lambda x.x$. As a result, we need to restrict the set of allowable typings to prohibit this kind of non-conservative approximation of the behavior of the expression in question. We need to assert that for each f^σ in the typing environment, if the output of f is useful, then so should be its input. Assume $\sigma = \sigma_{\text{in}} \rightarrow \sigma_{\text{out}}$. We can formalize what we require of σ as $tail(\sigma_{\text{in}}) \leq tail(\sigma_{\text{out}})$; that is, if σ_{out} is useful (i.e. $tail(\sigma_{\text{out}}) = L$), then so is σ_{in} (i.e. $tail(\sigma_{\text{in}}) \leq tail(\sigma_{\text{out}})$ implies $tail(\sigma_{\text{in}}) = L$). This is formalized in the concept of *top-level IO-relatedness*.

DEFINITION 3.2
A type $\sigma_1 \rightarrow \sigma_2$ is *top-level IO-related* iff $tail(\sigma_1) \leq tail(\sigma_2)$.

However, this definition is not enough to take care of higher-order types in their full generality. Consider $f^{L\rightarrow(D\rightarrow L)} \triangleright (f^{L\rightarrow(D\rightarrow L)} 3^L)^{D\rightarrow L}$. It is not possible to instantiate f with $\lambda x.(\lambda y.y + x)$. The type of f requires that if a live input is given, it will return a function, which will return a live output, if its input is dead. It is not possible that function is $\lambda y.y + x$. Hence, if $\sigma = \sigma_1 \rightarrow (\sigma_2 \rightarrow \sigma_3)$, then in a way similar to the above discussion, we need to assert that $tail(\sigma_1) \leq tail(\sigma_2 \rightarrow \sigma_3)$ and $tail(\sigma_2) \leq tail(\sigma_3)$. This can be generalized using the concept of *positively IO-relatedness*, given below. But first we need to define *positive and negative occurrences*.

DEFINITION 3.3

A subexpression σ' of type σ is at a *positive occurrence*, if $\sigma = \sigma'$
or $\sigma = \sigma_1 \rightarrow \sigma_2$ and either σ' is at a positive occurrence in σ_2 or σ' is at a negative

occurrence in σ_1. A subexpression of σ' of type σ is at a *negative occurrence*, if $\sigma = \sigma_1 \to \sigma_2$ and σ' is at a positive occurrence in σ_1.

DEFINITION 3.4
A type σ is *positively IO-related* iff all subexpressions of σ at positive occurrences are top-level IO-related. Similarly, a type σ is *negatively IO-related* iff all subexpressions of σ at negative occurrences are top-level IO-related.

Now, we can define *IO-correctness*. *Positive IO-relatedness* already puts the necessary restrictions on σ, s.t. for some f, $f^\sigma \in \Gamma$. A similar condition needs to be put on the type of the expression whose dead code we want to eliminate. Consider, for instance, $\emptyset \rhd (\lambda f^{D\to L}.(f^{D\to L}3^L)^L)^{(D\to L)\to L}$. If this were a legitimate typing, then there is no possible typing to apply it to $\lambda x.x$. Thus we also need to constrain σ in $\Gamma \rhd e^\sigma$ to be negatively IO-related.

DEFINITION 3.5
A typing assertion $\Gamma \rhd e^\sigma$ is *IO-correct* iff for all $f^{\sigma_f} \epsilon dom(\Gamma)$, σ_f are positively IO-related, σ is negatively IO-related and $tail(\sigma) \leq L$

In the above definition, the first assertion makes sure that the assumptions about the variables in the environment are conservative, i.e. they can be instantiated by any expression with the appropriate (standard) type. Similarly, the second guarantees that the assumptions about the inputs of e are conservative. The last assertion makes sure that e as a whole is always considered useful. We would like to emphasize that IO-correctness for $\Gamma \rhd e^\sigma$ is concerned with only Γ and σ.

THEOREM 3.6
If $\vdash \Gamma \rhd e^\sigma$ is IO-correct and e is in normal form, then e contains no dead code, i.e. no subexpression ${e_1}^{\sigma_1}$ such that $useless(\sigma_1)$.

In order to prove our main result we must be able to establish a correspondence between raw and decorated λ-terms during a series of reduction steps. We define a reduction rule for decorated λ-terms which, in addition to the ordinary β-reduction, takes care of the relations between the types.

DEFINITION 3.7
Substitution $e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\}$ on decorated expressions is recursively defined by the following rules:

$$
\begin{array}{ll}
x^{\sigma_1}\{|e^{\sigma_2}/x|\} & = e^{\sigma_1} \\
y^{\sigma_1}\{|e^{\sigma_2}/x|\} & = y^{\sigma_1} \\
(\lambda y^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\to\sigma_4}\{|e^{\sigma_5}/x|\} & = (\lambda y^{\sigma_1}.(e_1^{\sigma_2}\{|e^{\sigma_5}/x|\}))^{\sigma_3\to\sigma_4} \\
(e_1^{\sigma_1\to\sigma_2}e_2^{\sigma_3})^{\sigma_4}\{|e^{\sigma_5}/x|\} & = ((e_1^{\sigma_1\to\sigma_2}\{|e^{\sigma_5}/x|\})(e_2^{\sigma_3}\{|e^{\sigma_5}/x|\}))^{\sigma_4}
\end{array}
$$

The *reduction* $\to_b$ *for decorated expressions* is defined by $((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\to\sigma_4}e_2^{\sigma_3})^{\sigma_5} \to_b (e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\})$

The following theorem asserts that the correspondence between raw and decorated λ-terms are kept intact after a reduction step. This can be regarded as a kind of subject reduction property.

THEOREM 3.8
Assume ε_1 is a decoration of e_1, $\vdash \Gamma \rhd \varepsilon_1$ is provable, $e_1 \to_\beta e_2$, and $\varepsilon_1 \to_b \varepsilon_2$, where the reduction steps correspond to each other. Then ε_2 is a decoration of e_2 and $\vdash \Gamma \rhd \varepsilon_2$ is provable.

The correspondence between decorated and sieved expressions is kept intact after a reduction:

THEOREM 3.9
If $\vdash \Gamma \triangleright \varepsilon_1$, $\varepsilon_1 \rightarrow_b \varepsilon_2$ and $sieve(\varepsilon_1) = \epsilon_1$, then either

1. $\epsilon_1 \rightarrow_c \epsilon_2$ and $sieve(\varepsilon_2) = \epsilon_2$, if there is a corresponding reduction step for ϵ_1, or
2. $sieve(\varepsilon_2) = \epsilon_1$, if no corresponding reduction step exists for ϵ_1.

IO-correctness remains invariant under reduction by "$\rightarrow_b$":

THEOREM 3.10
If $\Gamma \triangleright \varepsilon_1$ is IO-correct and $\varepsilon_1 \rightarrow_b \varepsilon_2$, then $\Gamma \triangleright \varepsilon_2$ is IO-correct.

We now come to our main result: under certain conditions, a given expression and its sieved out counter-part evaluate to the same value. Hence, subterms marked as useless are indeed dead code.

THEOREM 3.11
If $\vdash \Gamma \triangleright \varepsilon_1$ is an IO-correct decoration of e_1, $e_1 \Downarrow v_1$, $sieve(\varepsilon_1) = \epsilon_1$ and $\epsilon_1 \Downarrow_c v_2$, then $v_1 = v_2$.

4 Type Inference

In this section we introduce a type inference algorithm for the type system. Since the type system can be viewed as a simple logic for specifying dead code, type inference is an automatic method to discover dead code. An important point to note is that the flow information is made explicit by the use of variable dependence types, constraints and the function BC(), all to be defined below. These concepts make it possible to use our method for other program analysis problems as well.

EXAMPLE 4.1
Let $\Gamma \triangleright e = [y : t_1, z : t_2] \triangleright \lambda x.yz : t_2$. *First, we decorate it with* dependence type variables *(defined below):* $\Gamma \triangleright e^{\rho} = [y^{\alpha_1}, z^{\alpha_2}] \triangleright ((\lambda x^{\alpha_3}.y^{\alpha_4})^{\alpha_5 \rightarrow \alpha_6} z^{\alpha_7})^{\alpha_8}$. *Then, we get the relevant constraints between these variables:* $C = \{\alpha_1 \leq \alpha_4, \alpha_2 \leq \alpha_7, \alpha_3 \rightarrow \alpha_4 \leq \alpha_5 \rightarrow \alpha_6, \alpha_7 \leq \alpha_5, \alpha_6 \leq \alpha_8, \alpha_8 \leq L\}$. *Note the similarity of this stage with the setting of data-flow equations in [1] and the phase for collecting type equalities in the Hindley-Milner type inference algorithm [15]. The last constraint is added, because the overall expression is always useful. The solution of these constraints gives the following decoration:* $[y^L, z^D] \triangleright ((\lambda x^D.y^L)^{D \rightarrow L} z^D)^L$. *When we* sieve *this, we end up with* $(\lambda x.y)_{\square_{t_2}}$.

Before going further, we need to define some concepts that are used in the type inference algorithm.

DEFINITION 4.2
A *variable dependence type* is defined by the grammar $\rho := \alpha \mid \rho_1 \rightarrow \rho_2$, where α is a dependence type variable. A variable dependence type ρ *refines* an ordinary type τ iff $\tau = t$ and $\rho = \alpha$ or $\tau = \tau_1 \rightarrow \tau_2$, $\rho = \rho_1 \rightarrow \rho_2$, ρ_1 refines τ_1 and ρ_2 refines τ_2.

The aim of *pre-decoration* is to annotate expressions and environments with dependence type variables. The relationships between these variables are found by the type inference algorithm. We want the pre-decoration to be disjoint to ensure it contains as much information as possible:

$$
\begin{array}{ll}
\text{mk\_pos\_IO\_related}(\alpha) & = \emptyset \\
\text{mk\_pos\_IO\_related}(\rho_1 \to \rho_2) & = \{tail(\rho_1) \leq tail(\rho_2)\} \cup \text{mk\_neg\_IO\_related}(\rho_1) \cup \text{mk\_pos\_IO\_related}(\rho_2) \\
\text{mk\_neg\_IO\_related}(\alpha) & = \emptyset \\
\text{mk\_neg\_IO\_related}(\rho_1 \to \rho_2) & = \text{mk\_pos\_IO\_related}(\rho_1) \cup \text{mk\_neg\_IO\_related}(\rho_2)
\end{array}
$$

$$
\begin{array}{ll}
\text{BC}(\Gamma \rhd x^{\rho}) & = \{\Gamma(x) \leq \rho\} \\
\text{BC}(\Gamma \rhd (\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3 \to \rho_4}) & = \text{BC}(\Gamma, x^{\rho_1} \rhd e^{\rho_2}) \cup \{\rho_1 \to \rho_2 \leq \rho_3 \to \rho_4\} \\
\text{BC}(\Gamma \rhd (e_1^{\rho_1 \to \rho_2} e_2^{\rho_3})^{\rho_4}) & = \text{BC}(\Gamma \rhd e_1^{\rho_1 \to \rho_2}) \cup \text{BC}(\Gamma \rhd e_2^{\rho_3}) \cup \{\rho_2 \leq \rho_4, \rho_1 \to \rho_2 \leq \rho_3 \to \rho_4\}
\end{array}
$$

$$
\begin{array}{ll}
\text{solve}(\{\alpha \leq L\} \cup C) & = \text{solve}(C[\alpha := L]) \circ [\alpha := L] \\
\text{solve}(\{\rho_1 \to \rho_2 \leq \rho_3 \to \rho_4\} \cup C) & = \text{solve}(\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C) \\
\text{solve}(C) & = (\lambda\beta.\text{if } \beta \,\epsilon\, \text{freeVars}(C) \text{ then } D), \quad \textit{if the above rules don't apply anymore}
\end{array}
$$

$$
\begin{array}{ll}
TI(\Gamma_\lambda \rhd e_\lambda) & = \text{let } \Gamma \rhd e^{\rho} = \text{annotate}(\Gamma_\lambda \rhd e_\lambda) \\
& \quad \text{and } C = \text{BC}(\Gamma \rhd e^{\rho}) \cup \text{mk\_pos\_IO\_related}(\Gamma) \cup \text{mk\_neg\_IO\_related}(\rho) \cup \{tail(\rho) \leq L\} \\
& \quad \text{and } i = \text{solve}(C) \\
& \quad \text{in} \\
& \qquad i(\Gamma \rhd e^{\rho}) \\
A(\Gamma_\lambda \rhd e_\lambda) & = \text{let } \Gamma \rhd e^{\sigma} = \text{TI}(\Gamma_\lambda \rhd e_\lambda) \text{ in } sieve(e^{\sigma})
\end{array}
$$

FIG. 2. Type Inference Algorithm

DEFINITION 4.3
$\Gamma \rhd e^{\rho}$ is a *pre-decoration* of $\Gamma_\lambda \rhd e_\tau$ (where $\vdash_\lambda \Gamma_\lambda \rhd e_\tau$ in typed λ-calculus) iff

1. For each $x_\tau \epsilon \Gamma_\lambda$, $x^{\rho} \epsilon \Gamma$ where ρ is a variable dependence type and ρ refines τ.
2. For each subexpression $e_{1_{\tau_1}}$ of e_τ, $e_1^{\rho_1}$ is the corresponding subexpression of e^{ρ}, ρ_1 is a variable dependence type and ρ_1 refines τ_1.

$\Gamma \rhd e^{\rho}$ is *disjoint* if all the occurring dependence type variables are distinct.

We will assume the existence of a function *annotate*, that gives a disjoint pre-decoration as output, when given as input a simply typed λ-calculus typing. The formal definition can be found in [8]. As an example, $\text{annotate}([g^{t_1 \to t_2}, x^{t_1}] \rhd (g^{t_1 \to t_2} x^{t_1})^{t_2}) = [g^{\alpha_1 \to \alpha_2}, x^{\alpha_3}] \rhd (g^{\alpha_4 \to \alpha_5} x^{\alpha_6})^{\alpha_7}$.

The type inference algorithm is given in Figure 2. The functions mk_pos_IO_related() and mk_neg_IO_related() collect the necessary constraint set for making the final typing IO-related.

BC() in Figure 2 is a function that, given a pre-decoration, forms the constraint set necessary for the final typing to be correct in the dependence type system. Note the similarity to the setting of data-flow equations in [1]. The resulting constraint set can be reused in other analysis problems.

DEFINITION 4.4
A substitution i *satisfies* a constraint set C, if $\vdash i(\rho_1) \leq i(\rho_2)$ holds for all constraints $(\rho_1 \leq \rho_2) \epsilon C$,

The function solve() in Figure 2 takes a set of constraints as input and outputs a substitution of the dependence type variables. It is proven in [8] that this substitution makes the constraint set true under the rules of the type system. It is assumed that D does not occur in the constraints. Note that solve() simply "propagates data backward" and hence has similarities with other backward analysis techniques. An obvious example would be backward data-flow analysis [1], but other examples can be found in related work (Section 6). We have given a function specialized for the problem at hand to simplify proofs, but it can generalized to take care of other backward analysis, discussed in Section 5.

THEOREM 4.5
If $\Gamma \triangleright e^{\rho}$ is a disjoint pre-decoration and $C = \mathrm{BC}(\Gamma \triangleright e^{\rho}) \cup \mathrm{mk\_pos\_IO\_related}(\Gamma) \cup \mathrm{mk\_neg\_IO\_related}(\rho) \cup \{tail(\rho) \leq L\}$ and $i = \mathrm{solve}(C)$, then $\vdash i(\Gamma) \triangleright i(e^{\rho})$ and it is IO-correct.

Given a λ-expression and an environment, $TI()$ carries out type inference and outputs the original expression and environment decorated with dependence types, whereas $A()$ "sieves out" the useless parts from the original expression. We assume that the input to both is decorated with ordinary types. This can be easily achieved by extending the Hindley-Milner type inference algorithm $\mathcal{W}$ [15] to save the intermediate type information.

Theorem 4.6 proves that what $A()$ finds is in fact irrelevant to the result of the computation and hence dead code. So, the function $A()$ works as intended.

THEOREM 4.6
Assume $\vdash_{\lambda} \Gamma_{\lambda} \triangleright e_{\lambda}$ and $\epsilon = A(\Gamma_{\lambda} \triangleright e_{\lambda})$. Then $e_{\lambda} \Downarrow v_1$ and $\epsilon \Downarrow_c v_2$ imply $v_1 = v_2$.

We can also prove that the inference algorithm finds all the dead code expressible in the type system:

THEOREM 4.7
Let $\Gamma \triangleright e^{\sigma} = \mathrm{TI}(\Gamma_{\lambda} \triangleright e_{\lambda})$. For all IO-related $\Gamma_c \triangleright e^{\sigma_c}$, s.t. $\vdash \Gamma_c \triangleright e^{\sigma_c}$, for all subexpressions ${e_1}^{\sigma'}$ of e^{σ} and the corresponding subexpressions ${e_1}^{\sigma'_c}$ of e^{σ_c}, $useless(\sigma'_c)$ implies $useless(\sigma')$. Hence, $A()$ finds the maximum amount of dead code expressible in the type system.

5 Extensions to Other Analysis Problems

In this section, we will briefly mention some of the analysis problems that can be solved by the method of this paper. Program slicing [24, 19] is a technique for finding those parts of a program that can affect a given program point (backward slice) and those parts that a given program point can affect (forward slice). This problem can be solved by making the initial assumption that the program point in question is *useful* and propagating this information through subtyping constraints backward or forward respectively. Strictness analysis is a technique used in the optimization of programs written in a lazy functional programming language [23], which determines whether a given function f is strict, i.e. $f(\bot) = \bot$. This can be done in the framework of this paper, by letting the input of f to be $\bot$ and propagating this information forward. Partial evaluation is the specialization of programs w.r.t. some part of the input (designated as *static*), so that they will run faster, when the rest of the input (designated as *dynamic*) becomes available. Binding time analysis attempts to find those parts of the program that depend only on *static* input. It is equivalent to strictness analysis [13]. Automatic program debugging is the inference of program properties through automatic means with minimum user intervention. In this sense, they cover the area between type checkers like ML type inference algorithm and full-scale program verifiers. Information about the values that program variables can take are propagated through subtyping constraints and checks are made whether some assertions are satisfied. One example would be array bounds checking.

6 Related Work

In [1], dead code elimination is investigated for a first order imperative language. Inspite of this, there are parallels between our method for collecting flow information and data flow analysis.

The work in [14, 19, 23, 13] makes use of projections for different program analysis problems for first-order languages. In comparison, our use of a type system offers several benefits. Type systems are a widely used and natural tool for expressing properties about higher-order functions, which the above work does not address. Since we make the flow information explicit, we can use the methods used in this paper for a large variety of analysis problems, some of which are discussed in the preceding sections. Another reason for using a program analysis method readily adaptable to different analysis problems is the trend in modern compilers for using types in the compilation process [21, 18, 5]. Additional benefits are a type-based approach does not require the complete program to be available for analysis and types provide a good user interface, esp. important in software engineering applications.

Set constraints is another method for analyzing programs [10]. Although constraints are used, the system is not type based and hence shares some of the difficulties projection based methods in the applications we are considering. The mechanism for the extraction of constraints is different as well. To our knowledge, there is no work on dependence analysis using set constraints.

Another group of researchers became interested in dead-code elimination in trying to optimize programs extracted from proofs, which usually contain a lot of dead code. Representative examples are [17, 3, 4, 7]. Although types are used, the type systems are complicated and are not readily extendable to different problems, since typing rules are tailored to one specific problem. Similarly, type inference is non-standard and complicated. A more complete discussion can be found in [8].

7 Conclusion

In this paper, we have introduced a method for dependence analysis for typed functional programming languages, in the context of dead code elimination.

We currently aim at implementing our system in the context of a distributed communication system *Ensemble* [9]. Layering of protocols offers well-known advantages, but leads to performance inefficiencies. In [9], optimization techniques in the context of a particular system *Ensemble* are introduced which depend critically on dead code elimination. *Ensemble* is written in *ML* and hence traditional techniques for dead code elimination don't work in this context. Relevant work can be found in [11, 12].

We also intend to investigate further the implications of our system for the other program analysis problems we have cited. We are currently investigating this in the context of automatic debugging.

8 Acknowledgments

We would like to thank R. Constable, G. Morrisett, J. Hickey, C. Chang, V. Menon and the anonymous referees for valuable comments on this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] M. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[3] Berardi, S. Pruning simply typed lambda terms. *Journal of Symbolic Computation*, to appear.

[4] Berardi, S. and Boerio, L. Using subtyping in program optimization. In *Typed Lambda Calculus and Applications*. Springer Verlag, 1995.

[5] Birkedal, L. and Rothwell, N. and Tofte, M. and Turner, D.N. . The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[6] D. Bjorner, A.P. Ershov, and N.D. Jones. *Partial Evaluation and Mixed Computation*. North-Holland, 1987.

[7] M. Coppo, F. Damiani, and P. Giannini. Refinement Types for Program Analysis. In *SAS'96*, LNCS 1145, pages 143–158. Springer, 1996.

[8] O. Hafizogullari and C. Kreitz. Dead code elimination through type inference. Technical Report TR98-1673, Department of Computer Science, Cornell University, 1998.

[9] Hayden, M. and van Renesse, R. Optimizing layered communication protocols. Technical Report TR96-1613, Cornell University, 1996.

[10] Nevin Heintze. Set-based analysis of ML programs. In *1994 ACM Conf. on LISP and Functional Prog.*, pages 42–51, June 1994.

[11] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, editor, *Fifteenth International Conference on Automated Deduction*, LNAI. Springer Verlag, 1998.

[12] Kreitz, C. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR 97-1637, Cornell University, July 1997.

[13] J. Launchbury. *Projection Factorizations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, Glasgow, Scotland, June 1989.

[14] Y.A. Liu. Dependence analysis for recursive data. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, May 1998.

[15] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17):348–375, 1978.

[16] J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, july 1991.

[17] C. Paulin-Mohring. Extracting F_ω's programs from proofs in second order *lambda*-calculus. In ACM, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[18] Peyton Jones, S.L. and Hall, C.V. and Hammond, K. and Partain, W. and Wadler, P. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[19] T. Reps and T. Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, 1996.

[20] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.

[21] Tarditi, D. and Morrisett, G. and Cheng, P. and Stone, C. and Harper, R. and Lee, P. Til: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[22] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *S. Abramsky and C. Hankin, editors, Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood Limited, 1987.

[23] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer Verlag, September 1987.

[24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.