Dead Code Elimination through Dependent Types*

Hongwei Xi

Department of Computer Science and Engineering
Oregon Graduate Institute
P.O. Box 91000
Portland, OR 97291, USA

e-mail: hongwei@cse.ogi.edu

Abstract. Pattern matching is an important feature in various functional programming languages such as SML, Caml, Haskell, etc. In these languages, unreachable or redundant matching clauses, which can be regarded as a special form of dead code, are a rich source for program errors. Therefore, eliminating unreachable matching clauses at compile-time can significantly enhance program error detection. Furthermore, this can also lead to significantly more efficient code at run-time.

We present a novel approach to eliminating unreachable matching clauses through the use of the dependent type system of DML, a functional programming language that enriches ML with a restricted form of dependent types. We then prove the correctness of the approach, which consists of the major technical contribution of the paper. In addition, we demonstrate the applicability of our approach to dead code elimination through some realistic examples. This constitutes a practical application of dependent types to functional programming, and in return it provides us with further support for the methodology adopted in our research on *dependent types in practical programming*.

1 Introduction

There is no precise definition of dead code in the literature. In this paper, we refer dead code as the code which can never be executed at run-time. Notice that this is essentially different from dead computation [1], that is, the computation producing values which are never used. For instance, in the following C code,

```
x = 1; x = 2;
```

the part `x = 1` is dead computation but not dead code since it is executed but its execution does not affect the entire computation. Also dead code is different from partially dead code, which is not executed only on *some* computation paths [10]. For instance, the part `y = 1` in the following C code is partially dead since it is not executed when `x` is not zero.

* The research reported in this paper was supported in part by the United States Air Force Materiel Command (F19628-96-C-0161) and the Department of Defense.

```
if (x == 0) { y = 1; }
```

Pattern matching is an important feature in many functional programming languages such as Standard ML[12], Caml[16], Haskell[6], etc. A particular form of dead code in these languages is unreachable or redundant matching clauses, that is, matching clauses which can never be chosen at run-time. The following is a straightforward but unrealistic example of dead code in Standard ML.

```
exception Unreachable
fun foo l =
  case l of nil => l | _::_ => l | _ => raise Unreachable
```

The function `foo` is of type `'a list -> 'a list`. Clearly, the third matching clause in the definition of `foo` can never be chosen since every value of type `'a list` matches either the first or the second clause. This form of unreachable matching clauses can be readily detected in practice. For instance, if the above code is passed to the (current version of) SML/NJ compiler, an *error* message is reported describing the redundancy of the third matching clause. However, there are realistic cases which are much more subtle. Let us consider the following example.

```
exception ZipException
fun zip(nil, nil) = nil
  | zip(x::xs, y::ys) = (x, y)::zip(xs, ys)
  | zip _ = raise ZipException
```

The function `zip` is meant to zip two lists of *equal* length into one. This is somewhat hinted in the program since an exception is raised if two given lists are of different lengths. If `zip` is applied to a pair of lists of equal length, then the third matching clause in its definition can never be chosen for the obvious reason. Therefore, we can eliminate the third matching clause as long as we can guarantee that `zip` is *always* applied to a pair of lists of equal length. Unfortunately, it is *impossible* in Standard ML (or other languages with similar or weaker typing systems) to restrict the application of `zip` only to pairs of lists of equal length since its type system cannot distinguish lists of different lengths. This partially motivated our research on strengthening the type system of ML with dependent types so that we can formulated more accurate types such as the type of all pairs of lists of equal length.

The type system of DML (Dependent ML) in [20] enriches that of ML with *a restricted form* of dependent types. The primary goal of this enrichment is to provide for specification and inference of significantly more accurate information, facilitating both program error detection and compiler optimization. For instance, the following declaration in DML enables the type system to distinguish lists of different lengths.

```
datatype 'a list = nil | :: of 'a * 'a list
typeref 'a list of nat with
  nil <| 'a list(0)
| :: <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

The declaration defines a (polymorphic) datatype `'a list` to represent the type of lists. This datatype is then indexed by a natural number, which stands for the length of a list in this case. The constructors associated with the datatype `'a list` are then assigned dependent types:

- `nil <| 'a list(0)` states that `nil` is a list of length 0, and
- `:: <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)` states that `::` yields a list of length $n+1$ when given a pair consisting of an element and a list of length n. Note that `{n:nat}` means that n is universally quantified over natural numbers, usually written as $\Pi n : nat$.

Now the following definition of the zip function in DML guarantees that this function can only be applied to a pair of lists of equal length.

```
fun('a,'b)
  zip_safe(nil, nil) = nil
| zip_safe(x::xs, y::ys) = (x, y)::zip_safe(xs, ys)
where zip_safe <| {n:nat} 'a list(n)*'b list(n)->('a * 'b) list(n)
```

The use of `fun('a,'b)` is a recent feature of Standard ML [12], which allows the programmer to explicitly control the scope of the type variables `'a` and `'b`. The type of `zip_safe` is `{n:nat} 'a list(n) * 'b list(n)->('a * 'b)list(n)` which states that `zip_safe` can only accept a pair of lists of equal length and always returns a list of the same length. Notice that the `where` clause is a type annotation which must be provided by the programmer. If this function is applied to a pair of lists of unequal lengths, the code cannot pass the type-checking in DML and therefore is rejected. In this case, the programmer can certainly choose to use the previously defined `zip` if necessary in order to pass type-checking.

The programmer often knows that certain matching clauses can never be executed if a program is implemented correctly. Therefore, it can lead to program error detection at compile-time if we can verify whether these clauses can indeed be eliminated in an implementation. For example, when implementing an evaluator for the pure call-by-value λ-calculus, we know that we can never encounter a variable which is not bound to a closure during the evaluation of a *closed* λ-expression. As shown in Section 4.2, this can be readily verified by our approach. Eliminating dead code can also lead to more efficient execution at run-time. For instance, `zip_safe` need not check the tag of the second list in its argument since it is always the same as that of the first one, and this can contribute to 20% speedup on a Sun Sparc 20 running SML/NJ version 110. In general, our approach can be regarded as an example which supports that safety and efficiency can be *complementary*. These advantages yield some strong justification for our approach to eliminating unreachable matching clauses at compile-time.

We organize the paper as follows. In Section 2, we give some preliminaries on the core of DML, which provides all the machinery needed to establish the correctness of our approach to dead code elimination. We then present in Section 3 the derivation rules which formalize the approach, and prove the correctness of

these rules. This constitutes the main technical contribution of this paper. In Section 4, we use some realistic examples to demonstrate the applicability of our approach. The rest of the paper discusses some related work and then concludes.

2 Preliminaries

It is both infeasible and unnecessary to present in this paper the entire DML, which can be found in [20]. We refer the reader to [18] for an overview of DML. The essence of our approach will be fully captured in an core language $\mathrm{ML}_0^{\Pi}(C)$, which is a monomorphic extension of mini-ML with general pattern matching and universal dependent types.

Note that the omission of ML let-polymorphism in $\mathrm{ML}_0^{\Pi}(C)$, which is fully supported in DML, is simply for the sake of brevity of the presentation. Since $\mathrm{ML}_0^{\Pi}(C)$ parameterizes over a constraint domain C from which type index objects are drawn, we start with a brief introduction of the formation of a constraint domain.

2.1 Constraint Domains

We emphasize that the general constraint language itself is typed. In order to avoid potential confusion we call the types in the constraint language *index sorts*. We use b for base index sorts such as o for propositions and int for integers. A signature Σ declares a set of function symbols and associates with every function symbol an *index sort* defined below. A *Σ-structure* $\mathcal{D}$ consists of a set $\mathbf{dom}(\mathcal{D})$ and an assignment of functions to the function symbols in Σ. A constraint domain $C = \langle \Sigma, \mathcal{D} \rangle$ consists of a signature Σ and a Σ-structure $\mathcal{D}$.

We use f for interpreted function symbols (constants are 0-ary functions), p for atomic predicates (that is, functions of sort $\gamma \to o$) and we assume that we have constants such as equality, truth values $\top$ and $\bot$, conjunction $\wedge$, and disjunction $\vee$, all of which are interpreted as usual.

$$\begin{array}{ll} \text{index sorts} & \gamma ::= b \mid \mathbf{1} \mid \gamma_1 * \gamma_2 \mid \{a : \gamma \mid P\} \\ \text{index propositions} & P ::= \top \mid \bot \mid p(i) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \end{array}$$

Here $\{a : \gamma \mid P\}$ is the subset index sort for those elements of index sort γ satisfying proposition P, where P is an index proposition. For instance, nat is an abbreviation for $\{a : int \mid a \geq 0\}$, that is, nat is a subset index sort of int.

We use a for index variables in the following formation. We assume that there exists a predicate $\doteq$ of sort $\gamma * \gamma \to o$ for every index sort γ, which is interpreted as equality. Also we emphasize that all function symbols declared in Σ must be associated with index sorts of form $\gamma \to b$ or b. In other words, the constraint language is first-order.

$$\begin{array}{lrl} \text{index objects} & i, j & ::= a \mid \mathrm{f}(i) \mid \langle\rangle \mid \langle i, j \rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\ \text{index contexts} & \phi & ::= \cdot_{\mathrm{ind}} \mid \phi, a : \gamma \mid \phi, P \\ \text{index constraints} & \Phi & ::= i \doteq j \mid \top \mid \Phi_1 \wedge \Phi_2 \mid P \supset \Phi \mid \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi \\ \text{index substitutions} & \theta & ::= [] \mid \theta[a \mapsto i] \\ \text{satisfiability relation} & & \phi \models \Phi \end{array}$$

Note $\cdot_{\text{ind}}$ is for the empty index context. We omit the standard sorting rules for this language for the sake of brevity. The satisfiability relation $\phi \models \Phi$ means that $(\phi)\Phi$ is satisfiable in the domain $\mathbf{dom}(\mathcal{D})$ of the Σ-structure $\mathcal{D}$, where $(\phi)\Phi$ is defined as follows.

$$
\begin{array}{l}
(\cdot_{\text{ind}})\Phi = \Phi \\
(a : b)\Phi = \forall a : b.\Phi \\
(a : \gamma_1 * \gamma_2)\Phi = (a_1 : \gamma_1)(a_2 : \gamma_2)\Phi[a \mapsto \langle a_1, a_2 \rangle] \\
(\phi, \{a : \gamma \mid P\})\Phi = (\phi)(a : \gamma)(P \supset \Phi) \\
(\phi, P)\Phi = (\phi)(P \supset \Phi)
\end{array}
$$

In this paper, we are primarily interested in the integer constraint domain, where the signature Σ_{int} is given in Figure 1 and the domain of Σ_{int}-structure is simply the set of integers. For instance, let $\phi = n : nat, a : nat, a + 1 \doteq n, 0 \doteq n$, then $\phi \models \bot$ holds in this domain since $(\phi)\bot$, defined as follows, is true.

$$\forall n : int.n \geq 0 \supset \forall a : int.a \geq 0 \supset (a + 1 \doteq n \supset (0 \doteq n \supset \bot))$$

Basically, $a+1 \doteq n$ implies $n > 0$ since a is a natural number, and this contradicts $0 \doteq n$. In other words, the index context ϕ is inconsistent. This example will be used later.

$$
\begin{array}{rlll}
\Sigma_{\text{int}} = & abs : int \to int & sgn : int \to int & + : int * int \to int \\
& - : int * int \to int & * : int * int \to int & div : int * int \to int \\
& min : int * int \to int & max : int * int \to int & mod : int * int \to int \\
& < : int * int \to o & \leq : int * int \to o & = : int * int \to o \\
& \geq : int * int \to o & > : int * int \to o & \neq : int * int \to o
\end{array}
$$

Fig. 1. The signature of the integer domain

2.2 The Language $\mathbf{ML}_0^{\Pi}(C)$

Given a constraint domain C, the syntax of $\mathrm{ML}_0^{\Pi}(C)$ is given in Figure 2. The syntax is relatively involved because we must include general pattern matching in order to present our approach. This is an explicitly typed language since the dead code elimination we propose is performed when type-checking a program is done and an explicitly typed version of the program has been constructed.

We use δ for base type families, where we use $\delta(\langle\rangle)$ for an unindexed type. Also we use $\cdot_{\text{ms}}$, $\cdot_{\text{ind}}$ and $\cdot$ for empty matches, empty index context and empty context, respectively. The difference between values and value forms is that the former is only closed under value substitutions while the latter is closed under all substitutions. We write $e[i]$ for the application of an expression to a type index and $\lambda a : \gamma.e$ for index variable abstraction.

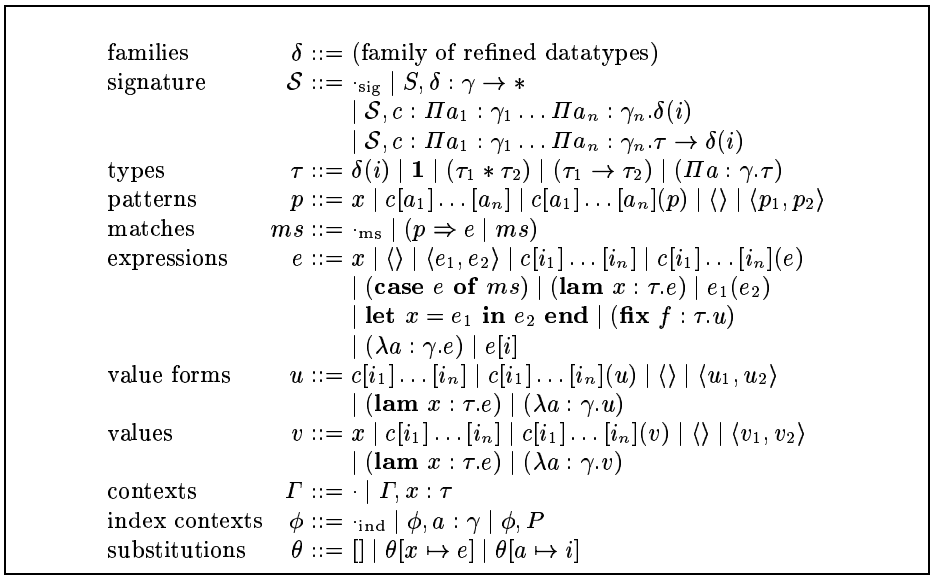

Fig. 2. The syntax for $\mathrm{ML}_0^{\Pi}(C)$

We leave out polymorphism in $\mathrm{ML}_0^{\Pi}(C)$ because polymorphism is largely orthogonal to the development of dependent types and has little effect on the dead code elimination presented in this paper. This tremendously simplifies the presentation.

In the rest of the paper, we assume that datatypes *intList* (for integer lists) and *intPairList* (for integer pair lists) have been declared with associated constructors *intNil*, *intCons* and *intPairNil*, *intPairCons*, respectively, and refined as follows.

$$
\begin{array}{ll}
\mathit{intNil} & : \mathit{intList}(0) \\
\mathit{intCons} & : \Pi n : \mathit{nat}.\mathit{int} * \mathit{intList}(n) \to \mathit{intList}(n+1) \\
\mathit{intPairNil} & : \mathit{intPairList}(0) \\
\mathit{intPairCons} & : \Pi n : \mathit{nat}.(\mathit{int} * \mathit{int}) * \mathit{intPairList}(n) \to \mathit{intPairList}(n+1)
\end{array}
$$

However, we will write *nil* for either *intNil* or *intPairNil* and *cons* for either *intCons* or *intPairCons* if there is no danger of confusion. The following expression *zipdef* in $\mathrm{ML}_0^{\Pi}(C)$ corresponds a monomorphic version of the previously defined function `zip_safe` (we use b for an index variable here).

$$
\begin{array}{l}
\mathbf{fix}\ \mathit{zip} : \Pi n : \mathit{nat}.\mathit{intList}(n) * \mathit{intList}(n) \to \mathit{intPairList}(n). \\
\quad \lambda n : \mathit{nat}.\mathbf{lam}\ l : \mathit{intList}(n) * \mathit{intList}(n). \\
\quad\quad \mathbf{case}\ l\ \mathbf{of}\ (\mathit{nil}, \mathit{nil}) \Rightarrow \mathit{nil} \\
\quad\quad\quad\quad \mid (\mathit{cons}[a](x, \mathit{xs}), \mathit{cons}[b](y, \mathit{ys})) \Rightarrow \mathit{cons}[a]((x, y), \mathit{zip}[a](\mathit{xs}, \mathit{ys}))
\end{array}
$$

Static Semantics We use $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i \doteq j$ from index objects to types, which is determined by the following rules.

$$\frac{\phi \models i \doteq j}{\phi \models \delta(i) \equiv \delta(j)} \qquad \frac{\phi \models \tau_1 \equiv \tau_1' \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'}$$

$$\frac{\phi \models \tau_1' \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Pi a : \gamma.\tau \equiv \Pi a : \gamma.\tau'}$$

We start with the typing rules for patterns, which are listed in Figure 3. These rules are essential to the formulation of our approach to eliminating unreachable matching clauses. The judgment $p \downarrow \tau \rhd (\phi; \Gamma)$ expresses that the index and ordinary variables in pattern p have the sorts and types in ϕ and Γ, respectively, if we know that p must have type τ.

$$\frac{}{x \downarrow \tau \rhd (\cdot_{\text{ind}}; x : \tau)}\ \textbf{(pat-var)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \rhd (\cdot_{\text{ind}}; \cdot)}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \rhd (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \rhd (\phi_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}\ \textbf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\delta(i)}{c[a_1]\ldots[a_n] \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j; \cdot)}\ \textbf{(pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \to \delta(i)) \quad p \downarrow \tau \rhd (\phi; \Gamma)}{c[a_1]\ldots[a_n](p) \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)}\ \textbf{(pat-cons-w)}$$

Fig. 3. Typing rules for patterns

We omit the rest of typing rules for $\text{ML}_0^{\Pi}(C)$, which can be found in [20]. The following lemma is at the heart of our approach to dead code elimination.

Lemma 1. *(Main Lemma) Suppose that $p \downarrow \tau \rhd \phi; \Gamma$ is derivable and $\phi \models \bot$ satisfiable. If $\cdot_{\text{ind}}; \cdot \vdash v : \tau$ is derivable, then v* does not *match the pattern p.*

Proof. The proof proceeds by structural induction on the derivation of $p \downarrow \tau \rhd \phi; \Gamma$. We present one interesting case where p is of form $c[a_1]\ldots[a_n](p')$ for some constructor c. Then the derivation of $p \downarrow \tau \rhd \phi; \Gamma$ must be of the following form.

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau' \to \delta(i)) \quad p' \downarrow \tau' \rhd (\phi'; \Gamma)}{c[a_1]\ldots[a_n](p') \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi'; \Gamma)}\ \textbf{(pat-cons-w)}$$

where $\tau = \delta(j)$, $\phi = a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi'$. Assume that v matches p. Then v is of form $c[i_1]\ldots[i_n](v')$, v' matches p' and $\cdot_{\text{ind}} \vdash i_k : \gamma_k$ are derivable for $1 \le k \le n$. Since $c[i_1]\ldots[i_n](v')$ is of type $\delta(i[\theta])$ for $\theta = [a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$, $\cdot_{\text{ind}} \models i[\theta] \doteq j$ is satisfiable. This leads to the satisfiability of $\phi'[\theta] \models \bot$ since

$\phi \models \bot$ is satisfiable. Notice that we can also derive $\cdot_{\text{ind}}; \cdot \vdash v' : \tau'[\theta]$ and $p' \downarrow \tau'[\theta] \rhd (\phi'[\theta], \Gamma[\theta])$. By induction hypothesis, v' does not match p'. This yields a contradiction, and therefore v does not match p.

All other cases can be treated similarly.

Dynamic Semantics The operational semantics of $\text{ML}_0^{\Pi}(C)$ can be given as usual in the style of natural semantics [9]. Again we omit the standard evaluation rules, which can be found in [20]

We use $e \hookrightarrow_d v$ to mean that e reduces to a value v in this semantics. These evaluation rules are only needed for proving the correctness of our approach. The following type-preservation theorem is also needed for this purpose.

Theorem 1. *(Type preservation in $\text{ML}_0^{\Pi}(C)$) Given e, v in $\text{ML}_0^{\Pi}(C)$ such that $e \hookrightarrow_d v$ is derivable. If $\cdot_{\text{ind}}; \cdot \vdash e : \tau$ is derivable, then $\cdot_{\text{ind}}; \cdot \vdash v : \tau$ is derivable.*

Proof. Please see Section 4.1.2 in [20] for details.

Notice that there is some nondeterminism associated with the rule for evaluating a case statement. If more than one matching clauses can match the value, there is no order to determine which one should be chosen. This is different from the deterministic strategy adopted in ML, which always chooses the first one which matches. We shall come back to this point later.

2.3 Operational Equivalence

In order to prove the correctness of our approach to dead code elimination, we must show that this approach does not alter the semantics of a program. We introduce the operational equivalence relation $\cong$ as follows for this purpose.

Definition 1. *We present the definition of contexts as follows.*

$$
\begin{array}{lrcl}
\textit{(matching contexts)} & C_m & ::= & \mid (p \Rightarrow C \mid ms) \mid (p \Rightarrow e \mid C_m) \\
\textit{(contexts)} & C & ::= & [] \mid \langle C, e\rangle \mid \langle e, C\rangle \mid c(C) \mid \textbf{lam}\ x : \tau.C \mid C(e) \mid e(C) \\
 & & & \mid \textbf{case}\ C\ \textbf{of}\ ms \mid \textbf{case}\ e\ \textbf{of}\ C_m \mid \textbf{fix}\ f : \tau.C \\
 & & & \mid \textbf{let}\ x = C\ \textbf{in}\ e\ \textbf{end} \mid \textbf{let}\ x = e\ \textbf{in}\ C\ \textbf{end}
\end{array}
$$

Given a context C and an expression e, $C[e]$ stands for the expression formulated by replacing with e the hole $[]$ in C. We emphasize that *this replacement is variable capturing*.

Definition 2. *Given two expression e_1 and e_2 in $\text{ML}_0^{\Pi}(C)$, e_1 is operationally equivalent to e_2 if the following holds.*

- *Given any context C such that $\cdot_{\text{ind}}; \cdot \vdash C[e_i] : \mathbf{1}$ are derivable for $i = 1, 2$, $C[e_1] \hookrightarrow_d \langle\rangle$ is derivable if and only if $C[e_2] \hookrightarrow_d \langle\rangle$ is.*

We write $e_1 \cong e_2$ if e_1 is operationally equivalent to e_2.

Clearly $\cong$ is an equivalence relation. Our aim is to prove that if a program e is transformed into $\underline{e}$ after dead code elimination, the $e \cong \underline{e}$. In other words, we intend to show that dead code elimination does not alter the operational semantics of a program.

3 Dead Code Elimination

We now go through an example to show how dead code elimination is performed on a program. This approach is then formalized and proven correct. This constitutes the main technical contribution of the paper.

3.1 An Example

Let us declare the function `zip_safe` in DML as follows.

```
fun zip_safe(intNil, intNil) = intPairNil
  | zip_safe(intCons(x,xs), intCons(y,ys)) =
    intPairCons((x, y), zip_safe(xs, ys))
  | zip_safe _ = raise ZipException
where zip_safe <| {n:nat} intList(n)*intList(n)->intPairList(n)
```

This declaration is then elaborated (after type-checking) to the following expression in $\mathrm{ML}_0^{\Pi}(C)$ (we assume that $raise(ZipException)$ is a legal expression). Notice that the third matching clause in the above definition is transformed into two *non-overlapping* matching clauses. This is necessary since pattern matching is done sequentially in ML, and therefore the value must match either pattern `(intNil, intCons(_,_))` or `(intCons(_,_), intNil)` if the third clause is chosen. The approach to performing such a transform is standard and therefore omitted.

$$
\begin{array}{l}
\textbf{fix}\ zip : \Pi n : nat.intList(n) * intList(n) \to intPairList(n). \\
\quad \lambda n : nat.\textbf{lam}\ l : intList(n) * intList(n). \\
\quad\quad \textbf{case}\ l\ \textbf{of}\ (nil, nil) \Rightarrow nil \\
\qquad\qquad\qquad \mid\ (cons[a](x, xs), cons[b](y, ys) \Rightarrow cons[a]((x, y), zip[a](xs, ys)) \\
\qquad\qquad\qquad \mid\ (nil, cons[b](y, ys)) \Rightarrow raise(ZipException) \\
\qquad\qquad\qquad \mid\ (cons[a](x, xs), nil) \Rightarrow raise(ZipException)
\end{array}
$$

If the third clause in the above expression can be chosen, then $(nil, cons[b](y, ys))$ must match a value of type $intList(n) * intList(n)$ for some natural number n. Checking $(nil, cons[b](y, ys))$ against $intList(n) * intList(n)$, we derive the following for $\phi = (0 \doteq n, b : nat, b + 1 \doteq n)$.

$$(nil, cons[b](y, ys)) \downarrow intList(n) * intList(n) \rhd (\phi; y : int, ys : intList(b))$$

Notice that ϕ is inconsistent since $\phi \models \bot$ is satisfiable. Therefore, we know by Lemma 1 that the third clause is unreachable. Similarly, the fourth clause is also unreachable. We can thus eliminate the dead code when compiling the program.

3.2 Formalization

While the above informal presentation of dead code elimination is intuitive, we have yet to demonstrate why this approach is correct and how it can be

$$\frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau \gg x} \textbf{ (elim-var)}$$

$$\frac{\mathcal{S}(c) = \Pi \overrightarrow{a} : \overrightarrow{\gamma}.\delta(i)}{\phi; \Gamma \vdash c[\overrightarrow{i}] : \delta(i[\overrightarrow{a} \mapsto \overrightarrow{i}]) \gg c[\overrightarrow{i}]} \textbf{ (elim-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi \overrightarrow{a} : \overrightarrow{\gamma}.\tau \to \delta(i) \quad \phi; \Gamma \vdash e : \tau[\overrightarrow{a} \mapsto \overrightarrow{i}] \gg \underline{e}}{\phi; \Gamma \vdash c[\overrightarrow{i}](e) : \delta(i[\overrightarrow{a} \mapsto \overrightarrow{i}]) \gg c[\overrightarrow{i}](\underline{e})} \textbf{ (elim-cons-w)}$$

$$\frac{}{\phi; \Gamma \vdash \langle\rangle : \mathbf{1} \gg \langle\rangle} \textbf{ (elim-unit)}$$

$$\frac{\phi; \Gamma \vdash e_1 : \tau_1 \gg \underline{e_1} \quad \phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e_2}}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2 \gg \langle \underline{e_1}, \underline{e_2} \rangle} \textbf{ (elim-prod)}$$

$$\frac{\phi \vdash \Gamma[\mathbf{ctx}] \quad \phi \vdash \tau_1 \Rightarrow \tau_2 : *}{\phi; \Gamma \vdash \cdot_{\mathrm{ms}} : \tau_1 \Rightarrow \tau_2 \gg \cdot_{\mathrm{ms}}} \textbf{ (elim-mat-empty)}$$

$$\frac{p \downarrow \tau_1 \rhd (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2 \gg \underline{e} \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2 \gg (p \Rightarrow \underline{e} \mid \underline{ms})} \textbf{ (elim-mat)}$$

$$\frac{p \downarrow \tau_1 \rhd (\phi'; \Gamma') \quad \phi, \phi' \models \bot \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}} \textbf{ (elim-mat-dead)}$$

$$\frac{\phi; \Gamma \vdash e : \tau_1 \gg \underline{e} \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms : \tau_2 \gg \mathbf{case}\ \underline{e}\ \mathbf{of}\ \underline{ms}} \textbf{ (elim-case)}$$

$$\frac{\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}}{\phi; \Gamma \vdash \lambda a : \gamma.e : (\Pi a : \gamma.\tau) \gg \lambda a : \gamma.\underline{e}} \textbf{ (elim-ilam)}$$

$$\frac{\phi; \Gamma \vdash e : \Pi a : \gamma.\tau \gg \underline{e} \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e[i] : \tau[a \mapsto i] \gg \underline{e}[i]} \textbf{ (elim-iapp)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2 \gg \underline{e}}{\phi; \Gamma \vdash \mathbf{lam}\ x : \tau_1.e : \tau_1 \to \tau_2 \gg \mathbf{lam}\ x : \tau_1.\underline{e}}$$

$$\frac{\phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \gg \underline{e_1} \quad \phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e_2}}{\phi; \Gamma \vdash e_1(e_2) : \tau_2 \gg \underline{e_1}(\underline{e_2})}$$

$$\frac{\phi; \Gamma \vdash e_1 : \tau_1 \gg \underline{e_1} \quad \phi; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \gg \underline{e_2}}{\phi; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2 \gg \mathbf{let}\ x = \underline{e_1}\ \mathbf{in}\ \underline{e_2}\ \mathbf{end}} \textbf{ (elim-let)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u : \tau \gg \underline{u}}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) : \tau \gg \mathbf{fix}\ f : \tau.\underline{u}} \textbf{ (elim-fix)}$$

Fig. 4. The rules for dead code elimination

implemented. For this purpose, we formalize the approach with derivation rules. A judgment of form $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ means that an expression e of type τ under $\phi; \Gamma$ transforms into the expression $\underline{e}$ through dead code elimination. The rules for deriving such a judgment are presented in Figure 4. Notice that the rule **(elim-mat-dead)** is the only rule which eliminates dead code.

Proposition 1. *We have the following.*

1. *If $\phi; \Gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash e : \tau \gg e$ is also derivable.*
2. *if $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ and $\cdot_{\mathrm{ind}}; \cdot \vdash C[e] : \mathbf{1}$ are derivable, then $\cdot_{\mathrm{ind}}; \cdot \vdash C[e] : \mathbf{1} \gg C[\underline{e}]$ is also derivable.*
3. *If $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ is derivable, then both $\phi; \Gamma \vdash e : \tau$ and $\phi; \Gamma \vdash \underline{e} : \tau$ are derivable.*

Proof. (1) and (2) are straightforward, and (3) follows from structural induction on the derivation of $\phi; \Gamma \vdash e : \tau \gg \underline{e}$.

Proposition 1 (1) simply means that we can always choose not to eliminate any code, and (2) means that dead code elimination is independent of context, and (3) means that dead code elimination is type-preserving.

Lemma 2. *(Substitution)*

1. *If both $\phi; \Gamma, x : \tau_2 \vdash e_1 : \tau_1 \gg \underline{e_1}$ and $\phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e_2}$ are derivable, then so is $\phi; \Gamma \vdash e_1[x \mapsto e_2] : \tau_1 \gg \underline{e_1}[x \mapsto \underline{e_2}]$.*
2. *If both $\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}$ and $\phi \vdash i : \gamma$ are derivable, then so is $\phi; \Gamma[a \mapsto i] \vdash e[a \mapsto i] : \tau[a \mapsto i] \gg \underline{e}[a \mapsto i]$.*

Proof. (1) and (2) follow from structural induction on the derivations of $\phi; \Gamma, x : \tau_2 \vdash e_1 : \tau_1 \gg \underline{e_1}$ and $\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}$, respectively.

Lemma 3. *Assume that $\cdot_{\mathrm{ind}}; \cdot \vdash e : \tau \gg \underline{e}$ is derivable.*

1. *If $e \hookrightarrow_d v$ is derivable, then $\underline{e} \hookrightarrow_d \underline{v}$ is derivable for some $\underline{v}$ such that $\cdot_{\mathrm{ind}}; \cdot \vdash v : \tau \gg \underline{v}$ is also derivable.*
2. *If $\underline{e} \hookrightarrow_d \underline{v}$ is derivable, then $e \hookrightarrow_d v$ is derivable for some v such that $\cdot_{\mathrm{ind}}; \cdot \vdash v : \tau \gg \underline{v}$ is also derivable.*

Proof. (1) and (2) follow from structural induction on the derivations of $e \hookrightarrow_d v$ and $\underline{e} \hookrightarrow_d \underline{v}$, respectively, using both Lemma 1 and Lemma 2.

Theorem 2. *If $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ is derivable, then $e \cong \underline{e}$ holds.*

Proof. Let C be a context such that both $\cdot_{\mathrm{ind}}; \cdot \vdash C[e] : \mathbf{1}$ and $C[e] \hookrightarrow_d \langle\rangle$ are derivable. By Proposition 1, $\cdot_{\mathrm{ind}}; \cdot \vdash C[e] : \mathbf{1} \gg C[\underline{e}]$ is derivable. Hence, by Lemma 3 (1), $C[\underline{e}] \hookrightarrow_d \underline{v}$ is derivable for some $\underline{v}$ such that $\cdot_{\mathrm{ind}}; \cdot \vdash \langle\rangle : \mathbf{1} \gg \underline{v}$. This implies that $\underline{v}$ is $\langle\rangle$. Similarly, by Lemma 3 (2), we can prove that $C[e] \hookrightarrow_d \langle\rangle$ is derivable for every context C such that both $\cdot_{\mathrm{ind}}; \cdot \vdash C[\underline{e}] : \mathbf{1}$ and $C[\underline{e}] \hookrightarrow_d \langle\rangle$ are derivable. Therefore, $e \cong \underline{e}$ holds.

4 Examples

We present some realistic examples in this section to demonstrate the effect of dead code elimination through dependent types. Note that polymorphism is allowed in this section.

4.1 The nth function

When applied to (l, n), the following `nth` function returns the nth element in the list l if n is less than the length of l and raises the `Subscript` exception otherwise. This function is frequently called in functional programming, where the use of lists is pervasive.

```
fun nth(nil, _) = raise Subscript
  | nth(x::xs, n) = if n = 0 then x else nth(xs, n-1)
```

If we assign `nth` the following type, that is, we restrict the application of `nth` to pairs (l, n) such that n is always less than the length of l,

```
{len:nat}{index:nat | index < len} 'a list(len) * int(index) -> 'a
```

then the first matching clause in the definition of `nth` is unreachable, and therefore can be safely eliminated. Note that we have refined the built-in type `int` into infinitely many singleton types such that `int(n)` contains *only* `n` for each integer `n`. Let us call this version `nth_safe`, and we have measured that `nth_safe` is about 25% faster than `nth` on a Sparc 20 station running SML/NJ version 110. The use of a similar idea to eliminate array bound checks can be found in [17].

4.2 An Evaluator for the Call-By-Value λ-Calculus

The code in Figure 5 implements an evaluator for the pure call-by-value λ-calculus. We use de Bruijn's notation to represent λ-expressions. For example, $\lambda x.\lambda y.y(x)$ is represented as `Lam(Lam(App(One, Shift(One))))`. We then refine the datatype `lambda_exp` into infinitely many types `lambda_exp(n)`. For each natural numbers `n`, `lambda_exp(n)` roughly stands for the type of all λ-terms in which there are *at most* `n` free variables.

If a value of type `lambda_exp(n) * closure list(n)` for some `n` matches the last clause in the definition of `cbv` then it matches either pattern `(One, nil)` or pattern `(Shift _, nil)`. In either case, a contradiction is reached since a value which matches either `One` or `Shift _` can not be of type `lambda_exp(0)` but `nil` is of type `closure list(0)`. Therefore, the last clause can be safely eliminated.

The programmer knows that the last clause is unreachable. After this is mechanically verified, the programmer gains confidence in the above implementation. On the other hand, if the last clause could not be safely eliminated, it would have been an indication of some program errors in the implementation.

```
datatype lambda_exp =
  One | Shift of lambda_exp |
  Lam of lambda_exp | App of lambda_exp * lambda_exp

datatype closure = Closure of lambda_exp * closure list

typeref lambda_exp of nat
with One <| {n:nat} lambda_exp(n+1)
   | Shift <| {n:nat} lambda_exp(n) -> lambda_exp(n+1)
   | Lam <| {n:nat} lambda_exp(n+1) -> lambda_exp(n)
   | App <| {n:nat} lambda_exp(n) * lambda_exp(n) -> lambda_exp(n)
   | Closure <| {n:nat} lambda_exp(n) * closure list(n) -> closure

exception Unreachable

fun callbyvalue(exp) = let
  fun cbv(One, clo::_) = clo
    | cbv(Shift(exp), _::env) = cbv(exp, env)
    | cbv(exp as Lam _, env) = Closure(exp, env)
    | cbv(App(exp1, exp2), env) = let
        val Closure(Lam(body), env1) = cbv(exp1, env)
        and clo = cbv(exp2, env)
      in cbv(body, clo::env1) end
    | cbv _ = raise Unreachable (* this can be safely eliminated *)
  where cbv <| {n:nat} lambda_exp(n) * closure list(n) -> closure
in
  cbv(exp, nil)
end
where callbyvalue <| lambda_exp(0) -> closure
(* Note: callbyvalue can only apply to CLOSED lambda expressions *)
```

Fig. 5. An evaluator for the call-by-value λ-calculus

4.3 Other Examples

So far all the presented examples involve the use of lists, but this is not necessary. We also have examples involving other data structures such as trees. For instance, the reader can find in [19] a red/black tree implementation containing unreachable matching clauses which can be eliminated in the same manner. In general, unreachable matching clauses are abundant in practice, of which many can be eliminated with our approach.

5 Related Work and Conclusion

It is beyond reasonable hope to mention even a moderate amount of research related or similar to dead code or dead computation elimination because of the vastness of the field. The reader can find further references in [8, 1, 7, 13, 10,

14, 11, 15]. Our approach to dead code elimination differs significantly from the previous approaches in several aspects.

We have adopted a type-based approach while most of the previous approaches are based on flow analysis. This gives us a great advantage when the issue of crossing module boundaries is concerned. For instance, after assigning the zip function the type

```
{n:nat} 'a list(n) * 'b list(n) -> ('a * 'b) list(n)
```

and eliminating the dead code, we can use this function *anywhere* as long as type-checking is passed. On the other hand, an approach based on flow analysis usually analyzes an instance of a function call and check whether there is dead code associated with this *particular* function call. One may argue that our approach must be supported by a dependent type system while an approach based on flow analysis need not. However, there would be no dead code in the zip function if we had not assigned it the above dependent type. It is the use of a dependent type system that enables us to exploit opportunities which do not exist otherwise.

Also we are primarily concerned with program error detection while most of the previous approaches were mainly designed for compiler optimization, which is only our secondary goal. Again, this is largely due to our adoption of a type-based approach.

We emphasize that our approach must rely on the type annotations supplied by the programmer in order to detect redundant matching clauses. It seems exceedingly difficult at this moment to find a procedure which can synthesize type annotations automatically. For instance, without the type annotation in the `zip_safe` example, it is unclear whether the programmer intends to apply the function to a pair lists of unequal lengths, and therefore unclear whether the last matching clause is redundant.

Our approach is most closely related to the research on refinement types [4, 2], which also aims for assigning programs more accurate types. However, the restricted form of dependent types in DML allows the programmer to form types which are not captured by the regular tree grammar [5], e.g., the type of all pairs of lists of equal length, but this is beyond the reach of refinement types. The price we pay is the loss of principal types, which may consequently lead to a more involved type-checking algorithm.

We have experimented our approach to dead code elimination in a prototype implementation of a type-checker for DML. We plan to incorporate this approach into the compiler for DML which we are building on top of Caml-light. Clearly, our approach can also be readily adapted to detecting uncaught exceptions [21], and we expect it to work well in this direction when combined with the approach in [3]. We shall report the work in the future.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.

2. Rowan Davies. Practical refinement-type checking. Thesis Proposal, Computer Science Department, Carnegie Mellon University, November 1997.
3. Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, September 1997.
4. Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, 1991.
5. F. Gecseg and M Steinb. *Tree automata*. Akademiai Kiado, 1991.
6. Paul Hudak, S. L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992.
7. John Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, pages 117–153. Addison-Wesley, 1990.
8. N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Conference Record of 6th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 244–256, January 1979.
9. Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.
10. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Implementation and Design*, pages 147 – 158, June 1994.
11. Y. Liu and D. Stoller. Dead code elimination using program-based regular tree grammars. Available at `http://ftp.cs.indiana.edu/pub/liu/ElimDeadRec-TR97.ps.Z`.
12. Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.
13. Heintze Nevin. *Set-based Program Analysis of ML programs*. Ph. D dissertation, Carnegie Mellon University, 1992.
14. F. Tip. A survey of program slicing. *Journal of Programming Languages*, 3(3):121–189, 1995.
15. P. Wadler and J. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385 – 407. Springer-Verlag, 1987.
16. Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, Paris, 1993.
17. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, June 1998.
18. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, January 1999. (To appear)
19. Hongwei Xi. Some examples of DML programming. Available at `http://www.cs.cmu.edu/~hwxi/DML/examples/`, November 1997.
20. Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Forthcoming. The current version is available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.
21. K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in standard ml programs. In *Proceedings of the 4th International Static Analysis Symposium*, September 1997.