Dead Code Elimination Through Type Inference

Ozan Hafızoğulları Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853, USA
{ozan,kreitz}@cs.cornell.edu

Abstract. We introduce a method to detect and eliminate dead code 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 other areas of software engineering.

1 Introduction

Dead code in a given program are those parts that do not affect the final result of the execution of the program. 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. In this paper, we give a method for dead code elimination for typed λ-calculus based languages, such as ML and typed intermediate languages.

One trend in compiler research for typed functional programming languages is to use types extensively in the compilation process [23, 32, 26, 14, 5, 15] via typed intermediate languages. Our method is both an instance of and a model for program analysis in these compilers. A closely related issue is the optimization of highly modular systems. Systems built out of modules offer well-known advantages, but lead to performance inefficiencies. Dead code elimination at the source level is a fundamental ingredient in the optimization of these systems. We discuss this more thoroughly in the context of a particular distributed communication system *Ensemble*, at the conclusion. Another usage is in software engineering, where dead code arises during the evolution of programs. Detection of dead code is useful in tools for software engineering, both as it is and as a model for other closely related program analysis problems, explained below. Another area where the problem of dead code elimination appears is the

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

optimization of proof extracts from formal proofs [2, 6, 3, 4, 25, 31]. The code generated by extraction from proofs usually contains a lot of redundancy and dead code elimination is one technique used to optimize this code.

As it is noted in [16, 17], there are other program analysis problems that do not explicitly involve dead code, but depend on closely related notions. Examples are program slicing [33, 29], specialization [29], incrementalization [18, 19] and compile-time garbage collection [9, 10, 24]. Hence, [16] prefers to use a more general name and call it *dependence analysis*. The techniques developed in our paper can be used to address these problems.

In this paper, we present a technique for dead code elimination in typed functional programming languages. The central idea of our approach is to introduce a type system based on simple subtypes [21], 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, etc., 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.

In Section 2 we introduce a simple type system for refinement types which is built on top of the standard typed λ-calculus type system [28] and uses simple subtypes as in [21]. We present an algorithm $sieve()$ which converts a typed expression to a *sieved expression* and thus eliminates dead code. In Section 3 we investigate the properties of our type system and show that the application of $sieve()$ does in fact preserve the value of an expression. In Section 4 we introduce an extended type inference algorithm. We show that its combination with $sieve()$ does in fact lead to the desired result (Theorem 7) and that it pinpoints all the dead code specifiable in the type system (Theorem 8). In Section 5 we explain how our approach can be extended to a real programming language such as ML. In Section 6, we discuss related work. We conclude with a brief discussion of future work.

2 A Simple Type System for Refinement Types

We give a discussion of the ideas involved in this section, before getting to the formal work. The first idea is that we have a type system, called *refinement types*, built 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 to the value of the expression.

Consider $e = (\lambda x.y)z$. Here, the subexpression z is irrelevant to the result of the computation. In the system we propose, e would be decorated with *refinement types* 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" all the subexpressions with their *refinement types.*

The intuition behind *refinement types* is as follows. 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 y has some value which can affect the result of the overall computation and $\lambda x.y$ is a function, whose input does not affect the computation, but its result may.

Given a "decorated" ε, we can "sieve out" dead code as follows:

$$\epsilon = (\lambda x.y)\Box_t$$

We use $\Box_t$ to stand for a subexpression with type t in typed λ- calculus. We do not care what is inside $\Box_t$, which is z in this case, since it does not affect the result of the computation.

So, the general strategy we follow is to take a typed λ-calculus term, "decorate" it with "refinement types" and finally locate dead code using these types. The formal definitions of these concepts are given later in this chapter.

Example 1. The expression $e = (\lambda y.((\lambda f.f)(\lambda x.gu))y)((\lambda z.z)w)$ contains two subexpressions y and $(\lambda z.z)w$ which are irrelevant for the result of its evaluation. In our type system, the expression e would be decorated with types as follows:

$$\varepsilon = ((\lambda y^D.((\lambda f^{D\to L}.f^{D\to L})^{(D\to L)\to(D\to L)}\ (\lambda x^D.(g^{L\to L}u^L)^L)^{D\to L})^{D\to L}\ y^D\ )^L \\ \quad (\ (\lambda z^D.z^D)^{D\to D}w^D)^D\ )^L$$

y^D intuitively means that y has a value that does not affect the result of the overall computation while $(g^{L\to L}u^L)^L$ expresses that the value of gu will be used during computation. $\lambda x.gu$ is a function whose input does not affect the computation, but its result may.

Given a decorated expression, we can *sieve out* all dead code, i.e. *to a first approximation*, all the expressions which are decorated with D. If the original type of such an expression in typed λ-calculus is τ we replace the expression by the term $\Box_\tau$. Eliminating the dead code from ε thus results in $\epsilon = (\lambda y.((\lambda f.f)(\lambda x.gu))\ \Box_t)\ \Box_t$ where t is a type variable.

In the rest of the examples, we will stick to this convention of using ε_i for decorated expressions and ϵ_i for *sieved* expressions.

In the rest of this section we formally define a type system of refinement types which refines the simple types of the standard typed λ-calculus [28]. Several issues that arise in the extension to a full-scale programming language, like *let-polymorphism*, recursion and implementation are postponed to Section 5.

Definition 1. *The simply typed λ-calculus is defined by:*
$\tau := t \mid \tau_1 \to \tau_2$
$e := x \mid \lambda x : \tau.e \mid e_1 e_2$

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

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

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

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

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

Fig. 1. Type system for the typed λ-calculus with refinement types

Definition 2. Refinement types *are defined by the grammar* $\sigma := L \mid D \mid \sigma_1 \rightarrow \sigma_2$. *A refinement type* σ refines *a simple type* τ *iff*

1. $\tau = t$ *and* ($\sigma = L$ *or* $\sigma = D$) *or*
2. $\tau = \tau_1 \rightarrow \tau_2$, $\sigma = \sigma_1 \rightarrow \sigma_2$, σ_1 *refines* τ_1, *and* σ_2 *refines* τ_2.

Hence, a refinement type refines any ordinary type which has the same syntactical structure. It does not distinguish between different type variables but instead 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 on top of 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 [21]. However, since refinement types contain no variables, there is no need for a constraint environment. Instead of giving a single type to a λ-expression e, we *decorate* it, i.e. annotate all subexpressions with their corresponding refinement types.The reason for this is not only the final type σ of e, but the types of all the subexpressions are utilized for the purpose of dead code elimination. A *decorated expression* is denoted by e^σ.

As usual the type system is presented in the form of 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 refinement type σ of a decorated expression e^σ determines whether e is dead or alive. Intuitively, a subexpression e may be useful to the result of the overall computation if it has a refinement type σ whose final result type is L. Only this *tail* of σ is relevant, since the usefulness of a function is determined by the usefulness of its result.

Definition 3. *The* tail *of a refinement type* σ *is defined recursively as follows:*

$$tail(L) = L, \;\; tail(D) = D, \;\; tail(\sigma_1 \rightarrow \sigma_2) = tail(\sigma_2).$$

A refinement type σ is useful *if $tail(\sigma) = L$ and* useless *if $tail(\sigma) = D$. We denote this by* useful(σ) *and* useless(σ).

A subexpression of a given expression e whose refinement 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 thus 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 $\rightarrow_c$.

Definition 4. Sieved expressions *are defined by the grammar*

$$\epsilon = x \mid \Box_\tau \mid \lambda x.e \mid e_1 e_2.$$

The reduction $\rightarrow_c$ for sieved expressions is defined by $(\lambda x.e_1)e_2 \rightarrow_c e_1\{e_2/x\}$.

Example 2. Consider the sieved expression $\epsilon = (\lambda y.((\lambda f.f)(\lambda x.u))\Box_t)\Box_t$ from Example 1. Then $\epsilon \rightarrow_c ((\lambda f.f)(\lambda x.gu))\Box_t \rightarrow_c (\lambda x.gu)\Box_t \rightarrow_c gu$

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

Definition 5. *The function sieve is defined recursively as follows*

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

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

Example 3. Let ε be the decorated term given in Example 1. Then $sieve(\varepsilon)$ results in $\epsilon = (\lambda y.((\lambda f.f)(\lambda x.u))\Box_t)\Box_t$.

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 that thus it is in fact dead code. In more specific terms, given an expression e_1, if ε_1 is a "decorated" version of it, then under certain assumptions, $sieve(\varepsilon_1)$ and e_1 reduce to the same normal form. This is proven in Theorem 5.

First, let us explain what we mean by "normal form".

Definition 6. *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. $\rightarrow_\beta$ ($\rightarrow_c$ respectively).*

Definition 7.

- $e_1 \Downarrow e_n$ *if* $e_1 \rightarrow_\beta \ldots \rightarrow_\beta e_n$ *and* e_n *is in* β*-normal form.*
- $e_1 \Downarrow_c e_n$ *if* $e_1 \rightarrow_c \ldots \rightarrow_c e_n$ *and* e_n *is in* c*-normal form.*

From now on, we will use *normal form*, instead of β-normal form or c-normal form, if the meaning can be inferred from the context.

Example 4. Let e and ϵ be as in Example 1. Then the corresponding reductions are:

$$\begin{array}{ll}
e \;=\; (\lambda y.((\lambda f.f)(\lambda x.gu))y)((\lambda z.z)w) & \epsilon \;=\; (\lambda y.((\lambda f.f)(\lambda x.gu))\Box_t)\Box_t \\
\quad\rightarrow_\beta (\lambda y.(\lambda x.gu)y)((\lambda z.z)w) & \quad\rightarrow_c (\lambda y.(\lambda x.gu)\Box_t)\Box_t \\
\quad\rightarrow_\beta (\lambda y.(\lambda x.gu)y)w & \\
\quad\rightarrow_\beta (\lambda x.gu)w & \quad\rightarrow_c (\lambda x.gu)\Box_t \\
\quad\rightarrow_\beta gu & \quad\rightarrow_c gu
\end{array}$$

Here is a rough discussion of what is to follow. We want to prove Theorem 5, 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 $\epsilon_1 \Downarrow_c v$. The method of the proof can be summarized by the following:

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

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 2, 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: 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 1 states that under certain conditions, an expression in normal form has no dead code. Hence, in our diagram, $e_n = \epsilon_k$.

To this aim, we first introduce the concept of "IO-correctness". Here is an intuitive discussion of how it arises.

Example 5. Consider the typing $f^{D\rightarrow L} \rhd (f^{D\rightarrow L} 3^D)^L$ 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. So, if the output of $\lambda x.x$ is useful, then so is its input. 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. In particular, 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. The existence of higher order functions and lambda abstractions brings in some complications. The exact formulation is below.

Definition 8. *A type $\sigma_1 \rightarrow \sigma_2$ is top-level IO-related iff $tail(\sigma_1) \leq tail(\sigma_2)$.*

Definition 9. *A type σ is* positively IO-related *iff all subtypes of σ at positive occurrences are top-level IO-related. Similarly, a type σ is* negatively IO-related *iff all subtypes of σ at negative occurrences are top-level IO-related.*

Definition 10. *A typing assertion $\Gamma \rhd e^\sigma$ is* IO-correct *iff*

1. *For all $f^{\sigma_f} \epsilon dom(\Gamma)$, σ_f are positively IO-related.*
2. *σ is negatively IO-related.*
3. *$tail(\sigma) \leq L$*

In the above definition, the first assertion makes sure that the assumptions about the variables in the environment are conservative. Similarly, the second assertion guarantees that the assumptions about the inputs to the expression e are conservative. The last assertion makes sure that the whole expression e is always considered useful.

The following lemma is used in the proof of 1.

Lemma 1.

1. *If $\sigma_1 \leq \sigma_2$ and σ_1 is positively IO-related, then σ_2 is positively IO-related.*
2. *If $\sigma_1 \leq \sigma_2$ and σ_2 is negatively IO-related, then σ_1 is negatively IO-related.*

Proof: We use mutual induction on the structure of σ_1 and σ_2.

Base Case: Let $\sigma_1 = \eta_1 \rightarrow \kappa_1$ and $\sigma_2 = \eta_2 \rightarrow \kappa_2$, where η_1, η_2,κ_1 and κ_2 are either or L. The assumption σ_1 is positively IO-related implies that $\vdash \eta_1 \leq \kappa_1$. Since $\vdash \sigma_1 \leq \sigma_2$, this implies that σ_2 is positively IO-related.
The second case can be proven similarly.

Induction Case: We deal with the first case first. Let $\sigma_1 = \eta_1 \rightarrow \kappa_1$ and $\sigma_2 = \eta_2 \rightarrow \kappa_2$. By the assumptions of the lemma, $\vdash \eta_1 \rightarrow \kappa_1 \leq \eta_2 \rightarrow \kappa_2 (1)$ and $\eta_1 \rightarrow \kappa_1$ is positively IO-related(2). These imply that η_1 is negatively IO-related and $\vdash \eta_2 \leq \eta_1$, which in turn imply that η_2 is negatively IO-related. Similarly we have κ_1 is positively IO-related and $\vdash \kappa_1 \leq \kappa_2$, which in turn imply κ_2 is positively IO-related. Hence, we only need to prove $\vdash tail(\eta_2) \leq tail(\kappa_2)$. Now (1) implies $\vdash tail(\eta_2) \leq tail(\eta_1)$ and $\vdash tail(\kappa_1) \leq tail(\kappa_2)$. (2) implies $\vdash tail(\eta_1) \leq tail(\kappa_1)$. These together imply $\vdash tail(\eta_2) \leq tail(\kappa_2)$, as required.

$\square$

The following theorem proves that under certain assumptions, expressions in normal form have no dead code.

Theorem 1. *If* $\vdash \Gamma \triangleright e^{\sigma}$ *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(σ_1).

Proof: By induction on the proof tree for $\vdash \Gamma \triangleright e^{\sigma}$. The proof cases are the last rule applied in the proof.

$\dfrac{}{\Gamma \triangleright x^{\Gamma(x)}}$: From the third condition of being IO-correct, we have $tail(\sigma) \leq L$. So, σ is not useless.

$\dfrac{\Gamma, x^{\sigma_1} \triangleright e^{\sigma_2}}{\Gamma \triangleright (\lambda x^{\sigma_1}.e)^{\sigma_1 \rightarrow \sigma_2}}$: If we prove $\vdash \Gamma, x^{\sigma_1} \triangleright e^{\sigma_2}$ is IO-correct and e is in normal form, then using the induction hypothesis, we have our result. The second part follows from the fact that $\lambda x.e$ is in normal form.As for the first part:

1. We have $\sigma_1 \rightarrow \sigma_2$ is negatively IO-related, which implies that σ_1 is positively IO-related. Hence, Γ, x^{σ_1} is positively IO-related.
2. We have $\sigma_1 \rightarrow \sigma_2$ is negatively IO-related, which implies that σ_2 is negatively IO-related.
3. $tail(\sigma_1 \rightarrow \sigma_2) = tail(\sigma_2) \leq L$.

$\dfrac{\Gamma \triangleright e_1^{\sigma_1 \rightarrow \sigma_2} \quad \Gamma \triangleright e_2^{\sigma_1}}{\Gamma \triangleright (e_1^{\sigma_1 \rightarrow \sigma_2} e_2^{\sigma_1})^{\sigma_2}}$: By the assumptions of the theorem, $e_1 e_2$ is in normal form. Hence, e_1 cannot be a λ-expression. So, either e_1 is a variable or an application, which is itself in normal form. By induction, it can be proven that for some ${\sigma_1}^{(n)}$ to ${\sigma_1}^{(1)}$, $e_1 = f^{{\sigma_1}^{(n)} \rightarrow \ldots \rightarrow {\sigma_1}^{(1)} \rightarrow \sigma_1 \rightarrow \sigma_2} e_{1n}^{{\sigma_1}^{(n)}} \ldots e_{11}^{{\sigma_1}^{(1)}} e_1^{\sigma_1}(1)$. From $f \epsilon dom(\Gamma)$ and first condition of being IO-correct, we have ${\sigma_1}^{(n)} \rightarrow \ldots \rightarrow {\sigma_1}^{(1)} \rightarrow \sigma_1 \rightarrow \sigma_2$ is positively IO-related.

e_1 has no dead code iff $e_{1i}^{{\sigma_1}^{(i)}}$ has none and f is not dead. f is not dead, because $tail({\sigma_1}^{(n)} \rightarrow \ldots \rightarrow {\sigma_1}^{(1)} \rightarrow \sigma_1 \rightarrow \sigma_2) = tail(\sigma_2) \leq L$, which follows from the third part of the assumption of the theorem asserting that $\Gamma \triangleright e_1 e_2$ is IO-correct.

It can be proven by the induction hypothesis that $\Gamma \triangleright e_{1i}^{\sigma_1^{(i)}}$ has no dead code:

1. Γ is positively IO-related, which follows from the assumption of the theorem.
2. (1) implies $\sigma_1^{(i)}$ is negatively IO-related.
3. We have $tail(\sigma_2) \leq L$ from the third part of the assumption, that $\Gamma \triangleright (e_1 e_2)^{\sigma_2}$ is IO-correct. (1) implies $\vdash tail(\sigma_1^i) \leq tail(\sigma_2)$. Hence, we have $\vdash tail(\sigma_1^i) \leq L$.

It can also be proven by the induction hypothesis that $\Gamma \triangleright e_2^{\sigma_1}$ has no dead code.

1. $\Gamma \triangleright e_1 e_2$ is IO-correct, hence Γ is positively IO-related.
2. (1) implies that σ_1 is negatively IO-related.
3. (1) implies that $\vdash tail(\sigma_1) \leq tail(\sigma_2)$ and we know that $\vdash tail(\sigma_2) \leq L$, hence $\vdash tail(\sigma_1) \leq L$.

Hence, $\Gamma \triangleright e_2^{\sigma_1}$ has no dead code. The application $(e_1 e_2)^{\sigma_2}$ is not dead, since $tail(\sigma_2) \leq L$, by the assumption of the theorem. Hence, $\Gamma \triangleright (e_1 e_2)^{\sigma_2}$ has no dead code.

$\dfrac{\Gamma \triangleright e^{\sigma_1} \qquad \vdash \sigma_1 \leq \sigma_2}{\Gamma \triangleright e^{\sigma_2}}$: We have to prove that $\Gamma \triangleright e^{\sigma_1}$ is IO-correct. That e^{σ_1} is in normal form follows from the assumption that e^{σ_2} is in normal form.

1. The assumption of the theorem gives that Γ is positively IO-related.

2. The assumption of the theorem implies σ_2 is negatively IO-related. We have $\vdash \sigma_1 \leq \sigma_2$. Hence, by the previous lemma, σ_1 is IO-related.
3. $\vdash \sigma_1 \leq \sigma_2$ implies $\vdash tail(\sigma_1) \leq tail(\sigma_2)$. The assumption of the theorem implies, $\vdash tail(\sigma_2) \leq L$. Hence $\vdash tail(\sigma_1) \leq L$.

Hence, we have proven $\vdash \Gamma \triangleright e^{\sigma_1}$ is IO-correct and e is in normal form. By the induction hypothesis, e^{σ_1} and hence e^{σ_2} has no dead code.

□

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. For this purpose 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 11. Substitution $e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\}$ *on decorated expressions is recursively defined by the following rules*

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

The reduction $\to_b$ for decorated expressions is defined by

$$((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\to\sigma_4}e_2^{\sigma_3})^{\sigma_5} \ \to_b \ (e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\})$$

It is important to note that the reduction rule keeps only the outermost type of an expression. For example, $((\lambda x^{\sigma}.x^{\sigma'})^{\sigma\to\sigma'}e^{\sigma_2})^{\sigma_1} \ \to_b \ x^{\sigma_1}\{|e^{\sigma_2}/x|\} = e^{\sigma_1}$.

Example 6. Let $\varepsilon = ((\lambda x^D.y^L)^{D\to L}z^D)^L$. Then, $\varepsilon \to_b z^L$.

Example 7. Let ε be the decorated term given in Example 1. Then
$\varepsilon \ \to_b \ ((\lambda y^D.((\lambda x^D.(g^{L\to L}u^L)^L)^{D\to L}y^D)^L)^{D\to L}((\lambda z^D.z^D)^{D\to D}w^D)^D)^L$

The following lemma is used in the proof of Theorem 2.

Lemma 2. *If* $\vdash \Gamma, x^{\sigma_1} \triangleright {e_1}^{\sigma_2}$, $\vdash \Gamma \triangleright {e_2}^{\sigma_3}$ *and* $\vdash \sigma_3 \leq \sigma_1$, *then* $\vdash \Gamma \triangleright {e_1}^{\sigma_2}\{|{e_2}^{\sigma_3}/x|\}$.

Proof: By straightforward induction on the structure of e:

$e = x$: We have $\vdash \Gamma, x^{\sigma_1} \triangleright x^{\sigma_2}$ (1), $\vdash \Gamma \triangleright {e_2}^{\sigma_3}$ (2) and $\vdash \sigma_3 \leq \sigma_1$ (3). In the proof of (1), only the rule SUB can be used. Hence, $\vdash \sigma_1 \leq \sigma_2$. This with (3) gives $\vdash \sigma_3 \leq \sigma_2$. (2) combined with this gives $\vdash \Gamma \triangleright {e_2}^{\sigma_2}$. Since, by definition ${e_2}^{\sigma_2} = x^{\sigma_2}\{|{e_2}^{\sigma_3}/x|\}$, we are done.

$e = y \neq x$: $\vdash \Gamma, x^{\sigma_1} \triangleright y^{\sigma_2}$ implies $\vdash \Gamma \triangleright y^{\sigma_2}$. Since, by definition, $y^{\sigma_2} = y^{\sigma_2}\{|{e_1}^{\sigma_3}/x|\}$, we are done.

$e = (\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\to\rho_4}$: Assume $\vdash \Gamma, x^{\sigma_1} \triangleright (\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\to\rho_4}$ (1), $\vdash \Gamma \triangleright {e_2}^{\sigma_3}$ (2) and $\vdash \sigma_3 \leq \sigma_1$ (3). (1) is true only if $\vdash \rho_1\to\rho_2 \leq \rho_3\to\rho_4$ (4) and $\vdash \Gamma, x^{\sigma_1}, y^{\rho_1} \triangleright d^{\rho_2}$ (5). Hence, by induction hypothesis and (2), we have $\vdash \Gamma, y^{\rho_1} \triangleright d^{\rho_2}\{|{e_2}^{\sigma_3}/x|\}$ (6). By LAM, SUB and (4), $\vdash \Gamma \triangleright (\lambda y^{\rho_1}.d^{\rho_2}\{|{e_2}^{\sigma_3}/x|\})^{\rho_3\to\rho_4}$. The last expression is equal to $(\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\to\rho_4}\{|{e_2}^{\sigma_3}/x|\}$, by definition.

$e = (d_1^{\rho_1\to\rho_2}d_2^{\rho_1})^{\rho_3}$: Similar to the case above.

□

The following theorem asserts that the correspondence between raw and decorated λ-terms are kept intact after a reduction step.

Theorem 2. *If $\vdash \Gamma \rhd \varepsilon_1$ is a decoration of e_1 with refinement types, $e_1 \rightarrow_\beta e_2$ and $\varepsilon_1 \rightarrow_b \varepsilon_2$, where these reduction steps correspond to each other, then $\vdash \Gamma \rhd \varepsilon_2$ is a decoration of e_2.*

Proof: Let $E[((\lambda x.e_1)e_2)] \rightarrow_\beta E[(e_1\{e_2/x\})]$. We will transform a proof of the decoration of the left hand side of the reduction to a proof of the decoration of the right hand side. Due to the nature of the proof system, all we need to do is to prove the part related to the reduction $((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\rightarrow\sigma_4} e_2^{\sigma_3})^{\sigma_5} \rightarrow_b (e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\})$. The proof of the left hand side is of the following form:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x^{\sigma_1} \rhd e_1^{\sigma_2}(1)}}{(\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_1\rightarrow\sigma_2}}}{\vdots\,\text{Appl. of SUB rule}}}{\Gamma \rhd (\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\rightarrow\sigma_4}} \quad \dfrac{\vdots}{\Gamma \rhd e_2^{\sigma_3}(2)}}{\Gamma \rhd ((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\rightarrow\sigma_4} e_2^{\sigma_3})^{\sigma_4}}}{\vdots\,\text{Appl. of SUB rule}}}{\Gamma \rhd ((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3\rightarrow\sigma_4} e_2^{\sigma_3})^{\sigma_5}}
$$

We have $\vdash \Gamma, x^{\sigma_1} \rhd e_1^{\sigma_2}$ (1) and $\vdash \Gamma \rhd e_2^{\sigma_3}$ (2). From the proof, it is seen that $\vdash \sigma_1 \rightarrow \sigma_2 \leq \sigma_3 \rightarrow \sigma_4$ (3) and $\vdash \sigma_4 \leq \sigma_5$ (4). Using (1), (2) and $\vdash \sigma_3 \leq \sigma_1$, which follows from (3), we have $\vdash \Gamma \rhd e_1^{\sigma_2}\{|e_2^{\sigma_3}/x|\}$, by the previous lemma. From (3) and (4), we have $\vdash \sigma_2 \leq \sigma_5$. We eventually have $\vdash \Gamma \rhd e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\}$. □

The following two lemmas are used in the proof of Theorem 3.

Lemma 3. *If $\vdash \Gamma \rhd e^{\sigma_1}$ and $\vdash \Gamma \rhd e^{\sigma_2}$ are two decorations that are only different in the types σ_1 and σ_2 and neither of these are useless, then $sieve(e^{\sigma_1}) = sieve(e^{\sigma_2})$.*

Proof: Straightforward induction. □

Lemma 4. *Let $\epsilon = (sieve(e_1^{\sigma_1}))\{sieve(e_2^{\sigma_2})/x\}$ and $\epsilon' = sieve(e_1^{\sigma_1}\{|e_2^{\sigma_2}/x|\})$, where $\vdash \Gamma, x^\sigma \rhd {e_1}^{\sigma_1}$ and $\vdash \Gamma \rhd {e_2}^{\sigma_2}$ for some Γ and σ. If $\vdash \sigma_2 \leq \sigma$, then $\epsilon = \epsilon'$.*

Proof: Proof by induction on the "usefulness" of σ_1 and the structure of ε_1:
If $useless(\sigma_1)$, then $\epsilon = \square_\tau = \epsilon'$, where σ_1 refines τ.
Else:
x: $\epsilon = x\{sieve(e_2^{\sigma_2})/x\} = sieve(e_2^{\sigma_2})$ and $\epsilon' = sieve(e_2^{\sigma_1})$. From the assumption of the lemma, $\vdash \sigma_2 \leq \sigma$ and since $\vdash \Gamma, x^\sigma \rhd x^{\sigma_1}$, we have $\vdash \sigma \leq \sigma_1$. So $\vdash \sigma_2 \leq \sigma_1$ and $useful(\sigma_1)$. Hence, $useful(\sigma_2)$. Finally, using the previous the lemma, $\epsilon = \epsilon'$.
y: If $x \neq y$, then $\epsilon = \epsilon' = y$.
$(\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\rightarrow\rho_4}$: We have

$$
\begin{aligned}
\epsilon &= sieve((\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\rightarrow\rho_4})\{sieve(e_2^{\sigma_2})/x\} \\
&= (\lambda y.sieve(d^{\rho_4}))\{sieve(e_2^{\sigma_2})/x\} \\
&= \lambda y.(sieve(d^{\rho_4})\{sieve(e_2^{\sigma_2})/x\}) \\
\epsilon' &= sieve((\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3\rightarrow\rho_4}\{|e_2^{\sigma_2}/x|\}) \\
&= sieve((\lambda y^{\rho_1}.d^{\rho_2}\{|e_2^{\sigma_2}/x|\})^{\rho_3\rightarrow\rho_4}) \\
&= \lambda y.sieve(d^{\rho_4}\{|e_2^{\sigma_2}/x|\})
\end{aligned}
$$

since $\rho_3 \to \rho_4 = \sigma_1$ is useful by assumption.

We know $\vdash \Gamma, x^\sigma \rhd (\lambda y^{\rho_1}.d^{\rho_2})^{\rho_3 \to \rho_4}$ (1) and assume $\vdash \sigma_2 \le \sigma$. (1) implies $\vdash \Gamma, y^{\rho_1}, x^\sigma \rhd d^{\rho_2}$ and $\vdash \rho_2 \le \rho_4$. Hence, $\vdash \Gamma, y^{\rho_1}, x^\sigma \rhd d^{\rho_4}$. We also know that $\vdash \Gamma \rhd {e_2}^{\sigma_2}$, which implies $\vdash \Gamma, y^{\rho_1} \rhd {e_2}^{\sigma_2}$ (we already remarked that we assume no name clashes, throughout the paper).

Hence, by the induction hypothesis, $sieve(d^{\rho_4})\{sieve(e_2^{\sigma_2})/x\} = sieve(d^{\rho_4}\{|e_2^{\sigma_2}/x|\})$.

Finally, we get the required result $\epsilon = \epsilon'$.

$(d_1^{\rho_1 \to \rho_2} d_2^{\rho_1})^{\rho_3}$: This case is proven similar to the previous case. $\square$

The following theorem shows that the correspondence between decorated and sieved expressions is kept intact after a reduction:

Theorem 3. *If $\vdash \Gamma \rhd \varepsilon_1$, $\varepsilon_1 \to_b \varepsilon_2$ and $sieve(\varepsilon_1) = \epsilon_1$, then either*

1. *$sieve(\varepsilon_2) = \epsilon_1$, if no corresponding reduction step exists for ϵ_1, or*
2. *$\epsilon_1 \to_c \epsilon_2$ and $sieve(\varepsilon_2) = \epsilon_2$.*

Proof: We can assume W.L.O.G.:

$$\begin{aligned} \varepsilon_1 &= ((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3 \to \sigma_4} e_2^{\sigma_3})^{\sigma_5} \\ \varepsilon_2 &= e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\} \end{aligned}$$

If σ_5 is useless, then $\epsilon_1 = \square_\tau = sieve(\varepsilon_2)$, where σ_5 refines τ.
If it is useful, then

$$\begin{aligned} \epsilon_1 = sieve(\varepsilon_1) &= sieve((\lambda x^{\sigma_1}.e_1^{\sigma_2})^{\sigma_3 \to \sigma_5}) sieve(e_2^{\sigma_3}) \\ &= (\lambda x.sieve(e_1^{\sigma_5}))(sieve(e_2^{\sigma_3})) \end{aligned}$$

since $useful(\sigma_5)$ implies $useful(\sigma_3 \to \sigma_5)$. Hence, $\epsilon_2 = sieve(e_1^{\sigma_5})\{sieve(e_2^{\sigma_3})/x\}$. We also have $sieve(\varepsilon_2) = sieve(e_1^{\sigma_5}\{|e_2^{\sigma_3}/x|\})$. We know that $\vdash \Gamma \rhd \varepsilon_1 = ((\lambda x^{\sigma_1}.{e_1}^{\sigma_2})^{\sigma_3 \to \sigma_4} {e_2}^{\sigma_3})^{\sigma_5}$. Hence, $\vdash \Gamma, x^{\sigma_1} \rhd {e_1}^{\sigma_2}$ (1), $\vdash \Gamma \rhd {e_2}^{\sigma_3}$ (2), $\vdash \sigma_1 \to \sigma_2 \le \sigma_3 \to \sigma_4$ (3) and $\vdash \sigma_4 \le \sigma_5$ (4). (1), (3) and (4) imply $\vdash \Gamma, x^{\sigma_1} \rhd {e_1}^{\sigma_5}$ (5). (3) implies $\vdash \sigma_3 \le \sigma_1$. Thus, we can use the previous lemma to prove $\epsilon_2 = sieve(\varepsilon_2)$. $\square$

The following theorem shows that IO-correctness remains invariant under reduction by "$\to_b$".

Theorem 4. *If $\vdash \Gamma \rhd \varepsilon_1$ is IO-correct and $\varepsilon_1 \to_b \varepsilon_2$, then $\vdash \Gamma \rhd \varepsilon_2$ is IO-correct.*

Proof: This is trivial. First condition of being IO-correct is true for ε_2, because remains unchanged. Second and third conditions are true, because the outermost type does not change during the reduction: $\varepsilon_1 = e_1^{\sigma_1}$ and $\varepsilon_1 \to_c \varepsilon_2 = e_2^{\sigma_2}$ imply $\sigma_1 = \sigma_2$. $\square$

This is the main theorem of this section. Intuitively, it asserts that under certain conditions, a given expression and its sieved out counter-part evaluate to the same value. Hence, what the function *sieve*() marks as useless is indeed dead code.

Theorem 5. *If $\vdash \Gamma \rhd \varepsilon_1$ 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 $v_1 = v_2$.*

Proof: Assume $e_1 \rightarrow_\beta e_2 \rightarrow_\beta \ldots \rightarrow_\beta e_n$, where e_n is in normal form. Let $\varepsilon_1 \rightarrow_\beta \varepsilon_2 \rightarrow_\beta \ldots \rightarrow_\beta \varepsilon_n$ be the corresponding reduction of a decoration of e_1, such that $\vdash \Gamma \rhd \varepsilon_1$.
By Theorem 2 and induction, $\vdash \Gamma \rhd \varepsilon_i$ is a decoration of e_i for $i \in \{1 \ldots n\}$.
Theorem 4 proves that $\Gamma \rhd \varepsilon_i$'s are IO-correct.
Now let $\epsilon_1 \rightarrow_c \epsilon_2 \rightarrow_c \ldots \rightarrow_c \epsilon_k$ be the reduction of $\epsilon_1 = sieve(\varepsilon_1)$, corresponding to the reduction $\varepsilon_1 \rightarrow_b \ldots \rightarrow_b \varepsilon_n$. By Theorem 3 and induction, we have

$$\begin{aligned} \exists l_1, \ldots, l_k, l_{k+1} : (l_1 = 1) \; and \; (l_{k+1} = n+1) \; and \; \forall j \epsilon \{1, \ldots, k\} : \\ \forall i \epsilon \{l_j, \ldots, l_{j+1} - 1\} : sieve(\varepsilon_i) = \epsilon_j \end{aligned}$$

Since, we assume e_n is in normal form, Theorem 1 proves that ε_n has no dead code. Hence, $v_1 = e_n = \epsilon_k = v_2$. □

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 to specify 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 refinement types, constraints and the function BC(), all to be defined below. These concepts make it possible to use our method in program analysis problems other than dead code elimination.

Example 8. Here, we give a simple example to convey some of the intuition behind the type inference algorithm. Let $\Gamma \rhd e = [y : t_1, z : t_2] \rhd \lambda x.yz : t_2$. First, we decorate it with *refinement type variables*, to be defined below:

$$\Gamma \rhd e^\rho = [y^{\alpha_1}, z^{\alpha_2}] \rhd ((\lambda x^{\alpha_3}.y^{\alpha_4})^{\alpha_5 \rightarrow \alpha_6} z^{\alpha_7})^{\alpha_8}$$

Then, we get the relevant constraints between these variables:

$$C = \{\alpha_1 \leq \alpha_4, \alpha_2 \leq \alpha_7, \alpha_3 \rightarrow \alpha_4 \leq \alpha_5 \rightarrow \alpha_6, \alpha_7 \leq \alpha_5, \alpha_6 \leq \alpha_8, \alpha_8 \leq L\}$$

The last constraint is added, because the overall expression is always useful. The solution of these constraints gives the following decoration:

$$[y^L, z^D] \rhd ((\lambda x^D.y^L)^{D \rightarrow L} z^D)^L$$

When we *sieve* this, we end up with $(\lambda x.y)\Box_{t_2}$.

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

Definition 12. *A* variable refinement type *is defined by* $\rho := \alpha \mid \rho_1 \rightarrow \rho_2$*, where is a refinement type variable.*

Definition 13. *A variable refinement type* refines *an ordinary type* τ *iff*

1. $\tau = t$ *and* $\rho = \alpha$ *or*
2. $\tau = \tau_1 \rightarrow \tau_2$, $\rho = \rho_1 \rightarrow \rho_2$, ρ_1 *refines* τ_1 *and* ρ_2 *refines* τ_2.

The aim of *pre-decoration*, defined below, is basically annotating expressions and environments with refinement type variables. The relationships between these variables are found by the type inference algorithm.

Definition 14. $\Gamma \rhd e^\rho$ *is a* pre-decoration *of* $\Gamma_\lambda \rhd e_\tau$ *(where* $\vdash_\lambda \Gamma_\lambda \rhd e_\tau$ *in typed lambda calculus) iff*

1. *For each* $x_\tau \epsilon \Gamma_\lambda$, $x^\rho \epsilon \Gamma$ *where* ρ *is a variable refinement 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 refinement type and* ρ_1 *refines* τ_1.

We want the pre-decoration to be disjoint to ensure it contains as much information as possible:

Definition 15. *A pre-decoration* $\Gamma \rhd e^\rho$ *is* disjoint *if all the occurring refinement type variables are distinct.*

Example 9. Consider $[g^{t_1 \rightarrow t_2}, x^{t_1}] \rhd (g^{t_1 \rightarrow t_2} x^{t_1})^{t_2}$, a typing in typed λ-calculus. $[g^{\alpha_1 \rightarrow \alpha_2}, x^{\alpha_1}] \rhd (g^{\alpha_1 \rightarrow \alpha_2} x^{\alpha_1})^{\alpha_2}$ and $[g^{\alpha_1 \rightarrow \alpha_2}, x^{\alpha_3}] \rhd (g^{\alpha_4 \rightarrow \alpha_5} x^{\alpha_6})^{\alpha_7}$ are two of its decorations. The second one is disjoint.

Definition 16. *The function annotate is used to make a pre-decoration from a typed* λ*-calculus typing. It is defined as*

$$
\begin{aligned}
annotate(\Gamma_\lambda \rhd e_\lambda) = \; & let\ \Gamma = a_3(\Gamma_\lambda)\ and\ e^\rho = a_2(e_\lambda) \\
& in \\
& \quad \Gamma \rhd e^\rho
\end{aligned}
$$

where

$$
\begin{aligned}
a_1(t) &= \alpha,\ where\ \alpha\ is\ a\ new\ refinement\ type\ variable \\
a_1(\tau_1 \rightarrow \tau_2) &= a_1(\tau_1) \rightarrow a_1(\tau_2) \\
\\
a_2(x^\tau) &= x^{a_1(\tau)} \\
a_2((\lambda x^{\tau_1}.e^{\tau_2})^{\tau_1 \rightarrow \tau_2}) &= (\lambda x^{a_1(\tau_1)} a_2(e^{\tau_2}).)^{a_1(\tau_1) \rightarrow a_1(\tau_2)} \\
a_2((e_1^{\tau_1} e_2^{\tau_2})^{\tau_3}) &= (a_2(e_1^{\tau_1}) a_2(e_2^{\tau_2}))^{a_1(\tau_3)} \\
\\
a_3(\emptyset) &= \emptyset \\
a_3(x^\tau :: \Gamma) &= x^{a_1(\tau)} :: a_3(\Gamma)
\end{aligned}
$$

Example 10. As an example, annotate$([g^{t_1 \rightarrow t_2}] \rhd (g^{t_1 \rightarrow t_2} x^{t_1})^{t_2}) = [g^{\alpha_1 \rightarrow \alpha_2}, x^{\alpha_3}] \rhd (g^{\alpha_4 \rightarrow \alpha_5} x^{\alpha_6})^{\alpha_7}$.

The following functions are used to collect the necessary constraint set in making the final typing IO-related.

Definition 17. *Definitions of mk_pos_IO_related() and mk_neg_IO_related() are given by mutual induction:*

1. $mk\_pos\_IO\_related(\alpha) = \emptyset$
   $mk\_pos\_IO\_related(\rho_1 \rightarrow \rho_2) =$
   $\{tail(\rho_1) \leq tail(\rho_2)\} \cup mk\_neg\_IO\_related(\rho_1) \cup mk\_pos\_IO\_related(\rho_2)$
2. $mk\_neg\_IO\_related(\alpha) = \emptyset$
   $mk\_neg\_IO\_related(\rho_1 \rightarrow \rho_2) =$
   $mk\_pos\_IO\_related(\rho_1) \cup mk\_neg\_IO\_related(\rho_2)$

Example 11. As examples, consider

$$\begin{aligned} \text{mk\_pos\_IO\_related}(\alpha_1 \rightarrow \alpha_2) &= \{\alpha_1 \leq \alpha_2\} \\ \text{mk\_neg\_IO\_related}((\alpha_1 \rightarrow \alpha_2) \rightarrow \alpha_3) &= \{\alpha_1 \leq \alpha_2\} \end{aligned}$$

Definition 18. *$BC()$ is a function that, given a pre-decoration, forms the constraint set necessary for the final typing to be correct in the refinement type system. It is defined as*

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

Example 12. As an example, consider

$$\begin{aligned} &\text{BC}([g^{\alpha_1 \rightarrow \alpha_2}, x^{\alpha_3}] \rhd (g^{\alpha_4 \rightarrow \alpha_5} x^{\alpha_6})^{\alpha_7}) = \\ &\qquad \{\alpha_1 \rightarrow \alpha_2 \leq \alpha_4 \rightarrow \alpha_5, \alpha_3 \leq \alpha_6, \alpha_6 \leq \alpha_4, \alpha_5 \leq \alpha_7\} \end{aligned}$$

Definition 19. *A substitution i* satisfies *a constraint set C, if for all constraints $(\rho_1 \leq \rho_2) \epsilon C$, $\vdash i(\rho_1) \leq i(\rho_2)$*

The following function is used to solve a special type of constraint set.

Definition 20. *$solve()$ is a function that takes a set of constraints as input and outputs a substitution of the refinement type variables. It is proven in Lemma 6 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.*

$solve(\{\alpha \leq L\} \cup C) = solve(C[\alpha := L]) o [\alpha := L]$
$solve(\{\rho_1 \rightarrow \rho_2 \leq \rho_3 \rightarrow \rho_4\} \cup C) = solve(\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C)$
$solve(C) = (\lambda\beta. \textit{if } \beta \epsilon fvars(C) \textit{ then } D)$,
if the above rules don't apply anymore

The following is an easy lemma that will later be used in the proofs.

Lemma 5. *If $\vdash \sigma_1 \leq \sigma_2$ and $\vdash \sigma_2 \leq \sigma_1$ then $\sigma_1 = \sigma_2$.*

Proof: Trivial induction on the structure of σ_i's. $\square$

The following three lemmas prove that the functions defined above really do what they are intended to do. They are used in the proof of Theorem 6, which, given a pre-decoration $\Gamma \rhd e^{\rho}$, provides a way to get a substitution i, such that $\vdash i(\Gamma) \rhd i(e^{\rho})$. This will be the basis of an algorithm for dead code elimination.

Lemma 6 shows that BC() works as it is intended to.

Lemma 6. *Let $BC(\Gamma \rhd e^{\rho}) = C$. For all substitutions i, if i satisfies C then $\vdash i(\Gamma) \rhd i(e^{\rho})$.*

Proof: By induction on the structure of e:

x: We have $\mathrm{BC}(\Gamma \rhd x^{\rho}) = \{\Gamma(x) \leq \rho\}$. Hence, we have $\vdash i(\Gamma(x)) \leq \rho$. So, we have the following proof:

$$\frac{\dfrac{}{i(\Gamma) \rhd x^{i(\Gamma(x))}} \qquad \vdash i(\Gamma(x)) \leq i(\rho)}{i(\Gamma) \rhd x^{i(\rho)}}$$

$(\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3 \to \rho_4}$:
Let $C = \mathrm{BC}(\Gamma \rhd (\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3 \to \rho_4}) = \mathrm{BC}(\Gamma, x^{\rho_1} \rhd e^{\rho_2}) \cup \{\rho_1 \to \rho_2 \leq \rho_3 \to \rho_4\}$ and assume i satisfies C. Then i satisfies $\mathrm{BC}(\Gamma, x^{\rho_1} \rhd e^{\rho_2})$ and it is true that $\vdash i(\rho_1 \to \rho_2) \leq i(\rho_3 \to \rho_4)$. By induction hypothesis, we have $\vdash i(\Gamma, x^{\rho_1}) \rhd i(e^{\rho_2})$. Hence, we have

$$\frac{\dfrac{\vdash i(\Gamma), x^{i(\rho_1)} \rhd i(e^{\rho_2})}{i(\Gamma) \rhd (\lambda x^{i(\rho_1)}.i(e^{\rho_2}))^{i(\rho_1) \to i(\rho_2)}} \qquad \vdash i(\rho_1) \to i(\rho_2) \leq i(\rho_3 \to \rho_4)}{i(\Gamma) \rhd (\lambda x^{i(\rho_1)}.i(e^{\rho_2}))^{i(\rho_3 \to \rho_4)}}$$

Since, $(\lambda x^{i(\rho_1)}.i(e^{\rho_2}))^{i(\rho_3 \to \rho_4)} = i((\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3 \to \rho_4})$, we have the required result.

$(e_1^{\rho_1 \to \rho_2} e_2^{\rho_1})^{\rho_3}$: Let $C = \mathrm{BC}(\Gamma \rhd (e_1^{\rho_1 \to \rho_2} e_2^{\rho_3})^{\rho_4}) = \mathrm{BC}(\Gamma \rhd e_1^{\rho_1 \to \rho_2}) \cup \mathrm{BC}(\Gamma \rhd e_2^{\rho_3}) \cup \{\rho_2 \leq \rho_4, \rho_3 \leq \rho_1, \rho_1 \leq \rho_3\}$ and i satisfies C. Then $\vdash i(\rho_2) \leq i(\rho_4)$ (1), $\vdash i(\rho_3) \leq i(\rho_1)$ and $\vdash i(\rho_1) \leq i(\rho_3)$. Using the last two assertions and the preceding lemma, we have $i(\rho_1) = i(\rho_3)$ (2). i satisfies C, hence it satisfies $\mathrm{BC}(\Gamma \rhd e_1^{\rho_1 \to \rho_2})$ and $\mathrm{BC}(\Gamma \rhd e_2^{\rho_3})$. Hence, by the induction hypothesis, $\vdash i(\Gamma) \rhd i(e_1^{\rho_1 \to \rho_2})$ and $\vdash i(\Gamma) \rhd i(e_2^{\rho_3})$. Since, $i(e_1^{\rho_1 \to \rho_2}) = i(e_1)^{i(\rho_1) \to i(\rho_2)}$ and $i(e_2^{\rho_3}) = i(e_2)^{i(\rho_3)}$, we have the following proof:

$$\frac{\dfrac{\vdash i(\Gamma) \rhd i(e_1)^{i(\rho_1) \to i(\rho_2)} \qquad \vdash i(\Gamma) \rhd i(e_2)^{i(\rho_1)}}{i(\Gamma) \rhd (i(e_1)^{i(\rho_1) \to i(\rho_2)} i(e_2)^{i(\rho_1)})^{i(\rho_2)}} \qquad \vdash i(\rho_2) \leq i(\rho_4)}{i(\Gamma) \rhd (i(e_1)^{i(\rho_1) \to i(\rho_2)} i(e_2)^{i(\rho_1)})^{i(\rho_4)}}$$

Since, by (2), $(i(e_1)^{i(\rho_1) \to i(\rho_2)} i(e_2)^{i(\rho_1)})^{i(\rho_4)} = i((e_1^{\rho_1 \to \rho_2} e_2^{\rho_3})^{\rho_4})$, we have $\vdash i(\Gamma) \rhd i((e_1^{\rho_1 \to \rho_2} e_2^{\rho_3})^{\rho_4})$. $\square$

Lemma 7 shows that solve() does what it is intended to do.

Lemma 7. *If C has no occurrence of D and i = solve(C), then i satisfies C.*

Proof: By induction on the complexity of the constraints in C and the size of C:

Base case: By the hypothesis, the constraints in C are either of the form $L \leq \alpha$ or $\alpha \leq \beta$. In both cases, it is obvious that if $i = (\lambda\beta.\text{if } \beta\epsilon\text{fvars}(C) \text{ then } D)$, then i satisfies C.

$C = \{\alpha \leq L\} \cup C'$**:** Let $i = \text{solve}(C) = \text{solve}(C'[\alpha := L]) o[\alpha := L]$ and $i' = \text{solve}(C'[\alpha := L])$. Hence, by the induction hypothesis, i' satisfies $C'[\alpha := L]$ (1).

By definition, $i(\alpha) = L$ and $i(L) = L$. Hence, $\vdash i(\alpha) \leq i(L)$, which implies that i satisfies $\{\alpha \leq L\}(2)$.

Assume $(\rho_1 \leq \rho_2)\epsilon C'$. By (1), we have $\vdash i'(\rho_1[\alpha := L]) \leq i'(\rho_2[\alpha := L])$. Thus, $\vdash (i' o[\alpha := L])(\rho_1) \leq (i' o[\alpha := L])(\rho_2)$ and hence $\vdash i(\rho_1) \leq i(\rho_2)$. This implies that i satisfies C' (3).

(2) and (3) imply that i satisfies $\{\alpha \leq L\} \cup C'$.

$C = \{\rho_1 \rightarrow \rho_2 \leq \rho_3 \rightarrow \rho_4\} \cup C'$**:** Let $i = \text{solve}(\{\rho_1 \rightarrow \rho_2 \leq \rho_3 \rightarrow \rho_4\} \cup C') =$ $\text{solve}(\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C')$. By induction hypothesis, i satisfies $\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C'$. Hence, i satisfies C' (1) and $\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\}$ (2). (2) implies $\vdash i(\rho_3) \leq i(\rho_1)$ and $\vdash i(\rho_2) \leq i(\rho_4)$. These imply $\vdash i(\rho_1 \rightarrow \rho_2) \leq i(\rho_3 \rightarrow \rho_4)$. So, i satisfies $\{\rho_1 \rightarrow \rho_2 \leq \rho_3 \rightarrow \rho_4\}$ (3). (1) and (3) give the required result.

□

Lemma 8 shows that mk_pos_IO_related() and mk_neg_IO_related() do what they are intended to do.

Lemma 8. *Let* $C = BC(\Gamma \rhd e^{\rho}) \cup mk\_pos\_IO\_related(\Gamma) \cup mk\_neg\_IO\_related(\rho) \cup \{tail(\rho) \leq L\}$. *For all substitutions i, if i satisfies C, then* $i(\Gamma) \rhd i(e^{\rho})$ *is IO-correct.*

Proof: Proof is straightforward. □

The previous lemmas culminate in the following theorem, which suggests an algorithm for dead code elimination.

Theorem 6. *If* $\Gamma \rhd e^{\rho}$ *is a disjoint pre-decoration and* $C = BC(\Gamma \rhd e^{\rho}) \cup mk\_pos\_IO\_related(\Gamma) \cup mk\_neg\_IO\_related(\rho) \cup \{tail(\rho) \leq L\}$ *and* $i = solve(C)$, *then* $\vdash i(\Gamma) \rhd i(e^{\rho})$ *and it is IO-correct.*

Proof: By Lemma 7, i satisfies C. Hence, it satisfies $\text{BC}(\Gamma \rhd e^{\rho})$. By Lemma 6, this implies $\vdash i(\Gamma) \rhd i(e^{\rho})$. The fact that it is IO-correct follows from Lemma 8. □

The following gives the definitions of the functions $TI()$ and $A()$. Given a typed λ-expression and an environment, $TI()$ does the type inference and outputs the original expression and environment decorated with refinement types, whereas $A()$ "sieves out" the useless parts from the original expression. We assume that the input to both are decorated with ordinary typed λ-calculus types. This can be easily achieved by extending the Hindley-Milner type inference algorithm $\mathcal{W}$ [20] to save the intermediate type information.

Definition 21.

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

Example 13. Let us give an example, that is simple, but necessarily does not show all the capability of the system.

Let $\Gamma_\lambda \rhd e_\lambda = [y^{t_2}, z^{t_1}] \rhd ((\lambda x^{t_1}.y^{t_2})^{t_1 \to t_2} z^{t_1})^{t_2}$. Hence,
$\Gamma \rhd e^\rho = \text{annotate}(\Gamma_\lambda \rhd e_\lambda) = [y^{\alpha_1}, z^{\alpha_2}] \rhd ((\lambda x^{\alpha_3}.y^{\alpha_4})^{\alpha_5 \to \alpha_6} z^{\alpha_7})^{\alpha_8}$,
$\text{BC}(\Gamma \rhd e^\rho) = \{\alpha_1 \leq \alpha_4, \alpha_2 \leq \alpha_7, \alpha_7 \leq \alpha_5, \alpha_6 \leq \alpha_8, \alpha_3 \to \alpha_4 \leq \alpha_5 \to \alpha_6\}$,
$\text{mk\_pos\_IO\_related}(\Gamma) = \emptyset$ and $\text{mk\_neg\_IO\_related}(\alpha_8) = \emptyset$,
$C = \{\alpha_1 \leq \alpha_4, \alpha_2 \leq \alpha_7, \alpha_7 \leq \alpha_5, \alpha_6 \leq \alpha_8, \alpha_3 \to \alpha_4 \leq \alpha_5 \to \alpha_6, \alpha_8 \leq L\}$,
$i = \text{solve}(C) = [\alpha_1, \alpha_4, \alpha_6, \alpha_8 := L; \alpha_2, \alpha_3, \alpha_5, \alpha_7 := D]$,
$i(e^\rho) = ((\lambda x^D.y^L)^{D \to L} z^D)^L$ and finally
$sieve(i(e^\rho)) = (\lambda x.y)\Box_t$.

The following theorem 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 it is intended to.

Theorem 7. *Assume* $\vdash_\lambda \Gamma_\lambda \rhd e_\lambda$ *and* $\epsilon = A(\Gamma_\lambda \rhd e_\lambda)$. *Then* $e_\lambda \Downarrow v_1$ *and* $\epsilon \Downarrow_c v_2$ *imply* $v_1 = v_2$.

Proof: Let $\Gamma \rhd e^\rho$, C and i be as defined in the algorithm $A()$. By Theorem 6, $\vdash i(\Gamma) \rhd i(e^\rho)$ and this is IO-related. Let $\epsilon = sieve(i(e^\rho)) = A(\Gamma_\lambda \rhd e_\lambda)$. By Theorem 5, $e_\lambda \Downarrow v_1$ and $\epsilon \Downarrow_c v_2$ imply $v_1 = v_2$, as required. □

The following lemmas are used in the proof of Theorem 8, which asserts that the algorithm $A()$ finds all the dead code expressible in the type system.

Lemma 9. *Let* $C = BC(\Gamma \rhd e^\rho)$. *If* $\vdash i(\Gamma) \rhd i(e^\rho)$, *then* i *satisfies* C.

Proof:

x: In this case, $C = \text{BC}(\Gamma \rhd x^\rho) = \{\Gamma(x) \leq \rho\}$ and $\vdash i(\Gamma) \rhd i(x^\rho)$. Consider the following proof:

$$
\dfrac{\dfrac{}{i(\Gamma) \rhd x^{i(\Gamma(x))}}}{\dfrac{\vdots \;\text{Appl. of SUB}}{i(\Gamma) \rhd x^{i(\rho)}}}
$$

Hence, we have $\vdash i(\Gamma(x)) \leq i(\rho)$. So, i satisfies C.

$(\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3\to\rho_4}$**:** We have $C = \mathrm{BC}(\Gamma, x^{\rho_1} \rhd e^{\rho_2}) \cup \{\rho_1\to\rho_2 \leq \rho_3\to\rho_4\}$. Hence, by the induction hypothesis, i satisfies $\mathrm{BC}(\Gamma, x^{\rho_1} \rhd e^{\rho_2})$ (1). By the assumption of the theorem, we have $\vdash i(\Gamma) \rhd i((\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3\to\rho_4})$. Hence, look at the proof:

$$\dfrac{\dfrac{\dfrac{i(\Gamma, x^{\rho_1}) \rhd i(e^{\rho_2})}{i(\Gamma) \rhd (\lambda x^{i(\rho_1)}.e^{i(\rho_2)})^{i(\rho_1)\to i(\rho_2)}}}{\vdots \text{ Appl. of SUB}}}{i(\Gamma) \rhd i((\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3\to\rho_4})}$$

From this proof, we have $\vdash i(\rho_1\to\rho_2) \leq i(\rho_3\to\rho_4)$. This and (1) above gives that i satisfies C.

$(e_1^{\rho_1\to\rho_2} e_2^{\rho_1})^{\rho_3}$**:** The proof is similar to the previous case. □

The following lemma is used in the one after it, which states formally that given a set of constraints, the algorithm solve() finds *the principal instantiation* of the refinement type variables, in some sense to be made clear in the lemma.

Lemma 10. *If i satisfies C and $i(\alpha) = L$, then i satisfies $C[L/\alpha]$.*

Proof: Trivial. □

Lemma 11. *If C does not contain D, $i = solve(C)$ and i_1 satisfies C, then for all refinement type variables occurring in C, $i(\beta) = L$ implies $i_1(\beta) = L$.*

Proof: By induction on the structure of the types occurring in C and the number of inequalities in C:

C **doesn't contain** L**:** Vacuously true.

$C = \{\alpha \leq L\} \cup C'$**:** Let $i = \mathrm{solve}(C'[\alpha := L]) o [\alpha := L]$ (1), $i' = \mathrm{solve}(C'[\alpha := L])$ (2) and assume i_1 satisfies C (3).
(3) implies i_1 satisfies C' and $\{\alpha \leq L\}$. Hence, by the previous lemma, i_1 satisfies $C'[\alpha := L]$.
By the induction hypothesis, we have
$\forall\beta \,\epsilon\, \mathrm{free\_var}(C'[\alpha := L]) \,.\, i'(\beta) = L \Rightarrow i_1(\beta) = L (4)$.
Since α is not in $\mathrm{free\_var}(C'[\alpha := L])$ and $i = i' o [\alpha := L]$, (4) implies
$\forall\beta \,\epsilon\, \mathrm{free\_var}(C'[\alpha := L]) \,.\, i(\beta) = L \Rightarrow i_1(\beta) = L (5)$.
From (3), we have $\vdash i_1(\alpha) \leq i_1(L) = L$. Hence, $i_1(\alpha) = L$. This with (5) implies
$\forall\beta \,\epsilon\, (\mathrm{free\_var}(C'[\alpha := L]) \cup \{\alpha\}) \,.\, i(\beta) = L \Rightarrow i_1(\beta) = L (6)$.
Since, $\mathrm{free\_var}(C' \cup \{\alpha \leq L\}) = \mathrm{free\_var}(C'[\alpha := L]) \cup \{\alpha\}$, we have
$\forall\beta \,\epsilon\, \mathrm{free\_var}(C) \,.\, i(\beta) = L \Rightarrow i_1(\beta) = L (6)$, as required.

$C = \{\rho_1\to\rho_2 \leq \rho_3\to\rho_4\} \cup C'$**:** Let $i = \mathrm{solve}(\{\rho_1\to\rho_2 \leq \rho_3\to\rho_4\} \cup C') =$ $\mathrm{solve}(\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C')$ (1) and assume i_1 satisfies $\{\rho_1\to\rho_2 \leq \rho_3\to\rho_4\} \cup C'$ (2). (2) implies that i_1 satisfies $\{\rho_3 \leq \rho_1, \rho_2 \leq \rho_4\} \cup C'$. Hence, by the induction hypothesis, we get the required result. □

Lemma 12. *If $\Gamma \rhd e^\rho$ is a disjoint pre-decoration and $C = BC(\Gamma \rhd e^\rho) \cup$ $\ldots \cup mk\_pos\_IO\_related(\Gamma) \cup mk\_neg\_IO\_related(\rho) \cup \{tail(\rho) \leq L\}$ and $i = solve(C)$, then for all i', s.t. $\vdash i'(\Gamma) \rhd i'(e^\rho)$ and $i'(\Gamma) \rhd i'(e^\rho)$ is IO-related, for all $e_1^{\rho_1}$ that are subexpressions of e^ρ,* useless$(i'(\rho_1))$ *implies* useless$(i(\rho_1))$ *i.e. $i'(e_1^{\rho_1})$ is dead implies $i(e_1^{\rho_1})$ is dead.*

Proof: Assume $\vdash i'(\Gamma) \rhd i'(e^\rho)$. For this to be true, i' must satisfy $\mathrm{BC}(\Gamma \rhd e^\rho)$, by Lemma 9. For $i'(\Gamma) \rhd i'(e^\rho)$ to be IO-correct, i' must satisfy mk_pos_IO_related$(\Gamma) \cup$ mk_neg_IO_related$(\rho) \cup \{tail(\rho) \leq L\}$. Hence, i' must satisfy C.
By Lemma 11, for all β occurring in C, $i(\beta) = L$ implies $i'(\beta) = L$ (1).
We have $useful(i(\rho_1)) \Rightarrow tail(i(\rho_1)) = L \Rightarrow i(tail(\rho_1)) = L \Rightarrow \langle \text{by (1)} \rangle\ i'(tail(\rho_1)) = L \Rightarrow tail(i'(\rho_1)) = L \Rightarrow useful(i'(\rho_1))$.
This proves that $useless(i'(\rho_1))$ implies $useless(i(\rho_1))$. □

Lemma 13. *For all decorations $\Gamma_c \rhd e^{\sigma_c}$ of e with constant refinement types and for all disjoint pre-decorations $\Gamma \rhd e^\rho$ of e with variable refinement types, there is a substitution i, s.t. $i(\Gamma \rhd e^\rho) = \Gamma_c \rhd e^{\sigma_c}$.*

Proof: Since $\Gamma \rhd e^\rho$ is disjoint, no variable appears twice. Hence, simply let $i(\alpha) =$ (the corresponding constant in $\Gamma_c \rhd e^{\sigma_c}$). □

Theorem 8. *Let $\Gamma \rhd e^\sigma = TI(\Gamma_\lambda \rhd e_\lambda)$. For all IO-related $\Gamma_c \rhd e^{\sigma_c}$, s.t. $\vdash \Gamma_c \rhd e^{\sigma_c}$, for all subexpressions ${e_1}^{\sigma'}$ of e^σ and the corresponding subexpressions ${e_1}^{\sigma'_c}$ of e^{σ_c},* useless(σ'_c) *implies* useless(σ'). *Hence, $A()$ finds the maximum amount of dead code expressible in the type system.*

Proof: The proof follows from Lemma 12 and 13. □

In other words, if it is expressible in the type system that a subexpression is dead code, TI() finds this out.

With this theorem, we have finished our presentation of a type inference based program analysis method for discovering dead code.

5 Extension to a Full-scale Programming Language

The extension to a full-scale typed functional programming language presents no difficulties. In this section, we discuss briefly issues related to the *let* construct, recursion and implementation.

The polymorphism via the *let* construct in ML [20] can be solved easily. In "let $x = e_1$ in e_2", the simple solution is to provide a different version of e_1 for each occurrence of x in e_2 and duplicating the constraints. When the refinement types for all of these instantiations of e_1 are known, they are unified to ensure that a subexpression of e_1 is useful, if one of its instantiations is useful. Naturally, this scheme can be made more efficient by saving and using the information gathered about previous instantiations.

In the preceding pages, we concentrated mainly on typed λ-calculus, because the problem of dead code elimination has already a well-established solution for first-order languages with recursion [1]. The main problem to be solved is posed by higher-order functions. If we want to add the *fix* operator to our language, there will be some minor additions, both to the algorithm and the proof. Below is an outline.

The changes to algorithm are as follows. we must extend BC() to:

$$\mathrm{BC}(\mathrm{fix}((f^{\rho_1}.e^{\rho_2})^{\rho_3})) = \{\rho_2 \leq \rho_1 \leq \rho_3\} \cup \mathrm{BC}(\Gamma, f^{\rho_1} \rhd e^{\rho_2})$$

We also have to make the assumption that the f in the environment is IO-correct. For this purpose, the following function is used to collect the necessary constraints:

$$\begin{aligned}\text{mk\_rec\_IO\_correct}(e) = {} & \text{if } e = x \text{ then } \emptyset \\ & \text{else if } (\lambda x^{\rho_1}.e^{\rho_2})^{\rho_3 \to \rho_4} \text{ then} \\ & \qquad \text{mk\_rec\_IO\_correct}(e^{\rho_2}) \\ & \text{else if } e = ({e_1}^{\rho_1} {e_2}^{\rho_2})^{\rho_3} \text{ then} \\ & \qquad \text{mk\_rec\_IO\_correct}({e_1}^{\rho_1}) \cup \text{mk\_rec\_IO\_correct}({e_2}^{\rho_2}) \\ & \text{else if } e = (\text{fix}(f^{\rho_1}.{e_1}^{\rho_2})^{\rho_3}) \text{ then} \\ & \qquad \text{mk\_pos\_IO\_related}(\rho_1) \cup \text{mk\_rec\_IO\_correct}(e^{\rho_2})\end{aligned}$$

The modification of the proof of Theorem 5 is as follows. Theorem 5 asserts that *"If ... and $e_1 \Downarrow v_1$ and $\epsilon_1 \Downarrow_c v_2$, then $v_1 = v_2$"*. So, basically we are trying to assert that *" e_1 and its dead code eliminated version ϵ_1 have the same value."* In the presence of *fix*, some legitimate expressions have no normal form. What we can do is to use the standard definition from operational semantics [27]. In operational semantics, types are put into two categories: *observable* and *non-observable* types. For example, in PCF, the observable types are the ground types *int* and *bool* [22]. Two expressions e_1 and e_2 are operationally equal, if for all closed contexts $C[\;]_\tau$ (where τ is an observable type), $C[e_1] \Downarrow v$ iff $C[e_2] \Downarrow v$. Note the similarity with the statement of Theorem 5. Since, we have Curry-style polymorhism,but still are working in a typed lambda calculus without constants, the definitions need to be slightly modified. Hence, the observable types (and ground types for that matter) are those, that are referred to, using a type variable t. A closed context $C[\;]$ is one, in which any free variable x of $C[\;]$ has a ground type. A value v is an expression variable x, with a ground type. After these modifications and the appropriate change to "sieve()", the proof works as before.

The implementation of our method would benefit from several programming tricks. The situation is similar to the implementation of algorithm $\mathcal{W}$ [20] in modern compilers. Algorithm $\mathcal{W}$ is conceptually clean, but modern compilers trade this with efficiency through programming tricks. Similar ideas can be employed in the integration of our method in a compiler.

6 Related Work

There has been a lot of work on dead code elimination and closely related program analysis problems. Some examples are [1, 17, 16, 9, 10, 18, 29]. On the other hand, type systems for various program analysis problems have been devised [13, 8, 3, 4, 6, 30]. The research presented in [3, 4, 6] specifically deals with the problem of dead code elimination. The papers [4] and [6] use similar type systems, that are extensions of simple subtypes. We will discuss [6], since it is an improvement over the rest and the methods they use are simpler and more intuitive. [6] introduces w-r-types $\mathcal{O}$ inductively as follows (using our notation): $D\epsilon\mathcal{O}$ and if σ_1 is a refinement type and $\sigma_2\epsilon\mathcal{O}$, then $(\sigma_1 \to \sigma_2)\epsilon\mathcal{O}$. Then, they introduce a typing rule that asserts that if $\sigma \epsilon \mathcal{O}$ and ρ and σ refine the same type, then $\rho \leq \sigma$. Because of this setup, [6] uses *guarded constraints* $\mathcal{G} \Rightarrow \mathcal{C}$, instead of conventional simple subtype constraints.

Our approach uses the simple subtype discipline [21] and nothing more. We encapture the flow information through the type system and make the necessary conservative approximations specific to the dead code problem through the concept of IO-correctness. We claim this seperation of flow information and problem specific data is crucial in using the methods in other program analysis problems easily. Having one framework helps compiler writers (or other users of program analysis) avoid multiple passes of the code for different analysis problems, using incompatible methods. We also believe that avoidance of ad hoc additions to the basic type system is important for conceptual clarity. As an example, we didn't need to introduce guarded constraints.

7 Conclusion

In this paper, we introduced a method for the detection and elimination of dead code in typed λ-calculus based languages. This is a problem, which arises in compiler research, optimization of modular systems, software engineering and automated theorem provers. Apart from solving the specific problem of dead code elimination, our method makes the flow information explicit and hence can be readily used for other program analysis problems. It has a clean and extensible conceptual basis, relying on well-understood concepts, like type system, type inference, simple subtypes, etc., and is specifically designed to operate smoothly on typed λ-calculus-based languages. One point to note is that our approach pinpoints the *dead* parts of a program. This is especially important for software engineering applications, where we may want to see the dependencies before deleting anything.

For future work, we aim to implement the system in the context of a distributed communication system *Ensemble* [7]. Layering of protocols offers well-known advantages, but leads to performance inefficiencies. In [7], optimization techniques in the context of a particular system *Ensemble* are introduced. These techniques depend critically on dead code elimination. *Ensemble* is written in *ML*, a typed functional programming language. Using the expression of its builders, "Ensemble benefits from a design which has tremendously improved performance, and the use of ML has been essential in being able to rapidly experiment and refine Ensemble's architecture ... " [7]. This work is part of a larger project for the optimization and verification of layered communication protocols in *Nuprl* [11, 12].

Our method can be used for other program analysis problems. Another direction we are following is extending this method to a general framework for program analysis in software engineering. You will find the details in our upcoming paper.

8 Acknowledgments

We would like to thank Robert Constable for leading this project and his encouragement and valuable advice throughout. We are also thankful to Gregory Morrisett, Jason Hickey, Chi-Chao Chang and Vijay Menon 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. M. Coppo, F. Damiani, and P. Giannini. Refinement Types for Program Analysis. In *SAS'96*, LNCS 1145, pages 143–158. Springer, 1996.
7. Hayden, M. and van Renesse, R. Optimizing layered communication protocols. Technical Report TR96-1613, Cornell University, 1996.
8. F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, 1992.
9. J. Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, chapter 5, pages 117–153. Addison-Wesley, 1990.
10. S.B. Jones and Metaéyer D. Le. Compile-time garbage collection by sharing analysis. In *Proceedings of the Fourth International Conference on FPCA*, pages 54–74, September 1989.
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. Kuo, T.M. and Mishra, P. Strictness analysis: a new perspective based on type inference. In *Functional Programming Languages and Computer Architecture.* ACM, 1989.
14. Leroy, X. . Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, Jan. 1992.
15. Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
16. Y.A. Liu. Dependence analysis for recursive data. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, May 1998.
17. Y.A. Liu and D. Stoller. Dead code elimination using program-based regular tree grammars. Technical report, Computer Science Department, Indiana University, November 1997.

18. Y.A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, pages 190–201, June 1995.
19. Y.A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.
20. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17):348–375, 1978.
21. J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, july 1991.
22. J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.
23. Morrisett, G. and Walker, D. and Crary, K and Glew, N. From System F to typed assembly language. Technical Report TR97-1651, Cornell University, 1997.
24. Y.G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, pages 116–127, June 1992.
25. C. Paulin-Mohring. Extracting F_ω's programs from proofs in second order λ-calculus. In ACM, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.
26. 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.
27. G.D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University Computer Science Department, 1981.
28. R. Hindley. The principal type schemes for an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
29. 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.
30. Solberg, K.L. and Nielson, H.R. and Nielson, F. Strictness and totality analysis. In *First International Static Analysis Symposium*, 1994.
31. Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.
32. 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.
33. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.