TECHNICAL REPORT TR99-02

# Type-Based Useless Variable Elimination

Naoki Kobayashi

July 1999

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

**TITLE**
Type-Based Useless Variable Elimination

**AUTHORS**
Naoki Kobayashi

**KEY WORDS AND PHRASES**
useless variable elimination, types, type-based analysis

**ABSTRACT**
We show a type-based method for useless variable elimination, i.e., transformation that eliminates variables whose values contribute nothing to the final outcome of a computation, and prove its correctness. The algorithm is a surprisingly simple extension of the usual type reconstruction algorithm. Our method seems more attractive than Wand and Siveroni's 0CFA-based method in many respects. First, it is efficient: it runs in time almost linear in the size of an input expression for a simply-typed $\lambda$-calculus, while the 0CFA-based method may require a cubic time. Second, our transformation can be shown to be *optimal* among those that preserve well-typedness, both for the simply-typed language and for an ML-style polymorphically-typed language. On the other hand, the 0CFA-based method is *not* optimal for the polymophically-typed language.

| **REPORT DATE** | **WRITTEN LANGUAGE** |
|---|---|
| July 1999 | English |
| **TOTAL NO. OF PAGES** | **NO. OF REFERENCES** |
| 29 | 21 |

**ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT**
Summary has been submitted for publication. Up-to-date version of this report will be available through http://www.yl.is.s.u-tokyo.ac.jp/ koba/publications.html

**DISTRIBUTION STATEMENT**
This technical report is available ONLY via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

**SUPPLEMENTARY NOTES**

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

# Type-Based Useless Variable Elimination

Naoki Kobayashi
Department of Information Science, University of Tokyo
7-3-1 Hongo,Bunkyo-ku, Tokyo 113-0033, Japan
email:koba@is.s.u-tokyo.ac.jp

## 1 Introduction

### 1.1 Background

Useless variable elimination is transformation that eliminates variables whose value contribute nothing to the final outcome of a computation. For example, as given in [20], 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 a useless variable `bogus`. In higher-order languages, it can be considered as an extension of deadcode elimination [20]. As argued in [20], this kind of useless variable is unlikely to be produced in human-produced code, but it may be produced in machine-generated code or as a result of various compiler optimizations such as arity-raising [6] and constant propagation. 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, it is transformed into `fn (x,y)=>x` and the variable `y` becomes useless.

Wand and Siveroni [20] proposed a 0CFA-based method of such useless variable elimination for an untyped $\lambda$-calculus (by extending Shiver's analysis for first-order languages [17]) and proved its correctness.

### 1.2 Our Proposal in This Paper

In this paper, we show a *type-based* method of useless variable elimination for a simply-typed $\lambda$-calculus and an ML-style polymorphically-typed language. We formalize useless variable

elimination 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 useless variable elimination as the two-step transformation clarifies the essence of useless variable elimination. It also seems advantageous in that no special treatment is necessary for dealing with elimination of part of structured data argument (on the other hand, it is not clear how to deal with structured data in [20]).

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 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. Therefore, we can perform useless variable elimination in a type-directed manner: 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 explain how to obtain an optimal transformation a little bit more in detail in this paragraph. What we essentially need to do is to perform type inference in a top-down [8] 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 $int \times \alpha \to int$ and $(1,2)$ must have type $int \times \alpha$. The usual type inference then checks 2 against type $\alpha$ and obtain $\alpha = int$, 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 Advantages of Our Method

Our type-based method seems more attractive for statically-typed languages than the $0CFA$-based method in many respects. First of all, our type-based method is guaranteed to preserve well-typeness; because many recent compilers of statically-typed languages keep type information until later stages of compilation [11], this seems to be an important point. Second, the correctness can be proved easily, in a similar manner to the soundness of the usual type systems; this is in contrast to the somewhat complex proof of the correctness of the $0CFA$-based method [20]. Third, our type-based method can eliminate not only useless top-level

parameters of functions, but also part of structured data passed to functions: for example, `fn (x, y:int*real, z)=>...` can be transformed into `fn (x, y:int, z)=>...` if the second element of `y` is found to be useless; on the other hand, it is not clear how to do it in Wand and Siveroni's method [20]. Fourth, our algorithm is efficient: for a simply-typed language, it runs in almost linear time in the size of an input expression, while the $0CFA$-based method requires a cubic time. For an ML-style polymorphic language, the worst-case time complexity is worse than that of the $0CFA$-based method. However, it does not cost much more than type reconstruction, which is known to cost an exponential time in the worst case [7] but work well in practice. So, our algorithm seems efficient in practice even in the case of the polymorphic language. Moreover, it can be easily shown that our useless variable elimination is *optimal* among those that preserve well-typedness of programs, for both of the simply-typed and polymorphically-typed languages. On the other hand, as far as the author knows, no result has been reported on the optimality of the $0CFA$-based method (although we do not doubt it for the simply-typed case), and indeed, in the case of the polymorphic language, the $0CFA$-based method is not optimal.

### 1.4 Structure of the Rest of This Paper

The rest of this paper is structured as follows. Section 2 introduces the syntax and typing of a simply-typed language. Section 3 gives translation rules and shows its correctness and optimality. Section 4 describes an algorithm to find an optimal translation and discusses its correctness and time complexity. Section 5 extend the method to handle ML-style let-polymorphism. Section 6 touches upon related work. In particular, we discuss why a type-based method like ours seems more suitable for useless variable elimination than the $CFA$-based method. Section 7 concludes this paper.

## 2 Syntax and Operational Semantics of the Language

In this section, we introduce the syntax and typing of the target language: a simply-typed $\lambda$-calculus. It will be extended with let-polymorphism in Section 5.

### 2.1 Terms

**Definition 2.1 [terms]:**
The syntax of terms is given by the following syntax.

$$\begin{array}{lll} M & ::= & () \mid n \mid x \mid \lambda x.M \mid M_1 + M_2 \mid \mathbf{fix}(f, x, M) \mid M_1 M_2 \mid proj_1(M) \mid proj_2(M) \mid (M_1, M_2) \\ V & ::= & () \mid n \mid \lambda x.M \mid (V_1, V_2) \end{array}$$

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 $f = \lambda x.M$, and $proj_i(M)$ denotes a pair projection. $()$ is a special constant; it plays a special role in the formalization of useless variable elimination.

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 $\lambda x.M$. 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 occurences 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 following rules.

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

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

$$\frac{n \text{ is the sum of } n_1 \text{ and } n_2}{n_1 + n_2 \Downarrow n} \qquad \text{(E-Add)}$$

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

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

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

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

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

## 2.3 Typing

A type system is also defined in a standard manner.

**Definition 2.2 [types]:** The set of types, ranged over by $\tau$, is given by the following syntax.

$$\tau \quad ::= \quad unit \mid int \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

**Definition 2.3:** A typing relation $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau$ is the least relation closed under the following rules.

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

$$\Gamma, x : \tau \vdash x : \tau \qquad \text{(T-Var)}$$

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

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

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

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

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2} \tag{T-App}$$

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

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathit{proj}_i(M) : \tau_i} \tag{T-Proj}$$

# 3 Useless Variable Elimination

As explained in Section 1, we formalize useless variable elimination as a type-preserving, source-to-source transformation. The transformation is divided into two steps: the first step replaces some of the terms whose values do not contribute to the result with (), and then the second step eliminates such (). For example, $(\lambda x.\mathit{proj}_1(x))(1, M)$ is first transformed to $(\lambda x.\mathit{proj}_1(x))(1, ())$, and then to $(\lambda x.x)1$. (As this example shows, the transformation does not preserve non-termination: even if evaluation of a source term diverges, evaluation of the translated term may terminate. See Section 6 for a remedy for this.) In this paper, we focus on the first transformation: the second transformation can be easily performed by looking at the type of a term.

In Section 3.1, we give rules for deriving correct transformations (i.e., transformation that does not change the result of computation) and prove that they are indeed correct. Then, we show in Section 3.2 that they are optimal in the sense that any correct transformation can be derived from those rules. An algorithm for finding an optimal transformation is deferred until Section 4.

## 3.1 Typing (translation) rules

We want to formalize rules for, given a typed term $\Gamma \vdash M : \tau$, deriving $M'$ such that $M'$ is a term 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 below. They are almost the same as typing rules, except for the rule (Tr-Unit). The rule (Tr-Unit) 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 ().

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

$$\Gamma, x : \tau \vdash x : \tau \Rightarrow x \tag{Tr-Var}$$

$$\Gamma \vdash n : \mathit{int} \Rightarrow n \tag{Tr-Int}$$

$$\frac{\Gamma \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \mathit{int} \Rightarrow M_2'}{\Gamma \vdash M_1 + M_2 : \mathit{int} \Rightarrow M_1' + M_2'} \tag{Tr-Add}$$

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

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

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

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

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

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 unnecessary applications of ()) by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma, z : \mathit{unit} \vdash N : \mathit{int} \Rightarrow N}}{\Gamma \vdash \lambda z.N : \mathit{unit} \to \mathit{int} \Rightarrow \lambda z.N} \quad \Gamma \vdash \mathit{proj}_2(x) + 1 : \mathit{unit} \Rightarrow ()}{\Gamma \vdash (\lambda z.N)(\mathit{proj}_2(x) + 1) : \mathit{int} \Rightarrow (\lambda z.N)()}}{\emptyset \vdash \lambda x.(\lambda z.N)(\mathit{proj}_2(x) + 1) : \mathit{int} \times \mathit{unit} \to \mathit{int} \Rightarrow \lambda x.((\lambda z.N)())} \quad \dfrac{\emptyset \vdash 1 : \mathit{int} \Rightarrow 1 \quad \emptyset \vdash 2 : \mathit{unit} \Rightarrow ()}{\emptyset \vdash (1,2) : \mathit{int} \times \mathit{unit} \Rightarrow (1,())}}{\emptyset \vdash M : \mathit{int} \Rightarrow M'}$$

Here, $N = \mathit{proj}_1(x) + 1$ and $\Gamma = x : \mathit{int} \times \mathit{unit}$.

**Example 3.2:** Consider the example given in Section 1:

```
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
```

It is expressed in our language (extended with booleans, $n$-tuples, and conditionals) by the following term $M$:

$(\lambda \mathit{loop}.\mathit{loop}(a, 3, 1))$
$\quad \mathbf{fix}(\mathit{loop}, x, \mathbf{if}\ \mathit{proj}_3(x) > 100\ \mathbf{then}\ \mathit{proj}_1(x)\ \mathbf{else}\ \mathit{loop}(f(\mathit{proj}_1(x), \mathit{proj}_3(x)), \mathit{proj}_2(x) + 2, \mathit{proj}_3(x) + 1)$

We can add the following rule for conditionals:

$$\frac{\Gamma \vdash M_1 : bool \Rightarrow M_1' \qquad \Gamma \vdash M_2 : \tau \Rightarrow M_2' \qquad \Gamma \vdash M_3 : \tau \Rightarrow M_3'}{\Gamma \vdash \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \ : \tau \Rightarrow \textbf{if } M_1' \textbf{ then } M_2' \textbf{ else } M_3'} \qquad (\textsc{Tr-If})$$

Let $\Gamma = loop : int \times unit \times int \to int, f : int \to int$. Then, the body of the function $loop$ and the main body are transformed as follows:

$\Gamma, x : int \times unit \times int \vdash$
  **if** $proj_3(x) > 100$ **then** $proj_1(x)$ **else**
    $loop(f(proj_1(x), proj_3(x)), proj_2(x) + 2, proj_3(x) + 1) : int$
    $\Rightarrow$ **if** $proj_3(x) > 100$ **then** $proj_1(x)$ **else** $loop(f(proj_1(x), proj_3(x)), (), proj_3(x) + 1)$
$\Gamma, a : int \vdash loop(a, 3, 1) : int \Rightarrow loop(a, (), 1)$

So, the whole term is transformed into:

$(\lambda loop.loop(a, (), 1))$
  $\textbf{fix}(loop, x, \textbf{if } proj_3(x) > 100 \textbf{ then } proj_1(x) \textbf{ else } loop(f(proj_1(x), proj_3(x)), (), proj_3(x) + 1)\ )$

under the typing assumption $f : int \times int \to int, a : int$. By eliminating unnecessary (), we obtain:

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

**Example 3.3:** Consider the following term taken from [20]:

```
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
```

It is expressed in our language as the following term $M$:

$$(\lambda f_1.\lambda f_2.\lambda f_3.\lambda g.\lambda h.g(x,h))(\lambda z.proj_1(z))(\lambda z.proj_1(z) + proj_1(z))$$
$$(\lambda z.proj_2(z))(\textbf{if } p(a,b) \textbf{ then } f_1 \textbf{ else } f_2\ )(\textbf{if } q(a,b) \textbf{ then } f_1 \textbf{ else } f_3\ ).$$

By assigning the following types to bound variables:

$$f_1, f_2, g : (int \times unit) \to int$$
$$f_3, h : unit,$$

we can derive

$$\Gamma \vdash M : int \Rightarrow M'$$

for

$$\Gamma = p : (int \times int) \to bool, q : (int \times int) \to bool, a : int, b : int, \text{ and}$$
$$M' = (\lambda f_1.\lambda f_2.\lambda f_3.\lambda g.\lambda h.g(x,h))$$
$$(\lambda z.proj_1(z))(\lambda z.proj_1(z) + proj_1(z))()(\textbf{if } p(a,b) \textbf{ then } f_1 \textbf{ else } f_2\ )().$$

If we simplify the resulting term by removing (), we obtain

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

which is the same result as given in [20]. (Here, because our language has a monomorphic type system, we assumed that $a$ and $b$ are integers; the result is the same for different assumptions on the types of $a$ and $b$.)

The following theorem says that there is at least one translation that preserves typing.

**Theorem 3.4:** If $\Gamma \vdash M : \tau$, then $\Gamma \vdash M : \tau \Rightarrow M'$ for some $M'$.

**Proof:** A derivation of $\Gamma \vdash M : \tau \Rightarrow M$ can be obtained easily from a derivation of $\Gamma \vdash M : \tau$: replace each each application of rule (T-XX) with an application of (TR-XX). □

The following theorem means that the output of the translation is a well-typed term. (Note here that the well-typedness of a source term is not required.)

**Theorem 3.5 [Well-typedness of translated terms]:** If $\Gamma \vdash M : \tau \Rightarrow M'$, then $\Gamma \vdash M' : \tau$.

**Proof:** Straightforward induction on the derivation of $\Gamma \vdash M : \tau \Rightarrow M'$. Notice that for each translation rule of the form

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \cdots \qquad \Gamma_n \vdash M_n : \tau_n \Rightarrow M_n'}{\Gamma \vdash M : \tau \Rightarrow M'}$$

the following is an instance of a typing rule:

$$\frac{\Gamma_1 \vdash M_1' : \tau_1 \qquad \cdots \qquad \Gamma_n \vdash M_n' : \tau_n}{\Gamma \vdash M' : \tau}$$

□

Theorem 3.6 below means that the translated term evaluates to the same value as a source term. Here, we show this only for terms that evaluate to integers. (We believe this is a sufficient criteria for the correctness because terms that yield function values can be compared by putting them into arbitrary contexts yielding integers.) A substitution $[V_1/x_1, \ldots, V_n/x_n]$ models an environment under which $M$ and $M'$ are evaluated. Note that this theorem does not imply that the transformation preserves divergence. For example, $(\lambda x.1)(\mathbf{fix}(f, y, f(y))1)$ diverges but it can be transformed into $(\lambda x.1)()$.

**Theorem 3.6 [Correctness]:** If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \mathit{int} \Rightarrow M'$, $\vdash V_1 : \tau_1, \ldots, \vdash V_n : \tau_n$ and $[V_1/x_1, \ldots, V_n/x_n]M \Downarrow n$, then $[V_1/x_1, \ldots, V_n/x_n]M' \Downarrow n$.

**Proof:** A special case of Theorem 5.3 proved in Section 5. □

Alternatively, we can state the correctness as the following corollary.

**Corollary 3.7 [Contextual Equivalence]:** Suppose $\Gamma \vdash M : \tau$ and $\Gamma \vdash M : \tau \Rightarrow M'$. Suppose also $N$ is a term containing only one occurrence of a variable $x$. If $\vdash [M/x]N : \mathit{int}$ is obtained from $\Gamma \vdash M : \tau$ and $[M/x]N \Downarrow n$, then $[M'/x]N \Downarrow n$.

**Proof:** By translation rules and the assumption, we have $\vdash [M/x]N : int \Rightarrow [M'/x]N$. So, the result follows from Theorem 3.6. □

The argument on the correctness in the above theorem is based on the operational semantics, but the following very informal reasoning based on denotational semantics may be more intuitive. Think of a cpo-based denotational semantics of lazy languages [9] and consider that () is representing the bottom $\perp$ of 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 $M$). So, if $M^* = n$, then $M'^*$ is either $\perp$ or $n$. If $M'^* = \perp$, then $M'$ evaluates to () or diverges, but this is not the case because $M'$ is a well-typed term and because $M$ must diverge if $M'$ diverge by the construction of $M'$. Therefore, $M'^* = n$. Since the evaluation of $M'$ terminates if that of $M$ does also in our call-by-value semantics, we know that $M'$ evaluates to the same integer as $M$.

### 3.2 Optimality of the translation

We formalized useless variable elimination as transformation that replaces some subterms with () while preserving the evaluation result. It is easy to check that our type-based transformation is optimal among those preserving well-typedness.

First, let us formally define replacement of subterms with (). The following relation $M \succeq M'$ means that $M'$ is obtained from $M$ by such replacement.

**Definition 3.8:** A relation $\succeq$ on terms is the least relation satisfying the following rules:

$$
\begin{array}{l}
M \succeq () \\
n \succeq n \\
x \succeq x \\
M + N \succeq M' + N' \text{ if } M \succeq M' \text{ and } N \succeq N' \\
\lambda x.M \succeq \lambda x.M' \text{ if } M \succeq M' \\
\mathbf{fix}(f, x, M) \succeq \mathbf{fix}(f, x, M') \text{ if } M \succeq M' \\
MN \succeq M'N' \text{ if } M \succeq M' \text{ and } N \succeq N' \\
(M, N) \succeq (M', N') \text{ if } M \succeq M' \text{ and } N \succeq N' \\
proj_i(M) \succeq proj_i(M') \text{ if } M \succeq M'
\end{array}
$$

The following theorem means that if $M'$ is obtained by replacement of some subterms with () and if $M'$ is well-typed, then $M'$ is obtained by our type-based transformation.

**Theorem 3.9 [optimality]:** If $\Gamma \vdash M : \tau$, $\Gamma \vdash M' : \tau$, and $M \succeq M'$, then $\Gamma \vdash M : \tau \Rightarrow M'$

**Proof:** A special case of Theorem 5.7. □

Note that the above theorem does not exclude the possibility of a better transformation that does more than merely replacing subterms with (). For example, consider the following program:

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

By replacing `if true then x else y` with `x`, we can eliminate the second parameter of `f`:

```
let fun f(x) = x in f(1) end
```

We regard this as a combination of useless variable elimination 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 useless variable elimination can be applied to it (2 is replaced with () and then unnecessary () is eliminated).

The following example illustrates a more fundamental limitation of the above claim of optimality.

```
let
  fun f(x,y)=x
  fun g(u,v)=f(u, very_expensive_computation(v))
in
  (h(f), g(1,2))
end
```

Suppose `h` is an external function of type $(int \times int \to int) \to int$. `f` does not use the second parameter, but `very_expensive_computation(v)` cannot be replaced with () because of typing constraints. However, if we allowed a subterm to be replaced with a value other than (), then we could transform it into:

```
let
  fun f(x,y)=x
  fun g(u)=f(u, 0)
in
  (h(f), g(1))
end
```

Here, 0 is passed as a dummy parameter in order to adjust types. See Section 6 for more discussion on this issue.

# 4 Translation Algorithm

Theorem 3.9 in Section 3 says that there is *some* type-based translation that achieves the best useless variable elimination (among those that preserve typing). In this section, we show that such an optimal type-based translation can be found in an almost linear time. We first overview the algorithm in Section 4.1, and then formally describe it in Section 4.2. Then, we show its correctness and discuss its time complexity in Section 4.3 and 4.4.

## 4.1 Overview of the Algorithm

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 $\Gamma \vdash M : \tau \Rightarrow M'$. Alternatively, we can assume that only $M$ is given as an input; in that case, we can obtain $\Gamma \vdash M : \tau$ in time linear in the size of $M$, by first computing a principal typing [7] and then replacing remaining type variables with *int*.

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 far 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*.

For example, consider a term $\emptyset \vdash (\lambda x.proj_1(x))(1,2) : int$. Because the translated term must also produce an integer, the whole term cannot be replaced with (). So, we know that $\lambda x.proj_1(x)$ and $(1,2)$ must have type of the form $\alpha \to int$ and $\alpha$ respectively. Now because we know that $\lambda x.proj_1(x)$ cannot have type *unit*, it must be transformed by using (Tr-Abs). So, $proj_1(x)$ must be transformed into a term of type *int*, which implies $proj_1(x)$ must be transformed by using (Tr-Proj), not (Tr-Unit). So, $\alpha$ must be a type of the form $int \times \gamma$. Now we know that $(1,2)$ must be transformed into a term of type $int \times \gamma$. So, 1 must be transformed into 1. Since we have no further constraints, we can let $\gamma$ be *unit*, and obtain $(\lambda x.proj_1(x))(1,())$ as an output.

Because we want to obtain an almost linear time algorithm, we use an algorithm similar to the linear-time type reconstruction algorithm for the simply-typed $\lambda$-calculus (see Proposition 3.3 of [7]). Imagine a type derivation each node of which is $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau_N \Rightarrow N'$. For each node, there are always two possible last rules: one is (Tr-Unit) and the other is the rule whose conclusion matches the syntactic constructor of a term. If the type of the term $N$ is not *unit*, then the rule must be the latter one and some equality constraints must be met. For example, if $N$ is a variable $x_i$, it must be the case that the rule must be (Tr-Var) and $\tau_i = \tau_N$; if $N$ is $N_1 N_2$, then it must be the case that $\tau_{N_1} = \tau_{N_2} \to \tau_N$. Otherwise, the rule may be the former one (which is preferred because it produces a better output) and no further constraints need to be met. Therefore, 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 generate a set of constraints of the form $\tau \neq unit \Rightarrow E$ where $E$ is a set of equality constraints on types. For example, for a subterm $N_1 N_2$, the constraint $(\alpha_{N_1 N_2} \neq unit) \Rightarrow \{\alpha_{N_1} = \alpha_{N_2} \to \alpha_{N_1 N_2}\}$ is generated. Every type derivation can be obtained by properly instantiating type variables so that the constraints are met.

After generating constraints, we can solve the constraints by using a unification algorithm. The differences from an ordinary unification algorithm are that equality constraints may be added lazily (if $\tau$ in a constraint $\tau \neq unit \Rightarrow E$ is a type variable, $E$ is added only when it is found that $\tau$ cannot be instantiated with *unit*) and that the occurrence check can be omitted (because it is guaranteed by Theorem 3.4 that a solution always exists if an input term is well typed).

After obtaining the most general substitution for type variables, we can obtain an optimal derivation by instantiating remaining type variables with *unit*.

**Example 4.1:** Consider a typed term $\emptyset \vdash (\lambda x.proj_1(x))(1,2) : int$. We can construct the following *template* of a derivation tree.

$$\dfrac{\dfrac{\dfrac{x : \beta_x \vdash x : \alpha_x \Rightarrow N_7}{x : \beta_x \vdash proj_1(x) : \alpha_{proj_1(x)} \Rightarrow N_6}}{\emptyset \vdash \lambda x.proj_1(x) : \alpha_{\lambda x.proj_1(x)} \Rightarrow N_2} \qquad \dfrac{\emptyset \vdash 1 : \alpha_1 \Rightarrow N_4 \qquad \emptyset \vdash 2 : \alpha_2 \Rightarrow N_5}{\emptyset \vdash (1,2) : \alpha_{(1,2)} \Rightarrow N_3}}{\emptyset \vdash (\lambda x.proj_1(x))(1,2) : \alpha_{(\lambda x.proj_1(x))(1,2)} \Rightarrow N_1}$$

We can obtain a derivation tree by replacing subtrees whose conclusion is of the form $\Gamma \vdash N : unit \Rightarrow N'$ with an instance of the axiom $\Gamma \vdash N : unit \Rightarrow ()$, if type variables are instantiated

so that the following conditions are met,

$$\alpha_{(\lambda x.proj_1(x))(1,2)} \neq unit \Rightarrow \{\alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \rightarrow \alpha_{(\lambda x.proj_1(x))(1,2)}\}$$
$$\alpha_{\lambda x.proj_1(x)} \neq unit \Rightarrow \{\alpha_{\lambda x.proj_1(x)} = \beta_x \rightarrow \alpha_{proj_1(x)}\}$$
$$\alpha_{proj_1(x)} \neq unit \Rightarrow \{\alpha_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\}$$
$$\alpha_x \neq unit \Rightarrow \{\alpha_x = \beta_x\}$$
$$\alpha_{(1,2)} \neq unit \Rightarrow \{\alpha_{(1,2)} = \alpha_1 \times \alpha_2\}$$
$$\alpha_1 \neq unit \Rightarrow \{\alpha_1 = int\}$$
$$\alpha_2 \neq unit \Rightarrow \{\alpha_2 = int\}$$

It should be clear that every valid derivation can be obtained in this manner.

## 4.2 Formal Description of the Algorithm

Now we are ready to describe the algorithm more formally.

### 4.2.1 Extracting Constraints

As explained above, we assign a type variable $\alpha_{M'}$ to each subterm $M'$ of $M$ and a type variable $\beta_x$ to each variable $x$ appearing in $M$. For multiple occurrences of the same term in $M$, we assign different type variables; specifically, if $M$ contains $n$ occurrences of the same variable $x$, then we introduce $n$ type variables $\alpha_{x_1}, \ldots, \alpha_{x_n}$ although the indexes are omitted below. We also assign a type variable $\gamma_{M'}$ to each subterm $M'$ of the form $proj_i(N)$.

$\mathcal{C}(N)$ defined below gives a set of equality constraints to be met in order for the rule matching $N$ to be applied.

**Definition 4.2:** Let $M$ be a term. For each subterm $M'$ of $M$, the set $\mathcal{C}(M')$ of constraints is defined by:

$$\mathcal{C}(x) = \{\alpha_x = \beta_x\}$$
$$\mathcal{C}(n) = \{\alpha_n = int\}$$
$$\mathcal{C}(M_1 + M_2) = \{\alpha_{M_1+M_2} = int, \alpha_{M_1} = int, \alpha_{M_2} = int\}$$
$$\mathcal{C}(\lambda x.N) = \{\alpha_{\lambda x.N} = \beta_x \rightarrow \alpha_N\}$$
$$\mathcal{C}(\mathbf{fix}(f, x, N)) = \{\alpha_{\mathbf{fix}(f,x,N)} = \beta_x \rightarrow \alpha_N, \beta_f = \beta_x \rightarrow \alpha_N\}$$
$$\mathcal{C}(M_1 M_2) = \{\alpha_{M_1} = \alpha_{M_2} \rightarrow \alpha_{M_1 M_2}\}$$
$$\mathcal{C}((M_1, M_2)) = \{\alpha_{(M_1,M_2)} = \alpha_{M_1} \times \alpha_{M_2}\}$$
$$\mathcal{C}(proj_1(N)) = \{\alpha_N = \alpha_{proj_1(N)} \times \gamma_{proj_1(N)}\}$$
$$\mathcal{C}(proj_2(N)) = \{\alpha_N = \gamma_{proj_2(N)} \times \alpha_{proj_2(N)}\}$$

Notice that the type derivation tree each node of which is $x_1 : \beta_{x_1}, \ldots, x_n : \beta_{x_n} \vdash N : \alpha_N \Rightarrow N'$ is a valid derivation tree for $\Gamma \vdash M : \tau$ if the following conditions hold:

$$\{(\alpha_N \neq unit) \Rightarrow \mathcal{C}(N) \mid N \text{ is a subterm of } M\} \cup \{\Gamma(x) = \beta_x \mid x \in dom(x)\} \cup \{\tau = \alpha_M\}.$$

### 4.2.2 Solving Constraints

We solve the above constraints by computing a valid equivalence relation [15] (see Definition 4.3 below) on type terms. Let $\mathcal{N}(M)$ be the set of nodes of dags (directed acyclic graphs) representing type terms appearing in $\{\mathcal{C}(N) \mid N \text{ is a subterm of } M\}$. We write $\sim$ for an equivalence relation over $\mathcal{N}(M)$. $[\tau]_\sim$ is the representative element of the equivalence class containing $\tau$, and we assume that the representative element is chosen so that if $[\tau]_\sim$ is a type variable then the equivalence class of $\tau$ contains only type variables. We write $\sim \cup \{\tau_{11} = \tau_{12}, \ldots, \tau_{n1} = \tau_{n2}\}$

for the least equivalence relation $\sim'$ such that $\sim' \supseteq \sim \cup \{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$. An algorithm for solving constraints is given in terms of the following rewriting relation on pairs consisting of a set $S$ of equality constraints and an equivalence relation $\sim$.

$$(S \cup \{\tau_1 = \tau_2\}, \sim) \rightsquigarrow \begin{cases} (S, \sim) & \text{if } [\tau_1]_\sim = [\tau_2]_\sim \\ (S \cup subC([\tau_1]_\sim, [\tau_2]_\sim) \cup newC(\tau_1, \tau_2, \sim), \sim \cup \{\tau_1 = \tau_2\}) & \\ \qquad \text{if } [\tau_1]_\sim \neq [\tau_2]_\sim \text{ and } subC(\tau_1, \tau_2) \neq \textbf{fail} & \\ \textbf{fail} \quad \text{if } [\tau_1]_\sim \neq [\tau_2]_\sim \text{ and } subC(\tau_1, \tau_2) = \textbf{fail} & \end{cases}$$

Here, $subC(\tau_1, \tau_2)$ and $newC(\tau_1, \tau_2, \sim)$ are defined by:

$$subC(\tau_1, \tau_2) = \begin{cases} \emptyset & \text{if } \tau_1 \text{ or } \tau_2 \text{ is a type variable} \\ \{\tau_{11} = \tau_{21}, \tau_{21} = \tau_{22}\} & \text{if (i)} \tau_1 = \tau_{11} \times \tau_{12} \wedge \tau_2 = \tau_{21} \times \tau_{22} \text{ or} \\ & \text{(ii)} \tau_1 = \tau_{11} \rightarrow \tau_{12} \wedge \tau_2 = \tau_{21} \rightarrow \tau_{22} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

$$newC(\tau_1, \tau_2, \sim) = \begin{cases} \bigcup\{\mathcal{C}(N) \mid \alpha_N \sim \tau_1\} & \\ \qquad \text{if } [\tau_1]_\sim \text{ is a type variable and } [\tau_2]_\sim \text{ is neither a type variable nor } \textit{unit} & \\ \bigcup\{\mathcal{C}(N) \mid \alpha_N \sim \tau_2\} & \\ \qquad \text{if } [\tau_2]_\sim \text{ is a type variable and } [\tau_1]_\sim \text{ is neither a type variable nor } \textit{unit} & \\ \emptyset \quad \text{otherwise} & \end{cases}$$

$subC(\tau_1, \tau_2)$ adds equality constraints on subterms of $\tau_1$ and $\tau_2$. $newC$ adds $\mathcal{C}(N)$ if $\alpha_N$ is found not to be *unit* for some $N$.

Starting with $(\{\Gamma(x) = \beta_x \mid x \in dom(x)\} \cup \{\tau = \alpha_M\}, \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\})$, apply the above rewriting rule repeatedly until it becomes a pair of the form $(\emptyset, \sim)$. Then, each type variable $\alpha$ can be instantiated with $mgu_\sim(\alpha)$ given below.

**Definition 4.3:** An equivalence relation $\sim$ on $\mathcal{N}(M)$ is *valid* [15] if (i)$\tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2'$ or $\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'$ implies $\tau_1 \sim \tau_1'$ and $\tau_2 \sim \tau_2'$, (ii)there are no $\tau_1, \tau_2, \tau_1', \tau_2'$ such that $\tau_1 \rightarrow \tau_2 \sim \tau_1' \times \tau_2'$, and (iii) a binary relation $(\{[\tau]_\sim \mid \tau \in \mathcal{N}(\tau)\}, \{([\alpha]_\sim, [\tau]_\sim) \mid \text{the type variable } \alpha \text{ appears in } \tau\})$ is a partial order.

Let $\sim$ be a valid equivalence relation on $\mathcal{N}(M)$. A substitution $mgu_\sim$ is defined by:

$$mgu_\sim(\alpha) = [mgu_\sim(\alpha_1)/\alpha_1, \ldots, mgu_\sim(\alpha_n)/\alpha_n][\alpha]_\sim$$

where $\alpha_1, \ldots, \alpha_n$ are type variables appearing in $[\alpha]_\sim$.

$mgu_\sim(\alpha)$ is well defined because of the third condition of the valid equivalence relation.

An optimized term $M'$ is obtained from $M$ by replacing each subterm $N$ of $M$ with () if $mgu_\sim(\alpha_N)$ is a type variable (replace larger terms first).

**Example 4.4:** Let $\emptyset \vdash (\lambda x.proj_1(x))(1, 2) : int$ be an input. The initial configuration is:

$$(\{\alpha_M = int\}, \sim_{id})$$

where $M = (\lambda x.proj_1(x))(1,2)$ and $\sim_{id} = \{(\tau,\tau) \mid \tau \in \mathcal{N}(M)\}$. It is rewritten as follows:

$$
\begin{array}{ll}
 & (\{\alpha_M = int\}, \{(\tau,\tau) \mid \tau \in \mathcal{N}(M)\}) \\
\leadsto & (\{\alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M\}, \sim_{id} \cup \{\alpha_M = int\}) \\
 & \quad (\mathcal{C}(M) \text{ has been added}) \\
\leadsto & (\{\alpha_{\lambda x.proj_1(x)} = \beta_x \to \alpha_{proj_1(x)}\}, \sim_{id} \cup \{\alpha_M = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M\}) \\
 & \quad (\mathcal{C}(\lambda x.proj_1(x)) \text{ has been added}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_{proj_1(x)} = int\}, \\
 & \sim_{id} \cup \{\alpha_M = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)}\}) \\
 & \quad (\text{subconstraints of } \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)} \text{ has been added}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\}, \\
 & \sim_{id} \cup \{\alpha_M = \alpha_{proj_1(x)} = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)}\} (= \sim_1)) \\
 & \quad (\mathcal{C}(proj_1(x)) \text{ has been added}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_x = \beta_x\}, \sim_1 \cup \{\alpha_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}\}, \sim_1 \cup \{\alpha_x = \beta_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\}) \\
\leadsto & (\{\alpha_1 \times \alpha_2 = \alpha_{(1,2)}\}, \sim_1 \cup \{\alpha_x = \beta_x = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)}\}) \\
 & \quad (\mathcal{C}((1,2)) \text{ has been added}) \\
\leadsto & (\{\alpha_1 = \alpha_{proj_1(x)}, \alpha_2 = \gamma_{proj_1(x)}\}, \\
 & \sim_1 \cup \{\alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)} = \alpha_1 \times \alpha_2\}) \\
 & \quad (\text{subconstraints of } \alpha_{proj_1(x)} \times \gamma_{proj_1(x)} = \alpha_1 \times \alpha_2 \text{ has been added}) \\
\leadsto^* & (\emptyset, \sim_{id} \cup \{\alpha_M = \alpha_{proj_1(x)} = \alpha_1 = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)}, \\
 & \quad \alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{proj_1(x)} \times \gamma_{proj_1(x)} = \alpha_1 \times \alpha_2, \alpha_2 = \gamma_{proj_1(x)}\})
\end{array}
$$

By instantiating $\alpha_2$ with *unit*, we obtain

$$
\begin{array}{l}
\alpha_M = \alpha_{proj_1(x)} = \alpha_1 = int \\
\alpha_{\lambda x.proj_1(x)} = int \times unit \to int \\
\alpha_x = \alpha_{(1,2)} = int \times unit \\
\alpha_2 = unit.
\end{array}
$$

Because $\alpha_2$ is type *unit*, we obtain $(\lambda x.proj_1(x))((1,()))$ by replacing 2 with ().

### 4.3 Correctness

The correctness of the above algorithm follows from the fact that each rewriting transforms a set of constraints into an equivalent set of constraints, the fact that rewriting always terminates without failure, and the fact that the resulting $\sim$ is a valid equivalence relation.

Because the constraint $(\alpha_N \neq unit) \Rightarrow \mathcal{C}(N)$ is implicitly assumed for each subterm $N$, the intended meaning of the pair $(S, \sim)$ is given as follows.

**Definition 4.5:** $[\![(S,\sim)]\!]$ is defined as

$$S \cup \{\tau_1 = \tau_2 \mid \tau_1 \sim \tau_2\} \cup \{(\alpha_N \neq unit) \Rightarrow \mathcal{C}(N) \mid [\alpha_N]_\sim \text{ is a type variable}\}.$$

We can show that $[\![(S,\sim)]\!]$ is preserved by rewriting.

**Notation 4.6:** We write $\Theta_M$ for the set of substitutions that maps each type variable in

$$
\begin{array}{l}
\{\alpha_N \mid N \text{ is a subterm of } M\} \cup \{\beta_x \mid x \text{ appears in } M\} \\
\cup \{\gamma_N \mid N \text{ is a subterm of } M \text{ and is of the form } proj_j(N')\}
\end{array}
$$

to a type term containing no type variables.

**Definition 4.7:** Let $\theta \in \Theta_M$. We write $\theta \models \tau_1 = \tau_2$ if $\theta\tau_1$ and $\theta\tau_2$ are syntactically equal. We also write $\theta \models (\tau \neq \mathit{unit}) \Rightarrow \{\tau_{11} = \tau_{12}, \ldots, \tau_{n1} = \tau_{n2}\}$ if either $\theta\tau = \mathit{unit}$ or $\theta\tau_{j1} = \theta\tau_{j2}$ for each $j \in \{1, \ldots, n\}$. Let $c_j, c'_j$ range over a constraint of the form $\tau_1 = \tau_2$ or $(\tau \neq \mathit{unit}) \Rightarrow \{\tau_{11} = \tau_{12}, \ldots, \tau_{n1} = \tau_{n2}\}$. We write $\theta \models \{c_1, \ldots, c_n\}$ if $\theta \models c_j$ for each $j \in \{1, \ldots, n\}$, and write $\{c_1, \ldots, c_n\} \models \{c'_1, \ldots, c'_k\}$ if $\theta \in \Theta_M$ and $\theta \models \{c_1, \ldots, c_n\}$ implies $\theta \models \{c'_1, \ldots, c'_k\}$. We write $\sim\!\models \{c_1, \ldots, c_n\}$ if $\{\tau_1 = \tau_2 \mid \tau_1 \sim \tau_2\} \models \{c_1, \ldots, c_n\}$.

**Lemma 4.8:** If $(S, \sim) \rightsquigarrow (S', \sim')$, then $\theta \models [\![(S, \sim)]\!] \Leftrightarrow \theta \models [\![(S', \sim')]\!]$ for any $\theta \in \Theta_M$.

We need some lemmas to prove the above lemma.

**Lemma 4.9:** If $(S, \sim) \rightsquigarrow (S', \sim')$ and $[\alpha_N]_\sim$ is a type variable but $[\alpha_N]_{\sim'}$ is neither a type variable nor *unit*, then $\mathcal{C}(N) \subseteq S'$.

**Proof:** By the definition of $\rightsquigarrow$, it must be the case that $S = S_1 \cup \{\tau_1 = \tau_2\}$, $S' = S_1 \cup \mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim) \cup \mathit{newC}(\tau_1, \tau_2, \sim)$, and $\sim'=\sim \cup\{\tau_1 = \tau_2\}$. Because $[\alpha_N]_\sim$ is a type variable but $[\alpha_N]_{\sim'}$ is not, $\alpha_N \sim \tau_i$, and $[\tau_i]_\sim$ is a type variable but $[\tau_j]_\sim$ is not for $i, j = 1, 2$ or $i, j = 2, 1$. Since $[\tau_j]_\sim = [\alpha_N]_{\sim'}$, $\mathcal{C}(N) \subseteq \mathit{newC}(\tau_1, \tau_2, \sim) \subseteq S'$. □

**Lemma 4.10:** If $(S, \sim) \rightsquigarrow (S', \sim')$, then $\theta \models S' \cup \{\tau_1 = \tau_2 \mid \tau_1 \sim' \tau_2\}$ implies $\theta \models S \cup \{\tau_1 = \tau_2 \mid \tau_1 \sim \tau_2\}$.

**Proof:** Trivial by the definition of $\rightsquigarrow$ (notice that if $(S_1 \cup \{\tau_1 = \tau_2\}, \sim_1) \rightsquigarrow (S_2, \sim_2)$ and $\tau_1 = \tau_2 \notin S_2$, then $\tau_1 \sim_2 \tau_2$). □

**Proof of Lemma 4.8:** $\Leftarrow$) We show that $\theta \models [\![(S', \sim')]\!]$ implies $\theta \models C$ for each constraint $C \in [\![(S, \sim)]\!]$. The case where $C$ comes from the first or second set of $[\![(S, \sim)]\!]$ follows immediately from Lemma 4.10. So, we need to consider only the case where $C$ comes from the third set of $[\![(S, \sim)]\!]$ and it is not an element of the third set of $[\![(S', \sim')]\!]$, i.e., case where $C$ is of the form $(\alpha_N \neq \mathit{unit}) \Rightarrow \mathcal{C}(N)$ and $[\alpha_N]_\sim$ is a type variable but $[\alpha_N]_{\sim'}$ is not. If $[\alpha_N]_{\sim'} = \mathit{unit}$, then $\theta \models C$ is vacuously true. So, suppose $[\alpha_N]_{\sim'} \neq \mathit{unit}$. In this case, $\mathcal{C}(N) \subseteq S'$ by Lemma 4.9, which implies $\theta \models C$.

$\Rightarrow$) We show that $\theta \models [\![(S, \sim)]\!]$ implies $\theta \models C$ for each constraint $C \in [\![(S', \sim')]\!]$. The case where $C$ comes from the third set of $[\![(S', \sim')]\!]$ is trivial, since, by the definition of $\rightsquigarrow$, $\sim' \supseteq \sim$, which implies that if $[\alpha_N]_{\sim'}$ is a type variable then so is $[\alpha_N]_\sim$. The case where $C$ comes from the second set is also trivial, since $\sim' \subseteq \sim \cup S$. If $C$ comes from the first set $S'$ and is not contained in $S$, then $C$ is either in $\mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim)$ or in $\mathit{newC}(\tau_1, \tau_2, \sim)$. In the former case, $\theta \models C$ follows from $(\tau_1 = \tau_2) \in S$. In the latter case, $C$ must be an element of $\mathcal{C}(N)$ such that $\alpha_N \sim \tau_i$, $[\tau_1]_\sim$ is a type variable, and $[\tau_2]_\sim$ is neither a type variable nor *unit*. So, $\theta \models C$ follows from the fact $\theta \models (\alpha_N \neq \mathit{unit}) \Rightarrow \mathcal{C}(N)$.

□

Next, we show that rewriting does not fail if the initial constraint has a solution.

**Lemma 4.11:** If $[\![(S, \sim)]\!]$ is satisfiable, then $(S, \sim) \not\rightsquigarrow^*$ **fail**.

**Proof:** By the definition of $\rightsquigarrow$, $(S, \sim) \not\rightsquigarrow$ **fail** if $[\![(S, \sim)]\!]$ is satisfiable. So, the result follows from Lemma 4.8. □

Next, we want to check that the final $\sim$ is a valid equivalence relation. For that purpose, it suffices to check that the following well-formedness condition is preserved by rewriting.

**Definition 4.12:** We say that $(S, \sim)$ is *well-formed* if, whenever $\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'$ or $\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'$, (i)$\tau_i \sim \tau_i'$ for $i = 1, 2$ hold or (ii)there exist $\tau_i''$ and $\tau_i'''$ s.t. $\tau_i \sim \tau_i''$, $\tau_i' \sim \tau_i'''$, and $\tau_i'' = \tau_i''' \in S$ for $i = 1, 2$.

**Lemma 4.13:** If $(S, \sim) \rightsquigarrow (S', \sim')$ and $(S, \sim)$ is well-formed, then $(S', \sim')$ is also well-formed.

**Proof:** Trivial by the definition of $\sim$. □

From the above lemmas, we obtain the correctness of rewriting.

**Lemma 4.14:** Let $\sim = \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\}$ and suppose $[\![(S, \sim)]\!]$ is satisfiable, i.e., $\theta \models [\![(S, \sim)]\!]$ for some $\theta \in \Theta_M$. If $(S, \sim) \rightsquigarrow^* (\emptyset, \sim')$, then $\sim'$ is the least valid equivalence relation $\mathcal{R}$ such that $\mathcal{R} \models [\![(S, \sim)]\!]$. (Here, we say that a valid equivalence relation $\mathcal{R}$ is *less than* $\mathcal{R}'$ if $\theta \models \mathcal{R}'$ implies $\theta \models \mathcal{R}$ for any $\theta \in \Theta_M$.)

**Proof:** First, we show that $\sim'$ is a valid equivalence relation. The first condition of a valid equivalence relation follows from Lemma 4.13. Because $[\![(S, \sim)]\!]$ is satisfiable, by Lemma 4.8, $\{(\tau_1, \tau_2) \mid \tau_1 \sim' \tau_2\}$ is also satisfiable. Therefore, $\sim'$ satisfies the second and third conditions of the valid equivalence relation.

The condition $\mathcal{R} \models [\![(S, \sim)]\!]$ follows immediately from Lemmas 4.9 and 4.10. The condition $\sim' \models S$ also follows immediately from Lemma 4.10.

Leastness follows from the fact that the conditions (i) and (ii) imply $\mathcal{R} \models [\![(S, \sim)]\!]$, which implies $\mathcal{R} \models [\![(\emptyset, \sim')]\!]$ by Lemma 4.8. □

Now we can check that our algorithm outputs a correct and optimal term

**Theorem 4.15:** Suppose $\Gamma \vdash M : \tau$. Then, $(\{\Gamma(x) = \beta_x \mid x \in dom(x)\} \cup \{\tau = \alpha_M\}, \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\})$ can be always rewritten by $\rightsquigarrow$ to $(\emptyset, \sim)$ for some $\sim$. Let $M'$ be a term obtained by replacing each subterm $N$ of $M$ s.t. $[\alpha_N]_\sim$ is a type variable with $()$. Then, $\Gamma \vdash M : \tau \Rightarrow M'$. Moreover, $M'$ is an optimal output, i.e., if $\Gamma \vdash M : \tau \Rightarrow M''$, then $M'' \succeq M'$.

**Proof sketch:** By Lemma 4.11, the rewriting does not fail. Termination of the rewriting is shown in Section 4.4.

Let $\tau_N$ and $\eta_x$ be type terms obtained by substituting *unit* for all type variables in $mgu_\sim(\alpha_N)$ and $mgu_\sim(\beta_x)$ respectively. By Lemma 4.14, we can construct a derivation tree for $\Gamma \vdash M : \tau \Rightarrow M'$, whose each node is of the form $x_1 : \eta_{x_1}, \ldots, x_n : \eta_{x_n} \vdash N : \tau_N \Rightarrow N'$. On the other hand, suppose there is a derivation tree $\Gamma \vdash M : \tau \Rightarrow M''$, whose node is $x_1 : \eta'_{x_1}, \ldots, x_n : \eta'_{x_n} \vdash N : \tau'_N \Rightarrow N''$. Let $\tau'_N$ be *unit* if there is no node of the form $\Delta \vdash N : \tau \Rightarrow N'$, and $\eta'_x$ be also *unit* if no type binding on $x$ appears in the derivation. Define also $\zeta_N$ by:

$$\zeta_{proj_1(N)} = \begin{cases} \rho_2 & \text{if } \tau'_N \text{ is of the form } \rho_1 \times \rho_2 \\ unit & \text{otherwise} \end{cases}$$

$$\zeta_{proj_2(N)} = \begin{cases} \rho_1 & \text{if } \tau'_N \text{ is of the form } \rho_1 \times \rho_2 \\ unit & \text{otherwise} \end{cases}$$

$\zeta_N$ is undefined if $N$ is not of the form $proj_j(N')$

Then, the reflexive and transitive closure of

$$\{(\tau_1, \tau_2) \mid \tau_1, \tau_2 \in \mathcal{N}(M) \text{ and } \theta\tau_1 = \theta\tau_2 \text{ for } \theta = [\overrightarrow{\tau'_N}/\overrightarrow{\alpha_N}, \overrightarrow{\eta'_x}/\overrightarrow{\beta_x}, \overrightarrow{\zeta_{proj_2(N)}}/\overrightarrow{\gamma_{proj_2(N)}}, \overrightarrow{\zeta_{proj_1(N)}}/\overrightarrow{\gamma_{proj_1(N)}}]\}$$

is a valid equivalence relation such that

$$\sim' \models [\![(\{\Gamma(x) = \beta_x \mid x \in dom(x)\} \cup \{\tau = \alpha_M\}, \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\})]\!] .$$

Since $\sim$ is the least one, $\tau'_N = \theta'(mgu_{\mathcal{R}}(\alpha_N))$ for some $\theta'$. Therefore, $M'' \succeq M'$. □

### 4.4 Efficiency of the algorithm

We show that the time complexity of the algorithm is almost linear (strictly speaking, $O(n\alpha(n,n))$, where $\alpha$ is the inverse of the Ackerman function and $n$ is the size of an input $\Gamma \vdash M : \tau$). We assume here that type terms in the input is expressed by directed acyclic graphs. because the initial set-up (for allocating type variables and computing the pair $(\{\Gamma(x) = \beta_x \mid x \in dom(x)\} \cup \{\tau = \alpha_M\}, \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\})$) and the final step (for replacing subterms of $M$ with ()) can be performed in a linear time, it is sufficient to show that the rewriting step can be performed in an almost linear time.

First, we estimate the cost of each rewriting step. Notice that non-constant operations involved in each rewriting step are only those for merging two equivalence classes and looking up the representative element of an equivalent class. So, the time complexity of each rewriting step is bound by those of merge and lookup-operations, whose amortized cost is $O(\alpha(n))$ [19].

Next, we show that the number of rewriting steps is $O(n)$. Because one element is removed from the set $S$ in each rewriting step, the number of rewriting steps is the size of the initial $S$ plus the number of elements added by $subC$ and $newC$. The size of the initial $S$ is clearly $O(n)$. Whenever $subC(\tau_1, \tau_2)$ is added, two equivalence classes are merged into one. So, $subC(\tau_1, \tau_2)$ can be added at most $(\mathcal{N}(M) - 1)$ times. Moreover, the size of $subC(\tau_1, \tau_2)$ is at most 2. Therefore, the number of elements added by $subC$ is $O(n)$. The number of elements added by $newC$ is also $O(n)$, because for each subterm $N$ of $M$, the size of $\mathcal{C}(N)$ is at most 3 and $\mathcal{C}(N)$ is added at most once.

By the above arguments, the time complexity of the algorithm is $O(n\alpha(n))$.

## 5 Extension with Polymorphism

In this section, we show that the results of Section 3, except for the cost of finding an optimal translation, hold also for a language with ML-style let polymorphism. An algorithm for finding an optimal translation is again obtained as a minor modification of a type reconstruction algorithm. Its worst-case cost is exponential in the size of an input, as for the ordinary type reconstruction porblem for ML [7]. We think, however, that the algorithm is efficient for realistic, well-typed programs: because the algorithm is just a simple modification of Henglein's type reconstruction algorithm [4] based on semi-unification (indeed, our algorithm just delays adding some semi-unification constraints until some type variable is instantiated), performing useless variable elimination after the usual type reconstruction is likely to just double the cost of perfoming only the usual type reconstruction. We show rules for deriving a translation in Section 5.1, prove their correctness and optimality in Section 5.2, and then explain an algorithm for obtaining an optimal translation.

### 5.1 Translation

The syntax of terms, types and type environments are extended as follows:

$$
\begin{array}{l}
M ::= \cdots \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \\
\tau ::= \cdots \mid \alpha \\
\sigma ::= \tau \mid \forall \alpha.\sigma \\
\Gamma ::= \emptyset \mid \Gamma, x : \sigma
\end{array}
$$

Here, $\alpha$ denotes a type variable, and $\tau$ and $\sigma$ are called a monotype and a polytype respectively. We write $\forall \overrightarrow{\alpha}.\tau$ for $\forall \alpha_1. \cdots \forall \alpha_n.\tau$.

The evaluation relation $M \Downarrow V$ is extended by adding the following rule:

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

Type judgement and translation judgement are of the form $\Gamma \vdash M : \tau$ and $\Gamma \vdash M : \tau \Rightarrow M'$ respectively, and rules are the same except for the rules given below. Here, $\rho$ ranges over monotypes. $FV(\tau)$ and $FV(\Gamma)$ denote the set of free type variable (i.e., type variables not bound by $\forall$) appearing in $\tau$ and $\Gamma$ respectively.

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

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

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

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

The realtion $\succeq$ is also extended by adding the following rule:

$$\textbf{let}\ x = M_1\ \textbf{in}\ M_2 \succeq \textbf{let}\ x = M_1'\ \textbf{in}\ M_2' \text{ if } M_1 \succeq M_1' \text{ and } M_2 \succeq M_2'$$

**Example 5.1:** 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:

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

It is transformed into

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

The following is a part of the derivation:

$$\frac{f : \sigma \vdash f : \tau \to int \Rightarrow f \qquad \dfrac{\cdots}{f : \sigma \vdash (\lambda v.proj_1(v), (1,2)) : \tau \Rightarrow (\lambda v.proj_1(v), (1,()))}}{f : \forall \alpha, \beta.(((\alpha \to \beta) \times \alpha) \to \beta) \vdash f(\lambda v.proj_1(v), (1,2)) : int \Rightarrow f(\lambda v.proj_1(v), (1,()))}$$

where $\sigma = \forall \alpha, \beta.(((\alpha \to \beta) \times \alpha) \to \beta)$ and $\tau = ((int \times unit) \to int) \times (int \times unit)$.

By eliminating 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 in the monomorphic case. It is indeed invalid in the monomorphic case because `f` is used polymorphically in the translated program.

## 5.2 Correctness

As in the monomorphic case, there is always at least one translation for a well-typed term.

**Theorem 5.2:** If $\Gamma \vdash M : \tau$, then $\Gamma \vdash M : \tau \Rightarrow M'$ for some $M'$.

**Proof:** $\Gamma \vdash M : \tau \Rightarrow M$ can be derived by straightfoward induction on the derivation of $\Gamma \vdash M : \tau$. □

The following theorem guarantees that the translated term is evaluated to the same value as the source term.

**Theorem 5.3 [Correctness]:** If $x_1 : \forall \overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall \overrightarrow{\alpha_n}.\tau_n \vdash M : int \Rightarrow M'$, $\vdash V_1 : \tau_1, \ldots, \vdash V_n : \tau_n$ and $[V_1/x_1, \ldots, V_n/x_n]M \Downarrow n$, then $[V_1/x_1, \ldots, V_n/x_n]M' \Downarrow n$.

This is obtained as a corollary of the following lemma (Lemma 5.4).

**Lemma 5.4:** 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 $V'$.

We need some lemmas to prove the above lemma.

**Lemma 5.5:** If $\Gamma \vdash M : \tau \Rightarrow M'$, then $[\tau'/\alpha]\Gamma \vdash M : [\tau'/\alpha]\tau \Rightarrow M'$

**Proof:** Straightforward induction on derifation of $\Gamma \vdash M : \tau \Rightarrow M'$. □

**Lemma 5.6 [Substitution]:** If $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M : \tau \Rightarrow M'$ and $\Gamma \vdash N : \tau_1 \Rightarrow N'$ and $\overrightarrow{\alpha}$ do not appear in $\Gamma$, then $\Gamma \vdash [N/x]M : \tau \Rightarrow [N'/x]M'$.

**Proof:** Induction on the derivation of $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M : \tau \Rightarrow M'$, with case analysis on the last rule.

- Case (Tr-Unit): In this case, $\tau = unit$ and $M' = ()$. Therefore, by (Tr-Unit), we have $\Gamma \vdash [N/x]M : \tau \Rightarrow ()(= [N'/x]M')$.
- Case (Tr-Var): In this case, $M$ is a variable. We have two cases to consider: the case where $M = x$ and the case where $M \neq x$. If $M = x$, then $[\overrightarrow{\rho}/\overrightarrow{\alpha}]\tau_1 = \tau$ and $M' = x$. So, $\Gamma \vdash [N/x]M : \tau \Rightarrow [N'/x]M'$ follows immediately from the assumption and Lemma 5.5. If $M = y \neq x$, then $[N/x]M = [N'/x]M' = y$. So, we obtain $\Gamma \vdash [N/x]M : \tau \Rightarrow [N/x]M'$ from the assumption.
- Case (Tr-Int): Trivial, since $M = M' = [N/x]M = [N'/x]M' = n$.
- Case (Tr-Add): In this case, $M = M_1 + M_2$, $M' = M_1' + M_2'$, $\tau = int$, $(\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M_1 : int)$, and $(\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M_2 : int)$. By induction hypothesis, $\Gamma \vdash [N/x]M_1 : int \Rightarrow [N'/x]M_1'$ and $\Gamma \vdash [N/x]M_2 : int \Rightarrow [N'/x]M_2'$. Therefore, we obtain $\Gamma \vdash [N/x]M : \tau \Rightarrow M'$ by using (Tr-Add).

- Case (Tr-Abs): In this case, $M = \lambda y.M_1$, $M' = \lambda y.M_1'$, $\tau = \tau_2 \to \tau_3$, and $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1, y : \tau_2 \vdash M_1 : \tau_3 \Rightarrow M_1'$. By induction hypothesis, $\Gamma, y : \tau_2 \vdash [N/x]M_1 : \tau_3 \Rightarrow [N'/x]M_1'$, from which we obtain $\Gamma \vdash \lambda y.[N/x]M_1 : \tau \Rightarrow \lambda y.[N'/x]M_1' (= [N'/x](\lambda y.M_1'))$.

- Case for (Tr-Fix): Similar to the case for (Tr-Abs).

- Cases for (Tr-App), (Tr-Pair), and (Tr-Proj): Similar to the case for (Tr-Add).

□

**Proof of Lemma 5.4:** We prove this lemma by induction on derivation of $[V_1/x_1, \ldots, V_n/x_n]M \Downarrow V$, but in order to simplify the case analysis, we show two cases independently: the case where $M' = ()$ and the case where $M$ is a variable.

- Case where $M' = ()$: In this case, $\tau = \mathit{unit}$. Therefore, the required properties hold for $V' = ()$.

- Case where $M$ is a variable: it must be the case that $M = x_i$, $\tau = [\overrightarrow{\rho}/\overrightarrow{\alpha_i}]\tau_i$, and $V = V_i$. $M'$ must be $x_i$ or $()$. The latter case has been shown above. So, suppose $M'$ is $x_i$. From $\vdash V_i : \tau_i$, we have $\vdash V_i : [\overrightarrow{\rho}/\overrightarrow{\alpha_i}]\tau_i$. So, the result holds for $V' = V_i$.

Now, we show the other cases by induction on derivation of $[V_1/x_1, \ldots, V_n/x_n]M \Downarrow V$, with case analysis on the last rule used. We can assume that $M$ is not a variable and that $M' \neq ()$, since we have already shown those cases. We write $\Gamma$ for $x_1 : \tau_1, \ldots, x_n : \tau_n$ and write $\theta, \theta'$ for substitutions $[V_1/x_1, \ldots, V_n/x_n]$, $[V_1'/x_1, \ldots, V_n'/x_n]$.

- Case for (E-Unit): In this case, $M = M' = ()$, so it is subsumed by the above case for $M' = ()$.

- Case for (E-Int): In this case, $M = M' = n$. So, the required properties hold for $V' = n$.

- Case for (E-Add): It must be the case that $M = M_1 + M_2$, $M' = M_1' + M_2'$, $\tau = \mathit{int}$, $\Gamma \vdash M_i : \mathit{int} \Rightarrow M_i'$, $\theta M_i \Downarrow n_i$, and $V$ is the sum of $n_1$ and $n_2$. By induction hypothesis, $\theta' M_i' \Downarrow n_i$ for $i = 1, 2$. So, by using (E-Add), we obtain $\theta' M' \Downarrow V$. The result follows, since $\vdash V : \mathit{int} \Rightarrow V$.

- Case for (E-Abs): In this case, $M = \lambda y.M_1$, $M' = \lambda y.M_1'$, $\tau = \tau' \to \tau''$, and $\Gamma, y : \tau' \vdash M_1 : \tau'' \Rightarrow M_1'$. By Lemma 5.6, we have $y : \tau' \vdash [V_1/x_1, \ldots, V_n/x_n]M_1 : \tau'' \Rightarrow [V_1'/x_1, \ldots, V_n'/x_n]M_1'$, from which we obtain

  $$\vdash \lambda y.[V_1/x_1, \ldots, V_n/x_n]M_1 : \tau \Rightarrow \lambda y.[V_1'/x_1, \ldots, V_n'/x_n]M_1'.$$

  Therefore, the required result hold for $V' = \lambda y.M_1'$.

- Case for (E-Fix): In this case, $M = \mathbf{fix}(f, y, M_1)$, $M' = \mathbf{fix}(f, y, M_1')$, $\tau = \tau' \to \tau''$, $V = \lambda y.[M/f]M_1$, and $\Gamma, f : \tau, y : \tau' \vdash M_1 : \tau'' \Rightarrow M_1'$. By (Tr-Abs), $\Gamma, f : \tau \vdash \lambda y.M_1 : \tau \Rightarrow \lambda y.M_1'$. By (Tr-Fix), we also have $\Gamma \vdash \mathbf{fix}(f, y, M_1) : \tau \Rightarrow \mathbf{fix}(f, y, M_1')$. By Lemma 5.6, $\Gamma \vdash \lambda y.[M/f]M_1 (= V) : \tau \Rightarrow \lambda y.[M'/f]M_1'$. Let $V' = \theta' \lambda y.[M'/f]M_1' = \lambda y.[\theta' M'/f]\theta' M_1'$. By using Lemma 5.6 again, we have $\vdash V : \tau \Rightarrow V'$ and $\theta' M' \Downarrow V'$.

- Case for (E-App): In this case, $M = M_1 M_2$, $\theta M_1 \Downarrow \lambda y.M_3$, $\theta M_2 \Downarrow W$, and $[W/y]M_3 \Downarrow V$. Moreover, it must be that $M' = M_1' M_2'$, $\Gamma \vdash M_1 : \tau' \to \tau \Rightarrow M_1'$ and $\Gamma \vdash M_2 : \tau' \Rightarrow M_2'$ by $\Gamma \vdash M_1 M_2 : \tau \Rightarrow M'$. By $\theta M_1 \Downarrow \lambda y.M_3$ and induction hypothesis, $\theta' M_1' \Downarrow \lambda y.M_3'$

and $y : \tau' \vdash M_3 : \tau \Rightarrow M_3'$ for some $M_3'$. By $\theta M_2 \Downarrow M_2'$, we also have $\theta' M_2' \Downarrow W$ and $\vdash W : \tau' \Rightarrow W'$ for some $W'$. So, by applying induction hypothesis again to $y : \tau' \vdash M_3 : \tau \Rightarrow M_3'$ and $[W/y]M_3 \Downarrow V$, we have $[W'/y]M_3' \Downarrow V'$ (from which $\theta' M' \Downarrow V'$ follows) and $\vdash V : \tau \Rightarrow V'$ for some $V'$.

- Case for (E-Proj): In this case, it must be that $M = proj_i(M_1)$, $\theta M_1 \Downarrow (V_1, V_2)$, $V_i = V$, $M' = proj_i(M_1')$, and $\Gamma \vdash M_1 : \tau \times \tau_2 \Rightarrow M_1'$. By induction hypothesis, $\theta' M_1' \Downarrow W'$ and $\vdash (V_1, V_2) : \tau \times \tau_2 \Rightarrow W'$ for some $W'$. By translation rules, $W'$ must be of the form $(V_1', V_2')$ and $\vdash V : \tau \Rightarrow V_i'$. The result follows for $V' = V_i'$, since $\theta' M' = proj_i(\theta' M_1') \Downarrow V'$.

- Case for (E-Pair): In this case, it must be that $M = (M_1, M_2)$, $M' = (M_1', M_2')$, $\theta M_1 \Downarrow W_1$, $\theta M_2 \Downarrow W_2$, $V = (W_1, W_2)$, $\tau = \tau_1 \times \tau_2$, $\Gamma \vdash M_1 : \tau' \Rightarrow M_1'$, and $\Gamma \vdash M_2 : \tau_2 \Rightarrow M_2'$. By induction hypothesis, $\theta' M_1' \Downarrow W_1'$, $\theta' M_2' \Downarrow W_2'$, $\vdash W_1 : \tau_1 \Rightarrow W_1'$ and $\vdash W_2 : \tau_2 \Rightarrow W_2'$. Therefore, $\theta' M' \Downarrow V'$ and $\vdash V : \tau \Rightarrow V'$ for $V' = (W_1', W_2')$.

- Case where the last rule is (E-Let): In this case, $M = \textbf{let } x = M_1 \textbf{ in } M_2$, $M' = \textbf{let } x = M_1' \textbf{ in } M_2'$, $\Gamma \vdash M_1 : \rho \Rightarrow M_1'$, and $\Gamma, x : \forall \overrightarrow{\alpha}.\rho \vdash M_2 \Rightarrow M_2'$. Moreover, by $\theta M \Downarrow V$, there must be $W$ such that $\theta M_1 \Downarrow W$ and $[W/x]\theta M_2 \Downarrow V$. By induction hypothesis, $\theta' M_1' \Downarrow W'$ and $\vdash W : \rho \Rightarrow W'$ for some $W'$. By applying induction hypothesis again to $[W/x]\theta M_2 \Downarrow V$ and $\Gamma, x : \forall \overrightarrow{\alpha}.\rho \vdash M_2 : \tau \Rightarrow M_2'$, we obtain $[W'/x]\theta' M_2' \Downarrow V'$ (which implies $\theta' M' \Downarrow V'$) and $\vdash V : \tau \Rightarrow V'$ for some $V'$, as required.

□

**Theorem 5.7 [optimality]:** If $\Gamma \vdash M : \tau$, $\Gamma \vdash M' : \tau$, and $M \succeq M'$, then $\Gamma \vdash M : \tau \Rightarrow M'$

**Proof:** We show a stronger property "If $\Gamma \vdash M' : \tau$ and $M \succeq M'$, then $\Gamma \vdash M : \tau \Rightarrow M'$ by induction on derivation of $\Gamma \vdash M' : \tau$ with case analysis on the last rule used. We show only main cases. The other cases are similar and trivial.

- Case (T-Unit): $M'$ must be $()$. So, by using (Tr-Unit), we obtain $\Gamma \vdash M : unit \Rightarrow M'$.

- Case (T-Var): $M'$ and $M$ must be $x$. So, by using (Tr-Var), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

- Case (T-Abs): It must be the case that $M' = \lambda x.M_1'$, $M = \lambda x.M_1$, $M_1 \succeq M_1'$, $\tau = \tau_1 \rightarrow \tau_2$, and $\Gamma, x : \tau_1 \vdash M_1' : \tau_2$. By induction hypothesis, $\Gamma, x : \tau_1 \vdash M_1 : \tau_2 \Rightarrow M_1'$. So, by using (Tr-Abs), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

- Case (T-Let): It must be the case that $M' = \textbf{let } x = M_1' \textbf{ in } M_2'$, $M = \textbf{let } x = M_1 \textbf{ in } M_2$, $M_i \succeq M_i'$, $\Gamma \vdash M_1' : \tau_1$, and $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M_2' : \tau$. By induction hypothesis, $\Gamma \vdash M_1 : \tau_1 \Rightarrow M_1'$ and $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M_2 : \tau \Rightarrow M_2'$. So, by using (Tr-Let), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

- Case (T-App): It must be the case that $M' = M_1' M_2'$, $M = M_1 M_2$, $M_i \succeq M_i'$, $\Gamma \vdash M_1' : \tau_1 \rightarrow \tau$, and $\Gamma \vdash M_2' : \tau_1$. By induction hypothesis, $\Gamma \vdash M_1 : \tau_1 \rightarrow \tau \Rightarrow M_1'$ and $\Gamma \vdash M_2 : \tau_1 \Rightarrow M_2'$. By using (T-App), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

□

### 5.3 Finding an Optimal Translation

In this section, we sketch an algorithm to find an optimal translation in the polymorphic case. The basic strategy of the algorithm is the same as in the monomophic case described in Section 4: constraints on type variables are added *lazily* when some type variables are instantiated to some type other than (). Only difference from the monomorphic case is that Henglein's constraints based on semi-unification [4] are used, instead of unification constraints.

Following Henglein's typing rules [4], we use the following translation rules in which only monotypes appear.

$$\Gamma; \tau \vdash M : \mathit{unit} \Rightarrow () \qquad \text{(Tr2-Unit)}$$

$$\frac{\tau \times \rho \leq \tau' \times \rho}{\Gamma, x : \tau; \rho \vdash x : \tau' \Rightarrow x} \qquad \text{(Tr2-Var)}$$

$$\Gamma; \tau \vdash n : \mathit{int} \Rightarrow n \qquad \text{(Tr2-Int)}$$

$$\frac{\Gamma; \tau \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma; \tau \vdash M_2 : \mathit{int} \Rightarrow M_2'}{\Gamma; \tau \vdash M_1 + M_2 : \mathit{int} \Rightarrow M_1' + M_2'} \qquad \text{(Tr2-Add)}$$

$$\frac{\Gamma, x : \tau_1; \tau_1 \times \rho \vdash M : \tau_2 \Rightarrow M'}{\Gamma; \rho \vdash \lambda x.M : \tau_1 \to \tau_2 \Rightarrow \lambda x.M'} \qquad \text{(Tr2-Abs)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma, x : \tau_1; \rho \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma; \rho \vdash \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2' : \tau_2} \qquad \text{(Tr2-Let)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; (\tau_1 \to \tau_2) \times \rho \vdash M : \tau_2 \Rightarrow M'}{\Gamma; \rho \vdash \mathbf{fix}(f, x, M) : \tau_1 \to \tau_2 \Rightarrow \mathbf{fix}(f, x, M')} \qquad \text{(Tr2-Fix)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \to \tau_2 \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \tau_1 \Rightarrow M_2'}{\Gamma; \rho \vdash M_1 M_2 : \tau_2 \Rightarrow M_1' M_2'} \qquad \text{(Tr2-App)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma; \rho \vdash (M_1, M_2) : \tau_1 \times \tau_2 \Rightarrow (M_1', M_2')} \qquad \text{(Tr2-Pair)}$$

$$\frac{\Gamma; \rho \vdash M : \tau_1 \times \tau_2 \Rightarrow M'}{\Gamma; \rho \vdash \mathit{proj}_i(M) : \tau_i \Rightarrow \mathit{proj}_i(M')} \qquad \text{(Tr2-Proj)}$$

Here, instead of explicitly using type schemes, type variables that appear in $\Gamma$ but not in $\rho$ are implicitly assumed to be quantified in the translation relation $\Gamma; \rho \vdash M : \tau \Rightarrow M'$. For example, $x : (\alpha \times \beta) \to \alpha; \beta \vdash M : \tau \Rightarrow M'$ corresponds to $x : \forall\alpha.((\alpha \times \beta) \to \alpha) \vdash M : \tau \Rightarrow M'$ of the original translation relation. So, the role of $\rho$ of $\Gamma; \rho \vdash M : \tau \Rightarrow M'$ is used just for remembering which type variables cannot be generalized. That is why we put the types of $x$ and $f$ in (Tr2-Abs) and (Tr2-Fix) into the $\rho$-part (if we allow polymorphic recursion, it is unnecessary to put the type of $f$ into it in (Tr2-Fix)). The semi-unification constraint $\tau_1 \leq \tau_2$ holds if there exist substitutions $\theta$ such that $\theta\tau_1 = \tau_2$ [4]; so, for example, $(\alpha_1 \times \mathit{int}) \to \alpha_2 \leq \alpha_3 \times \mathit{int}$ is solved by substituting $\beta_1 \times \mathit{int}$ for $\alpha_3$. The constraint $\tau \times \rho \leq \tau' \times \rho$ in the rule (Tr2-Var) allows type variables that appear in $\tau$ but not in $\rho$ to be instantiated with arbitrary types. Please refer to [4] for more explanation.

It follows by the same argument as [4] that the above rules are equivalent to the original rules in the following sense.

**Theorem 5.8:** $x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau \Rightarrow M'$ if and only if $x_1 : \tau_1, \ldots, x_n : \tau_n; \rho_1 \times \cdots \times \rho_k \vdash M : \tau \Rightarrow M'$ where $\{\rho_1, \ldots, \rho_k\} = (FV(\tau_1) \cup \cdots \cup FV(\tau_n)) \backslash \{\overrightarrow{\alpha_1}, \ldots, \overrightarrow{\alpha_n}\}$.

We can now obtain an algorithm for finding an optimal translation in a similar manner to the monomorphic case. Let $x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau$ be an input term. Assign fresh type variables $\alpha_N, \alpha'_N$ to each subterm $N$ and a fresh type variable $\beta_x$ to each variable $x$. Assign a fresh type variable $\gamma_N$ also to each subterm $N$ of the form $proj_j(N')$. Then, a derivation whose conclusion is

$$x_1 : \tau_1, \ldots, x_n : \tau_n; \rho \vdash M : \tau \Rightarrow M'$$

and whose node is of the form

$$y_1 : \beta_{y_1}, \ldots, y_k : \beta_{y_k}; \alpha'_N \vdash N : \alpha_N \Rightarrow N'$$

is obtained if and only if the following conditions are met:

$$\{\tau_1 = \beta_{x_1}, \ldots, \tau_n = \beta_{x_n}, \tau = \alpha_M, \alpha'_M = \rho\} \cup \{(\alpha_N \neq unit) \Rightarrow \mathcal{C}_p(N) \mid N \text{ is a subterm of } M\}$$

Here, $\mathcal{C}_p(N)$ is defined as follows:

$\mathcal{C}_p(x) = \{\beta_x \times \alpha'_x \leq \alpha_x \times \alpha'_x\}$
$\mathcal{C}_p(n) = \{\alpha_n = int\}$
$\mathcal{C}_p(M_1 + M_2) = \{\alpha_{M_1+M_2} = \alpha_{M_1} = \alpha_{M_2} = int, \alpha'_{M_1+M_2} = \alpha'_{M_1} = \alpha'_{M_2}\}$
$\mathcal{C}_p(\lambda x.M) = \{\alpha_{\lambda x.M} = \beta_x \to \alpha_M, \alpha'_M = \beta_x \times \alpha'_{\lambda x.M}\}$
$\mathcal{C}_p(\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) = \{\alpha_{M_1} = \beta_x, \alpha_{\mathbf{let}\ x=M_1\ \mathbf{in}\ M_2} = \alpha_{M_2}, \alpha'_{\mathbf{let}\ x=M_1\ \mathbf{in}\ M_2} = \alpha'_{M_1} = \alpha'_{M_2}\}$
$\mathcal{C}_p(\mathbf{fix}(f, x, M)) = \{\alpha_{\mathbf{fix}(f,x,M)} = \beta_f = \beta_x \to \alpha_M, \alpha'_M = \beta_f \times \alpha'_{\mathbf{fix}(f,x,M)}\}$
$\mathcal{C}_p(M_1 M_2) = \{\alpha_{M_1} = \alpha_{M_2} \to \alpha_{M_1 M_2}, \alpha'_{M_1} = \alpha'_{M_2} = \alpha_{M_1 M_2}\}$
$\mathcal{C}_p((M_1, M_2)) = \{\alpha_{(M_1,M_2)} = \alpha_{M_1} \times \alpha_{M_2}, \alpha'_{M_1} = \alpha'_{M_2} = \alpha_{(M_1,M_2)}\}$
$\mathcal{C}_p(proj_1(M)) = \{\alpha_M = \alpha_{proj_1(M)} \times \gamma_{proj_1(M)}, \alpha'_M = \alpha'_{proj_1(M)}\}$
$\mathcal{C}_p(proj_2(M)) = \{\alpha_M = \gamma_{proj_2(M)} \times \alpha_{proj_2(M)}, \alpha'_M = \alpha'_{proj_2(M)}\}$

The above constraints can be solved in a similar manner to the monomorphic case. Starting with the initial set of constraints $\{\tau_1 = \beta_{x_1}, \ldots, \tau_n = \beta_{x_n}, \tau = \alpha_M, \alpha'_M = \rho\}$, we can solve it step by step by using Henglein's graph rewriting-based semiunification algorithm [4]. When it is known that $\alpha_N$ cannot be instantiated to () (i.e., when the constraint $\alpha_N = \tau$ or $\alpha_N \leq \tau$ is found for a type $\tau$ that is neither a type variable nor *unit*), $\mathcal{C}_p(N)$ is added to the constraints.

When the rewriting terminates and no more constraint needs to be added, find type variables $\alpha_N$ for which no constraint of the form $\alpha_N = \tau$ or $\alpha_N \leq \tau$ for a type $\tau$ other than a type variable and *unit* is imposed. Then, we can obtain an optimal output $M'$ by replacing every such $N$ of $M$ with ().

# 6 Discussion

## 6.1 Comparison with the 0CFA-based method

Our type-based transformation is optimal and can be performed in almost linear time for the simply-typed language. On the other hand, Wand and Siveroni's method is based on 0CFA, which costs cubic time in the worst case [3]. So, the question arises why the 0CFA-based method takes extra time. The reason 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
```

In the 0CFA-based method, the flow information into $f$ and $g$ are computed separately and it is analyzed that `fn (x,y)=>x` flows into `f` but `fn (x,y)=>x+y` does not. However, this precise information is unnecessary: even if we know it, we cannot transform `f(1,2)` into `f(1)`. In order to perform it, we must transform `fn (x,y)=>x` into `fn x=>x` , but we cannot, because it also flows into `g` and therefore `g(2,3)` must also be changed to `g(2)`, which is not valid since `fn (x,y)=>x+y` also flows into it. In other words, because `fn (x,y)=>x` flows into both $f$ and $g$, $f$ and $g$ must be called by using the same interface; so, it is useless to keep flow information on $f$ and $g$ separately. Therefore, an equality-based analysis like type-based analysis without structural subtyping and 0CFA$_=$[12] is sufficient for the useless variable elimination.

Moreover, in the case of a polymorphic language, our type-based method is still optimal but the 0CFA-based method is not. Consider the following program:

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

Our type-based method can transform it into

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

by assigning a type $int \times unit \to int$ to `fn (x,y)=>x` and a type scheme $\forall\alpha,\beta.((\alpha \to \beta)\times\alpha \to \beta)$ to $h$. It can be further simplified to

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

by eliminating the unnecessary parameter (). On the other hand, the 0CFA-based method cannot perform such translation: because `fn (x,y)=>x` and `fn (x,y)=>x+y` flow into the variable `f`, they are judged to depend on the same arguments (see Section 6, [20]).

Replacing 0CFA with $n$CFA [17] does not help much. Of course, there is a case where $n$CFA can produce a better result than 0CFA. Consider the following program:

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

With 1CFA, we know that only `fn (x,y)=>x` can flow into `f`, and therefore, we can transform it into:

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

However, 1CFA does not work for the slightly different program below: Because it cannot analyze that only `fn (x,y)=>x` flow into `f`, the function call `f(1,2)` cannot be replaced with `f(1)`.

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

Similarly, CFA with polymorphic splitting [21] would not be so effective for useless variable elimination. Indeed, it does not work for the last example, either.

In addition to the above problem of not being optimal, CFA-based methods seem to suffer from the cost of unnecessary computation. $n$CFA computes flow information for each pair of a program point and a $n$-call string, but in order to change a function call `f(x,y)` to `f(x)`, every closure that may flow into `f` must be changed in the same way, *irrespectively of call strings.*

### 6.2 Other related work

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 [10] (although his analysis has been proposed for a different purpose). We can extend the usual type system by annotating each type constructor with use 0 (indicating values are not used at all) or use $\omega$ (indicating values may be used once or more). Then, we can perform type reconstruction and replace an expression of use 0 with () (so, a type annotated with use 0 corresponds to *unit* in our translation). However, for a polymorphic language, it is not clear how to extend the use analysis with polymorphism so that the same transformation can be performed as our method.

The underlying idea behind our method is similar to those of aggressive garbage collection schemes [1, 5] based on Reynolds' abstraction/parametricity theorem [16], 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 useless variable elimination from those garbage collection schemes are that type inference is performed at compile time, and that any expressions on which no type constraint is imposed by the context, not just heap values, are eliminated.

### 6.3 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 described in Section 5 wrongly transforms `(fn (x,y)=>x=y)(1,2)` into `(fn (x, y)=>()=())((), ())`. In order to avoid this kind of wrong transformation, we can just 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-effect** Because our translation replaces whatever expression that can have type *unit* with (), it does not preserve non-termination and side-effect. 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 `()`.

In order to preserve non-termination and side effect, we just need 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 [18], depending on what kind of side effect we want to preserve. By regarding calls to recursive functions and `print` as having side effect, 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 powerful useless variable elimination** In this paper, we viewed useless variable elimination as a transformation replacing some subexpressions with (). It is possible to eliminate more useless variables by allowing more aggressive transformation. Such transformations can be performed by an appropriate extension of the type system.

For example, consider the following program:

```
let
  val f = fn (x, y)=>x
  val g = fn (x, y)=>x+y
  val h = if b then f else g
in
  (f(1,2), g(2,3), h(3, 4))
end
```

Our method cannot eliminate the second argument of `f` because `f` and `g`, which requires the second argument, flow into the same variable `h`. However, if we allow explicit coercion to be inserted for adjusting the calling interface, the above program can be transformed into

```
let
  val f = fn x=>x
  val g = fn (x, y)=>x+y
  val h = if b then fn (x,y)=>f(x) else g
in
  (f(1), g(2,3), h(3, 4))
end
```

Here, `f` of `if b then f else g` was replaced with `fn (x,y)=>f(x)`. This kind of translation can be enabled by introducing structural subtyping with an axiom $\tau \leq \mathit{unit}$. Or, we can use 0CFA for a monomorphic language (in this case, the flow information computed by 0CFA is useful).

Instead of allowing explicit coercion, we can also eliminate useless variables by allowing replacement of subexpressions with some *dummy* values other than (). For example, consider the following program:

```
let
  fun f(x,y) = if p(x) then x else f(x-1, expensive_computation(x+y, v))
  val g = fn (x, y)=>x+y
  val h = if b then f else g
in
  (f(1,2), h(3, 4))
end
```

The second argument of `f` does not affect the result. So, it can be transformed into

```
let
  fun f(x, y) = if p(x) then x else f(x-1, 0)
  val g = fn (x, y)=>x+y
  val h = if b then f else g
in
  (f(1,0), h(3, 4))
end
```

Here, `expensive_computation(x+y, v)` was replaced with a dummy value 0 of the same type. By this transformation, we could eliminate useless free variable `v` and redundant computation. This kind of transformation can be achieved by extending the type system with uses (as mentioned in the above subsection) and a use-based subtyping relation (such that, for example, $\mathit{int}^0 \rightarrow^\omega \mathit{int}^\omega \leq \mathit{int}^\omega \rightarrow^\omega \mathit{int}^0$ holds). A subterm of type $\tau$ with use 0 can be replaced with a dummy value of type $\tau$ instead of ().

For the above program, polymorphic recursion [4] is also effective. By assigning a type scheme $\forall\alpha.(\mathit{int} \times \alpha \rightarrow \mathit{int})$ to `f`, we can replace it with the following program:

```
let
  fun f(x,y) = if p(x) then x else f(x-1, ())
  val g = fn (x, y)=>x+y
  val h = if b then f else g
in
  (f(1,()), g(2,3), h(3, 4))
end
```

## 7 Concluding Remarks

We have formalized a type-based method for useless variable elimination and shown its correctness and optimality.

It is widely known that type systems and flow analyses are closely related [13, 12, 14, 2]. 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 useless variable elimination.

For an ML-style polymorphic language, polymorphism provides an additional power of useless variable elimination (recall Example 5.1). Because no reasonable control flow analysis (except for an obvious, but inefficient analysis that inlines all let-definitions before control flow analysis) is known to the author's knowledge that corresponds exactly to the polymorphic type inference, 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, unlike the flow analysis-based method.

Note, however, that the above argument does not completely deny the CFA-based method. CFA may be necessary anyway for other optimizations (for example, for replacing **if** $b$ **then** $M_1$ **else** $M_2$ with $M_1$): in that case, it would be a good idea to reuse the result of CFA for useless variable elimination.

## References


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

[2] Nevin Heintze. Control-flow analysis and type systems. In *International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206, 1995.

[3] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 342–351, 1997.

[4] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[5] Haruo Hosoya and Akinori Yonezawa. Formal description for collecting reachable garbage via dynamic type inference. In *Proceedings of the Second International Workshop on Types in Compilation (TIC98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 215–239, 1998.

[6] Simon Peyton Jones and Andreé Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.

[7] Paris C. Kanellakis, Harry G. Mairson, and John 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.

[8] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[9] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[10] Torben Mogensen. Types for 0, 1 or many uses. In *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 112–122, 1998.

[11] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[12] Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.

[13] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.

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

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

[16] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Proceedings of Information Processing 83*, pages 513–523, 1983.

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

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

[19] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[20] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[21] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998.