Type-Based Useless-Variable Elimination [†]

Naoki Kobayashi (koba@is.s.u-tokyo.ac.jp)
Department of Information Science, University of Tokyo

2000/04/29

Abstract. Useless-variable elimination is a transformation that eliminates variables whose values contribute nothing to the final outcome of a computation. We present a type-based method for useless-variable elimination and prove its correctness. The algorithm is a surprisingly simple extension of the usual type-reconstruction algorithm. Our method has several attractive features. First, it is simple, so that the proof of the correctness is clear and the method can be easily extended to deal with a polymorphic language. Second, it is efficient: for a simply-typed λ-calculus, it runs in time almost linear in the size of an input expression. Moreover, our transformation is *optimal* in a certain sense among those that preserve well-typedness, both for the simply-typed language and for an ML-style polymorphically-typed language.

Keywords: control-flow analysis, type-based analysis, type inference, useless-variable elimination

1. Introduction

1.1. Background

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

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

into:

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

by eliminating the useless variable `bogus`. (Here, we ignore exceptions that may be caused by arithmetic operations: See Section 1.4 below.)

[†] A revised and extended version of the paper that has appeared in Proceedings of 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00), pp.84-93, 2000.

[1] This example is taken from [29] and originally comes from [25].

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

Shivers [25] first defined the problem of UVE and gave an algorithm for it. His algorithm uses control-flow information. Wand and Siveroni [29] recently reformalized Shivers' UVE and proved its correctness. Wand and Siveroni's algorithm is based on 0CFA [25], a context-insensitive control-flow analysis, and it takes cubic time in the worst case. The proof of the correctness of the algorithm is rather complicated.

1.2. Our Proposal in This Paper

In this paper, we present a *type-based* UVE for an ML-style polymorphically-typed, call-by-value functional language. We define UVE as the following two-step transformation: In the first step, we replace as many subexpressions as possible with an empty value () so that the evaluation result of the whole expression does not change. In the second step, we simplify the resulting expression by eliminating unnecessary ()'s. 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 simplify the program to obtain the optimized program shown before, by eliminating `()`'s. This view of UVE as a two-step transformation clarifies the essence of UVE: we think the essence lies in the first transformation. It also seems advantageous in that no special treatment is necessary for eliminating part of a structured data argument.

Since the second transformation — simplifying a program by eliminating ()'s — 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.[2] Intuitively, this is because no operations (like pair destructors and arithmetic operations) that inspect the values of the subexpressions can be applied inside the expression: If there were such operations, replacing the subexpressions with () would make the expression ill-typed. Therefore, we can replace a subexpression with () when the type of the subexpression can be assumed to be *unit* (the type of ()) in the surrounding context.

We now explain how to obtain a transformation in more detail. What we essentially need to do is to perform type inference in a top-down [14] and on-demand manner: we need to perform type inference for a subterm N only when it is found from type inference of the surrounding context that N must produce a non-empty value (because otherwise N can be just replaced with (), whatever expression N is). If some subterm N remains unchecked, then N can be replaced with an empty value (). For example, consider a term $(\lambda(x,y).x)(1,2)$ of integer type. By performing type inference in a top-down manner, we know that $\lambda(x,y).x$ must have type $\mathit{int}\times\alpha \to \mathit{int}$ and $(1,2)$ must have type $\mathit{int}\times\alpha$. The usual type inference then checks 2 against type α and obtains $\alpha = \mathit{int}$, but in our case, 2 is not checked because α 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 terms of type α with (). It is then simplified to $(\lambda x.x)1$ by eliminating ().

1.3. Contributions

This is not the first work that studied the problem of UVE for functional languages: Shivers [25] and Wand and Siveroni [29] proposed CFA-based methods, and Fischbach and Hannan [5] recently, independently from us, proposed a type-based method in which types are annotated with information on whether values are needed or not. Damiani et al. [2, 3] also proposed a non-standard type system of PCF for a slightly different problem. Our contributions can be summarized as follows:

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

[2] The same observation was also made by Berardi [1]. It is also related with Reynolds' abstraction/parametricity theorem [24].

- Efficiency: Wand and Siveroni's method costs cubic time, while ours costs only almost linear time for the simply-typed case. With polymorphism, the cost can be exponential in the worst case, but we believe the algorithm works well in practice (just as ML type inference does). Detailed analysis of the cost, which is missing in other work, is also one of our contributions.
- Treatment of polymorphism: Our method can deal with Hindley-Milner polymorphism, while other type-based methods [2, 3, 5] do not. Although it is mentioned in their papers that they can extend the method to deal with polymorphism, it does not seem obvious how to do so while preserving the optimality of their methods, because polymorphism brings more opportunities for UVE and UVE in turn brings more polymorphism, as will be demonstrated in Section 3.
- Optimality of the method: Our method is optimal among any type-preserving UVE, that is, (i) any sound (i.e., semantics-preserving) and type-preserving transformation can be obtained by using our method, and (ii) there is indeed an algorithm to obtain an optimal transformation (that replaces more subexpressions with () than any other transformation) by using our method. In order to avoid confusion, we often refer to the property (i) alone by the *completeness* of our method. As mentioned above, how to find an optimal transformation in the presence of polymorphism is not so trivial. In fact, Wand and Siveroni's method [29] is *not* optimal for the polymophically-typed language.

The optimality of our method implies that, while subset-based analyses (like 0CFA) have been used in the previous methods for UVE [25, 29], it is actually sufficient to use equality-based analyses like our type-based analysis without subtyping or $0\mathrm{CFA}_=$[19] for our definition of UVE. An informal account of this is given in Section 7.

1.4. Limitations

Our type-based UVE has the following limitations.

- Non-termination and side-effects: The transformation method sketched above does not preserve side effects (including non-termination). For example, consider an expression $(\lambda x.1)M$ and suppose that M diverges. The expression is transformed into $(\lambda x.1)()$. In order to preserve side effects, we must perform an effect-analysis [26] and disallow the replacement of a side-effecting expression. For simplicity, we first ignore the problem of side effects and then discuss a remedy for this later (in Section 6.2). Although the remedy is very simple, the optimality no longer holds. If we must preserve divergence in the call-by-value semantics, $(\lambda x.1)M$ can be replaced with $(\lambda x.1)()$ if and only if M converges. It is therefore undecidable whether or not a transformation is optimal.

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

1.5. Structure of the Rest of This Paper

The rest of this paper is structured as follows. Section 2 introduces the syntax and typing of the source language. Section 3 gives transformation rules and shows their correctness and completeness. Section 4 describes how to find an optimal transformation. Section 5 briefly describes how to simplify the transformed program by removing (). Section 6 discusses extensions to deal with side effects, equality types, etc. Section 7 discusses related work, and Section 8 concludes this paper.

2. Syntax and Operational Semantics of the Language

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

2.1. Terms

DEFINITION 2.1 (terms). *The syntax of terms is defined as follows:*

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

Here, n ranges over integers.

The expression $M_1{+}M_2$ denotes the summation of integers, $\mathbf{fix}(f, x, M)$ denotes a recursive function f such that $f = \lambda x.M$,[3] and $\mathit{proj}_i(M)$ denotes a pair projection. The expression $\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3$ evaluates M_2 if M_1 is evaluated to 0, and evaluates M_3 otherwise. The value () is a special constant: its semantics is the same as the () in ML [16], but it plays a special role in the formalization of UVE in this paper.

Bound and free variables of M are defined as usual: f and x are bound in $\mathbf{fix}(f, x, M)$ and x is bound in M of $\lambda x.M$ and $\mathbf{let}\ x = N\ \mathbf{in}\ M$. We assume that α-conversion is implicitly performed as necessary, so that all bound variables are always different from each other and from free variables. We write $[N/x]M$ for a term obtained by substituting N (with α-conversion if necessary) for all free occurrences of x in M.

2.2. Operational Semantics

We define a natural-semantics-style operational semantics by using the relation $M \Downarrow V$, which means that the term M is evaluated to the value V. It is defined as the least relation closed under the rules given in Figure 1. In the rule (E-Add), "+" in $n_1{+}n_2$ denotes the semantic addition of two integers.

2.3. Typing

DEFINITION 2.2 (types). *The sets of types and type schemes, ranged over by τ and σ respectively, are given by the following syntax.*

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

We also use a meta-variable ρ for types. We write $FV(\sigma)$ for the set of free type variables (i.e., type variables not bound by $\forall$) in σ. We identify types up to α-conversion of bound type variables.

DEFINITION 2.3. *A type environment, denoted by Γ, is a mapping from a finite set of variables to the set of type schemes.*

NOTATION 2.4. *We write $\mathit{dom}(\Gamma)$ for the domain of Γ. When $x_1, \ldots, x_n$ are distinct from each other, the sequence $x_1\!:\!\sigma_1, \ldots, x_n\!:\!\sigma_n$ denotes the type environment Γ such that $\mathit{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \sigma_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin \mathit{dom}(\Gamma)$ holds, $\Gamma, x\!:\!\sigma$*

[3] We do not use the general fixed-point operator because our language is call-by-value.

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

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

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

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

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

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

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

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

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

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

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

Figure 1. Operational Semantics

denotes the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \sigma$, and $\Delta(y) = \Gamma(y)$ for each $y \in dom(\Gamma)$. $FV(\Gamma)$ denotes the set $\bigcup_{x \in dom(\Gamma)} FV(\Gamma(x))$.

DEFINITION 2.5. *A typing relation $\Gamma \vdash M : \tau$ is the least relation closed under the rules given in Figure 2. (In the figure, $\overrightarrow{\alpha}$ stands for a sequence $\alpha_1, \ldots, \alpha_n$, and $\forall\overrightarrow{\alpha}.\tau$ stands for $\forall\alpha_1.\cdots\forall\alpha_n.\tau$.)*

3. Useless-Variable Elimination

We formalize UVE as a type-preserving, source-to-source transformation that replaces some subterms with an empty value (). In Sections 3.1 and 3.2, we give rules for deriving such transformations and prove their

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

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

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

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

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

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

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

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

$$\text{(T-Let)}$$

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

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

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

Figure 2. Typing Rules

correctness. Then, we show in Section 3.3 that they are optimal in the sense that any valid transformation can be derived from those rules. An algorithm for finding an optimal transformation is deferred until Section 4.

3.1. Transformation Rules

We want to formalize rules for deriving from a well-typed term M a term M' such that M' is obtained by replacing some subterms of M with () and the evaluation result of M' is the same as that of 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 τ under Γ by replacing some terms with (). It is defined as the least relation closed under the rules given in Figure 3. The key idea is embodied in the rule (Tr-Unit): It means that if we want some part of a term to produce a value of type *unit*, then we can just replace that part with $()$ *regardless of what that part is.* More intuitively, it means that if we do not need information on the evaluation result of some part of a term, we can just replace it with an empty value $()$. The other rules are almost the same as the corresponding typing rules.

Note that the transformation rules are non-deterministic (in addition to the usual non-determinism on how generic type variables are instantiated in (Tr-Var)): for example, if $M = M_1 M_2$ and $\tau = \mathit{unit}$, then we can apply either (Tr-Unit) or (Tr-App). This is just for technical convenience: Thanks to this, we always have a valid identity transformation $\Gamma \vdash M : \tau \Rightarrow M$ for any well-typed term (see Theorem 3.7). If we want to avoid the non-determinism, we can just add the side condition $\tau_2 \neq \mathit{unit}$ to (Tr-App), etc. (Note that $\Gamma \vdash M : \tau \Rightarrow M$ would no longer hold if we did so.)

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 that a function f does not use the second argument and that f has a monomorphic type. 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-Var) and (Tr-App), $f(1,2)$ can be transformed into $f(1,())$ only when $\Gamma(f)$ is $\mathit{int} \times \mathit{unit} \to \tau$. 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 $()$ and applying β-reduction) by the derivation in Figure 4. In the figure, N_1, N_2, and Γ denote $\mathit{proj}_1(x) + 1$, $\mathit{proj}_2(x) + 1$, and $x : \mathit{int} \times \mathit{unit}$ respectively. □

EXAMPLE 3.2. Let us reconsider the example given in Section 1. The source program is expressed in our language (extended with booleans,

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Transformation Rules

n-tuples, and conditionals) by the following term M:

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

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

$$\pi_1 = \dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, z : \mathit{unit} \vdash N_1 : \mathit{int} \Rightarrow N_1}}{\Gamma \vdash \lambda z.N_1 : \mathit{unit}{\rightarrow}\mathit{int} \Rightarrow \lambda z.N_1} \quad \Gamma \vdash N_2 : \mathit{unit} \Rightarrow ()}{\Gamma \vdash (\lambda z.N_1)N_2 : \mathit{int} \Rightarrow (\lambda z.N_1)()}\ \textsc{Tr-App}}{\emptyset \vdash \lambda x.(\lambda z.N_1)N_2 : \mathit{int} \times \mathit{unit}{\rightarrow}\mathit{int} \Rightarrow \lambda x.((\lambda z.N_1)())}\ \textsc{Tr-Abs}$$

$$\pi_2 = \dfrac{\pi_1 \quad \dfrac{\emptyset \vdash 1 : \mathit{int} \Rightarrow 1 \quad \emptyset \vdash 2 : \mathit{unit} \Rightarrow ()}{\emptyset \vdash (1,2) : \mathit{int} \times \mathit{unit} \Rightarrow (1,())}\ \textsc{Tr-Pair}}{\emptyset \vdash M : \mathit{int} \Rightarrow M'}\ \textsc{Tr-App}$$

Figure 4. An Example of Transformation Derivation

follows:

$$\begin{array}{l}
\Gamma, x : \mathit{int} \times \mathit{unit} \times \mathit{int} \vdash \\
\qquad \mathbf{if}\ \mathit{proj}_3(x) > 100\ \mathbf{then}\ \mathit{proj}_1(x) \\
\qquad \mathbf{else}\ \mathit{loop}(f(\mathit{proj}_1(x), \mathit{proj}_3(x)), \mathit{proj}_2(x) + 2, \mathit{proj}_3(x) + 1) : \mathit{int} \\
\qquad \Rightarrow \mathbf{if}\ \mathit{proj}_3(x) > 100\ \mathbf{then}\ \mathit{proj}_1(x) \\
\qquad\qquad \mathbf{else}\ \mathit{loop}(f(\mathit{proj}_1(x), \mathit{proj}_3(x)), (), \mathit{proj}_3(x) + 1) \\
\Gamma, a : \mathit{int} \vdash \mathit{loop}(a, 3, 1) : \mathit{int} \Rightarrow \mathit{loop}(a, (), 1)
\end{array}$$

So, the whole term is transformed into:

$$\begin{array}{l}
(\lambda \mathit{loop}.\mathit{loop}(a, (), 1)) \\
\qquad \mathbf{fix}(\mathit{loop}, x, \mathbf{if}\ \mathit{proj}_3(x) > 100\ \mathbf{then}\ \mathit{proj}_1(x) \\
\qquad\qquad \mathbf{else}\ \mathit{loop}(f(\mathit{proj}_1(x), \mathit{proj}_3(x)), (), \mathit{proj}_3(x) + 1))
\end{array}$$

under the typing assumption $f : \mathit{int} \times \mathit{int} \rightarrow \mathit{int}, a : \mathit{int}$. By eliminating the unnecessary $()$, we obtain the optimized program given in Section 1. □

EXAMPLE 3.3. Consider the following program taken from [29]:

```
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 (extended with if-expressions) as the following term M:

$$\begin{array}{l}
(\lambda f_1.\lambda f_2.\lambda f_3.\lambda g.\lambda h.g(x, h)) \\
\qquad (\lambda z.\mathit{proj}_1(z))(\lambda z.\mathit{proj}_1(z) + \mathit{proj}_1(z))(\lambda z.\mathit{proj}_2(z)) \\
\qquad (\mathbf{if}\ p(a, b)\ \mathbf{then}\ f_1\ \mathbf{else}\ f_2\ )(\mathbf{if}\ q(a, b)\ \mathbf{then}\ f_1\ \mathbf{else}\ f_3\ ).
\end{array}$$

By assigning the following types to bound variables:

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

we can derive

$$\Gamma \vdash M : \mathit{int} \Rightarrow M'$$

for

$$\Gamma = p : (\mathit{int} \times \mathit{int}) \to \mathit{bool}, q : (\mathit{int} \times \mathit{int}) \to \mathit{bool}, a : \mathit{int}, b : \mathit{int}, x : \mathit{int} \text{ and}$$
$$M' = (\lambda f_1.\lambda f_2.\lambda f_3.\lambda g.\lambda h.g(x, ()))(\lambda z.\mathit{proj}_1(z))$$
$$(\lambda z.\mathit{proj}_1(z) + \mathit{proj}_1(z))()(\textbf{if } p(a,b) \textbf{ then } f_1 \textbf{ else } f_2\ )().$$

(Here, we assumed that a and b are integers; the result does not change for different assumptions on the types of a and b.) By eliminating unnecessary projections and function applications to values containing (), we obtain the following program:

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

This is the same as the result of Wand and Siveroni's method [29]. □

EXAMPLE 3.4. We can eliminate not only top-level parameters of functions, but also a part of structured data. Consider the following program:

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

It is expressed as

$$(\lambda f.f(1,(2,3)))\lambda u.(\mathit{proj}_1(u) + \mathit{proj}_2(\mathit{proj}_2(u)))$$

in our language. By assigning type $\mathit{int} \times (\mathit{unit} \times \mathit{int}) \to \mathit{int}$ to f, we obtain

$$(\lambda f.f(1,((),3)))\lambda u.(\mathit{proj}_1(u) + \mathit{proj}_2(\mathit{proj}_2(u)))$$

as an output. By removing the unnecessary (), we obtain the following program:

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

□

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

EXAMPLE 3.5. 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:

$$\begin{array}{l}\textbf{let } f = \lambda u.\mathit{proj}_1(u)(\mathit{proj}_2(u)) \textbf{ in}\\ \quad f(\lambda v.\mathit{proj}_1(v), (1,2)) + f(\lambda v.\mathit{proj}_1(v) + \mathit{proj}_2(v), (2,3))\end{array}$$

We can transform it into

$$\begin{array}{l}\textbf{let } f = \lambda u.\mathit{proj}_1(u)(\mathit{proj}_2(u)) \textbf{ in}\\ \quad f(\lambda v.\mathit{proj}_1(v), (1,())) + f(\lambda v.\mathit{proj}_1(v) + \mathit{proj}_2(v), (2,3))\end{array}$$

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

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

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

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

EXAMPLE 3.6. The function `f` defined by:

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

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

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

□

3.2. Correctness

Correctness of our type-based UVE is expressed by the three theorems given below. Theorem 3.7 says that there is at least one valid transformation for any well-typed term. Theorem 3.8 shows that the output of the transformation is a well-typed term. Theorem 3.10 shows that the transformed term evaluates to the same value as a source term. Note that the transformation may not preserve divergence. See Section 6 for a remedy of this.

THEOREM 3.7. *If $\Gamma \vdash M : \tau$, then $\Gamma \vdash M : \tau \Rightarrow M'$ for some M'.*

Proof. Because $\Gamma \vdash M : \tau \Rightarrow M$ always holds. □

THEOREM 3.8 (Well-typedness of transformed 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 transformation 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:

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

□

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

DEFINITION 3.9 (contexts). *A context is an expression obtained from a term by replacing one subterm with [ ]. It is defined by the following syntax:*

$$\begin{array}{rl} C[\,] ::= & [\,] \mid C[\,] + M \mid M + C[\,] \mid \lambda x.C[\,] \mid \mathbf{fix}(f, x, C[\,]) \mid (C[\,])M \\ & \mid M(C[\,]) \mid \mathbf{let}\ x = C[\,]\ \mathbf{in}\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ C[\,] \\ & \mid \mathbf{if0}\ C[\,]\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \ \mid \mathbf{if0}\ M_1\ \mathbf{then}\ C[\,]\ \mathbf{else}\ M_2 \\ & \mid \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ C[\,] \\ & \mid \mathit{proj}_1(C[\,]) \mid \mathit{proj}_2(C[\,]) \mid (C[\,], M) \mid (M, C[\,]) \end{array}$$

$C[M]$ denotes a term obtained by replacing [ ] with M.

THEOREM 3.10 (Contextual equivalence). [4] *Suppose that $\Gamma \vdash M : \tau \Rightarrow M'$ holds and that $C[\,]$ is a context. If $C[M] \Downarrow n$ holds and $\emptyset \vdash C[\,] : int$ is derivable from $\Gamma \vdash [\,] : \tau$, then $C[M'] \Downarrow n$.*

This is obtained as a corollary of the following lemma, which states that a transformed term is evaluated to a value obtained by transforming the evaluation result of the original term. The substitution $[V_1/x_1, \ldots, V_n/x_n]$ (where $V_1, \ldots, V_n$ are closed value terms) in the lemma models the environment in which terms are evaluated.

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

LEMMA 3.11. *If* $x_1 : \forall \overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall \overrightarrow{\alpha_n}.\tau_n \vdash M : \tau \Rightarrow M'$, $\emptyset \vdash V_i : \tau_i \Rightarrow V_i'$ *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* $\emptyset \vdash V : \tau \Rightarrow V'$ *for some* V'.
Proof. See Appendix A.

Using this lemma, we can prove Theorem 3.10 as follows.

Proof of Theorem 3.10. From the derivation of $\emptyset \vdash C[\,] : \mathit{int}$, we can obtain a derivation of $\emptyset \vdash C[M] : \mathit{int} \Rightarrow C[M']$ by replacing $\Gamma \vdash [\,] : \tau$ with $\Gamma \vdash M : \tau \Rightarrow M'$ and replacing each application of a typing rule (T-XX) with that of the corresponding transformation rule (TR-XX). (Here, we exploit that the identity transformation is always valid. Formally, this can be proved by induction on the derivation of $\emptyset \vdash C[\,] : \mathit{int}$.) So, it must be the case that $C[M'] \Downarrow V$ and $\emptyset \vdash n : \mathit{int} \Rightarrow V$ for some V by Lemma 3.11 (let $n = 0$ in the lemma). By the transformation rules in Figure 3, $\emptyset \vdash n : \mathit{int} \Rightarrow V$ must be derived using (TR-INT). So, V must be n. □

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

3.3. Optimality of the Transformation

We formalized UVE as a transformation that replaces some subterms with () while preserving the evaluation result. It is easy to check that our type-based UVE is complete in the sense that it can derive any sound type-preserving transformations that replace subterms with ()." The algorithm described in Section 4 finds the optimal transformation (that replaces more subterms with () than any other transformations) among transformations obtained by using our type-based method (Theorem 4.4). So, from the completeness result (Theorem 3.13 below), we know that the algorithm finds the optimal transformation among *any* (i.e., not limited to those obtained by our type-based method) type-preserving UVE transformations.

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

DEFINITION 3.12. *A binary relation $\succeq$ on terms is the least relation satisfying the following rules:*

$$
\begin{array}{l}
M \succeq () \\
n \succeq n \\
x \succeq x \\
M + N \succeq M' + N' \textit{ if } M \succeq M' \textit{ and } N \succeq N' \\
\lambda x.M \succeq \lambda x.M' \textit{ if } M \succeq M' \\
\mathbf{fix}(f, x, M) \succeq \mathbf{fix}(f, x, M') \textit{ if } M \succeq M' \\
MN \succeq M'N' \textit{ if } M \succeq M' \textit{ and } N \succeq N' \\
\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \succeq \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2' \textit{ if } M_1 \succeq M_1' \textit{ and } M_2 \succeq M_2' \\
\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ \succeq \mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3' \\
\qquad\qquad\qquad\qquad\qquad \textit{if } M_1 \succeq M_1', M_2 \succeq M_2', \textit{ and } M_3 \succeq M_3' \\
(M, N) \succeq (M', N') \textit{ if } M \succeq M' \textit{ and } N \succeq N' \\
\mathit{proj}_i(M) \succeq \mathit{proj}_i(M') \textit{ if } M \succeq M'
\end{array}
$$

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

THEOREM 3.13 (completeness). *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 the structure of M'. We show only main cases. The other cases are similar and trivial.

– Case $M' = ()$. In this case, τ must be *unit*. So, by using (Tr-Unit), we obtain $\Gamma \vdash M : \mathit{unit} \Rightarrow M'$.

– Case $M' = x$: By the assumption $M \succeq M'$, M must be x. So, by using (Tr-Var), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

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

- Case $M' = \textbf{let } x = M'_1 \textbf{ in } M'_2$: It must be the case that $M = \textbf{let } x = M_1 \textbf{ in } M_2$, $M_i \succeq M'_i$, $\Gamma \vdash M'_1 : \tau_1$, and $\Gamma, x : \forall \overrightarrow{\alpha}.\tau_1 \vdash M'_2 : \tau$ with $\{\overrightarrow{\alpha}\} = FV(\tau_1) \backslash FV(\Gamma)$. 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 $M' = M'_1 M'_2$: It must be the case that $M = M_1 M_2$, $M_i \succeq M'_i$, $\Gamma \vdash M'_1 : \tau_1 \to \tau$, and $\Gamma \vdash M'_2 : \tau_1$. By induction hypothesis, $\Gamma \vdash M_1 : \tau_1 \to \tau \Rightarrow M'_1$ and $\Gamma \vdash M_2 : \tau_1 \Rightarrow M'_2$. By using (T-App), we obtain $\Gamma \vdash M : \tau \Rightarrow M'$.

$\square$

REMARK 3.14. *The above theorem does not exclude the possibility of a better transformation that does more than merely replacing subterms with* (). *For example, the program:*

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

can be simplified to: `let fun f(x) = x in f(1) end`. *We regard this as a combination of UVE and other transformations. The program can be first transformed into* `let fun f(x, y) = x in f(1, 2) end` *by using the equality* **if** *true* **then** M_1 **else** $M_2 = M_1$ *and then UVE can be applied to it (*2 *is replaced with* () *and then the unnecessary* () *is eliminated). Section 6.3 discusses more fundamental limitations.*

4. Transformation Algorithm

In this section, we give an algorithm for finding an optimal transformation and discuss its computational cost. In the monomorphic case (i.e., without let-polymorphism), an optimal transformation is found in almost linear time. In the polymorphic case, the algorithm costs exponential time in the worst case, but we believe that it works well in practice.

4.1. Monomorphic Case

4.1.1. *Overview of the Algorithm*

An input is a triple (Γ, M, τ) such that $\Gamma \vdash M : \tau$. 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 a principal typing $\Gamma \vdash M : \tau$ in time linear in the size of M [12].) Without loss of generality, we can assume that Γ and τ do not contain free type variables: Otherwise, we can replace them with arbitrary non-*unit* types (*int*, for example) that do not contain free type variables. (This ensures that those variables are not instantiated with *unit* by the transformation algorithm below. The output of the algorithm does not depend on which types are chosen.)

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

We use an algorithm similar to the linear-time type-reconstruction algorithm for the simply-typed λ-calculus (see Proposition 3.3 of [12]). 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 rules: one is (Tr-Unit) and the other is the rule whose conclusion matches the constructor of the 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 is (Tr-Var) and $\tau_i = \tau_N$; if N is $N_1 N_2$, then it must be the case that $\tau_{N_1} = \tau_{N_2} \rightarrow \tau_N$. Otherwise, the rule may be (Tr-Unit) (which is preferred because it produces a better output) and no further constraints need to be met. Therefore, by introducing a type variable α_N representing the type of each subterm N and a type variable β_x representing the type of each variable bound in type environments, we can generate a set of constraints of the form $\alpha_N \neq \mathit{unit} \Rightarrow \mathcal{C}(N)$ where $\mathcal{C}(N)$ is a set of equality constraints on types. For example, for a subterm $N_1 N_2$, the constraint $(\alpha_{N_1 N_2} \neq \mathit{unit}) \Rightarrow \{\alpha_{N_1} = \alpha_{N_2} \rightarrow \alpha_{N_1 N_2}\}$ 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 ($\mathcal{C}(N)$ is added only when it is found that α_N cannot be instantiated with *unit*) and that the occurrence check can be omitted (because we assume that 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 $\emptyset \vdash (\lambda x.\mathit{proj}_1(x))(1,2) : \mathit{int}$. We can construct the following *template* of a derivation tree.

$$\dfrac{\dfrac{\dfrac{x:\beta_x \vdash x : \alpha_x \Rightarrow N_7}{x:\beta_x \vdash \mathit{proj}_1(x) : \alpha_{\mathit{proj}_1(x)} \Rightarrow N_6}}{\emptyset \vdash \lambda x.\mathit{proj}_1(x) : \alpha_{\lambda x.\mathit{proj}_1(x)} \Rightarrow N_2} \qquad \dfrac{\emptyset \vdash 1 : \alpha_1 \Rightarrow N_4 \qquad \emptyset \vdash 2 : \alpha_2 \Rightarrow N_5}{\emptyset \vdash (1,2) : \alpha_{(1,2)} \Rightarrow N_3}}{\emptyset \vdash (\lambda x.\mathit{proj}_1(x))(1,2) : \alpha_{(\lambda x.\mathit{proj}_1(x))(1,2)} \Rightarrow N_1}$$

Instantiate type variables so that the following conditions are met:

$$\begin{array}{l}
\alpha_{(\lambda x.\mathit{proj}_1(x))(1,2)} \neq \mathit{unit} \Rightarrow \{\alpha_{\lambda x.\mathit{proj}_1(x)} = \alpha_{(1,2)} \to \alpha_{(\lambda x.\mathit{proj}_1(x))(1,2)}\} \\
\alpha_{\lambda x.\mathit{proj}_1(x)} \neq \mathit{unit} \Rightarrow \{\alpha_{\lambda x.\mathit{proj}_1(x)} = \beta_x \to \alpha_{\mathit{proj}_1(x)}\} \\
\alpha_{\mathit{proj}_1(x)} \neq \mathit{unit} \Rightarrow \{\alpha_x = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)}\} \\
\alpha_x \neq \mathit{unit} \Rightarrow \{\alpha_x = \beta_x\} \\
\alpha_{(1,2)} \neq \mathit{unit} \Rightarrow \{\alpha_{(1,2)} = \alpha_1 \times \alpha_2\} \\
\alpha_1 \neq \mathit{unit} \Rightarrow \{\alpha_1 = \mathit{int}\} \\
\alpha_2 \neq \mathit{unit} \Rightarrow \{\alpha_2 = \mathit{int}\}
\end{array}$$

We can obtain a derivation tree by replacing subtrees whose conclusion is of the form $\Gamma \vdash N : \mathit{unit} \Rightarrow N'$ with instances of the axiom $\Gamma \vdash N : \mathit{unit} \Rightarrow ()$. $\square$

4.1.2. *Formal Description of the Algorithm*

Now we are ready to describe the algorithm more formally.

4.1.2.1. *Extracting Constraints*

As explained above, we assign a type variable α_N to each subterm N of M and a type variable β_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 γ_N to each subterm N of the form $\mathit{proj}_i(N)$.

The set $\mathcal{C}(N)$ 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 N of M, the set $\mathcal{C}(N)$ of constraints is defined by:*

$$\begin{array}{l}
\mathcal{C}(()) = \{\alpha_{()} = \mathit{unit}\}\\
\mathcal{C}(x) = \{\alpha_x = \beta_x\}\\
\mathcal{C}(n) = \{\alpha_n = \mathit{int}\}\\
\mathcal{C}(M_1 + M_2) = \{\alpha_{M_1+M_2} = \mathit{int}, \alpha_{M_1} = \mathit{int}, \alpha_{M_2} = \mathit{int}\}\\
\mathcal{C}(\lambda x.N) = \{\alpha_{\lambda x.N} = \beta_x \to \alpha_N\}\\
\mathcal{C}(\mathbf{fix}(f,x,N)) = \{\alpha_{\mathbf{fix}(f,x,N)} = \beta_x \to \alpha_N, \beta_f = \beta_x \to \alpha_N\}\\
\mathcal{C}(M_1 M_2) = \{\alpha_{M_1} = \alpha_{M_2} \to \alpha_{M_1 M_2}\}\\
\mathcal{C}(\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ ) =\\
\quad \{\alpha_{M_1} = \mathit{int}, \alpha_{M_2} = \alpha_{M_3} = \alpha_{\mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3}\ \}\\
\mathcal{C}((M_1, M_2)) = \{\alpha_{(M_1,M_2)} = \alpha_{M_1} \times \alpha_{M_2}\}\\
\mathcal{C}(\mathit{proj}_1(N)) = \{\alpha_N = \alpha_{\mathit{proj}_1(N)} \times \gamma_{\mathit{proj}_1(N)}\}\\
\mathcal{C}(\mathit{proj}_2(N)) = \{\alpha_N = \gamma_{\mathit{proj}_2(N)} \times \alpha_{\mathit{proj}_2(N)}\}
\end{array}$$

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 $\Gamma \vdash M : \tau \Rightarrow M'$ if the following conditions are satisfiable:

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

4.1.2.2. *Solving Constraints* We solve the above constraints by computing a valid equivalence relation [22] (see Definition B.8 in Appendix B) 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 $\mathcal{N}(M)$. The expression $[\tau]_\sim$ is the representative element of the equivalence class containing τ, and we assume that the representative element is chosen so that if $[\tau]_\sim$ is a type variable then the equivalence class of τ contains only type variables. We write $\sim \oplus\{\tau_{11} = \tau_{12}, \ldots, \tau_{n1} = \tau_{n2}\}$ 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 $\sim$.

$$\begin{array}{l}
(S \uplus \{\tau_1 = \tau_2\}, \sim)\\
\leadsto \begin{cases}
(S, \sim) \qquad \text{if } [\tau_1]_\sim = [\tau_2]_\sim\\
(S \cup \mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim) \cup \mathit{newC}(\tau_1, \tau_2, \sim), \sim \oplus\{\tau_1 = \tau_2\})\\
\qquad\qquad \text{if } [\tau_1]_\sim \neq [\tau_2]_\sim \text{ and } \mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim) \neq \mathbf{fail}\\
\mathbf{fail} \qquad \text{if } [\tau_1]_\sim \neq [\tau_2]_\sim \text{ and } \mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim) = \mathbf{fail}
\end{cases}
\end{array}$$

Here, $S_1 \uplus S_2$ denotes the union of disjoint sets S_1 and S_2. $subC(\tau_1, \tau_2)$ and $newC(\tau_1, \tau_2, \sim)$ are defined by:

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

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

$subC(\tau_1, \tau_2)$ adds equality constraints on subterms of τ_1 and τ_2. $newC$ adds $\mathcal{C}(N)$ if α_N is found not to be *unit* for some N.

Given a valid type judgment $\Gamma \vdash M : \tau$ as an input, apply the above rewriting rule repeatedly to the initial pair $(\{\Gamma(x) = \beta_x \mid x \in dom(\Gamma)\} \cup \{\tau = \alpha_M\}, \{(\tau,\tau) \mid \tau \in \mathcal{N}(M)\})$, until it becomes a pair of the form $(\emptyset, \sim)$. Then, an optimized term M' is obtained from M by replacing each subterm N of M with () if $[\alpha]_\sim$ is a type variable or *unit* (replace larger terms first).

EXAMPLE 4.3. 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:

$$\begin{array}{ll} & (\{\alpha_M = int\}, \{(\tau,\tau) \mid \tau \in \mathcal{N}(M)\}) \\ \leadsto & (\{\alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M\}, \sim_{id} \oplus \{\alpha_M = int\}) \\ & (\mathcal{C}(M) \text{ has been added}) \\ \leadsto & (\{\alpha_{\lambda x.proj_1(x)} = \beta_x \to \alpha_{proj_1(x)}\}, \\ & \sim_{id} \oplus \{\alpha_M = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M\}) \\ & (\mathcal{C}(\lambda x.proj_1(x)) \text{ has been added}) \\ \leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_{proj_1(x)} = \alpha_M\}, \\ & \sim_{id} \oplus \{\alpha_M = int, \alpha_{\lambda x.proj_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)}\}) \\ & (\text{subconstraints of } \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{proj_1(x)} \text{ has been added}) \end{array}$$

$$\begin{array}{ll}
\leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_x = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)}\}, \\
& \sim_{id} \oplus \{\alpha_M = \alpha_{\mathit{proj}_1(x)} = \mathit{int}, \\
& \qquad\qquad \alpha_{\lambda x.\mathit{proj}_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{\mathit{proj}_1(x)}\}(= \sim_1)) \\
& (\mathcal{C}(\mathit{proj}_1(x)) \text{ has been added}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}, \alpha_x = \beta_x\}, \sim_1 \oplus \{\alpha_x = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)}\}) \\
& (\mathcal{C}(x) \text{ has been added}) \\
\leadsto & (\{\beta_x = \alpha_{(1,2)}\}, \sim_1 \oplus \{\alpha_x = \beta_x = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)}\}) \\
& (\alpha_x = \beta_x \text{ has been moved to the equivalence relation}) \\
\leadsto & (\{\alpha_1 \times \alpha_2 = \alpha_{(1,2)}\}, \sim_1 \oplus \{\alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)}\}) \\
& (\mathcal{C}((1,2)) \text{ has been added}) \\
\leadsto & (\{\alpha_1 = \alpha_{\mathit{proj}_1(x)}, \alpha_2 = \gamma_{\mathit{proj}_1(x)}\}, \\
& \sim_1 \oplus \{\alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)} = \alpha_1 \times \alpha_2\}) \\
& (\text{subconstraints of } \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)} = \alpha_1 \times \alpha_2 \text{ has been added}) \\
\leadsto & (\{\alpha_1 = \mathit{int}, \alpha_2 = \gamma_{\mathit{proj}_1(x)}\}, \\
& \sim_1 \oplus \{\alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)} = \alpha_1 \times \alpha_2, \\
& \qquad\qquad \alpha_1 = \alpha_{\mathit{proj}_1(x)}\}) \\
& (\mathcal{C}(1) \text{ has been added}) \\
\leadsto^* & (\emptyset, \sim_{id} \oplus \{\alpha_M = \alpha_{\mathit{proj}_1(x)} = \alpha_1 = \mathit{int}, \\
& \qquad\qquad \alpha_{\lambda x.\mathit{proj}_1(x)} = \alpha_{(1,2)} \to \alpha_M = \beta_x \to \alpha_{\mathit{proj}_1(x)}, \\
& \qquad\qquad \alpha_x = \beta_x = \alpha_{(1,2)} = \alpha_{\mathit{proj}_1(x)} \times \gamma_{\mathit{proj}_1(x)} = \alpha_1 \times \alpha_2, \\
& \qquad\qquad \alpha_2 = \gamma_{\mathit{proj}_1(x)}\}) \\
& (\text{All the equations have been moved to the equivalence relation})
\end{array}$$

By instantiating α_2 with *unit*, we obtain

$$\begin{array}{l}
\alpha_M = \alpha_{\mathit{proj}_1(x)} = \alpha_1 = \mathit{int} \\
\alpha_{\lambda x.\mathit{proj}_1(x)} = \mathit{int} \times \mathit{unit} \to \mathit{int} \\
\alpha_x = \alpha_{(1,2)} = \mathit{int} \times \mathit{unit} \\
\alpha_2 = \mathit{unit}.
\end{array}$$

Because α_2 is type *unit*, we obtain $(\lambda x.\mathit{proj}_1(x))((1,()))$ by replacing 2 with (). □

4.1.3. *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 [22].

THEOREM 4.4. *Suppose* $\Gamma \vdash M : \tau$. *Then,* $(\{\Gamma(x) = \beta_x \mid x \in \mathit{dom}(\Gamma)\} \cup \{\tau = \alpha_M\}, \{(\tau,\tau) \mid \tau \in \mathcal{N}(M)\})$ *can be always rewritten by* $\leadsto$ *to* $(\emptyset, \sim)$ *for some* $\sim$. *Let* M' *be a term obtained by replacing with* () *each subterm* N *of* M *s.t.* $[\alpha_N]_\sim$ *is unit or a type variable.*

Then, $\Gamma \vdash M : \tau \Rightarrow M'$. *Moreover,* M' *is an optimal output, i.e.,* $M'' \succeq M'$ *holds for every* M'' *such that* $\Gamma \vdash M : \tau \Rightarrow M''$.

Proof. See Appendix B. □

4.1.4. *Efficiency of the algorithm*

We show that the time complexity of the algorithm is almost linear (strictly speaking, $O(n\alpha(n))$, where α is the inverse of the Ackerman function and n is the size of an input $\Gamma \vdash M : \tau$). We assume that types of the input term are expressed by directed acyclic graphs. Because the initial set-up (for allocating type variables and computing the pair $(\{\Gamma(x) = \beta_x \mid x \in dom(\Gamma)\} \cup \{\tau = \alpha_M\}, \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\}))$ and the final step (for replacing subterms of M with ()) can be performed in linear time, it is sufficient to show that the rewriting step can be performed in 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 $O(\alpha(n))$ [27].

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 $O(n\alpha(n))$.

4.2. Polymorphic Case

The basic idea of the algorithm is the same as in the monomorphic case: type inference for each subterm is performed *on-demand*, only when it is found that its type must not be *unit*. The only difference from the monomorphic case is that the unification constraints in $\mathcal{C}(N)$ above are replaced by semi-unification constraints [13] of the form $\tau_1 \leq \tau_2$ ($\tau_1 \leq \tau_2$ is defined to hold if and only if there exists a substitution θ such that $\theta\tau_1 = \tau_2$).

In order to obtain $\mathcal{C}(N)$, we can use Henglein's type system based on semi-unification constraints [10]. His type system has been developed for performing type inference in the presence of polymorphic recursion. In his polymorphic type system, a type judgment is of the form $\Gamma; \rho \vdash M : \tau$, where ρ ranges over monotypes and Γ contains only monotypes. He keeps in the additional type parameter ρ information on which type variables cannot be used polymorphically. So, the judgment $x : \forall\alpha.(\alpha \to \beta) \vdash M : \mathit{int}$ is expressed as $x : \alpha \to \beta; \beta \vdash M : \mathit{int}$. Following his type system, we can reformalize transformation rules as follows:

$$\Gamma; \rho \vdash M : \mathit{unit} \Rightarrow () \qquad \text{(TR2-UNIT)}$$

$$\frac{\tau \times \rho \leq \tau' \times \rho}{\Gamma, x : \tau; \rho \vdash x : \tau' \Rightarrow x} \qquad \text{(TR2-VAR)}$$

$$\Gamma; \rho \vdash n : \mathit{int} \Rightarrow n \qquad \text{(TR2-INT)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \mathit{int} \Rightarrow M_2'}{\Gamma; \rho \vdash M_1 + M_2 : \mathit{int} \Rightarrow M_1' + M_2'} \qquad \text{(TR2-ADD)}$$

$$\frac{\Gamma, x : \tau_1; \tau_1 \times \rho \vdash M : \tau_2 \Rightarrow M'}{\Gamma; \rho \vdash \lambda x.M : \tau_1 \to \tau_2 \Rightarrow \lambda x.M'} \qquad \text{(TR2-ABS)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma, x : \tau_1; \rho \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma; \rho \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2 \Rightarrow \mathbf{let}\ x = M_1'\ \mathbf{in}\ M_2'} \qquad \text{(TR2-LET)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \mathit{int} \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \tau \Rightarrow M_2' \qquad \Gamma; \rho \vdash M_3 : \tau \Rightarrow M_3'}{\Gamma; \rho \vdash \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ : \tau \Rightarrow \mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'} \qquad \text{(TR2-IF)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1; (\tau_1 \to \tau_2) \times \rho \vdash M : \tau_2 \Rightarrow M'}{\Gamma; \rho \vdash \mathbf{fix}(f, x, M) : \tau_1 \to \tau_2 \Rightarrow \mathbf{fix}(f, x, M')} \qquad \text{(TR2-FIX)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \to \tau_2 \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \tau_1 \Rightarrow M_2'}{\Gamma; \rho \vdash M_1 M_2 : \tau_2 \Rightarrow M_1' M_2'} \qquad \text{(TR2-APP)}$$

$$\frac{\Gamma; \rho \vdash M_1 : \tau_1 \Rightarrow M_1' \qquad \Gamma; \rho \vdash M_2 : \tau_2 \Rightarrow M_2'}{\Gamma; \rho \vdash (M_1, M_2) : \tau_1 \times \tau_2 \Rightarrow (M_1', M_2')} \qquad \text{(TR2-PAIR)}$$

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

The rule (Tr2-Var) allows the type variables in τ except for those appearing in ρ to be instantiated. The rule (Tr2-Abs) disallows x to be used polymorphically by recording its type τ_1 in the righthand side of ";". On the other hand, the rule (Tr2-Let) does not record the type of x, so the let-bound variable x can be used polymorphically.

It follows by the same argument as [10] that the above rules are equivalent to the original rules in the following sense.

THEOREM 4.5.

1. *Let* $\{\rho_1, \ldots, \rho_k\} = FV(x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n)$*, and suppose that the sets of type variables* $\{\overrightarrow{\alpha_1}\}, \ldots, \{\overrightarrow{\alpha_n}\}$*, and* $\{\overrightarrow{\rho}\}$ *are disjoint from each other. If* $x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau \Rightarrow M'$*, then* $x_1 : \tau_1, \ldots, x_n : \tau_n; \rho_1 \times \cdots \times \rho_k \vdash M : \tau \Rightarrow M'$ *holds.*

2. *Suppose* $\{\overrightarrow{\beta_i}\} = FV(\tau_i) \backslash FV(\rho)$*. If* $x_1 : \tau_1, \ldots, x_n : \tau_n; \rho \vdash M : \tau \Rightarrow M'$*, then* $x_1 : \forall\overrightarrow{\beta_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\beta_n}.\tau_n \vdash M : \tau \Rightarrow M'$ *holds.*

We can now obtain an algorithm for finding an optimal transformation in a similar manner to the monomorphic case. Let a (valid) type judgment $x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau$ be an input. Assign fresh type variables α_N, α'_N to each subterm N and a fresh type variable β_x to each variable x. The variable α'_N stands for the ρ-part of a judgment $\Gamma; \rho \vdash N : \tau \Rightarrow N'$. Assign a fresh type variable γ_N also to each subterm N of the form $\mathit{proj}_j(N')$. Then, consider a derivation tree for

$$x_1 : \tau_1, \ldots, x_n : \tau_n; \rho_1 \times \cdots \times \rho_m \vdash M : \tau \Rightarrow M'$$

such that each node is labelled by a judgment

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

and $\{\rho_1, \ldots, \rho_m\} = FV(x_1 : \forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n : \forall\overrightarrow{\alpha_n}.\tau_n)$. It is a valid derivation if and only if the following conditions are satisfiable:

$$\begin{array}{l}\{\tau_1 = \beta_{x_1}, \ldots, \tau_n = \beta_{x_n}, \tau = \alpha_M, \alpha'_M = \rho_1 \times \cdots \times \rho_m\} \\ \cup \{(\alpha_N \neq \mathit{unit}) \Rightarrow \mathcal{C}_p(N) \mid N \text{ is a subterm of } M\}.\end{array}$$

Here, $\mathcal{C}_p(N)$ is given by (Cases where N is (N_1, N_2), $\mathit{proj}_i(N_1)$, or **if0** N_1 **then** N_2 **else** N_3 are omitted.):

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

The main differences between $\mathcal{C}(N)$ and $\mathcal{C}_p(N)$ are that the equality constraint in $\mathcal{C}(x)$ has been replaced by a semi-unification constraint in $\mathcal{C}_p(x)$, and that the constraints on α'_N (which keeps the type variables that cannot be used polymorphically) have been added in $\mathcal{C}_p(N)$. We can solve them on-demand by using Henglein's graph-based semi-unification algorithm [10]. When the constraints have been solved, we can obtain an optimal output from the input M by replacing with () every subterm N such that there is no constraint of the form $\alpha_N = \tau$ or $\alpha_N \leq \tau$ for a type τ other than *unit*.

The worst-case cost in the polymorphic case is exponential in the size of an input, as for the ordinary type-reconstruction problem for ML [12]. We think, however, that the algorithm works well for realistic, well-typed programs: Henglein [10] has shown that his type-reconstruction algorithm runs in time polynomial in the size of the *explicitly-typed* program (which implies that type inference costs exponential time only if the size of inferred types is exponential in the size of the original program). We think that performing UVE after the usual type reconstruction is likely to just double the cost of performing only the usual type reconstruction.

5. Elimination of Useless Projections and Null Tuples

This section briefly explains the second step of UVE: how to simplify results of the transformation given in Sections 3 and 4 by eliminating useless constructions/destructions of pairs containing (). For simplicity, we first deal with the monomorphic case.

Elimination of useless projections and () can be again formalized as a type-based transformation. Assume that a valid type judgment

$\Gamma \vdash M : \tau$ (which is output by the transformation algorithm in Section 4) is given. Without loss of generality, we can assume that Γ and τ do not contain type *unit*: otherwise, we can rename the $()$ generated by UVE and its type *unit* so that they can be distinguished from those that were present in the source program. (Notice that without this assumption, replacing a pair $(M, ())$ with M is not always valid: Consider the case where the pair is passed to an external function $f = \lambda x.\mathit{proj}_1(x)$, whose code cannot be changed.) Under this assumption, Figure 5 shows rules for eliminating the $()$ and unnecessary constructions/destructions of pairs. (In this case, we need not perform type inference, since the type of each subexpression is known after the first step.) In the figure, $\mathit{empty}(\tau)$ is true if and only if τ is composed only of *unit* and $\times$. Key rules are (El-Unit), (El-Proj1), and (El-Pair1). The rules (El-Unit) and (El-Pair1) allow us to transform unnecessary pairing of $()$. For example, $((), ((), ()))$ is transformed into $()$, and $(1, (2, ()))$ is transformed into $(1, 2)$. The rule (El-Proj1) allows us to eliminate unnecessary projections. For example, the projection $\mathit{proj}_1(M : \mathit{int} \times ())$ is eliminated: It is valid because M is transformed into a value of type *int* by another key rule (El-Pair1). The rules in Figure 5 are exhaustive: The rule (El-Unit) covers the case where the type of the original term is empty. The other cases are covered by the other rules. Notice that the rules (El-Proj1) and (El-Pair1) cover the cases where one element of the pair has an empty type, while (El-Proj2) and (El-Pair2) cover the cases where neither element has an empty type.

We do not show the correctness of the transformation here, but we expect that we can prove a theorem corresponding to Theorem 3.10 in a similar manner, by first showing the property corresponding to Lemma 3.11.

Elimination of useless λ-abstractions and applications on $()$ (like $\lambda x : \mathit{unit}.M$ and $M())$ is not covered in the rules of Figure 5. It is not so trivial (except for the case where they appear in the form $(\lambda x.M)()$), because replacing $\lambda x : \mathit{unit}.M$ with M may increase the cost of evaluation. One solution is to classify the function types into those of functions whose bodies are values and those of functions whose bodies may not be values. We can then perform type inference and replace $\lambda x.M$ with M and $N()$ with N only when $\lambda x.M$ and N has types of the former class.

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

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

Because there is a call `f(2, 3)`, `f` must remain a function of type $\forall\alpha,\beta.(\alpha \times \beta \to \beta \times \alpha)$. So, the call `f((), 1)` cannot be replaced with `f(1)`: It is fine that `f` is used as a type *unit* $\times$ *int* $\to$ *int* $\times$ *unit*, but if the type *unit* is removed, the resulting type *int* $\to$ *int* is no longer an instance of $\forall\alpha,\beta.(\alpha \times \beta \to \beta \times \alpha)$. This problem can also be dealt with based on type information. We can replace *unit* with two new types, one that can be substituted for a polymorphic type variable and one that cannot. We only eliminate () of the latter type. Because `()` in `f((),1)` has the former type, it cannot be eliminated.

6. Discussions

6.1. Equality Type

In ML, `fn (x,y)=>x=y` is assigned a type `''a *''a ->bool`, where `''a` is an equality type variable. Since it can be instantiated with *unit*, naive application of our method wrongly transforms `(fn (x,y)=>x=y)(1,2)` into `(fn (x,y)=>()=())((), ())`. To avoid this, it is sufficient to distinguish the null tuple () and its type *unit* of ML from those used in our transformation. Let us rename () and *unit* in our transformation as ()′ and *unit*′. Then, we can replace the rule (Tr-Unit) with the following two rules:

$$\Gamma \vdash M : \mathit{unit}' \Rightarrow ()' \qquad \text{(Tr-Unit1)}$$

$$\Gamma \vdash () : \mathit{unit} \Rightarrow () \qquad \text{(Tr-Unit2)}$$

An equality type variable should not be instantiated with *unit*′.

In general, similar problems may occur whenever the source language has primitive operations that take the null tuple () as an argument. The same solution (of distinguishing the null tuple of the source language from () used in the transformation) would be applicable in such cases.

6.2. Preservation of Non-Termination and Side Effects

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

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

$$\frac{\mathit{empty}(\tau)}{\Gamma \vdash M : \tau \rightsquigarrow ()} \quad (\textsc{El-Unit})$$

$$\frac{\neg\mathit{empty}(\tau)}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow x} \quad (\textsc{El-Var})$$

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \tau_1 \to \tau_2 \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau_1 \rightsquigarrow M_2' \\ \neg\mathit{empty}(\tau_2)\end{array}}{\Gamma \vdash M_1 M_2 : \tau_2 \rightsquigarrow M_1' M_2'} \quad (\textsc{El-App})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \mathit{int} \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau \rightsquigarrow M_2' \\ \Gamma \vdash M_3 : \tau \rightsquigarrow M_3' \qquad \neg\mathit{empty}(\tau)\end{array}}{\Gamma \vdash \mathbf{if0}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3\ : \tau \rightsquigarrow \mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'} \quad (\textsc{El-If})$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2 \rightsquigarrow M' \qquad \neg\mathit{empty}(\tau_i) \qquad \mathit{empty}(\tau_{3-i})}{\Gamma \vdash \mathit{proj}_i(M) : \tau_i \rightsquigarrow M'} \quad (\textsc{El-Proj1})$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2 \rightsquigarrow M' \qquad \neg\mathit{empty}(\tau_1) \qquad \neg\mathit{empty}(\tau_2)}{\Gamma \vdash \mathit{proj}_i(M) : \tau_i \rightsquigarrow \mathit{proj}_i(M')} \quad (\textsc{El-Proj2})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \tau_1 \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau_2 \rightsquigarrow M_2' \\ \neg\mathit{empty}(\tau_i) \qquad \mathit{empty}(\tau_{3-i})\end{array}}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2 \rightsquigarrow M_i'} \quad (\textsc{El-Pair1})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \tau_1 \rightsquigarrow M_1' \qquad \Gamma \vdash M_2 : \tau_2 \rightsquigarrow M_2' \\ \neg\mathit{empty}(\tau_1) \qquad \neg\mathit{empty}(\tau_2)\end{array}}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2 \rightsquigarrow (M_1', M_2')} \quad (\textsc{El-Pair2})$$

Figure 5. Rules for Eliminating ()

is transformed into `()`.

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

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

Here, the condition "M is effect-free" can be expressed by using appropriate effect systems [26], depending on what kind of side effect we want to preserve. By regarding calls to recursive functions and `print` as having side effects (but ignoring overflow exceptions that may be caused by the expression `y+1`), we can transform the above program into:

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

6.3. More Aggressive Transformations

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

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

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

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

Even when replacement of a subterm with () is invalid, it may be valid to replace it with a *dummy* value of the same type. Consider the following example:

```
(fn f=>(h(f), f(1, M)))(fn (x,y)=>x)
```

Suppose `h` is an external function of type $(\mathit{int} \times \mathit{int} \to \mathit{int}) \to \mathit{int}$. The term `f(1, M)` cannot be replaced with `f(1, ())` because of typing constraints, but it can be replaced with a dummy value `0`. Although the replacement does not eliminate the second parameter of `f`, it may make variables in `M` useless. Damiani [2, 3] has already proposed a method for replacing subterms with dummy values (which he called *dead code elimination*). The optimality of this problem is also undecidable.

7. Related Work

7.1. Comparison with Other Methods for UVE

7.1.0.1. *Shivers's UVE [25]* Shivers [25] originated the idea of UVE and presented an algorithm for UVE using his control-flow analysis. His algorithm is the basis for Wand and Siveroni's UVE [29], but the description of the algorithm was rather informal and its correctness was not formally proved.

7.1.0.2. *Wand-Siveroni's work [29]* As mentioned in Section 1, Wand and Siveroni [29] reformalized Shivers's UVE [25] and proved its correctness.

Our method is optimal and can be performed in almost linear time for the simply-typed language, while Wand and Siveroni's method is based on 0CFA,[5] which costs cubic time in the worst case [9]. The reason for this is that the 0CFA-based method analyzes unnecessary flow information. For example, consider the following expression

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

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

In the case of a polymorphic language, our type-based method is still optimal but the 0CFA-based method is not. In fact, the 0CFA based method cannot perform the transformation shown in Example 3.5.

Replacing 0CFA with nCFA [25] does not help much. Of course, there is a case where nCFA 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),
```

[5] On the other hand, the description of Shivers's original UVE algorithm is independent of particular CFA.

```
      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)+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 [30] would not be so effective for UVE. Indeed, it does not work for the last example, either.[6]

In addition to the above problem of not being optimal, CFA-based methods seem to suffer from the cost of unnecessary computation. nCFA 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.*

7.1.0.3. *Fischbach-Hannan's work [5]* Fischbach and Hannan's method guarantees preservation of effects by integrating a simple effect analysis. It is, however, easy to extend our method to deal with effects, as mentioned above. We think that it is preferable to keep a type system for effect analysis and that for UVE separate, because complex type systems such as subtyping and polymorphic recursion [28] are necessary for precise analysis of effects, while simpler type systems suffice for UVE. Fischbach and Hannan's method does not deal with tuples and polymorphism.

[6] It is possible to refine CFA with polymorphic splitting to fully capture the behavior of Hindley-Milner type inference: Then, UVE based on the analysis would become optimal but would cost much time ([30], Sec.3.5).

7.2. Other Related Work

Prunning of simply-typed λ-terms Berardi [1] considered essentially the same problem (of replacing subterms with () so that typing is preserved) as ours, for the simply-typed λ-calculus. He showed that the semantics of a term is preserved as long as typing is preserved (as we show in Theorem 3.10 for a polymorphic language). He also sketched a transformation algorithm, but it seems less efficient than ours.

Deadcode elimination Damiani et al. [2, 3] treated a slightly different problem: to replace subterms with dummy values of the same type, rather than with (). It does not straightforwardly apply to UVE, since even if a function f does not use the second parameter, the function call $f(M_1, M_2)$ is replaced only with $f(M_1, d)$ (where d is a dummy value). In order to further replace $f(M_1, d)$ with $f(M_1)$, a method for UVE like ours must be performed. Because of this difference, the non-standard type system of Damiani et al. [2] allows subtyping (which we do not allow).

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

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

Type inference-based garbage collection The underlying idea behind our method is similar to those of garbage-collection schemes [6, 11] based on Reynolds' abstraction/parametricity theorem [24], where type inference is performed at run-time and a heap cell on which no type constraint is imposed by the context is collected as a garbage. The main differences between these techniques and ours are that our UVE performs type inference at compile time and eliminates any terms on which no type constraint is imposed by the context, not just heap values.

Other work Heintze and McAllester [8] have developed a linear-time algorithm for bounded-type programs that builds a directed graph whose transitive closure gives the information obtained by 0CFA. Interestingly, it is reported that some 0CFA-consuming analysis (for bounded-type programs) can also be performed in linear time, by using the obtained graph. Their work is not directly related with our work in this paper, because the reason our algorithm is almost linear time is that we need to perform only an equality-based analysis, not that we have specialized 0CFA for the problem of UVE. However, for the problem of deadcode elimination (replacing subterms with dummy values), it may be interesting to apply their algorithm.

There are also many other techniques for reducing procedure parameters (although the problems they deal with seem to be actually quite different from the one treated in this paper). For example, in the context of logic programming, Leuschel and Sørensen [15] presented a method for removing redundant arguments of predicates. Danvy and Schultz [4] presented a method for removing formal parameters by replacing them with references to non-local variables, in the context of lambda dropping.

8. Concluding Remarks

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

It is widely known that type systems and flow analyses are closely related [7, 19, 20, 21]. For a monomorphic language, the main difference between Wand and Siveroni's 0CFA-based method and ours is whether the analysis is subset-based or equality-based (or in the terminology of type systems, whether subtyping is allowed or not); our conclusion on this point is that an equality-based analysis is sufficient for UVE. For an ML-style polymorphic language, polymorphism provides additional power of UVE (recall Example 3.5). So, it would be safe to say that the type-based method is more suitable for an ML-style language. Another supporting argument for the type-based method is that it is guaranteed to preserve typing.

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

Acknowledgements

I would like to thank the anonymous referees for a number of useful comments and suggestions. I would also like to thank John Hannan, Atsushi Igarashi, and Makoto Tatsuta for useful comments and discussions.

References

1. Berardi, S.: 1996, 'Pruning Simply Typed Lambda-Terms'. *Journal of Logic and Computation* **6**(5), 663–681.
2. Coppo, M., F. Damiani, and P. Giannini: 1996, 'Refinement Types for Program Analysis'. In: *Proceedings of the 3rd International Static Analysis Symposium (SAS'96)*, Vol. 1145 of *Lecture Notes in Computer Science*. pp. 143–158.
3. Damiani, F.: 1998, 'Non-standard type inference for functional programs'. Ph.D. thesis, Università degli Studi di Torino.
4. Danvy, O. and U. P. Schultz: 2000, 'Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure'. *Theoretical Computer Science*. To appear. A preliminary version appeared as BRICS Report RS-99-27.
5. Fischbach, A. and J. Hannan: 1999, 'Type Systems and Algorithms for Useless-Variable Elimination'. draft. Available from `http://www.cse.psu.edu/~hannan/papers.html`.
6. Fradet, P.: 1994, 'Collecting More Garbage'. In: *Lisp and Functional Programming*. pp. 24–33.
7. Heintze, N.: 1995, 'Control-Flow Analysis and Type Systems'. In: *International Static Analysis Symposium (SAS'95)*, Vol. 983 of *Lecture Notes in Computer Science*. pp. 189–206.
8. Heintze, N. and D. McAllester: 1997a, 'Linear-time Subtransitive Control Flow Analysis'. In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 261–272.
9. Heintze, N. and D. McAllester: 1997b, 'On the Cubic Bottleneck in Subtyping and Flow Analysis'. In: *Proceedings of IEEE Symposium on Logic in Computer Science*. pp. 342–351.
10. Henglein, F.: 1993, 'Type Inference with Polymorphic Recursion'. *ACM Transactions on Programming Languages and Systems* **15**(2), 253–289.
11. Hosoya, H. and A. Yonezawa: 1998, 'Formal Description for Collecting Reachable Garbage via Dynamic Type Inference'. In: *Proceedings of the Second International Workshop on Types in Compilation (TIC98)*, Vol. 1473 of *Lecture Notes in Computer Science*. pp. 215–239.
12. Kanellakis, P. C., H. G. Mairson, and J. C. Mitchell: 1991, 'Unification and ML Type Reconstruction'. In: J.-L. Lassez and G. D. Plotkin (eds.): *Computational Logic: Essays in Honor of Alan Robinson*. The MIT Press, pp. 444–478.

13. Kapur, D., D. Musser, P. Narendran, and J. Stillman: 1991, 'Semi-unification'. *Theoretical Computer Science* **81**(2), 169–188.
14. Lee, O. and K. Yi: 1998, 'Proofs about a Folklore Let-Polymorphic Type Inference Algorithm'. *ACM Transactions on Programming Languages and Systems* **20**(4), 707–723.
15. Leuschel, M. and M. H. Sørensen: 1996, 'Redundant Argument Filtering of Logic Programs'. In: *Logic Program Synthesis and Transformation*, Vol. 1207 of *Lecture Notes in Computer Science*. pp. 83–103.
16. Milner, R., M. Tofte, R. Harper, and D. MacQueen: 1997, *The Definition of Standard ML (Revised)*. The MIT Press.
17. Mitchell, J. C.: 1996, *Foundations for Programming Languages*. The MIT Press.
18. Mogensen, T.: 1998, 'Types for 0, 1 or Many Uses'. In: *Implementation of Functional Languages*, Vol. 1467 of *Lecture Notes in Computer Science*. pp. 112–122.
19. Palsberg, J.: 1998, 'Equality-based flow analysis versus recursive types'. *ACM Transactions on Programming Languages and Systems* **20**(6), 1251–1264.
20. Palsberg, J. and P. O'Keefe: 1995, 'A Type System Equivalent to Flow Analysis'. *ACM Transactions on Programming Languages and Systems* **17**(4), 576–599.
21. Palsberg, J. and C. Pavlopoulou: 1998, 'From Polyvariant Flow Information to Intersection and Union Types'. In: *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. pp. 197–208.
22. Paterson, M. S. and M. N. Wegman: 1978, 'Linear Unification'. *Journal of Computer and System Sciences* **16**, 158–167.
23. Reps, T. and T. Turnidge: 1996, 'Program Specialization via Program Slicing'. In: *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, Vol. 1110 of *Lecture Notes in Computer Science*. pp. 409–429.
24. Reynolds, J. C.: 1983, 'Types, Abstraction and Parametric Polymorphism'. In: *Proceedings of Information Processing 83*. pp. 513–523.
25. Shivers, O.: 1991, 'Control-Flow Analysis of Higher-Order Languages'. Ph.D. thesis, Carnegie-Mellon University.
26. Talpin, J.-P. and P. Jouvelot: 1992, 'Polymorphic type, region and effect inference'. *Journal of Functional Programming* **2**(3), 245–271.
27. Tarjan, R. E.: 1975, 'Efficiency of a good but not linear set union algorithm'. *Journal of the ACM* **22**, 215–225.
28. Tofte, M. and J.-P. Talpin: 1994, 'Implementation of the call-by-value lambda-calculus using a stack of regions'. In: *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. pp. 188–201.
29. Wand, M. and I. Siveroni: 1999, 'Constraint Systems for Useless Variable Elimination'. In: *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. pp. 291–302.
30. Wright, A. K. and S. Jagannathan: 1998, 'Polymorphic Splitting: An Effective Polyvariant Flow Analysis'. *ACM Transactions on Programming Languages and Systems* **20**(1), 166–207.

Appendix

A. Proof of Lemma 3.11

We first need some auxiliary lemmas.

LEMMA A.1. *. If* $\Gamma, x : \forall\overrightarrow{\alpha}.\tau_1 \vdash M : \tau_2 \Rightarrow M'$ *and* $\{\overrightarrow{\alpha}\} \subseteq \{\overrightarrow{\beta}\}$ *hold, then* $\Gamma, x : \forall\overrightarrow{\beta}.\tau_1 \vdash M : \tau_2 \Rightarrow M'$ *also holds.*

Proof. This follows by straightforward induction on derivation of $\Gamma, x : \forall\overrightarrow{\alpha}.\tau_1 \vdash M : \tau_2 \Rightarrow M'$. □

LEMMA A.2. *If* $\Gamma \vdash M : \tau \Rightarrow M'$, *then* $[\tau'/\alpha]\Gamma \vdash M : [\tau'/\alpha]\tau \Rightarrow M'$

Proof. The proof is by induction on structure of M'. We show only the cases where M' is a variable or a let-expression: The other cases are trivial.

- Case where $M' = x$: It must be the case that $M = x$, $\Gamma = (\Gamma', x : \forall\overrightarrow{\beta}.\tau_1)$, and $[\overrightarrow{\rho}/\overrightarrow{\beta}]\tau_1 = \tau$. We can assume without loss of generality that $\{\overrightarrow{\beta}\} \cap (\{\alpha\} \cup FV(\tau')) = \emptyset$. So, we have $[\tau'/\alpha]\Gamma = [\tau'/\alpha]\Gamma', x : \forall\overrightarrow{\beta}.[\tau'/\alpha]\tau_1$. By using (TR-VAR), we obtain
$$[\tau'/\alpha]\Gamma \vdash x : [([\tau'/\alpha]\overrightarrow{\rho})/\overrightarrow{\beta}]([\tau'/\alpha]\tau_1) \Rightarrow x.$$
We have the required result because $[([\tau'/\alpha]\overrightarrow{\rho})/\overrightarrow{\beta}][\tau'/\alpha]\tau_1 = [\tau'/\alpha]([\overrightarrow{\rho}/\overrightarrow{\beta}]\tau_1) = [\tau'/\alpha]\tau$ holds.

- Case where $M' = \mathbf{let}\ x = M_1' \ \mathbf{in}\ M_2'$: We have
$$\begin{array}{l} M = \mathbf{let}\ x = M_1 \ \mathbf{in}\ M_2 \\ \Gamma \vdash M_1 : \tau_1 \Rightarrow M_1' \\ \Gamma, x : \forall\overrightarrow{\beta}.\tau_1 \vdash M_2 : \tau \Rightarrow M_2' \end{array}$$
with $M = \mathbf{let}\ x = M_1 \ \mathbf{in}\ M_2$ and $\{\overrightarrow{\beta}\} = FV(\tau_1)\backslash FV(\Gamma)$. Without loss of generality, we can assume that $\{\overrightarrow{\beta}\} \cap (\{\alpha\} \cup FV(\tau')) = \emptyset$. (Otherwise, we can rename β in $\Gamma \vdash M_1 : \tau_1 \Rightarrow M_1'$ by applying induction hypothesis to M_1'.) By induction hypothesis, we have
$$\begin{array}{l} [\tau'/\alpha]\Gamma \vdash M_1 : [\tau'/\alpha]\tau_1 \Rightarrow M_1' \\ [\tau'/\alpha]\Gamma, x : [\tau'/\alpha]\forall\overrightarrow{\beta}.\tau_1 \vdash M_2 : [\tau'/\alpha]\tau \Rightarrow M_2'. \end{array}$$
By the assumption $\{\overrightarrow{\beta}\} \cap (\{\alpha\} \cup FV(\tau')) = \emptyset$, we have $\{\overrightarrow{\beta}\} \subseteq FV([\tau'/\alpha]\tau_1)\backslash FV([\tau'/\alpha]\Gamma)$ and $[\tau'/\alpha]\forall\overrightarrow{\beta}.\tau_1 = \forall\overrightarrow{\beta}.[\tau'/\alpha]\tau_1$. So, by using (TR-LET) and Lemma A.1, we obtain $[\tau'/\alpha]\Gamma \vdash M : [\tau'/\alpha]\tau \Rightarrow M'$ as required.

□

LEMMA A.3 (Substitution). *If* $\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1 \vdash M : \tau \Rightarrow M'$ *and* $\Gamma \vdash N : \tau_1 \Rightarrow N'$ *and* $\{\overrightarrow{\alpha}\} \cap (FV(\Gamma) \cup FV(\tau)) = \emptyset$*, then* $\Gamma \vdash [N/x]M : \tau \Rightarrow [N'/x]M'$*.*

Proof. The proof is by induction on the structure of M'.

– Case for $M' = ()$: In this case, $\tau = \mathit{unit}$. Therefore, by (TR-UNIT), we have $\Gamma \vdash [N/x]M : \tau \Rightarrow ()(= [N'/x]M')$.

– Case for $M' = x$: By the assumption $\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1 \vdash M : \tau \Rightarrow M'$, it must be the case that $M = x$ and $[\overrightarrow{\rho}/\overrightarrow{\alpha}]\tau_1 = \tau$ for some $\overrightarrow{\rho}$. So, $\Gamma \vdash [N/x]M : \tau \Rightarrow [N'/x]M'$ follows immediately from the assumption $\Gamma \vdash N : \tau_1 \Rightarrow N'$ and Lemma A.2.

– Case for $M' = y(\neq x)$: It must be the case that $[N/x]M = [N'/x]M' = y$. So, we obtain $\Gamma \vdash [N/x]M : \tau \Rightarrow [N/x]M'$ from the assumption.

– Case for $M' = n$: Trivial, since $M = M' = [N/x]M = [N'/x]M' = n$.

– Case for $M' = M_1' + M_2'$: In this case, we have $M = M_1 + M_2$, $\tau = \mathit{int}$, $(\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1 \vdash M_1 : \mathit{int} \Rightarrow M_1')$, and $(\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1 \vdash M_2 : \mathit{int} \Rightarrow M_2')$. By induction hypothesis, $\Gamma \vdash [N/x]M_1 : \mathit{int} \Rightarrow [N'/x]M_1'$ and $\Gamma \vdash [N/x]M_2 : \mathit{int} \Rightarrow [N'/x]M_2'$. Therefore, we obtain $\Gamma \vdash [N/x]M : \tau \Rightarrow [N'/x]M'$ by using (TR-ADD).

– Case for $M' = \lambda y.M_1'$: In this case, we have $M = \lambda y.M_1$ and $\tau = \tau_2 \to \tau_3$. Without loss of generality, we can assume that $y \neq x$ (by the assumption on implicit α-conversion). So, we have $\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1, y:\tau_2 \vdash M_1 : \tau_3 \Rightarrow M_1'$. By the assumption $\{\overrightarrow{\alpha}\} \cap (FV(\Gamma) \cup FV(\tau)) = \emptyset$, we have $\{\overrightarrow{\alpha}\} \cap (FV(\Gamma, y:\tau_2) \cup FV(\tau_3)) = \emptyset$. So, by using induction hypothesis, we obtain $\Gamma, y:\tau_2 \vdash [N/x]M_1 : \tau_3 \Rightarrow [N'/x]M_1'$, from which $\Gamma \vdash \lambda y.[N/x]M_1 : \tau \Rightarrow \lambda y.[N'/x]M_1'(= [N'/x](\lambda y.M_1'))$ follows.

– Case for $M' = \mathbf{fix}(f, x, M)$: Similar to the case for $M' = \lambda x.M$.

– Case where M' is of the form $M_1'M_2'$, $\mathit{proj}_i(M_1')$, (M_1', M_2'), or $\mathbf{if0}\ M_1'\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'$: Similar to the case for (TR-ADD).

– Case for $M' = \mathbf{let}\ y = M_1'\ \mathbf{in}\ M_2'$: It must be the case that $M = \mathbf{let}\ y = M_1\ \mathbf{in}\ M_2$. We can assume without loss of generality that $y \neq x$. We also have $\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1 \vdash M_1 : \tau_2 \Rightarrow M_1'$ and $\Gamma, x:\forall\overrightarrow{\alpha}.\tau_1, y:\forall\overrightarrow{\beta}.\tau_2 \vdash M_2 : \tau \Rightarrow M_2'$, where $\{\overrightarrow{\beta}\} =$

$FV(\tau_2)\backslash(FV(\Gamma) \cup FV(\forall\overrightarrow{\alpha}.\tau_1))$. Without loss of generality, we can assume that $\{\overrightarrow{\alpha}\} \cap FV(\tau_2) = \emptyset$. (If it does not hold, apply α-conversion to $\forall\overrightarrow{\alpha}.\tau_1$.) So, by using induction hypothesis, we obtain $\Gamma \vdash [N/x]M_1 : \tau_2 \Rightarrow [N'/x]M_1'$, and $\Gamma, y:\forall\overrightarrow{\beta}.\tau_2 \vdash [N/x]M_2 : \tau \Rightarrow [N'/x]M_2'$. Let $\{\overrightarrow{\gamma}\} = FV(\tau_2)\backslash FV(\Gamma)$. Then because $\{\overrightarrow{\beta}\} = FV(\tau_2)\backslash(FV(\Gamma) \cup FV(\forall\overrightarrow{\alpha}.\tau_1)) \subseteq FV(\tau_2)\backslash FV(\Gamma) = \{\overrightarrow{\gamma}\}$ holds, we obtain $\Gamma, y:\forall\overrightarrow{\gamma}.\tau_2 \vdash [N/x]M_2 : \tau \Rightarrow [N'/x]M_2'$ by using Lemma A.1. So, we obtain the required result by using (TR-LET).

□

Now we can prove Lemma 3.11:

If $x_1:\forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n:\forall\overrightarrow{\alpha_n}.\tau_n \vdash M : \tau \Rightarrow M'$, $\emptyset \vdash V_i : \tau_i \Rightarrow V_i'$ 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 $\emptyset \vdash V : \tau \Rightarrow V'$ for some V'.

Proof of Lemma 3.11. 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. By applying Lemma A.2 to $\emptyset \vdash V_i : \tau_i \Rightarrow V_i'$, we get $\emptyset \vdash V_i : [\overrightarrow{\rho}/\overrightarrow{\alpha_i}]\tau_i \Rightarrow V_i'$. 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 Γ for $x_1:\forall\overrightarrow{\alpha_1}.\tau_1, \ldots, x_n:\forall\overrightarrow{\alpha_n}.\tau_n$ and write θ and θ' for substitutions $[V_1/x_1, \ldots, V_n/x_n]$ and $[V_1'/x_1, \ldots, V_n'/x_n]$ respectively. We can assume without loss of generality that $\overrightarrow{\alpha_1}, \ldots, \overrightarrow{\alpha_n}$ do not appear free in the derivation of $\Gamma \vdash M : \tau \Rightarrow M'$.

- 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): The derivation must be of the form:

$$\dfrac{\dfrac{\cdots}{\theta M_1 \Downarrow n_1} \quad \dfrac{\cdots}{\theta M_2 \Downarrow n_2}}{\theta M \Downarrow n_1{+}n_2}$$

with $M = M_1 + M_2$ and $V = n_1{+}n_2$. By the assumptions, we also have $M' = M'_1 + M'_2$, $\tau = \mathit{int}$, and $\Gamma \vdash M_i : \mathit{int} \Rightarrow M'_i$. By applying induction hypothesis to the derivations of $\theta M_i \Downarrow n_i$, we obtain $\theta' M'_i \Downarrow n_i$ for $i = 1, 2$. So, by using (E-Add), we obtain $\theta' M' \Downarrow V$. The result follows, since $\emptyset \vdash V : \mathit{int} \Rightarrow V$.

– Case for (E-Abs): In this case, $M = \lambda y.M_1$, $M' = \lambda y.M'_1$, $\tau = \tau' {\rightarrow} \tau''$, and $\Gamma, y : \tau' \vdash M_1 : \tau'' \Rightarrow M'_1$. By repeated applications of Lemma A.3 and the assumption that $\{\overrightarrow{\alpha_1}\}, \ldots, \{\overrightarrow{\alpha_n}\}$, and $FV(\Gamma) \cup FV(\tau)$ are disjoint from each other, we have $y : \tau' \vdash \theta M_1 : \tau'' \Rightarrow \theta' M'_1$, from which we obtain

$$\vdash \lambda y.\theta M_1 : \tau \Rightarrow \lambda y.\theta' M'_1.$$

Therefore, the required result holds for $V' = \lambda y.\theta' M'_1$.

– Case for (E-Fix): In this case, $M = \mathbf{fix}(f, y, M_1)$, $M' = \mathbf{fix}(f, y, M'_1)$, $\tau = \tau' {\rightarrow} \tau''$, $V = \lambda y.[\theta M/f]\theta M_1$, 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 Lemma A.3 and the assumption $\Gamma \vdash M : \tau \Rightarrow M'$, we have $\Gamma \vdash \lambda y.[M/f]M_1 : \tau \Rightarrow \lambda y.[M'/f]M'_1$. Let $V' = \theta' \lambda y.[M'/f]M'_1 = \lambda y.[\theta' M'/f]\theta' M'_1$. By applying Lemma A.3 repeatedly, we obtain $\emptyset \vdash V : \tau \Rightarrow V'$. We have also $\theta' M' \Downarrow V'$ as required.

– Case for (E-App): The derivation must be of the form:

$$\dfrac{\dfrac{\cdots}{\theta M_1 \Downarrow \lambda y.M_3} \quad \dfrac{\cdots}{\theta M_2 \Downarrow W} \quad \dfrac{\cdots}{[W/y]M_3 \Downarrow V}}{\theta M \Downarrow V}$$

with $M = M_1 M_2$. Moreover, by the assumption $\Gamma \vdash M_1 M_2 : \tau \Rightarrow M'$, it must be the case that $M' = M'_1 M'_2$, $\Gamma \vdash M_1 : \tau' {\rightarrow} \tau \Rightarrow M'_1$ and $\Gamma \vdash M_2 : \tau' \Rightarrow M'_2$. By applying induction hypothesis to the derivation of $\theta M_1 \Downarrow \lambda y.M_3$, we obtain $\theta' M'_1 \Downarrow \lambda y.M'_3$ and $\emptyset \vdash \lambda y.M_3 : \tau' {\rightarrow} \tau \Rightarrow \lambda y.M'_3$ for some M'_3. From the latter, we get $y : \tau' \vdash M_3 : \tau \Rightarrow M'_3$. By applying induction hypothesis to the derivation of $\theta M_2 \Downarrow W$, we also obtain $\theta' M'_2 \Downarrow W'$ and $\emptyset \vdash W : \tau' \Rightarrow W'$ for some W'. So, by applying induction hypothesis again to the derivation of $[W/y]M_3 \Downarrow V$, we have $[W'/y]M'_3 \Downarrow V'$ (from which $\theta' M' \Downarrow V'$ follows) and $\emptyset \vdash V : \tau \Rightarrow V'$ for some V'.

- Case where the last rule is (E-LET): The derivation must be of the form

$$\frac{\dfrac{\cdots}{\theta M_1 \Downarrow W} \quad \dfrac{\cdots}{[W/x]\theta M_2 \Downarrow V}}{\theta M \Downarrow V}$$

  with $M = \textbf{let } x = M_1 \textbf{ in } M_2$. we also have $M' = \textbf{let } x = M'_1 \textbf{ in } M'_2$, $\Gamma \vdash M_1 : \rho \Rightarrow M'_1$, and $\Gamma, x : \forall\overrightarrow{\alpha}.\rho \vdash M_2 : \tau \Rightarrow M'_2$ with $\{\overrightarrow{\alpha}\} = FV(\rho)\backslash FV(\Gamma)$. By applying induction hypothesis to the derivation of $\theta M_1 \Downarrow W$, we obtain $\theta' M'_1 \Downarrow W'$ and $\emptyset \vdash W : \rho \Rightarrow W'$ for some W'. By applying induction hypothesis again to the derivation of $[W/x]\theta M_2 \Downarrow V$, we obtain $[W'/x]\theta' M'_2 \Downarrow V'$ (which implies $\theta' M' \Downarrow V'$) and $\emptyset \vdash V : \tau \Rightarrow V'$ for some V', as required.

- Case for (E-IFT): The derivation must be of the form

$$\frac{\dfrac{\cdots}{\theta M_1 \Downarrow 0} \quad \dfrac{\cdots}{\theta M_2 \Downarrow V}}{\theta M \Downarrow V}$$

  with $M = \textbf{if0 } M_1 \textbf{ then } M_2 \textbf{ else } M_3$. We also have $M' = \textbf{if0 } M'_1 \textbf{ then } M'_2 \textbf{ else } M'_3$, $\Gamma \vdash M_1 : \mathit{int} \Rightarrow M'_1$, and $\Gamma \vdash M_i : \tau \Rightarrow M'_i$ for $i = 2, 3$. By applying induction hypothesis to the derivation of $\theta M_1 \Downarrow 0$ and the fact $\Gamma \vdash M_1 : \mathit{int} \Rightarrow M'_1$, we obtain $\theta' M'_1 \Downarrow V'_1$ and $\emptyset \vdash 0 : \mathit{int} \Rightarrow V'_1$ for some V'_1. By the transformation rules, V'_1 must be 0. By applying induction hypothesis again to $\theta M_2 \Downarrow V$ and $\Gamma \vdash M_2 : \tau \Rightarrow M'_2$, we also have $\theta' M'_2 \Downarrow V'$ and $\emptyset \vdash V : \tau \Rightarrow V'$ for some V'. By using (E-IFT), we get $\theta' M' \Downarrow V'$ as required.

- Case for (E-IFF): Similar to the case for (E-IFT).

- Case for (E-PROJ): We show only the case where i in the rule is 1: The case where $i = 2$ is similar. In this case, the derivation must be of the form

$$\frac{\dfrac{\cdots}{\theta M_1 \Downarrow (V, W_2)}}{\theta M \Downarrow V}$$

  with $M = \mathit{proj}_1(M_1)$. We also have $M' = \mathit{proj}_1(M'_1)$ and $\Gamma \vdash M_1 : \tau \times \tau_2 \Rightarrow M'_1$. By induction hypothesis, we have $\theta' M'_1 \Downarrow W'$ and $\emptyset \vdash (V, W_2) : \tau \times \tau_2 \Rightarrow W'$ for some W'. By transformation rules, W' must be of the form (W'_1, W'_2) and $\emptyset \vdash V : \tau \Rightarrow W'_1$. The result follows for $V' = W'_1$, since $\theta' M' = \mathit{proj}_1(\theta' M'_1) \Downarrow V'$.

- Case for (E-PAIR): The derivation must be of the form

$$\frac{\dfrac{\cdots}{\theta M_1 \Downarrow W_1} \quad \dfrac{\cdots}{\theta M_2 \Downarrow W_2}}{\theta M \Downarrow V}$$

with $M = (M_1, M_2)$ and $V = (W_1, W_2)$. We also have $M' = (M'_1, M'_2)$, $\tau = \tau_1 \times \tau_2$, $\Gamma \vdash M_1 : \tau_1 \Rightarrow M'_1$, and $\Gamma \vdash M_2 : \tau_2 \Rightarrow M'_2$. By induction hypothesis, we have $\theta' M'_1 \Downarrow W'_1$, $\theta' M'_2 \Downarrow W'_2$, $\emptyset \vdash W_1 : \tau_1 \Rightarrow W'_1$ and $\emptyset \vdash W_2 : \tau_2 \Rightarrow W'_2$ for some W'_1 and W'_2. Therefore, $\theta' M' \Downarrow V'$ and $\emptyset \vdash V : \tau \Rightarrow V'$ hold for $V' = (W'_1, W'_2)$.

□

B. Proof of Theorem 4.4

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

DEFINITION B.1. $[\![(S, \sim)]\!]$ *is defined as*

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

NOTATION B.2. *We write* Θ_M *for the set of substitutions that maps each type variable in*

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

to a type term containing no type variables.

DEFINITION B.3. *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 *and* c'_j *be constraints 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* $j \in \{1, \ldots, n\}$.

The following lemma states that $[\![(S, \sim)]\!]$ is an invariant condition preserved by the rewriting $\rightsquigarrow$.

LEMMA B.4. *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 additional lemmas to prove this.

LEMMA B.5. *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 $\leadsto$, it must be the case that $S = S_1 \uplus \{\tau_1 = \tau_2\}$, $S' = S_1 \cup \mathit{subC}([\tau_1]_\sim, [\tau_2]_\sim) \cup \mathit{newC}(\tau_1, \tau_2, \sim)$, and $\sim' = \sim \oplus \{\tau_1 = \tau_2\}$. 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 B.6. *If* $(S, \sim) \leadsto (S', \sim')$, *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 $\leadsto$ (notice that $\sim \subseteq \sim'$ and if $(S_1 \cup \{\tau_1 = \tau_2\}, \sim_1) \leadsto (S_2, \sim_2)$ and $\tau_1 = \tau_2 \notin S_2$, then $\tau_1 \sim_2 \tau_2$). □

Proof of Lemma B.4.

$\Leftarrow$) 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 B.6. 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 B.5, 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 second set is trivial, since $\sim' \subseteq \sim \oplus S$. 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 $\mathcal{C}(N)$ for some N such that $\alpha_N \sim \tau_1$, $[\tau_1]_\sim$ is a type variable, and $[\tau_2]_\sim$ is neither a type variable nor *unit*. So, $\theta \models c$ follows from the fact $\theta \models (\alpha_N \neq \mathit{unit}) \Rightarrow \mathcal{C}(N)$.

Suppose that c comes from the third set of $[\![(S', \sim')]\!]$. In this case, c is $(\alpha_N \neq \mathit{unit}) \Rightarrow \mathcal{C}(N)$ and $[\alpha_N]_{\sim'}$ is a type variable. By the definition of $\leadsto$, we have $\sim' \supseteq \sim$, which implies that $[\alpha_N]_\sim$ is also a type variable. So, it must be the case that $c \in [\![(S, \sim)]\!]$. By the assumption $\theta \models [\![(S, \sim)]\!]$, $\theta \models c$ follows.

□

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

LEMMA B.7. *If* $[\![(S, \sim)]\!]$ *is satisfiable, then* $(S, \sim) \not\leadsto^* \textbf{fail}$.

Proof. Suppose $(S, \sim) \rightsquigarrow^* (S', \sim') \rightsquigarrow$ **fail**. By the definition of $\rightsquigarrow$, it must be the case that $S' = S_1 \uplus \{\tau_1 = \tau_2\}$ and $subC([\tau_1]_{\sim'}, [\tau_2]_{\sim'}) =$ **fail**. By the definition of $subC$, $[\tau_1]_{\sim'}$ and $[\tau_2]_{\sim'}$ are not unifiable. So, $[\![(S', \sim')]\!]$ is not satisfiable. By Lemma B.4 and the assumption that $[\![(S, \sim)]\!]$ is satisfiable, however, $[\![(S', \sim')]\!]$ must be satisfiable, hence a contradiction. □

Next, we want to check that the final $\sim$ is a valid equivalence relation as defined below.

DEFINITION B.8. *An equivalence relation $\sim$ on $\mathcal{N}(M)$ is* valid *[22] 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 $\{([\alpha]_\sim, [\tau]_\sim) \mid \tau \in \mathcal{N}(M), \tau \neq \alpha$, and the type variable α appears in $\tau\}$ is a strict partial order.*

The third condition prevents cycles. For example, it excludes out the following unsatisfiable equivalence relation: $\{\alpha \sim \beta \times \beta, \beta \sim \alpha \times \alpha\}$. Intuitively, a valid equivalence relation is a "solved form" of equality constraints. From a valid equivalence relation $\sim$, we can obtain a most general unifier $mgu_\sim$ of $\{\tau_1 = \tau_2 \mid \tau_1 \sim \tau_2\}$ by:

$$mgu_\sim(\alpha) = \begin{cases} \alpha & \text{if } [\alpha]_\sim = \alpha \\ [mgu_\sim(\alpha_1)/\alpha_1, \dots, mgu_\sim(\alpha_n)/\alpha_n][\alpha]_\sim & \textit{otherwise} \end{cases}$$

Here, $\alpha_1, \dots, \alpha_n$ are type variables appearing in $[\alpha]_\sim$. (Note that $mgu_\sim(\alpha)$ is well defined because of the third condition of the valid equivalence relation.)

To prove that the rewriting rules always generate a valid equivalence relation, it suffices to check that the following well-formedness condition is preserved by rewriting.

DEFINITION B.9. *We say that $(S, \sim)$ is* well-formed *if $\tau_i \,(\sim \oplus S)\, \tau_i'$ holds for $i = 1, 2$ whenever $\tau_1 {\rightarrow} \tau_2 \sim \tau_1' {\rightarrow} \tau_2'$ or $\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'$ holds.*

LEMMA B.10. *Suppose that $[\![(S, \sim)]\!]$ is satisfiable. If $(S, \sim) \rightsquigarrow (S', \sim')$ and $(S, \sim)$ is well-formed, then $(S', \sim')$ is also well-formed.*

Proof. The only non-trivial is the case where $(\tau_1 \mathbf{op} \tau_2) \sim' (\tau_1' \mathbf{op} \tau_2')$ but $(\tau_1 \mathbf{op} \tau_2) \not\sim (\tau_1' \mathbf{op} \tau_2')$ for $\mathbf{op} = \times$ or $\rightarrow$. Suppose $\mathbf{op} = \times$. (The case where $\mathbf{op} = \rightarrow$ is similar.) In this case, there exist S_1, τ_3, and τ_3' such that:

$$\begin{array}{l} S = S_1 \uplus \{\tau_3 = \tau_3'\} \\ S' = S_1 \cup subC([\tau_3]_\sim, [\tau_3']_\sim) \cup newC(\tau_3, \tau_3', \sim) \\ \sim' = \sim \oplus \{\tau_3 = \tau_3'\} \\ \tau_3 \sim \tau_1 \times \tau_2 \\ \tau_3' \sim \tau_1' \times \tau_2' \end{array}$$

By the last two conditions, the well-formedness of $(S, \sim)$ and the satisfiability of $[\![(S, \sim)]\!]$, it must be the case that

$$\begin{array}{ll} [\tau_3]_\sim = \rho_1 \times \rho_2 & [\tau'_3]_\sim = \rho'_1 \times \rho'_2 \\ \rho_i(\sim \oplus S)\tau_i \text{ for } i = 1,2 & \rho'_i(\sim \oplus S)\tau'_i \text{ for } i = 1,2 \end{array}$$

for some ρ_1, ρ_2, ρ'_1, and ρ'_2. Because $(\sim \oplus S_1) \subseteq (\sim' \oplus S)$ holds, the last two conditions imply $\rho_i(\sim' \oplus S')\tau_i$ and $\rho'_i(\sim' \oplus S')\tau'_i$. By the definition of $subC$, we have:

$$\{\rho_1 = \rho'_1, \rho_2 = \rho'_2\} \subseteq subC([\tau_3]_\sim, [\tau'_3]_\sim) \subseteq S'.$$

Therefore, we have $\tau_i(\sim' \oplus S')\rho_i(\sim' \oplus S')\rho'_i(\sim' \oplus S')\tau'_i$ as required. □

LEMMA B.11. *Let* $\sim = \{(\tau, \tau) \mid \tau \in \mathcal{N}(M)\}$ *and suppose* $[\![(S, \sim)]\!]$ *is satisfiable. If* $(S, \sim) \rightsquigarrow^* (\emptyset, \sim')$*, then* $\sim'$ *is a valid equivalence relation.*

Proof. Because $\sim$ is the identity relation, $(S, \sim)$ is well-formed. By Lemmas B.4 and B.10, $(\emptyset, \sim')$ is also well-formed, from which the first condition of a valid equivalence relation follows. Because $[\![(S, \sim)]\!]$ is satisfiable, by Lemma B.4, $\{\tau_1 = \tau_2 \mid \tau_1 \sim' \tau_2\}$ is also satisfiable. Therefore, $\sim'$ satisfies the second and third conditions of the valid equivalence relation. □

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

Proof of Theorem 4.4. By Lemma B.7, the rewriting does not fail. In Section 4.1.4, we have shown (without using this theorem) that the rewriting always terminates within $O(n)$ steps. By Lemma B.11, $mgu_\sim$ is well defined. Let θ_1 be a substitution $[unit/\alpha_1, \ldots, unit/\alpha_n]$ where $\{\alpha_1, \ldots, \alpha_n\}$ is the set $\{\alpha \in \mathcal{N}(M) \mid [\alpha]_\sim$ is a type variable$\}$. Then, $\theta = \theta_1 \circ mgu_\sim$ satisfies $[\![(\emptyset, \sim)]\!] = \{\tau_1 = \tau_2 \mid \tau_1 \sim \tau_2\} \cup \{\alpha_N \neq unit \Rightarrow \mathcal{C}(N) \mid [\alpha_N]_\sim$ is a type variable$\}$. (Note that constraints in the second set are vacuously true because $\theta\alpha_N = unit$.) By Lemma B.4, it also satisfies

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

So, we can construct a derivation tree for $\Gamma \vdash M : \tau \Rightarrow M'$, whose each node is of the form $x_1 : \theta(\beta_{x_1}), \ldots, x_n : \theta(\beta_{x_n}) \vdash N : \theta(\alpha_N) \Rightarrow N'$ (Recall the remark after Definition 4.2).

To show the last property, suppose there is a derivation tree for $\Gamma \vdash M : \tau \Rightarrow M''$, whose each node is of the form $x_1 : \eta'_{x_1}, \ldots, x_n : \eta'_{x_n} \vdash N : \tau'_N \Rightarrow N''$. Let τ'_N be *unit* if there is no node of the form

$\Delta \vdash N : \tau'_N \Rightarrow N''$, and η'_x be also *unit* if there is no node labelled by $\Delta \vdash N : \tau'_N \Rightarrow N''$ such that $x \in dom(\Delta)$. Define also ζ_N by:

$$\zeta_{proj_1(N)} = \begin{cases} \rho_2 & \text{if } \tau'_N \text{ is of the form } \rho_1 \times \rho_2 \\ unit & \text{otherwise} \end{cases}$$
$$\zeta_{proj_2(N)} = \begin{cases} \rho_1 & \text{if } \tau'_N \text{ is of the form } \rho_1 \times \rho_2 \\ unit & \text{otherwise} \end{cases}$$
ζ_N is undefined if N is not of the form $proj_j(N')$

Then,

$$\theta' = [\overrightarrow{\tau'_N}/\overrightarrow{\alpha_N}, \overrightarrow{\eta'_x}/\overrightarrow{\beta_x}, \overrightarrow{\zeta_{proj_2(N)}}/\overrightarrow{\gamma_{proj_2(N)}}, \overrightarrow{\zeta_{proj_1(N)}}/\overrightarrow{\gamma_{proj_1(N)}}]$$

satisfies

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

By Lemma B.4, θ' also satisfies $\sim$.

Suppose a subterm N of M is replaced with *unit* in the transformation $\Gamma \vdash M : \tau \Rightarrow M''$. Then, $\tau'_N(= \theta'(\alpha_N))$ is *unit*. Since θ' satisfies $\sim$, $mgu_{\sim}(\alpha_N)$ must be either *unit* or a type variable. By the definition of M', N or a term containing N is replaced with $()$ in $\Gamma \vdash M : \tau \Rightarrow M'$, which implies $M'' \succeq M'$ as required. □