# Type-Based Useless Variable Elimination

Naoki Kobayashi
University of Tokyo
koba@yl.is.s.u-tokyo.ac.jp

## ABSTRACT

Useless variable elimination [25] is a transformation that eliminates variables whose values contribute nothing to the final outcome of a computation. We present a type-based method for useless variable elimination and prove its correctness. The algorithm is a surprisingly simple extension of the usual type reconstruction algorithm. Our method seems more attractive than other methods for useless variable elimination in several respects. First, it is simple, so that the proof of the correctness is clear and the method can be easily extended to deal with a polymorphic language. Second, it is efficient: it runs in time almost linear in the size of an input expression for a simply-typed $\lambda$-calculus, while Wand and Siveroni's 0CFA-based method may require a cubic time. Moreover, our transformation is *optimal* in a certain sense among those that preserve well-typedness, both for the simply-typed language and for an ML-style polymorphically-typed language. On the other hand, Wand and Siveroni's method is *not* optimal for the polymophically-typed language.

## 1. INTRODUCTION

### 1.1 Background

Useless variable elimination [25] (UVE, in short) is a transformation that eliminates variables whose values contribute nothing to the final outcome of a computation. For example,[1] we can transform the following program

```
let fun loop(a,bogus,j) =
  if (j>100) then a else loop(f(a,j),bogus+2,j+1)
in loop(a,3,1) end
```

into:

```
let fun loop(a,j) =
  if (j>100) then a else loop(f(a,j),j+1)
in loop(a,1) end
```

by eliminating the useless variable `bogus`.

This kind of useless variable is unlikely to appear in human-produced code, but it may appear in automatically generated code or in a result of various compiler optimizations such as constant propagation and partial evaluation. For example, consider a program fragment `fn (x,y)=>if b then x else y`. If $b$ is found to be always *true* by other program analysis, the program can be transformed into `fn (x,y)=>x`, making the variable `y` useless.

Shivers [25] first presented how to perform UVE using control flow information. Wand and Siveroni [28] recently reformalized Shivers' UVE and proved its correctness. Wand and Siveroni's algorithm is based on 0CFA [25], a context-insensitive control flow analysis, and hence it takes cubic time. The algorithm is somewhat complicated and the proof of its correctness is quite involved.

### 1.2 Our Proposal in This Paper

In this paper, we present a *type-based* UVE for an ML-style polymorphically-typed, call-by-value functional language. We formalize UVE as a problem of replacing as many subexpressions as possible with an empty value () so that the evaluation result of the whole expression does not change, and then cleaning up the resulting expression by eliminating unnecessary (). For example, the above program can be transformed into

```
let fun loop(a,(),j) =
  if (j>100) then a else loop(f(a,j),(),j+1)
in loop(a,(),1) end
```

by replacing the second argument of `loop` with `()`. We can then clean up the program to obtain the optimized program shown before, by eliminating `()`. This view of UVE as the two-step transformation clarifies the essence of UVE: we think the essence lies in the first translation. It also seems advantageous in that no special treatment is necessary for dealing with elimination of part of structured data argument.

Since the second transformation — cleaning up a program by eliminating `()` — seems fairly easy, we focus on the first transformation of replacing subexpressions with () without changing the evaluation result. We formalize rules for deriving a correct transformation as a simple extension of standard typing rules, and obtain an algorithm for finding an optimal transformation as a simple modification to a standard

[1] This example was taken from [28].

A corrected version of the summary in Proceedings of PEPM'00
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*PEPM* '00 Boston, Massachusett USA
Copyright 2000 ACM

type reconstruction algorithm. An important observation behind this formalization is that when some subexpressions are replaced with $()$, the evaluation result of the whole expression is unchanged as long as the typing of the whole expression is unchanged. (This observation basically follows from Reynolds' parametricity theorem [24].) Therefore, we can replace a subexpression with $()$ when the type of the subexpression can be assumed to be *unit* (the type of $()$) in the surrounding context.

We now explain how to obtain a transformation in more detail. What we essentially need to do is to perform type inference in a top-down [14] and on-demand manner: we just need to perform type inference for a subterm $N$ only when it is found from type inference of the surrounding context that $N$ must produce a non-empty value (because otherwise $N$ can be just replaced with $()$, whatever expression $N$ is). If some subterm $N$ remains unchecked, then $N$ can be replaced with an empty value $()$. For example, consider a term $(\lambda(x,y).x)(1,2)$ of integer type. By performing type inference in a top-down manner, we know that $\lambda(x,y).x$ must have type $\mathit{int} \times \alpha \rightarrow \mathit{int}$ and $(1,2)$ must have type $\mathit{int} \times \alpha$. The usual type inference then checks 2 against type $\alpha$ and obtains $\alpha = \mathit{int}$, but in our case, 2 is not checked because $\alpha$ can be instantiated to the type *unit* of an empty value without violating typing constraints of the surrounding context. We then obtain a term $(\lambda(x,()).x)(1,())$ by replacing a term of type $\alpha$ with $()$. It is then cleaned up as $(\lambda x.x)1$ by eliminating $()$.

## 1.3 Contributions

This is not the first work that studied the problem of UVE for (higher-order) functional languages: Wand and Siveroni [28] proposed a 0CFA-based method, and Fischbach and Hannan [4] recently, independently from us, proposed a type-based method in which types are annotated with information on whether values are needed or not. Damiani et al. [1; 2] also proposed a non-standard type system of PCF for a slightly different problem. Our contributions can be summarized as follows:

- Simple and elegant formalization: our type-based UVE is just a slight modification to the usual type reconstruction. So, it is easy to verify its correctness and to extend the method to deal with full-scale programming languages. Also, it would not be so hard to implement the algorithm, compared with Wand and Siveroni's method (although efficient implementation should require a fair amount of work just as that of 0CFA does).

- Efficiency: Wand and Siveroni's method costs cubic time, while ours costs only almost linear time for the simply-typed case. With polymorphism, the cost can be exponential in the worst case, but we believe the algorithm works well in practice (just as ML type inference does). Detailed analysis of the cost, which is missing in other works, is also one of our contributions.

- Treatment of polymorphism: Our method can elegantly deal with (ML-style) polymorphism, while other type-based methods [1; 2; 4] do not. Although it is mentioned in their papers that they can extend the method to deal with polymorphism, it does not seem so obvious to do so while preserving the optimality of their methods, because polymorphism brings more opportunities for UVE and UVE in turn brings more polymorphism, as will be demonstrated in Examples 3.4 and 3.5.

- Optimality of the method: Our method is optimal among any type-preserving UVE, that is, (i) any sound (i.e., semantics-preserving) and type-preserving transformation can be obtained by using our method, and (ii) there is indeed an algorithm to obtain an optimal transformation (that replaces more subexpressions with $()$ than any other transformation) by using our method. In order to avoid confusion, we often refer to the property (i) alone by the *completeness* of our method. As mentioned above, finding an optimal transformation in the presence of polymorphism is not so trivial.

## 1.4 Limitations

Our type-based UVE has the following limitations.

- Non-termination and side-effects: The transformation method sketched above does not preserve side effects (including non-termination). For example, consider an expression $(\lambda x.1)M$ and suppose that $M$ diverges. The expression is transformed into $(\lambda x.1)()$. In order to preserve side effects, we just need to perform an effect-analysis [26] and disallow the replacement of a side effecting expression. For simplicity, we first ignore the problem of side effects and then discuss a remedy of this later. Although the remedy is very simple, the optimality does not hold any more: If we must preserve divergence (in the call-by-value semantics), $(\lambda x.1)M$ can be replaced with $(\lambda x.1)()$ if and only if $M$ diverges. It is therefore undecidable whether or not a transformation is optimal.

- Possibility of more aggressive transformations: Our UVE is optimal only among the type-preserving transformations that replace subterms only with $()$. For some programs, there are better transformations (in the sense that they eliminate more useless variables) that are operationally-sound but non-type-preserving, or those that replace subterms with terms other than $()$. We discuss this point in more detail in Section 6.

## 1.5 Structure of the Rest of This Paper

The rest of this paper is structured as follows. Section 2 introduces the syntax and typing of the source language. Section 3 gives translation rules and shows their correctness and completeness. Section 4 describes how to find an optimal translation. Section 5 describes how to clean up the transformed program by removing $()$. Section 6 discusses related work in more detail and also discusses extensions to deal with side effects, equality types, etc. Section 7 concludes this paper. For the space restriction, we omit some definitions, detailed descriptions and analyses of the algorithm, and most of the proofs. They are found in the full version of this paper [13].

# 2. SYNTAX AND OPERATIONAL SEMANTICS OF THE LANGUAGE

In this section, we introduce the syntax and typing of an ML-style polymorphic, call-by-value functional language, which is used as the source and target language of UVE.

## 2.1 Terms

**Definition 2.1 [terms]:**
The syntax of terms is defined as follows:

$$\begin{array}{lcl} M \text{ (terms)} & ::= & () \mid n \mid x \mid M_1 + M_2 \\ & & \mid \lambda x.M \mid \mathbf{fix}(f, x, M) \\ & & \mid M_1 M_2 \mid \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \\ & & \mid \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \\ & & \mid \mathit{proj}_1(M) \mid \mathit{proj}_2(M) \mid (M_1, M_2) \\ V \text{ (values)} & ::= & () \mid n \mid \lambda x.M \mid (V_1, V_2) \end{array}$$

Here, $n$ ranges over integers.

$M_1 + M_2$ denotes the summation of integers, $\mathbf{fix}(f, x, M)$ denotes a recursive function $f$ such that $f = \lambda x.M$,[2] and $\mathit{proj}_i(M)$ denotes a pair projection. $\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3$ evaluates $M_2$ if $M_1$ is evaluated to 0, and evaluates $M_3$ otherwise. $()$ is a special constant: its semantics is the same as the $()$ in ML [16], but it plays a special role in the formalization of UVE in this paper.
Bound and free variables of $M$ are defined as usual: $f$ and $x$ are bound in $\mathbf{fix}(f, x, M)$ and $x$ is bound in $M$ of $\lambda x.M$ and $\mathbf{let}\ x = N\ \mathbf{in}\ M$. We assume that $\alpha$-conversion is implicitly performed so that all bound variables are different from each other and from free variables. We write $[N/x]M$ for a term obtained by substituting $N$ for all free occurrences of $x$ in $M$.

## 2.2 Operational Semantics

We define a natural semantics-style operational semantics by using the relation $M \Downarrow V$, which means that the term $M$ is evaluated to the value $V$. It is defined as the least relation closed under the rules given in Figure 1.

## 2.3 Typing

**Definition 2.2 [types]:** The sets of types and type schemes, ranged over by $\tau$ and $\sigma$ respectively, are given by the following syntax.

$$\begin{array}{lcl} \tau \text{ (types)} & ::= & \mathit{unit} \mid \mathit{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\ \sigma \text{ (type schemes)} & ::= & \tau \mid \forall \alpha.\sigma \end{array}$$

We write $FV(\sigma)$ and $FV(x_1 : \sigma_1, \dots, x_n : \sigma_n)$ for the set of free type variables (i.e., type variables not bound by $\forall$) in $\sigma$ and $x_1 : \sigma_1, \dots, x_n : \sigma_n$ respectively.

**Definition 2.3:** A typing relation $x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash M : \tau$ is the least relation closed under the rules given in Figure 2. (In the figure, $\overrightarrow{\alpha}$ stands for a sequence $\alpha_1, \dots, \alpha_n$, and $\forall \overrightarrow{\alpha}.\sigma$ stands for $\forall \alpha_1. \cdots \forall \alpha_n.\sigma$.)

# 3. USELESS VARIABLE ELIMINATION

We formalize UVE as a type-preserving, source-to-source transformation that replaces some subterms with an empty value $()$.
In Sections 3.1 and 3.2, we give rules for deriving such transformations and prove their correctness. Then, we show in Section 3.3 that they are optimal in the sense that any valid transformation can be derived from those rules. An algorithm for finding an optimal transformation is deferred until Section 4.

## 3.1 Translation Rules

We want to formalize rules for deriving from a typed term $\Gamma \vdash M : \tau$ a term $M'$ such that $M'$ is obtained by replacing some subterms of $M$ with $()$ and the result of evaluating $M'$ is the same as that of evaluating $M$. For that purpose, we introduce a relation $\Gamma \vdash M : \tau \Rightarrow M'$. Intuitively, it means that $M$ can be transformed into a term $M'$ that has type $\tau$ under $\Gamma$ by replacing some terms with $()$. It is defined as the least relation closed under the rules given in Figure 3. The key idea is embodied in the rule (Tr-Unit): It means that if we want some part of a term to produce a value of type *unit*, then we can just replace that part with $()$ *whatever it is*. More intuitively, it means that if we do not need information on the evaluation result of some part of a term, we can just replace it with an empty value $()$. The other rules are almost the same as the corresponding typing rules. Note that the translation rules are non-deterministic (in addition to the usual non-determinism on how generic type variables are instantiated in (Tr-Var)): for example, if $M = M_1 M_2$ and $\tau = \mathit{unit}$, then we can apply either (Tr-Unit) or (Tr-App). This is just for technical convenience: if we want to avoid it, we can just add the side condition $\tau_2 \neq ()$ to (Tr-App), etc.
A nice point about our type-based approach is that the consistency of transformation can be naturally maintained by using type information. Notice that replacing some useless terms with $()$ may require other parts to be modified accordingly. For example, suppose a function $f$ does not use the second argument. In order to change a function call $f(1, 2)$ into $f(1, ())$, we must replace any other calls $f(M_1, M_2)$ with $f(M_1, ())$. This is naturally taken care of by typing assumptions: By (Tr-App), $f(1, 2)$ can be transformed into $f(1, ())$ only when $\Gamma(f)$ has type $\mathit{int} \times \mathit{unit} \to \tau$. Because other parts must also be transformed under the same typing assumption, the second parameters of any other calls to $f$ are also replaced with $()$.

**Example 3.1:** Consider the following term:

$$M = (\lambda x.(\lambda z.\mathit{proj}_1(x) + 1)(\mathit{proj}_2(x) + 1))(1, 2)$$

It is transformed into:

$$M' = (\lambda x.(\lambda z.\mathit{proj}_1(x) + 1)())(1, ())$$

(which can be further simplified to $(\lambda x.x + 1)1$ by removing $()$) by the derivation in Figure 4. In the figure, $N$ denotes $\mathit{proj}_1(x) + 1$ and $\Gamma$ denotes $x : \mathit{int} \times \mathit{unit}$.

**Example 3.2:** The following program taken from [28]:

```
let fun f1(x,y)=x
    fun f2(x,y)=x+x
    fun f3(x,y)=y
    val g = if p(a,b) then f1 else f2
    val h = if q(a,b) then f1 else f3
in g(x, h) end
```

can be transformed into:

[2] We do not use the general fixpoint operator because our language is call-by-value.

$$() \Downarrow () \quad \text{(E-Unit)}$$

$$n \Downarrow n \quad \text{(E-Int)}$$

$$\frac{M_1 \Downarrow n_1 \qquad M_2 \Downarrow n_2}{M_1 + M_2 \Downarrow n_1 + n_2} \quad \text{(E-Add)}$$

$$\lambda x.M \Downarrow \lambda x.M \quad \text{(E-Abs)}$$

$$\mathbf{fix}(f, x, M) \Downarrow \lambda x.[\mathbf{fix}(f, x, M)/f]M \quad \text{(E-Fix)}$$

$$\frac{M_1 \Downarrow \lambda x.M \qquad M_2 \Downarrow V_2 \qquad [V_2/x]M \Downarrow V}{M_1 M_2 \Downarrow V} \quad \text{(E-App)}$$

$$\frac{M_1 \Downarrow V_1 \qquad [V_1/x]M_2 \Downarrow V_2}{\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \Downarrow V_2} \quad \text{(E-Let)}$$

$$\frac{M_1 \Downarrow 0 \qquad M_2 \Downarrow V}{\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ \Downarrow V} \quad \text{(E-IfT)}$$

$$\frac{M_1 \Downarrow n \qquad n \neq 0 \qquad M_3 \Downarrow V}{\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ \Downarrow V} \quad \text{(E-IfF)}$$

$$\frac{M \Downarrow (V_1, V_2)}{proj_i(M) \Downarrow V_i} \quad \text{(E-Proj)}$$

$$\frac{M_1 \Downarrow V_1 \qquad M_2 \Downarrow V_2}{(M_1, M_2) \Downarrow (V_1, V_2)} \quad \text{(E-Pair)}$$

Figure 1: Operational Semantics

$$\Gamma \vdash () : unit \quad \text{(T-Unit)}$$

$$\Gamma, x : \forall \overrightarrow{\alpha}.\tau \vdash x : [\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau \quad \text{(T-Var)}$$

$$\Gamma \vdash n : int \quad \text{(T-Int)}$$

$$\frac{\Gamma \vdash M_1 : int \qquad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 + M_2 : int} \quad \text{(T-Add)}$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \mathbf{fix}(f, x, M) : \tau_1 \to \tau_2} \quad \text{(T-Fix)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M_2 : \tau_2 \qquad \{\overrightarrow{\alpha}\} = FV(\tau_1) \backslash FV(\Gamma)}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2} \quad \text{(T-Let)}$$

$$\frac{\Gamma \vdash M_1 : int \qquad \Gamma \vdash M_2 : \tau \qquad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ : \tau} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash proj_i(M) : \tau_i} \quad \text{(T-Proj)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2} \quad \text{(T-Pair)}$$

Figure 2: Typing Rules

```
let fun f1(x)=x
    fun f2(x)=x+x
    val g = if p(a,b) then f1 else f2
in g(x) end
```

(which is the same result as given in [28]) by applying our method and then removing (). See the full version [13] for intermediate steps of this translation.

**Example 3.3:** We can eliminate not only top-level parameters of functions, but also a part of structured data. Consider the following program:

```
let fun f(x,(y,z))=x+z in f(1,(2,3)) end
```

By assigning type $int \times (unit \times int) \to int$ to `f`, we have

```
let fun f(x,((),z))=x+z in f(1,((),3)) end
```

In the next example, let-polymorphism plays an important role.

**Example 3.4:** Consider the following program:

```
let fun f(g,x)=g(x)
in
  f(fn (x,y)=>x,(1, 2))+f(fn (x,y)=>x+y,(2,3))
end
```

This is expressed in our language by the following term:

$$\mathbf{let}\ f = \lambda u.proj_1(u)(proj_2(u))\ \mathbf{in}$$
$$f(\lambda v.proj_1(v), (1, 2)) + f(\lambda v.proj_1(v) + proj_2(v), (2, 3))$$

We can transform it into

$$\mathbf{let}\ f = \lambda u.proj_1(u)(proj_2(u))\ \mathbf{in}$$
$$f(\lambda v.proj_1(v), (1, ())) + f(\lambda v.proj_1(v) + proj_2(v), (2, 3))$$

by assigning type $\forall \alpha, \beta.(((\alpha \to \beta) \times \alpha) \to \beta)$ to $f$. By eliminating the unnecessary (), we obtain the following program:

```
let fun f(g,x)=g(x)
in f(fn x=>x,1)+f(fn (x, y)=>x+y,(2,3)) end
```

Notice that this translation cannot be performed without let-polymorphism. It is indeed invalid without polymorphism because `f` is used polymorphically in the translated program.

The above example indicates that polymorphism provides more opportunities for UVE. The following example indicates that UVE in turn brings more polymorphism.

**Example 3.5:** The function `f` defined by:

```
fun f(x) = let val y = fst(x) in x end
```

$$\Gamma \vdash M : \mathit{unit} \Rightarrow () \quad (\textsc{Tr-Unit})$$

$$\Gamma, x : \forall\overrightarrow{\alpha}.\tau \vdash x : [\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau \Rightarrow x \quad (\textsc{Tr-Var})$$

$$\Gamma \vdash n : \mathit{int} \Rightarrow n \quad (\textsc{Tr-Int})$$

$$\frac{\Gamma \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \mathit{int} \Rightarrow M_2'}{\Gamma \vdash M_1 + M_2 : \mathit{int} \Rightarrow M_1' + M_2'} \quad (\textsc{Tr-Add})$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2 \Rightarrow M'}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2 \Rightarrow \lambda x.M'} \quad (\textsc{Tr-Abs})$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash M : \tau_2 \Rightarrow M'}{\Gamma \vdash \mathbf{fix}(f, x, M) : \tau_1 \to \tau_2 \Rightarrow \mathbf{fix}(f, x, M')} \quad (\textsc{Tr-Fix})$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \tau_1 \Rightarrow M_2'}{\Gamma \vdash M_1 M_2 : \tau_2 \Rightarrow M_1' M_2'} \quad (\textsc{Tr-App})$$

$$\frac{\Gamma \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \tau \Rightarrow M_2' \qquad \Gamma \vdash M_3 : \tau \Rightarrow M_3'}{\Gamma \vdash \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ : \tau \Rightarrow \mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'} \quad (\textsc{Tr-If})$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2 \Rightarrow M'}{\Gamma \vdash \mathit{proj}_i(M) : \tau_i \Rightarrow \mathit{proj}_i(M')} \quad (\textsc{Tr-Proj})$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2 \Rightarrow (M_1', M_2')} \quad (\textsc{Tr-Pair})$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma, x : \forall\overrightarrow{\alpha}.\tau_1 \vdash M_2 : \tau_2 \Rightarrow M_2' \qquad \{\overrightarrow{\alpha}\} = FV(\tau_1) \backslash FV(\Gamma)}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2 \Rightarrow \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2'} \quad (\textsc{Tr-Let})$$

Figure 3: Translation Rules

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, z : \mathit{unit} \vdash N : \mathit{int} \Rightarrow N}}{\Gamma \vdash \lambda z.N : \mathit{unit} \to \mathit{int} \Rightarrow \lambda z.N} \quad \Gamma \vdash \mathit{proj}_2(x) + 1 : \mathit{unit} \Rightarrow ()}{\Gamma \vdash (\lambda z.N)(\mathit{proj}_2(x) + 1) : \mathit{int} \Rightarrow (\lambda z.N)()}\ (\textsc{Tr-App})}{\emptyset \vdash \lambda x.(\lambda z.N)(\mathit{proj}_2(x) + 1) : \mathit{int} \times \mathit{unit} \to \mathit{int} \Rightarrow \lambda x.((\lambda z.N)())}\ (\textsc{Tr-Abs}) \quad \dfrac{\emptyset \vdash 1 : \mathit{int} \Rightarrow 1 \quad \emptyset \vdash 2 : \mathit{unit} \Rightarrow ()}{\emptyset \vdash (1, 2) : \mathit{int} \times \mathit{unit} \Rightarrow (1, ())}\ (\textsc{Tr-Pair})}{\emptyset \vdash M : \mathit{int} \Rightarrow M'}\ (\textsc{Tr-App})$$

Figure 4: An Example of Translation Derivation

has a type scheme $\forall\alpha, \beta.(\alpha \times \beta \to \alpha \times \beta)$, but it has a more general type scheme $\forall\alpha.(\alpha \to \alpha)$ after it is transformed into:

```
fun f(x) = x
```

## 3.2 Correctness

Correctness of our type-based UVE is stated as the three theorems given below. Theorem 3.6 says that there is at least one valid translation for any well-typed term. Theorem 3.7 means that the output of the translation is a well-typed term. Theorem 3.9 means that the translated term evaluates to the same value as a source term. Note that the transformation may not preserve divergence. See Section 6 for a remedy of this.

**Theorem 3.6:** If $\Gamma \vdash M : \tau$, then $\Gamma \vdash M : \tau \Rightarrow M'$ for some $M'$.

PROOF. Because $\Gamma \vdash M : \tau \Rightarrow M$ always holds. $\square$

**Theorem 3.7 [Well-typedness of translated terms]:** If $\Gamma \vdash M : \tau \Rightarrow M'$, then $\Gamma \vdash M' : \tau$.

PROOF. Straightforward induction on the derivation of $\Gamma \vdash M : \tau \Rightarrow M'$. $\square$

We use a kind of contextual equivalence to show that the translated term evaluates to the same value as a source term. The idea of the contextual equivalence is to put two terms into an arbitrary well-typed context producing an integer, and to compare the result; if two terms cannot be distinguished by any contexts, they are considered equivalent. The theorem follows after the definition of the contexts.

**Definition 3.8 [contexts]:** A context is an expression obtained from a term by replacing one subterm with $[\ ]$. Let $C[\ ]$ be a context. We write $C[M]$ for a term obtained by replacing $[\ ]$ with $M$.

**Theorem 3.9 [Contextual equivalence]:** [3] Suppose $\Gamma \vdash M : \tau$ and $\Gamma \vdash M : \tau \Rightarrow M'$. Suppose also that $C[\ ]$ is a context. If $C[M] \Downarrow n$ holds and $\vdash C[\ ] : \mathit{int}$ is derivable from $\Gamma \vdash [\ ] : \tau$, then $C[M'] \Downarrow n$.

This is obtained as a corollary of the following lemma (Lemma 3.10), which means that a transformed term is evaluated to a value obtained by transforming the evaluation result of the original term.

**Lemma 3.10:** If $x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau \Rightarrow M'$, $\vdash V_i : \tau_i$ for each $i \in \{1, \ldots, n\}$, and $[V_1/x_1, \ldots, V_n/x_n]M \Downarrow V$, then $[V_1/x_1, \ldots, V_n/x_n]M' \Downarrow V'$ and $\vdash V : \tau \Rightarrow V'$ for some $V'$.

[3] Since this theorem says nothing about the case where $M$ diverges, it might be better to say "contextual approximation."

The above lemma can be proved by fairly straightforward induction on derivation of $[V_1/x_1, \dots, V_n/x_n]M \Downarrow V$ (see [13]). Using this lemma, we can prove Theorem 3.9 as follows.

PROOF OF THEOREM 3.9. By translation rules and the assumptions, we have $\vdash C[M] : int \Rightarrow C[M']$. So, we have $C[M'] \Downarrow n'$ and $\vdash n : int \Rightarrow n'$ by Lemma 3.10. By translation rules, it must be the case that $n = n'$. $\square$

In the above theorem, the correctness is stated based on the operational semantics. The following reasoning based on denotational semantics is very informal but may be more intuitive. Think of a cpo-based denotational semantics (for example, see [17]) for a lazy functional language and consider a cpo $\leq$ that satisfies $\perp \leq x$ where $\perp$ represents (), divergence or an error, and $x$ is any other element of the cpo. If $M'$ is a term obtained by replacing some subterms of $M$ with (), then, the denotation $M'^*$ of $M'$ is less than $M^*$ (intuitively because $M'$ yields less information than $M$). Now, suppose $M$ is evaluated to an integer $n$ in the call-by-value semantics. Then, $M$ is evaluated to $n$ also in the call-by-need semantics, and hence $M^* = n$. So, $M'^*$ is either $\perp$ or $n$. If $M'^* = \perp$, then $M'$ evaluates to (), diverges, or causes an error in the call-by-need semantics, and hence also in the call-by-value semantics. However, this is not the case because $M'$ is a well-typed term and because $M$ must diverge if $M'$ diverges by the construction of $M'$. Therefore, $M'$ must be evaluated to $n$ in the call-by-value semantics.

## 3.3 Optimality of the Translation

We formalized UVE as a transformation that replaces some subterms with () while preserving the evaluation result. It is easy to check that our type-based UVE is complete in the sense that it can derive any type-preserving UVE transformation. The algorithm described in Section 4 finds the optimal transformation (that replaces more subterms with () than any other transformations) among transformations obtained by using our type-based method. So, from the completeness result (Theorem 3.12 below), we know that the algorithm finds the optimal transformation among *any* (i.e., not limited to those obtained by our type-based method) type-preserving UVE transformations.

First, we introduce the relation $M \succeq M'$, meaning that $M'$ is obtained from $M$ by replacing subterms with ().

**Definition 3.11:** A binary relation $\succeq$ on terms is the least relation satisfying the following rules:

$M \succeq ()$
$n \succeq n$
$x \succeq x$
$M + N \succeq M' + N'$ if $M \succeq M'$ and $N \succeq N'$
$\lambda x.M \succeq \lambda x.M'$ if $M \succeq M'$
$\mathbf{fix}(f, x, M) \succeq \mathbf{fix}(f, x, M')$ if $M \succeq M'$
$MN \succeq M'N'$ if $M \succeq M'$ and $N \succeq N'$
$\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \succeq \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2'$
    if $M_1 \succeq M_1'$ and $M_2 \succeq M_2'$
$\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \succeq \mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'$
    if $M_1 \succeq M_1'$, $M_2 \succeq M_2'$, and $M_3 \succeq M_3'$
$(M, N) \succeq (M', N')$ if $M \succeq M'$ and $N \succeq N'$
$proj_i(M) \succeq proj_i(M')$ if $M \succeq M'$

The following theorem means that any valid transformation can be obtained by using our type-based transformation.

**Theorem 3.12 [completeness]:** If $\Gamma \vdash M : \tau$, $\Gamma \vdash M' : \tau$, and $M \succeq M'$, then $\Gamma \vdash M : \tau \Rightarrow M'$

**Remark 3.13:** The above theorem does not exclude the possibility of a better transformation that does more than merely replacing subterms with (). For example, the program:

```
let fun f(x, y) = if true then x else y
in f(1, 2) end
```

can be simplified to: `let fun f(x) = x in f(1) end`. We regard this as a combination of UVE and other transformations. The program can be first transformed into `let fun f(x, y) = x in f(1, 2) end` by using the equality **if** *true* **then** $M_1$ **else** $M_2 = M_1$ and then UVE can be applied to it (2 is replaced with () and then the unnecessary () is eliminated). See Section 6 for discussions of the possibility of other aggressive transformations.

# 4. ALGORITHM FOR FINDING AN OPTIMAL TRANSFORMATION

In this section, we briefly sketch an algorithm for finding an optimal translation and discuss its computational cost. In the monomorphic case (i.e., without let-polymorphism), an optimal translation is found in an almost linear time. In the polymorphic case, the algorithm costs exponential time in the worst case, but we believe that it works well in practice. Please refer to the full paper [13] for details of the algorithm and a proof of its correctness.

## 4.1 Monomorphic Case

An input is a typed term $\Gamma \vdash M : \tau$. We assume that the interface of a program outside $M$ cannot be changed. Therefore, the goal is to find the least (w.r.t. $\succeq$) $M'$ such that $\Gamma \vdash M : \tau \Rightarrow M'$.

By the translation rules, in order to find an optimal translation of a typed term $\Gamma \vdash M : \tau$, it is sufficient to find a derivation $\Gamma \vdash M : \tau \Rightarrow M'$ that uses (TR-UNIT) as much as possible. Basically, we can obtain such an optimal derivation by performing type inference *on-demand*. By (TR-UNIT), a subterm $N$ can be replaced with () as long as it can have type *unit* in the translated term. So, we need to solve type constraints on $N$ only when it is found that $N$ must be transformed into a term of type other than *unit*.

We use an algorithm similar to the linear-time type reconstruction algorithm for the simply-typed $\lambda$-calculus (see Proposition 3.3 of [11]). By introducing a type variable $\alpha_N$ representing the type of each subterm $N$ and a type variable $\beta_x$ representing the type of each variable bound in type environments, we can construct a *template* of a derivation tree, each node of which is of the form $x_1 : \beta_{x_1}, \dots, x_n : \beta_{x_n} \vdash N : \alpha_N \Rightarrow N'$. For example, for a typed term $\emptyset \vdash (\lambda x.proj_1(x))(1, 2) : int$, we can construct the template in Figure 5.

If we instantiate type variables so that certain conditions on type variables are met, we can obtain a real derivation tree by replacing subtrees whose conclusion is of the form $\Delta \vdash N : unit \Rightarrow N'$ with an instance of the axiom $\Delta \vdash N : unit \Rightarrow ()$. For the above example, such conditions are

$$\cfrac{\cfrac{\cfrac{x:\beta_x \vdash x:\alpha_x \Rightarrow N_7}{x:\beta_x \vdash proj_1(x):\alpha_{proj_1(x)} \Rightarrow N_6}\ (\text{Tr-Proj})}{\emptyset \vdash \lambda x.proj_1(x):\alpha_{\lambda x.proj_1(x)} \Rightarrow N_2}\ (\text{Tr-Abs}) \qquad \cfrac{\emptyset \vdash 1:\alpha_1 \Rightarrow N_4 \qquad \emptyset \vdash 2:\alpha_2 \Rightarrow N_5}{\emptyset \vdash (1,2):\alpha_{(1,2)} \Rightarrow N_3}\ (\text{Tr-Pair})}{\emptyset \vdash (\lambda x.proj_1(x))(1,2):\alpha_{(\lambda x.proj_1(x))(1,2)} \Rightarrow N_1}\ (\text{Tr-App})$$

Figure 5: A Template for Translation Derivation

expressed as:

$$\begin{array}{l}
(\alpha_{(\lambda x.proj_1(x))(1,2)} \neq unit) \Rightarrow \\
\quad \{\alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \rightarrow \alpha_{(\lambda x.proj_1(x))(1,2)}\} \\
(\alpha_{\lambda x.proj_1(x)} \neq unit) \Rightarrow \{\alpha_{\lambda x.proj_1(x)} = \beta_x \rightarrow \alpha_{proj_1(x)}\} \\
(\alpha_{proj_1(x)} \neq unit) \Rightarrow \{\alpha_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\} \\
\ldots
\end{array}$$

In general, for each subterm $N$, one constraint of the form $(\alpha_N \neq unit) \Rightarrow E_N$ is generated. $E_N$ is given by:

$$\begin{array}{l}
E_x = \{\alpha_x = \beta_x\} \\
E_n = \{\alpha_n = int\} \\
E_{M_1+M_2} = \{\alpha_{M_1+M_2} = int, \alpha_{M_1} = int, \alpha_{M_2} = int\} \\
E_{\lambda x.N} = \{\alpha_{\lambda x.N} = \beta_x \rightarrow \alpha_N\} \\
E_{\mathbf{fix}(f,x,N)} = \{\alpha_{\mathbf{fix}(f,x,N)} = \beta_x \rightarrow \alpha_N, \beta_f = \beta_x \rightarrow \alpha_N\} \\
E_{M_1 M_2} = \{\alpha_{M_1} = \alpha_{M_2} \rightarrow \alpha_{M_1 M_2}\} \\
E_{\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3} = \\
\quad \{\alpha_{M_1} = int, \alpha_{M_2} = \alpha_{M_3} = \alpha_{\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3}\} \\
E(M_1, M_2) = \{\alpha_{(M_1,M_2)} = \alpha_{M_1} \times \alpha_{M_2}\} \\
E_{proj_1(N)} = \{\alpha_N = \alpha_{proj_1(N)} \times \gamma_{proj_1(N)}\} \\
E_{proj_2(N)} = \{\alpha_N = \gamma_{proj_2(N)} \times \alpha_{proj_2(N)}\}
\end{array}$$

Therefore, if we want to obtain a derivation of the form $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \Rightarrow M'$, we can first solve equality constraints $\{\tau_1 = \beta_{x_1}, \ldots, \tau_n = \beta_{x_n}, \tau = \alpha_M\}$; then when it is found that some $\alpha_N$ cannot be instantiated with *unit*, we can add equality constraints $E_N$ and solve them. Finally, when it is found that no more constraints need to be added, we can instantiate remaining type variables with *unit*, and obtain a derivation of $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \Rightarrow M'$. It should be clear that the obtained derivation uses (Tr-Unit) as much as possible, and hence the translation is optimal.
Because the number of equality constraints to be solved is at most linear in the size of the input program, the translation algorithm runs in $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackerman function. (Because equality constraints are added lazily, we cannot use exactly the same unification algorithm as Paterson and Wegman's linear-time one [22], but we can still solve equality constraints in an almost linear time.)

## 4.2 Polymorphic Case

The basic idea of the algorithm is the same as in the monomorphic case: type inference for each subterm is performed *on-demand*, only when it is found that its type must not be *unit*. The only difference from the monomorphic case is that the unification constraints in $E_N$ above are replaced by semi-unification constraints [12] of the form $\tau_1 < \tau_2$ ($\tau_1 < \tau_2$ is defined to hold if and only if there exists a substitution $\theta$ such that $\theta\tau_1 = \tau_2$).
In order to obtain $E_N$, we can use Henglein's type system based on semi-unification constraints [9]. His type system has been developed for performing type inference in the presence of polymorphic recursion. In his polymorphic type system, a type judgment is of the form $\Gamma; \tau_1 \vdash M : \tau_2$ and only monotypes can appear in $\Gamma$; instead, he keeps in the additional type parameter $\tau_1$ information on which type variables cannot be used polymorphically. So, the judgment $x : \forall\alpha.(\alpha \rightarrow \beta) \vdash M : int$ is expressed as $x : \alpha \rightarrow \beta; \beta \vdash M : int$. Following his type system, we can reformalize translation rules as follows:

$$\frac{\tau \times \tau_1 < \tau' \times \tau_1}{\Gamma, x : \tau; \tau_1 \vdash x : \tau' \Rightarrow x}$$

$$\frac{\Gamma, x : \tau_1; \tau \times \tau_1 \vdash M : \tau_2 \Rightarrow M'}{\Gamma; \tau \vdash \lambda x.M : \tau_1 \rightarrow \tau_2 \Rightarrow \lambda x.M'}$$

$$\frac{\Gamma; \tau \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma, x : \tau_1; \tau \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma; \tau \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2 \Rightarrow \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2'}$$

The first rule allows the type variables in $\tau$ except for those appearing in $\tau_1$ to be instantiated. The second rule disallows $x$ to be used polymorphically by recording its type $\tau_1$ in the righthand side of ";". On the other hand, the third rule does not record the type of $x$, so the let-bound variable $x$ can be used polymorphically. After semi-unification constraints are extracted from these rules, we can solve them on-demand by using Henglein's graph-based semi-unification algorithm [9]. The worst-case cost in the polymorphic case is exponential in the size of an input, as for the ordinary type reconstruction problem for ML [11]. We think, however, that the algorithm works well for realistic, well-typed programs: Henglein [9] has shown that his type reconstruction algorithm runs in time polynomial in the size of the *explicitly typed* program (which implies that type inference costs exponential time only if the size of inferred types is exponential in the size of the original program). We think that performing UVE after the usual type reconstruction is likely to just double the cost of performing only the usual type reconstruction.

# 5. ELIMINATION OF USELESS PROJECTIONS AND NULL TUPLES

This section briefly explains the second step of UVE: how to clean up translated programs by eliminating useless pairs constructors, projections and (). For simplicity, we first deal with the monomorphic case.
Elimination of useless projections and () can be again formalized as a type-based transformation. Assume that a typed term $\Gamma \vdash M : \tau$ is given. Without loss of generality, we can assume that $\Gamma$ and $\tau$ do not contain type *unit*: otherwise, we can rename the () generated by UVE and its type *unit* so that they can be distinguished from those that were present in the source program. Figure 6 shows some key rules for eliminating the () and unnecessary projections and pairs from a typed term $\Gamma \vdash M : \tau$. In the figure, $empty(\tau)$ is true if and only if $\tau$ is composed only of *unit* and $\times$. (In this case, we need not perform type inference

any more, since the type of each subexpression is known after the first step.)

Elimination of useless $\lambda$-abstractions and applications on () (like $\lambda x : unit.M$ and $M()$) is not so trivial (except for the case where they appear in the form $(\lambda x.M)()$), because replacing $\lambda x : unit.M$ with $M$ may increase the cost of evaluation. One solution is to classify the function types into those of functions whose bodies are values and those of functions whose bodies may not be values. We can then perform type inference and replace $\lambda x.M$ with $M$ and $N()$ with $N$ only when $\lambda x.M$ and $N$ has types of the former class.

In the polymorphic case, not all pairs of the form $((), M)$ can be replaced with $M$. For example, consider the following program:

```
let fun f(x, y) = (y, x)
in #1(f((), 1))+#2(f(2, ())) end
```

If we choose the boxed representation for the arguments `x` and `y` of the polymorphic function `f`, then the call `f((), 1)` cannot be replaced with `f(1)`. This problem can also be dealt with based on type information. We can divide *unit* into one that can be substituted for a polymorphic type variable and the other that cannot, and eliminate only () of the latter type.

# 6. DISCUSSION

## 6.1 Further Extensions

*Equality type.* In ML, `fn (x,y)=>x=y` is assigned a type `''a *''a -> bool`, where `''a` is an equality type variable. Since it can be instantiated with *unit*, naive application of our method wrongly transforms `(fn (x,y)=>x=y)(1,2)` into `(fn (x, y)=>()=())((), ())`. To avoid this, it is sufficient to divide *unit* into two types $unit^0$, which cannot be substituted for an equality type variable, and $unit^\omega$, which may be substituted for an equality type variable. Then, we can restrict the rule (Tr-Unit) as follows:

$$\Gamma \vdash M : unit^0 \Rightarrow () \qquad \text{(Tr-Unit)}$$

*Preservation of non-termination and side effects.* Because our translation replaces whatever expression that can have type *unit* with (), it does not preserve non-termination and side effects. For example,

```
let fun f(x, y) =
    if x<0 then () else (print x;f(x-1, y+1))
in f(2, 3) end
```

is transformed into `()`.

To avoid this, it suffices to change the rule (Tr-Unit) as follows:

$$\frac{M \text{ is effect-free}}{\Gamma \vdash M : unit \Rightarrow ()} \qquad \text{(Tr-Unit)}$$

Here, the condition "$M$ is effect-free" can be expressed by using appropriate effect systems [26], depending on what kind of side effect we want to preserve. By regarding calls to recursive functions and `print` as having side effects, we can safely transform the above program into:

```
let fun f(x, ()) =
    if x<0 then () else (print x;f(x-1, ()))
in f(2, ()) end
```

*More aggressive transformations.* As indicated in Section 1, our optimality result does not exclude the possibility of a better transformation that does not preserve types or does more than replacing subterms with ().

If the output need not be a well-typed term, more subterms may be replaced by (). For example, **if0** 0 **then** $M_1$ **else** $M_2$ can be replaced with **if0** 0 **then** $M_1$ **else** () . The optimality of non-type preserving transformation is undecidable: Notice that, in the term

$$\textbf{let } f = \lambda x.\textbf{if0 } x \textbf{ then } M_1 \textbf{ else } M_2 \textbf{ in } f(0) + f(M_3; 1),$$

$M_2$ can be replaced with () if and only if $M_3$ diverges.

Even when replacement of a subterm with () is invalid, it may be valid to replace it with a *dummy* value of the same type and enable elimination of useless variables in other places. For example, consider the following program fragment:

```
let val f = fn (x, y)=>x
    val g = fn (x, y)=>x+y+1
    val h = if b then f else g
in (f(1, M), h(3, 4)) end
```

For simplicity, suppose that all functions are monomorphic. Replacing `f(1, M)` with `f(1, ())` is invalid, because the type of `f` must be the same as that of `g`, $int \times int \rightarrow int$. However, replacing `M` with a dummy value `0` is valid. Although the replacement does not eliminate the second parameter of `f`, it may make variables in `M` useless. One way to perform such replacement would be to introduce a subtype relation $\tau \leq unit$, construct a type derivation tree, and then replace a subterm of type *unit* with a dummy value of the subterm's original type. Damiani [1; 2] has already proposed a method for replacing subterms with dummy values (which he called *dead code elimination*). The optimality of this problem is also undecidable.

## 6.2 Comparison with Other Methods for UVE

*Wand and Siveroni's work [28].* Our method is optimal and can be performed in almost linear time for the simply-typed language, while Wand and Siveroni's method is based on 0CFA, which costs cubic time in the worst case [8]. The reason for this is that the 0CFA-based method analyzes unnecessary flow information. For example, consider the following expression

```
let val f = fn (x,y)=>x
    val g = if b then f else fn (x,y)=>x+y
in f(1,2)+g(2,3) end
```

The 0CFA computes the flows into $f$ and $g$ separately and analyzes that `fn (x,y)=>x` flows into `f` but `fn (x,y)=>x+y` does not. However, this precise information is unnecessary: even with that information, we cannot transform `f(1,2)` into `f(1)`. Because `fn (x,y)=>x` flows into both $f$ and $g$, $f$ and $g$ must be called by using the same interface. So, once it is found that the same value flows into some variables, it is useless to keep separate flow information for those variables.

$$\frac{empty(\tau)}{\Gamma \vdash M : \tau \rightsquigarrow ()} \qquad \frac{\Gamma \vdash M : \tau_1 \times \tau_2 \rightsquigarrow M' \qquad \neg empty(\tau_1) \qquad \neg empty(\tau_2)}{\Gamma \vdash proj_i(M) : \tau_i \rightsquigarrow proj_i(M')}$$

$$\frac{\neg empty(\tau)}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow x} \qquad \frac{\Gamma \vdash M_1 : \tau_1 \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau_2 \rightsquigarrow M_2' \qquad empty(\tau_{3-i})}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2 \rightsquigarrow M_i'}$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2 \rightsquigarrow M' \qquad empty(\tau_{3-i})}{\Gamma \vdash proj_i(M) : \tau_i \rightsquigarrow M'} \qquad \frac{\Gamma \vdash M_1 : \tau_1 \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau_2 \rightsquigarrow M_2' \qquad \neg empty(\tau_1) \qquad \neg empty(\tau_2)}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2 \rightsquigarrow (M_1', M_2')}$$

Figure 6: Key Rules for Eliminating ()

Therefore, an equality-based analysis like type-based analysis without structural subtyping and $0CFA_=$[19] is sufficient for UVE.

In the case of a polymorphic language, our type-based method is still optimal but the 0CFA-based method is not. In fact, the 0CFA based method cannot perform the translation shown in Example 3.4.

*Fischbach and Hannan's work [4].* Fischbach and Hannan's method guarantees preservation of effects by integrating a simple effect analysis. It is, however, easy to extend our method to deal with effects, as mentioned above. We think that it is preferable to keep a type system for effect analysis and that for UVE separate, because complex type systems such as subtyping and polymorphic recursion [27] are necessary for precise analysis of effects, while simpler type systems suffice for UVE. Fischbach and Hannan's method does not deal with tuples and polymorphism.

## 6.3 Other Related Work

*Deadcode elimination.* As mentioned above, Damiani et al. [1; 2] treated a slightly different problem: to replace subterms with dummy values of the same type, rather than with (). It does not straightforwardly apply to UVE, since even if a function $f$ does not use the second parameter, the function call $f(M_1, M_2)$ is replaced only with $f(M_1, d)$ (where $d$ is a dummy value). In order to further replace $f(M_1, d)$ with $f(M_1)$, a method for UVE like ours must be performed. Because of this difference, the non-standard type system in [1] allows subtyping (which we do not allow).

*Program slicing.* Program specialization by using program slicing [23] is also closely related: If we are interested in only the first element of the output of a program $M$, then we can obtain a program slice by applying UVE and/or deadcode elimination to $proj_1(M)$. Detailed comparisons with each technique for program slicing are future work.

*Use analysis.* For a monomorphically-typed language, a transformation similar to ours can be performed by using a restricted form of Mogensen's use analysis for lazy functional languages [18] (although his analysis has been proposed for a different purpose). For a polymorphic language, however, it is not clear how to extend the use analysis with polymorphism so that the optimal transformation can be performed.

*Type inference-based garbage collection.* The underlying idea behind our method is similar to those of garbage collection schemes [5; 10] based on Reynolds' abstraction/parametricity theorem [24], where type inference is performed at run-time and a heap cell on which no type constraint is imposed by the context is collected as a garbage. The main differences of our UVE from them are that type inference is performed at compile time, and that any terms on which no type constraint is imposed by the context, not just heap values, are eliminated.

*Other works.* Heintze and McAllester [7] developed a linear-time algorithm for bounded-type programs that builds a directed graph whose transitive closure gives the information obtained by 0CFA. Interestingly, it is reported that some 0CFA-consuming analysis (for bounded-type programs) can also be performed in linear time, by using the obtained graph. Their work is not directly related with our work in this paper, because the reason why our algorithm is linear time is that we need to perform only an equality-based analysis, not that we have specialized 0CFA for the problem of UVE. However, for the problem of deadcode elimination (replacing subterms with dummy values), it may be interesting to apply their algorithm.

There are also many other works for reducing procedure parameters (although the problems they deal with seem to be actually quite different from the one treated in this paper). For example, in the context of logic programming, Leuschel and Sørensen [15] presented a method for removing redundant arguments of predicates. Danvy and Schultz [3] presented a method for removing redundant variables occurring as a result of a transformation called lambda dropping.

# 7. CONCLUDING REMARKS

We have formalized a type-based method for UVE and shown its correctness and optimality.

It is widely known that type systems and flow analyses are closely related [6; 19; 20; 21]. So, for a monomorphic language, the main difference between Wand and Siveroni's 0CFA-based method and ours is whether the analysis is subset-based or equality-based (or in the terminology of type systems, whether subtyping is allowed or not); our conclusion on this point is that an equality-based analysis is sufficient for UVE. For an ML-style polymorphic language, polymorphism provides an additional power of UVE (recall Example 3.4). So, it would be safe to say that the type-based method is more suitable at least for an ML-style language. Another supporting argument for the type-based method is that it is guaranteed to preserve typing.

One may argue that the CFA is anyway necessary for other optimizations and therefore the practical advantage of our method over Wand and Siveroni's method is small. However, because 0CFA costs cubic time in the program size, and UVE can reduce the program size (by eliminating some dead-code), it might be a good idea to perform the type-based UVE first, *and then* to perform 0CFA for other optimizations. Generally, because one optimization may enable another optimization, some compilers may perform several optimizations repeatedly; in that case, it is good to know what kind of analysis suffices for each optimization. Experiments are necessary anyway to know whether our UVE is really useful in practice, though.


## Acknowledgment

I would like to thank John Hannan, Atsushi Igarashi, and the anonymous referees for discussions and comments.


## 8. REFERENCES


[1] M. Coppo, F. Damiani, and P. Giannini. Refinement types for program analysis. In *Proceedings of the 3rd International Static Analysis Symposium (SAS'96)*, LNCS 1145, pages 143–158, 1996.

[2] F. Damiani. *Non-standard type inference for functional programs*. PhD thesis, Università degli Studi di Torino, 1998.

[3] O. Danvy and U. P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theor. Comput. Sci.*, Nov. 2000. A preliminary version appeared as BRICS Report RS-99-27.

[4] A. Fischbach and J. Hannan. Type systems and algorithms for useless-variable elimination, July 1999. submitted.

[5] P. Fradet. Collecting more garbage. In *Lisp and Functional Programming*, pages 24–33, 1994.

[6] N. Heintze. Control-flow analysis and type systems. In *International Static Analysis Symposium (SAS'95)*, LNCS 983, pages 189–206, 1995.

[7] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[8] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proc. of IEEE Symposium on Logic in Computer Science*, pages 342–351, 1997.

[9] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):253–289, 1993.

[10] H. Hosoya and A. Yonezawa. Formal description for collecting reachable garbage via dynamic type inference. In *Proc. of the 2nd International Workshop on Types in Compilation (TIC98)*, LNCS 1473, pages 215–239, 1998.

[11] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML Type Reconstruction. In J.-L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. The MIT Press, 1991.

[12] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. *Theoretical Computer Science*, 81(2):169–188, 1991.

[13] N. Kobayashi. Type-based useless variable elimination. Technical Report 99-02, Department of Information Science, University of Tokyo, July 1999. Available from `http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html`.

[14] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Prog. Lang. Syst.*, 20(4):707–723, 1998.

[15] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In *Logic Program Synthesis and Transformation*, LNCS 1207, pages 83–103, 1996.

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

[17] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[18] T. Mogensen. Types for 0, 1 or many uses. In *Implementation of Functional Languages*, LNCS 1467, pages 112–122, 1998.

[19] J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Trans. Prog. Lang. Syst.*, 20(6):1251–1264, 1998.

[20] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Prog. Lang. Syst.*, 17(4):576–599, 1995.

[21] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proc. of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 197–208, 1998.

[22] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[23] T. Reps and T. Turnidge. Program specialization via program slicing. In *Proc. of the Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 409–429, 1996.

[24] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Proc. of Information Processing 83*, pages 513–523, 1983.

[25] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[26] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[27] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[28] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proc. of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 291–302, 1999.