Constraint Systems for Useless Variable Elimination

Mitchell Wand and Igor Siveroni *
College of Computer Science
Northeastern University
Boston, MA 02115, USA
{wand,igor}@ccs.neu.edu

Abstract

A useless variable is one whose value contributes nothing to the final outcome of a computation. Such variables are unlikely to occur in human-produced code, but may be introduced by various program transformations. We would like to eliminate useless parameters from procedures and eliminate the corresponding actual parameters from their call sites. This transformation is the extension to higher-order programming of a variety of dead-code elimination optimizations that are important in compilers for first-order imperative languages.

Shivers has presented such a transformation. We reformulate the transformation and prove its correctness. We believe that this correctness proof can be a model for proofs of other analysis-based transformations. We proceed as follows:

- We reformulate Shivers' analysis as a set of constraints; since the constraints are conditional inclusions, they can be solved using standard algorithms.
- We prove that any solution to the constraints is sound: that two computations that differ only on variables marked as useless give the same answer up to useless variables, and hence the same answer when the answer is a constant.
- We observe that this notion of soundness is too weak to support the transformation of the analyzed program. We add additional closure conditions to the analysis and show that any solution to the new set of constraints justifies the transformation. The proof is by tree surgery: we show that from any computation of the original program, we can construct a computation of the transformed program that yields an answer related to the original, and the same answer when the original answer is a constant.

*Work supported by the National Science Foundation under grants numbered CCR-9404646 and CCR-9629801.

To appear in Proceedings of 26th ACM Symposium on Principles of Programming Languages, Houston, TX, 1999.

1 Introduction

In a series of papers, we have explored the question of how program transformations are justified by program analyses [Wan93, SW97, WC98]. In this paper we consider the example of *useless variable elimination*. A useless variable is one whose value contributes nothing to the final outcome of a computation [Shi91].

For example, consider, the following example, obtained by translating [Muc97, Figure 18.18] into a functional setting:

```
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
```

Here it is clear that the second argument does not contribute to the value of the computation. Our transformation replaces `loop` by the evident two-argument procedure, getting

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

Our analysis removes all the useless variables identified by Muchnick's first-order analysis [Muc97, Section 18]. In addition, it identifies useless variables even in higher-order procedures. For example, consider:

```
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)
```

`f1` and `f2` are the only functions that flow to the application site `g(x,h)`. These functions do not use their second argument and they do not flow to any other calling points. Hence, the second actual parameter can be removed from the application site and both functions can be replaced by their one-argument versions. We observe that `h` and `f3` do not contribute to the value of the computation. We can remove them from the initial program, getting

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

The decision of removing `f3` from the `let` expression depended on the elimination of `h`, which in turn depended on the removal of the second actual parameter from `g(x,h)`. The second formal parameter of `f1` and `f2` can be eliminated only if the second actual parameter from `g(x,h)` is removed, and viceversa. It is essential that these transformations are done at the same time.

While useless variables seem unlikely to occur in high-quality human-produced code, this transformation seems worthy of study for several reasons:

- Such variables may be introduced by various program transformations. For example, arity-raising [PJS98] often creates useless variables.
- This transformation is the extension to higher-order programming of a variety of dead-code elimination optimizations that are important in compilers for first-order imperative languages.
- Such variables may be result from program maintenance or other engineering considerations. For example, [ST98] finds that an average of 12.5% of the data members in realistic C++ benchmark programs were useless.

We present an analysis and transformation to eliminate useless variables from programs. This analysis and transformation is similar to one proposed by Shivers [Shi91]. Our major contribution is a proof of correctness of the transformation. The plan of the work is:

- We reformulate Shivers' analysis as a set of constraints; since the constraints are conditional inclusions, they can be solved using standard algorithms [PS94]. We later show that although there may be $O(n^3)$ constraints, the standard algorithm still solves them in $O(n^3)$ time.
- We prove that any solution to the constraints is *sound*: that two computations that differ only on variables marked as useless give the same answer up to useless variables, and hence the same answer when the answer is a constant.
- We observe that this notion of soundness is too weak to support the transformation of the analyzed program. We add additional closure conditions to the analysis and show that any solution to the new set of constraints *justifies* the transformation: we show that from any computation of the original program, we can construct a computation of the transformed program that yields an answer related to the original, and the same answer when the original answer is a constant.

We begin in Section 2 with a survey of related work. In Sections 3 and 4 we present the source language, its semantics, and a simple control-flow analysis. In Section 5 we present our reformulation of Shivers' analysis as a constraint-generation system and prove its soundness. In Section 6 we observe (as did Shivers) that the naive analysis is insufficient; we then show how to strengthen it. In Section 7 we present the transformation and show that it is correct. Last, we close with comments on the complexity of the analysis, possible additions to support effects and separate compilation, and conclusions.

2 Related Work

Our treatment of useless-variable elimination for higher-order languages is based on Shivers [Shi91]. We extend Shivers' work by proving the correctness of the transformation. Our transformation also eliminates useless `let`-bindings and replaces recursive abstractions by non-recursive ones when possible. Shivers' result is an application of control-flow analysis, which has been rediscovered several times [Jon81, JM82, Ses88, Shi91]. The formulation of control-flow analysis as a constraint problem is largely due to [Hei92, PS94].

Our transformation is the extension to higher-order languages of optimizations such as as dead code elimination, useless variable elimination, and unreachable code elimination, which are described in detail for first-order imperative languages by most standard compiler texts [App98, ASU86, Muc97]. Our definition of dead code comes closest to the one given by Muchnick, who defines dead code as code that is executable but that has no effect on the result of the computation being performed [Muc97]. Since a variable is useless when its value contributes nothing to the final outcome of the computation, any terms whose values are bound to useless variables are dead code. Our transformation eliminates all such terms. These also include terms that correspond to unreachable code in imperative programs.

This is not the same as live-variable analysis: A variable is dead at a program point if its contribution to the result of the computation has already been incorporated into some other value; a useless variable is one which is dead at its initial binding.

Peyton Jones [PJS98] points out that "whilst it is unusual for programmers to write functions with arguments that are completely unused, it is rather common for them to write functions that does not use some parts of their arguments". This fact is exposed by the wrapper/worker splitting transformation used in the Glasgow Haskell Compiler(GHC) [PJHH+93]. The transformation creates a *wrapper* function that evaluates the strict arguments of the original functions and passes the unboxed values to the *worker*, which implements the original function. Elements of data structures (i.e. products) can be passed separately - if some of them are known to be strict - to the worker, exposing components that are not used at all by the function. These useless parameters are detected by an *absence analyzer* and removed from the code.

Program slicing [Wei84, HRB84] is a method for decomposing programs by analyzing their data and control flow. A slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at this point. Our dependency sets are much like slices, but we take advantage of the simpler structure of expressions: our labels l are like program points, but we are concerned only with the value of the expression at that label, and we collect just the variables upon which that value depends. We use the term *dependence analysis* for this process, but our concept seems unrelated to the notion of *dependence graphs* in first-order languages, which are relations on basic blocks using assignment.

Abadi *et al* [ALL96] define a dependency system for a higher-order language. However, rather than relying on analysis, they instrument the interpreter to keep track of which parts of the original term the result depends. They then cache these results to avoid future recalculation. This scheme enables them to cache pairs of the form (input pattern, output), rather than just (input, output) as in traditional memoizing systems [Mic68]. They also succinctly explain the difference between dependence analysis and strictness analysis: "strictness analysis is concerned with what parts of a program *must* be evaluated, [whereas dependence analysis is concerned with] what parts of a program *may* affect the result." [ALL96, p. 84]

Our previous work includes a series of papers that describe transformations justified by program analyses [SW97, Wan93, WC98]. [Wan93] proves the correctness of a binding-time analysis and an off-line partial evaluator, using purely syntactic notions much like those in this paper. Steckler and Wand [SW97] use a control-flow analysis with additional constraints to introduce site-specific procedure-calling protocols; their protocols add extra parameters (to avoid saving them in closures), whereas we now consider eliminating useless parameters. [SW97] also use notions of occurrence index and occurrence closures that we refine by using labels, label configurations and a universe Σ of labeled terms that link labels and terms. [WC98] uses a sequence of three analyses to identify dead values in a first-order language of recursion equations. That result relies entirely on the soundness properties of the analyses, unlike the current paper.

3 The Source Language

The syntax of the source language is shown in Figure 1. It is an untyped lambda calculus extended with primitive functions and explicit operators for conditionals, `let`, and recursive procedures. 0-ary procedures are considered by allowing $n = 0$ in the production for *Fun*. Labels are used to identify the position of a term in a program; an expression is a labeled term.

The semantics is a big-step call-by-value environment semantics. It deals with *configurations* consisting of an expression label l and an environment ρ mapping variables to values. Values are either constants c or closures consisting of a pair $\langle l,\ \rho\rangle$, where l is the label of an abstraction. Note that a closure $\langle l,\ \rho\rangle$ is both a configuration and a value.

$$\begin{array}{ll} C \in \mathit{Config} ::= \langle l,\ \rho\rangle & \text{(configurations)} \\ u, v, w \in \mathit{Val} ::= c \mid \langle l,\ \rho\rangle & \text{(values)} \\ \rho \in \mathit{Env} ::= [\,] \mid \rho[x \mapsto v] & \text{(environments)} \end{array}$$

We connect labels to terms by hypothesizing a finite *universe* Σ of labeled terms, with the property that no label occurs more than once in Σ. We write $\Sigma(l) = t$ if the unique occurrence of l in Σ labels the term t.

This enables us to connect the semantics (which deals with terms) to the analysis (which deals with labels). It is useful to think of Σ as a store containing the nodes of the parse trees of some expressions, and labels l as L-values pointing to those nodes. In some development environments, an object like Σ is called the "storage map." However, since labeled terms are finite, we can use induction on substructures of a labeled term as an induction principle.

The semantics is then given by judgments $\Sigma \vdash C \Downarrow v$ defined by the system of Figure 2. Except for the systematic use of labels to refer to terms in the universe, this is an ordinary call-by-value big-step environment semantics.

4 Control Flow Analysis

Our analyses depend on a 0CFA specified by the judgment $(\Sigma, l) \models (\hat{C}, \hat{\rho})$, where $\hat{C}$ is (following [NN97]) a map from labels to sets of labels of abstractions and $\hat{\rho}$ is a map from variables to sets of labels of abstractions. This represents the judgment that $\hat{C}$ and $\hat{\rho}$ represent a consistent description of label l in universe Σ. This notation is reversed from that in [NN97]: to us an analysis is the proposition, the set of labeled terms constitutes a model, and a particular label constitutes a "point" inside the model.

The semantics of these propositions is standard and is given by the definition of $\models$ in Figure 3. The definition is given by induction on the size of l in Σ, unlike the gfp version of [NN97]. In this figure we have used vertical braces to indicate conjunctions.

We extend this definition to configurations and environments by:

$$\begin{array}{rl} (\Sigma, \langle l,\ \rho\rangle) \models (\hat{C}, \hat{\rho}) & \text{iff } (\Sigma, l) \models (\hat{C}, \hat{\rho}) \wedge (\Sigma, \rho) \models (\hat{C}, \hat{\rho}) \\ (\Sigma, \rho) \models (\hat{C}, \hat{\rho}) & \text{iff } \ \forall x \in dom(\rho): \\ & \qquad \left\{ \begin{array}{l} (\Sigma, \rho(x)) \models (\hat{C}, \hat{\rho}) \\ \wedge\ \rho(x) = \langle l,\ \rho'\rangle \Rightarrow l \in \hat{\rho}(x) \end{array} \right. \\ (\Sigma, c) \models (\hat{C}, \hat{\rho}) & \text{always} \end{array}$$

The soundness of the control flow analysis says that if a configuration evaluates to a closure, then that closure is one that is predicted by the flow analysis as a possible value for the original configuration, and furthermore $(\hat{C}, \hat{\rho})$ is a consistent description of the result. The latter property is particularly useful to prove the application case in some of our theorems.

Lemma 4.1 (Soundness of Control Flow Analysis) *If* $(\Sigma, \langle l,\ \rho\rangle) \models (\hat{C}, \hat{\rho})$ *and* $\Sigma \vdash \langle l,\ \rho\rangle \Downarrow \langle l',\ \rho'\rangle$*, then* $l' \in \hat{C}(l)$ *and* $(\Sigma, \langle l',\ \rho'\rangle) \models (\hat{C}, \hat{\rho})$*.*

Proof: By induction on the derivation of $\Sigma \vdash \langle l,\ \rho\rangle \Downarrow \langle l',\ \rho'\rangle$.

5 Dependency Analysis and Useless Variables

We next proceed to the dependency analysis. The dependency analysis computes for each label l a superset of the set of variables that contribute to the value of $\Sigma(l)$. A variable is deemed *useless* for label l if it does not appear in $\mathcal{D}(l)$.

A dependency analysis is a function $\mathcal{D}$

$$\mathcal{D} : \mathit{Lab} \to \mathcal{P}(\mathit{Var})$$

$$
\begin{array}{llr}
e \in \mathit{Exp} ::= t^l & \text{(expressions)} & \\
\\
t \in \mathit{Term} & \text{(terms)} & \\
t \ ::= \quad c & & [\mathbf{const}] \\
\quad \mid \quad x & & [\mathbf{var}] \\
\quad \mid \quad f & & [\mathbf{fn}] \\
\quad \mid \quad (g\ e_1 \ldots e_n) & & [\mathbf{prim}] \\
\quad \mid \quad (e_0\ e_1 \ldots e_n) & & [\mathbf{app}] \\
\quad \mid \quad (\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & & [\mathbf{if}] \\
\quad \mid \quad (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & & [\mathbf{let}] \\
\\
f \in \mathit{Fun} & \text{(function terms)} & \\
f \ ::= \quad (\texttt{fn}\ x_1 \ldots x_n \Rightarrow e_0) & & \\
\quad \mid \quad (\texttt{fun}\ y\ x_1 \ldots x_n \Rightarrow e_0) & & \\
\\
g \in \mathit{Prim} & \text{(primitives)} & \\
c, d \in \mathit{Const} & \text{(constants)} & \\
x, y, z \in \mathit{Var} & \text{(variables)} & \\
\phi : \mathit{Prim} \to [\mathit{Const}^n \to \mathit{Const}] & &
\end{array}
$$

Figure 1: Syntax of the Source Language

$[\mathbf{const}]$ $\quad \Sigma \vdash \langle l,\ \rho \rangle \Downarrow c \qquad$ if $\Sigma(l) = c$

$[\mathbf{var}]$ $\quad \Sigma \vdash \langle l,\ \rho \rangle \Downarrow v \qquad$ if $\Sigma(l) = x \wedge x \in dom(\rho) \wedge \rho(x) = v$

$[\mathbf{fn}]$ $\quad \Sigma \vdash \langle l,\ \rho \rangle \Downarrow \langle l,\ \rho \rangle \qquad$ if $\Sigma(l) \in \mathit{Fun}$

$$[\mathbf{prim}] \quad \frac{\Sigma(l) = (g^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) \wedge \Sigma \vdash \langle l_i,\ \rho \rangle \Downarrow c_i}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow [\phi(g)](c_1 \ldots c_n)} \quad i = 1, \ldots, n$$

$$[\mathbf{app_{fn}}] \quad \frac{\begin{array}{l} \Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) \wedge \\ \Sigma \vdash \langle l_0,\ \rho \rangle \Downarrow \langle l_f,\ \rho_0 \rangle \wedge \\ \Sigma \vdash \langle l_i,\ \rho \rangle \Downarrow v_i \quad i = 1, \ldots, n \quad \wedge \\ \Sigma(l_f) = (\texttt{fn}\ x_1 \ldots x_n \Rightarrow t_b^{l_b}) \wedge \\ \Sigma \vdash \langle l_b,\ \rho_0[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \rangle \Downarrow v \end{array}}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow v}$$

$$[\mathbf{app_{fun}}] \quad \frac{\begin{array}{l} \Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) \wedge \\ \Sigma \vdash \langle l_0,\ \rho \rangle \Downarrow \langle l_f,\ \rho_0 \rangle \wedge \\ \Sigma \vdash \langle l_i,\ \rho \rangle \Downarrow v_i \quad i = 1, \ldots, n \quad \wedge \\ \Sigma(l_f) = (\texttt{fun}\ y\ x_1 \ldots x_n \Rightarrow t_b^{l_b}) \wedge \\ \Sigma \vdash \langle l_b,\ \rho_0[y \mapsto \langle l_f,\ \rho_0 \rangle][x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \rangle \Downarrow v \end{array}}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow v}$$

$$[\mathbf{if_1}] \quad \frac{\Sigma(l) = (\texttt{if}\ t_0^{l_0}\ \texttt{then}\ t_1^{l_1}\ \texttt{else}\ t_2^{l_2}) \wedge \Sigma \vdash \langle l_0,\ \rho \rangle \Downarrow \mathit{true} \wedge \Sigma \vdash \langle l_1,\ \rho \rangle \Downarrow v}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow v}$$

$$[\mathbf{if_2}] \quad \frac{\Sigma(l) = (\texttt{if}\ t_0^{l_0}\ \texttt{then}\ t_1^{l_1}\ \texttt{else}\ t_2^{l_2}) \wedge \Sigma \vdash \langle l_0,\ \rho \rangle \Downarrow \mathit{false} \wedge \Sigma \vdash \langle l_2,\ \rho \rangle \Downarrow v}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow v}$$

$$[\mathbf{let}] \quad \frac{\Sigma(l) = (\texttt{let}\ x = t_1^{l_1}\ \texttt{in}\ t_2^{l_2}) \wedge \Sigma \vdash \langle l_1,\ \rho \rangle \Downarrow v_1 \wedge \Sigma \vdash \langle l_2,\ \rho[x \mapsto v_1] \rangle \Downarrow v}{\Sigma \vdash \langle l,\ \rho \rangle \Downarrow v}$$

Figure 2: Operational Semantics of the Source Language

$$(\Sigma, l) \models (\hat{C}, \hat{\rho}) \Leftrightarrow$$

[const] $\Sigma(l) = c \Rightarrow \mathit{true}$

[var] $\Sigma(l) = x \Rightarrow \hat{\rho}(x) \subseteq \hat{C}(l)$

[fn] $\Sigma(l) = (\mathtt{fn}\, x_1 \ldots x_n \Rightarrow t^{l'}) \Rightarrow \begin{cases} (\Sigma, l') \models (\hat{C}, \hat{\rho}) \\ l \in \hat{C}(l) \end{cases}$

[fun] $\Sigma(l) = (\mathtt{fun}\, y\, x_1 \ldots x_n \Rightarrow t^{l'}) \Rightarrow \begin{cases} (\Sigma, l') \models (\hat{C}, \hat{\rho}) \\ l \in \hat{C}(l) \\ l \in \hat{\rho}(y) \end{cases}$

[prim] $\Sigma(l) = (g^{l_0}\, t_1^{l_1} \ldots t_n^{l_n}) \Rightarrow (\Sigma, l_i) \models (\hat{C}, \hat{\rho}) \quad i = 1, \ldots, n$

[app] $\Sigma(l) = (t_0^{l_0}\, t_1^{l_1} \ldots t_n^{l_n}) \Rightarrow \begin{cases} (\Sigma, l_i) \models (\hat{C}, \hat{\rho}) \quad i = 0, \ldots, n \\ \forall l' : \begin{cases} (\Sigma(l') = (\mathtt{fn}\, x_1 \ldots x_n \Rightarrow t_f^{l_f}) \vee \\ \Sigma(l') = (\mathtt{fun}\, y\, x_1 \ldots x_n \Rightarrow t_f^{l_f})) \Rightarrow \\ \quad \begin{cases} l' \in \hat{C}(l_0) \Rightarrow \hat{C}(l_i) \subseteq \hat{\rho}(x_i) \quad i = 1, \ldots, n \\ l' \in \hat{C}(l_0) \Rightarrow \hat{C}(l_f) \subseteq \hat{C}(l)) \end{cases} \end{cases} \end{cases}$

[if] $\Sigma(l) = (\mathtt{if}\ t_0^{l_0}\ \mathtt{then}\ t_1^{l_1}\ \mathtt{else}\ t_2^{l_2}) \Rightarrow \begin{cases} (\Sigma, l_i) \models (\hat{C}, \hat{\rho}) \quad i = 1, 2, 3 \\ \hat{C}(l_1) \subseteq \hat{C}(l) \\ \hat{C}(l_2) \subseteq \hat{C}(l) \end{cases}$

[let] $\Sigma(l) = (\mathtt{let}\ x = t_1^{l_1}\ \mathtt{in}\ t_2^{l_2}) \Rightarrow \begin{cases} (\Sigma, l_1) \models (\hat{C}, \hat{\rho}) \\ (\Sigma, l_2) \models (\hat{C}, \hat{\rho}) \\ \hat{C}(l_1) \subseteq \hat{\rho}(x) \\ \hat{C}(l_2) \subseteq \hat{C}(l) \end{cases}$

Figure 3: Control Flow Analysis

such that if $\Sigma(l) = t$ then $\mathcal{D}(l) \subseteq FV(t)$. The semantics of a dependency analysis is specified as a judgment $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. This represents the judgment that the propositions $\hat{C}$, $\hat{\rho}$, and $\mathcal{D}$ are true at label l of universe Σ. The semantics of these propositions is given by the rules of Figure 4. Like the rules of Figure 3, these rules generate a finite set of constraints for any label l of universe Σ.

The rules of Figure 4 are fairly intuitive: For $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ to hold, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ must hold, as defined in Figure 3. In addition:

- A variable always depends on itself.
- An abstraction depends on all the variables on which its body depends, less its bound variables.
- An application of a primitive depends on all the variables that any of its arguments depends on.
- For a procedure application, we consider all of the closures that can flow to that site. For each i, if the body of any such closure depends on its i-th argument, then the entire expression depends on all the variables on which the i-th actual parameter depends.

Notice that this definition is inductive on $\Sigma(l)$, so for any l it generates a finite set of constraints. Furthermore, the constraints are all monotonic in $\mathcal{D}$, so we can solve the constraints using the standard closure algorithms [PS94].

In order to state a correctness property for this analysis, we extend the definition to environments, configurations, and values as follows:

$$\begin{aligned} (\Sigma, \rho) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \ &\Leftrightarrow\ \begin{cases} (\Sigma, \rho) \models (\hat{C}, \hat{\rho}) \wedge \\ \forall x \in \mathit{dom}(\rho), \\ \quad (\Sigma, \rho(x)) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \end{cases} \\ (\Sigma, \langle l,\ \rho \rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \ &\Leftrightarrow\ \begin{cases} (\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \wedge \\ (\Sigma, \rho) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \end{cases} \\ (\Sigma, c) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \ &\Leftrightarrow\ \text{true} \end{aligned}$$

It is then easy to prove that any dependency analysis satisfying these constraints is preserved under evaluation:

Lemma 5.1 *If* $(\Sigma, \langle l,\ \rho \rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *and* $\Sigma \vdash \langle l,\ \rho \rangle \Downarrow \langle l',\ \rho' \rangle$, *then* $l' \in \hat{C}(l)$ *and* $(\Sigma, \langle l',\ \rho' \rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$.

In order to express why variables not in $\mathcal{D}(l)$ are useless for l, we define an equivalence relation on configurations that says that they are equal except perhaps for useless variables:

$$(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \;\Leftrightarrow\; (\Sigma, l) \models (\hat{C}, \hat{\rho}) \wedge \mathcal{D}(l) \subseteq FV(\Sigma(l)) \wedge$$

$$\begin{array}{lcl}
\Sigma(l) = c & \Rightarrow & \text{always} \\[6pt]
\Sigma(l) = x & \Rightarrow & x \in \mathcal{D}(l) \\[6pt]
\Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_0^{l_0}) & \Rightarrow & \begin{cases} (\Sigma, l_0) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ \mathcal{D}(l_0) - \{x_1, \ldots, x_n\} \subseteq \mathcal{D}(l) \end{cases} \\[6pt]
\Sigma(l) = (\texttt{fun}\, y\, x_1 \ldots x_n \Rightarrow t_0^{l_0}) & \Rightarrow & \begin{cases} (\Sigma, l_0) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ \mathcal{D}(l_0) - \{y, x_1, \ldots, x_n\} \subseteq \mathcal{D}(l) \end{cases} \\[6pt]
\Sigma(l) = (g^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) & \Rightarrow & i = 1, \ldots, n : \begin{cases} (\Sigma, l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ \mathcal{D}(l_i) \subseteq \mathcal{D}(l) \end{cases}
\end{array}$$

$$\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})^l \quad \Rightarrow$$

$$\begin{cases} (\Sigma, l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \quad i = 0, \ldots, n \\ \mathcal{D}(l_0) \subseteq \mathcal{D}(l) \\ \forall l' : \ (\Sigma(l') = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) \vee \Sigma(l') = (\texttt{fun}\, x\ x_1 \ldots x_n \Rightarrow t_b^{l_b})) \Rightarrow \\ \qquad l' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_b) \Rightarrow \mathcal{D}(l_i) \subseteq \mathcal{D}(l) \quad i = 1, \ldots, n \end{cases}$$

$$\begin{array}{lcl}
\Sigma(l) = (\texttt{if}\ t_0^{l_0}\ \texttt{then}\ t_1^{l_1}\ \texttt{else}\ t_2^{l_2}) & \Rightarrow & \begin{cases} (\Sigma, l_0) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ (\Sigma, l_1) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ (\Sigma, l_2) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ \mathcal{D}(l_0) \cup \mathcal{D}(l_1) \cup \mathcal{D}(l_2) \subseteq \mathcal{D}(l) \end{cases} \\[6pt]
\Sigma(l) = (\texttt{let}\ x = t_1^{l_1}\ \texttt{in}\ t_2^{l_2}) & \Rightarrow & \begin{cases} (\Sigma, l_1) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ (\Sigma, l_2) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ \mathcal{D}(l_2) - \{x\} \subseteq \mathcal{D}(l) \\ x \in \mathcal{D}(l_2) \Rightarrow \mathcal{D}(l_1) \subseteq \mathcal{D}(l) \end{cases}
\end{array}$$

Figure 4: Dependency Analysis

$$\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})^l \Rightarrow$$

$$\begin{cases} (\Sigma, l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \quad i = 0, \ldots, n \\ \mathcal{D}(l_0) \subseteq \mathcal{D}(l) \\ \forall l' : \ (\Sigma(l') = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) \vee \ \Sigma(l') = (\texttt{fun}\, x\ x_1 \ldots x_n \Rightarrow t_b^{l_b})) \Rightarrow \\ \qquad \begin{cases} l' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_b) \Rightarrow \mathcal{D}(l_i) \subseteq \mathcal{D}(l) \quad i = 1, \ldots, n \\ \boxed{\begin{array}{l} \forall l'' : \ (\Sigma(l'') = (\texttt{fn}\, y_1 \ldots y_n \Rightarrow t_c^{l_c}) \vee \ \Sigma(l'') = (\texttt{fun}\, y\ y_1 \ldots y_n \Rightarrow t_c^{l_c})) \Rightarrow \\ \qquad l', l'' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_b) \Rightarrow y_i \in \mathcal{D}(l_c) \quad i = 1, \ldots, n \quad \wedge \end{array}} \end{cases} \end{cases}$$

$$(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \;\Leftrightarrow\; (\Sigma, l) \models (\hat{C}, \hat{\rho}) \wedge \begin{cases} \Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}),\ \mathcal{D}(l) \subseteq FV(\Sigma(l)) \cup \{x_1, \ldots, x_n\} \\ \Sigma(l) = (\texttt{fun}\, y\ x_1 \ldots x_n \Rightarrow t_b^{l_b}),\ \mathcal{D}(l) \subseteq FV(\Sigma(l)) \cup \{x_1, \ldots, x_n\} \\ \text{otherwise}\ ,\ \mathcal{D}(l) \subseteq FV(\Sigma(l)) \end{cases}$$

Figure 5: Extension to the Dependency Analysis

Definition 5.1 (Equality up to useless variables)

$$
\begin{array}{lll}
\mathcal{D} \models \rho \cong_l \rho' & \Leftrightarrow & \forall x \in \mathcal{D}(l),\ \mathcal{D} \models \rho(x) \cong \rho'(x) \\
\mathcal{D} \models \langle l,\ \rho\rangle \cong \langle l',\ \rho'\rangle & \Leftrightarrow & \begin{cases} l = l' \\ \mathcal{D} \models \rho \cong_l \rho' \end{cases} \\
\mathcal{D} \models c \cong c' & \Leftrightarrow & c = c'
\end{array}
$$

Note that this definition is by structural induction on configurations, and is independent of Σ.

Now we can prove the correctness of the analysis, by showing that configurations that differ only in useless variables produce results that differ only in useless variables:

Theorem 5.1 (Correctness of Dependency Analysis) *If* $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$, $\mathcal{D} \models C \cong C'$, $\Sigma \vdash C \Downarrow v$, *and* $\Sigma \vdash C' \Downarrow v'$, *then* $\mathcal{D} \models v \cong v'$.

Proof: By induction on the structure of $\Sigma \vdash C \Downarrow v$.

In particular, when v is a constant, then v' must be the same constant:

Corollary 5.1 *If* $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$, $\mathcal{D} \models C \cong C'$, $\Sigma \vdash C \Downarrow c$, *and* $\Sigma \vdash C' \Downarrow c'$, *then* $c = c'$

These results sharpen the usual results that the result of a computation depends only on its free variables.

Note that two configurations that differ only in useless variables may differ in their termination behavior, much like the example in Section 9.

6 Extending the Analysis to Support a Transformation

The analysis above, while correct, is not sufficient to support the desired transformation, because it does not guarantee that two closures flowing into a site will depend on the same set of arguments. Worse yet, one could have all the closures flowing into one site depending on the same set of arguments, but one of those closures could flow into another site where it met another closure that depended on a different set of arguments. For example, consider

```
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,y), h(x,y))
```

Here both the possible values of `g` depend only on their first argument. This suggests that we could rewrite `f1` and `f2` to take only one argument and remove the `y` from the call to `g`. However, `h` may depend on either of its arguments, so we cannot remove any arguments from the call to `h`. Since `f1` flows into `h`, we cannot remove any arguments from `f1`.[1] And since `f1` and `f2` flow together into `g`, we cannot remove any arguments from `f2` either.

[1] At least not without duplicating code, which would lead to a different, more complex transformation.

Merely analyzing dependencies does not shed any light on these difficulties, despite the correctness of the dependency analysis. To handle these problems, we need to extend the analysis to enforce these conditions.

Luckily, the remedy is fairly simple. For each closure, we can determine the argument positions on which it depends:

$$
\begin{array}{rl}
\mathit{DFormals}_{(\mathcal{D},\Sigma)}(l') = & \{j \mid (\Sigma(l') = (\mathtt{fn}\ x_1 \ldots x_n \Rightarrow t_b^{l_b}) \vee \\
& \quad \Sigma(l') = (\mathtt{fun}\ y\ x_1 \ldots x_n \Rightarrow t_b^{l_b})) \\
& \quad \wedge x_j \in \mathcal{D}(l_b)\}
\end{array}
$$

We want to guarantee for that for any l appearing in the operator position of an application, any two closures flowing into l depend on exactly the same arguments:

$$l', l'' \in \hat{C}(l) \Rightarrow \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l') = \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l'')$$

This can be ensured by adding extra entries to $\mathcal{D}$: If the body of *any* closure flowing into a site depends on its i-th parameter, then we add extra entries to $\mathcal{D}$ to declare that the bodies of *all* closures flowing into that site will also depend on their i-th parameters. This can be done by changing the rule for applications in the dependency analysis of Figure 4 to the one shown in figure 5.

Here the new constraints are in the box: we quantify over l'' and add the feedback constraint

$$l', l'' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_b) \Rightarrow y_i \in \mathcal{D}(l_c)$$

that forces each the sets $\mathit{DFormals}_{(\mathcal{D},\Sigma)}(l')$ and $\mathit{DFormals}_{(\mathcal{D},\Sigma)}(l'')$ to be equal. The last constraint forces us to add a special case to the condition $\mathcal{D}(l) \subseteq FV(\Sigma(l))$ because y_i is not necessarily free in $\Sigma(l_c)$. We add the new constraint:

$$
\begin{array}{l}
(\Sigma(l) = (\mathtt{fn}\ x_1 \ldots x_n \Rightarrow t_b^{l_b}) \vee \\
\ \Sigma(l) = (\mathtt{fun}\ y\ x_1 \ldots x_n \Rightarrow t_b^{l_b})) \\
\quad \Rightarrow \mathcal{D}(l) \subseteq FV(\Sigma(l)) \cup \{x_1, \ldots, x_n\}
\end{array}
$$

The dependency analysis satisfying the new constraints is also preserved under evaluation:

Lemma 6.1 *If* $(\Sigma, \langle l,\ \rho\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *and* $\Sigma \vdash \langle l,\ \rho\rangle \Downarrow \langle l',\ \rho'\rangle$, *then* $l' \in \hat{C}(l)$ *and* $(\Sigma, \langle l',\ \rho'\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$.

where $\models$ now refers to the definition as modified by Figure 5.

7 The Transformation

Now we are in a position to transform the program. The transformation takes place in three positions:

- In each application: Eliminate all the expressions that appear as arguments to parameters that are useless in the procedures called from this site. By the preceding constraints, these will be the same for all procedures that can reach this site.
- In each functional term: Eliminate all the arguments from the argument list that are useless in the body of the function, and replace `fun` by `fn` when the local name y is useless in the body.

- In each `let` term: The `let` term is replaced by its body if the let-variable is not part of the dependency set of the body term.

To determine which actual parameters should be retained at an application site, we define the set

$$\begin{array}{ll} DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) & = DFormals_{(\mathcal{D},\Sigma)}(l') \\ & \text{where } \Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) \wedge l' \in \hat{C}(l_0) \end{array}$$

where in the new analysis, the right-hand side is independent of the choice of l'. If there is no such l', we define the set to be empty, since $\hat{C}(l_0) = \emptyset$ implies that $t_0^{l_0}$ never returns a closure.

Given $\mathcal{A} = (\hat{C}, \hat{\rho}, \Sigma, \mathcal{D})$, the transformation $\mathcal{T}_{\mathcal{A}}(-)$ takes a label and produces a labeled term as described by the rules in figure 6,where in the last line for **let**, the notation $\mathcal{T}_{\mathcal{A}}(l_2)^l$ means an expression the same as $\mathcal{T}_{\mathcal{A}}(l_2)$ except that it is labeled with l.

As usual, in order to express the correctness of the transformation, we extend it to operate on configurations, values, and universes:

$$\begin{array}{lcl} \mathcal{T}_{\mathcal{A}}(\langle l,\ \rho\rangle) & = & \langle l,\ \{\mathcal{T}_{\mathcal{A}}(\rho(x)) \mid x \in \mathcal{D}(l)\}\rangle \\ \mathcal{T}_{\mathcal{A}}(c) & = & c \\ \mathcal{T}_{\mathcal{A}}(\Sigma) & = & \{\mathcal{T}_{\mathcal{A}}(l) \mid l \in \mathrm{dom}(\Sigma)\} \end{array}$$

The decision about removing a formal parameter from an abstraction is made separately from decisions about removing occurrences of terms containing that variable. Hence there is a danger that some variable might be removed from a formal parameter list while some occurrence of that variable still remained in the transformed body, resulting in an unbound variable. It would be easy to add more constraints to the analysis to prevent this, but luckily that is not necessary: we show that the transformation creates no new free variables.

Lemma 7.1 *If* $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *then* $FV(\mathcal{T}_{\mathcal{A}}(l)) \subseteq \mathcal{D}(l)$ *and* $FV(\mathcal{T}_{\mathcal{A}}(l)) \subseteq FV(\Sigma(l))$.

Proof: By induction on the size of l in Σ. We prove first that $FV(\mathcal{T}_{\mathcal{A}}(l)) \subseteq \mathcal{D}(l)$ by showing the abstraction and application cases:

[fn] $\quad \Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b})$

The transformed term is

$$\begin{array}{l} \mathcal{T}_{\mathcal{A}}(l) = (\texttt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b))^l, \\ \quad i_j \in DFormals_{(\mathcal{D},\Sigma)}(l) \end{array}$$

The constraints for $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ are:

$$\begin{array}{l} (\Sigma, l_b) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \wedge \\ \mathcal{D}(l_b) - \{x_1, \ldots, x_n\} \subseteq \mathcal{D}(l) \end{array}$$

Hence we can apply the inductive hypothesis to l_b and derive

$$FV(\mathcal{T}_{\mathcal{A}}(l_b)) \subseteq \mathcal{D}(l_b)$$

By applying the definition of free variables we get

$$\begin{array}{l} FV(\mathcal{T}_{\mathcal{A}}(l)) \\ = FV(\mathcal{T}_{\mathcal{A}}(l_b)) - \{x_{i_1}, \ldots, x_{i_m}\} \\ \subseteq \mathcal{D}(l_b) - \{x_{i_1}, \ldots, x_{i_m}\} \end{array}$$

If $x_i \notin DFormals_{(\mathcal{D},\Sigma)}(l)$ then $x_i \notin \mathcal{D}(l_b)$. This means that

$$\mathcal{D}(l_b) - \{x_{i_1}, \ldots, x_{i_m}\} = \mathcal{D}(l_b) - \{x_1, \ldots, x_n\}$$

We finally have

$$FV(\mathcal{T}_{\mathcal{A}}(l)) \subseteq \mathcal{D}(l_b) - \{x_1, \ldots, x_n\} \subseteq \mathcal{D}(l)$$

[app] $\quad \Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})$

The transformed term is:

$$\begin{array}{l} \mathcal{T}_{\mathcal{A}}(l) = (\mathcal{T}_{\mathcal{A}}(l_0)\ \mathcal{T}_{\mathcal{A}}(l_{i_1}) \ldots \mathcal{T}_{\mathcal{A}}(l_{i_m})) \\ \quad i_j \in DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) \end{array}$$

Let $l' \in \hat{C}(l_0)$ and $\Sigma(l') = (\texttt{fn}\ x_1 \ldots x_n \Rightarrow t_b^{l_b})$. This means that

$$\begin{array}{l} DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) = DFormals_{(\mathcal{D},\Sigma)}(l') \wedge \\ x_{i_j} \in \mathcal{D}(l_b) \end{array}$$

By applying the definition of the dependency analysis we can derive

$$\begin{array}{l} \mathcal{D}(l_0) \subseteq \mathcal{D}(l) \\ \mathcal{D}(l_{i_j}) \subseteq \mathcal{D}(l) \quad i_j \in DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) \end{array}$$

Hence we can apply the inductive hypothesis and derive

$$\begin{array}{l} FV(\mathcal{T}_{\mathcal{A}}(l_0)) \subseteq \mathcal{D}(l_0) \\ FV(\mathcal{T}_{\mathcal{A}}(l_{i_j})) \subseteq \mathcal{D}(l_{i_j}) \quad i_j \in DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) \end{array}$$

By applying the definition of free variables we get

$$\begin{array}{l} FV(\mathcal{T}_{\mathcal{A}}(l)) \\ = FV(\mathcal{T}_{\mathcal{A}}(l_0)) \cup FV(\mathcal{T}_{\mathcal{A}}(l_{i_1})) \cup \ldots \cup FV(\mathcal{T}_{\mathcal{A}}(l_{i_m})) \\ \subseteq \mathcal{D}(l_0) \cup \mathcal{D}(l_{i_1}) \cup \ldots \cup \mathcal{D}(l_{i_m}) \\ \subseteq \mathcal{D}(l) \cup \mathcal{D}(l) \cup \ldots \cup \mathcal{D}(l) \\ = \mathcal{D}(l) \end{array}$$

The second part of the lemma is trivial except for the abstraction case:

[fn] $\quad \Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b})$

The transformed term is:

$$\begin{array}{l} \mathcal{T}_{\mathcal{A}}(l) = (\texttt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b))^l, \\ \quad i_j \in DFormals_{(\mathcal{D},\Sigma)}(l) \end{array}$$

Using the definition of $DFormals_{(\mathcal{D},\Sigma)}$ and the first part of the lemma we have

$$\begin{array}{l} i_j \in DFormals_{(\mathcal{D},\Sigma)}(l) \Leftrightarrow x_{i_j} \in \mathcal{D}(l_b) \cap \{x_1, \ldots, x_n\} \\ FV(\mathcal{T}_{\mathcal{A}}(l_b)) \subseteq \mathcal{D}(l_b) \end{array}$$

If $x_i \notin DFormals_{(\mathcal{D},\Sigma)}(l)$ then $x_i \notin \mathcal{D}(l_b)$ and $x_i \notin FV(\mathcal{T}_{\mathcal{A}}(l_b))$. Then we can derive

$$\begin{array}{l} FV(\mathcal{T}_{\mathcal{A}}(l_b)) - \{x_{i_1}, \ldots, x_{i_m}\} \\ \quad = FV(\mathcal{T}_{\mathcal{A}}(l_b)) - \{x_1, \ldots, x_n\} \end{array}$$

By applying the definition of free variables we get

$$\begin{array}{ll} FV(\mathcal{T}_{\mathcal{A}}(l)) & \\ = FV(\mathcal{T}_{\mathcal{A}}(l_b)) - \{x_{i_1}, \ldots, x_{i_m}\} & i_j \in DFormals_{(\mathcal{D},\Sigma)}(l) \\ = FV(\mathcal{T}_{\mathcal{A}}(l_b)) - \{x_1, \ldots, x_n\} & \\ \subseteq FV(\Sigma(l_b)) - \{x_1, \ldots, x_n\} & \text{Ind. Hypothesis} \\ = FV(\Sigma(l)) & \end{array}$$

$$
\begin{array}{lll}
\Sigma(l) = c & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = c^l \\
\Sigma(l) = x & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = x^l \\
\Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = (\texttt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b))^l, \\
 & & \quad i_j \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l) \\
\Sigma(l) = (\texttt{fun}\, y\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) & \Rightarrow &
\end{array}
$$

$$
\mathcal{T}_{\mathcal{A}}(l) = \begin{cases} (\texttt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b))^l & \text{if } y \notin \mathcal{D}(l_b) \\ (\texttt{fun}\, y\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b))^l & \text{otherwise} \end{cases}
\quad i_j \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l)
$$

$$
\begin{array}{lll}
\Sigma(l) = (g^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = (g^{l_0}\ \ \mathcal{T}_{\mathcal{A}}(l_1)\ \ \ldots\ \ \mathcal{T}_{\mathcal{A}}(l_n))^l \\
\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = (\mathcal{T}_{\mathcal{A}}(l_0)\ \ \mathcal{T}_{\mathcal{A}}(l_{i_1})\ \ \ldots\ \ \mathcal{T}_{\mathcal{A}}(l_{i_m}))^l, \\
 & & \quad i_j \in \mathit{DActuals}_{(\hat{C},\mathcal{D},\Sigma)}(l) \\
\Sigma(l) = (\texttt{if}\ t_0^{l_0}\ \texttt{then}\ t_1^{l_1}\ \texttt{else}\ t_2^{l_2}) & \Rightarrow & \mathcal{T}_{\mathcal{A}}(l) = (\texttt{if}\ \mathcal{T}_{\mathcal{A}}(l_0)\ \texttt{then}\ \mathcal{T}_{\mathcal{A}}(l_1)\ \texttt{else}\ \mathcal{T}_{\mathcal{A}}(l_2))^l \\
\Sigma(l) = (\texttt{let}\ x = t_1^{l_1}\ \texttt{in}\ t_2^{l_2}) & \Rightarrow &
\end{array}
$$

$$
\mathcal{T}_{\mathcal{A}}(l) = \begin{cases} (\texttt{let}\ x = \mathcal{T}_{\mathcal{A}}(l_1)\ \texttt{in}\ \mathcal{T}_{\mathcal{A}}(l_2))^l & \text{if } x \in \mathcal{D}(l_2) \\ \mathcal{T}_{\mathcal{A}}(l_2)^l & \text{otherwise} \end{cases}
$$

Figure 6: The Transformation

[**QED**]

Now we can state our main theorem: the correctness of the transformation. The theorem says that given an evaluation of some configuration to some value, we can construct an evaluation of the transformed configuration to the transformed value.

Theorem 7.1 (Correctness of the Transformation) *If* $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$*,* $\Sigma \vdash C \Downarrow v$ *and* $\mathcal{D} \models C \cong C'$*, then* $\exists v'$ *s.t.* $\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash C' \Downarrow v'$ *and* $\mathcal{D} \models v \cong v'$*.*

Proof: By tree surgery: given a derivation of $\Sigma \vdash C \Downarrow v$, we recursively construct a derivation of $\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash C' \Downarrow v'$. The key step, corresponding to subject reduction in a small-step semantics, is that if $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ holds for the conclusion $\Sigma \vdash C \Downarrow v$ of some rule, then $(\Sigma, C'') \models (\hat{C}, \hat{\rho}, \mathcal{D})$ holds for any configuration C'' that appears on the left-hand side of any antecedent of the rule.[2]

We show the application case:

$$
[\mathbf{app_{fn}}]\quad \frac{\begin{array}{l} \Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n}) \wedge \\ \Sigma \vdash \langle l_0,\ \rho\rangle \Downarrow \langle l_f,\ \rho_0\rangle \wedge \\ \Sigma \vdash \langle l_i,\ \rho\rangle \Downarrow v_i \quad i = 1, \ldots, n \wedge \\ \Sigma(l_f) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) \wedge \\ \Sigma \vdash \langle l_b,\ \rho_0[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]\rangle \Downarrow v \end{array}}{\Sigma \vdash \langle l,\ \rho\rangle \Downarrow v}
$$

[2] This is of course an old idea, going back at least as far as [MW77].

In the transformed program, the corresponding evaluation tree has the subtrees corresponding to useless parameters removed, the remaining subtrees recursively transformed, and the environments of the various closures trimmed accordingly.

To make this precise, let $C = \langle l,\ \rho\rangle$ and $C' = \langle l,\ \rho'\rangle$ where $\mathcal{D} \models \rho \cong_l \rho'$. By applying the definition of the dependency analysis and lemma 6.1 we can derive

$$
\begin{array}{l}
(\Sigma, \langle l_0,\ \rho\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\
(\Sigma, \langle l_f,\ \rho_0\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \wedge l_f \in \hat{C}(l_0) \\
i = 1, \ldots, n \quad \begin{cases} (\Sigma, \langle l_i,\ \rho\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ v_i = \langle l_{v_i},\ \rho_i\rangle \Rightarrow \begin{cases} (\Sigma, v_i) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \\ l_{v_i} \in \hat{C}(l_i) \end{cases} \end{cases}
\end{array}
$$

Since $l_f \in \hat{C}(l_0)$ we know that

$$
\mathit{DActuals}_{(\hat{C},\mathcal{D},\Sigma)}(l) = \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f)
$$

so the transformed terms are

$$
\begin{array}{l}
\mathcal{T}_{\mathcal{A}}(l) = (\mathcal{T}_{\mathcal{A}}(l_0)\ \mathcal{T}_{\mathcal{A}}(l_{i_1}) \ldots \mathcal{T}_{\mathcal{A}}(l_{i_m})) \\
\mathcal{T}_{\mathcal{A}}(l_f) = (\texttt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b)) \\
\quad i_j \in \mathit{DActuals}_{(\hat{C},\mathcal{D},\Sigma)}(l) = \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f)
\end{array}
$$

The definition of the analysis says that $\mathcal{D}(l_0) \subseteq \mathcal{D}(l)$ and $\mathcal{D}(l_i) \subseteq \mathcal{D}(l)$, if $x_i \in \mathcal{D}(l_b)$. Now, since $\mathcal{D} \models \rho \cong_l \rho'$, if $\mathcal{D}(l_i) \subseteq \mathcal{D}(l)$, then $\mathcal{D} \models \rho \cong_{l_i} \rho'$. Hence

$$
\mathcal{D}(l_i) \subseteq \mathcal{D}(l) \Rightarrow \mathcal{D} \models \langle l_i,\ \rho\rangle \cong \langle l_i,\ \rho'\rangle
$$

Therefore we can apply the inductive hypothesis to $\langle l_0, \rho\rangle$ and to $\langle l_i,\, \rho\rangle$ when $x_i \in \mathcal{D}(l_b)$, getting

$$\begin{array}{l}
\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_0,\, \rho'\rangle \Downarrow \langle l_f,\, \rho_0'\rangle \\
\mathcal{D} \models \langle l_f,\, \rho_0\rangle \cong \langle l_f,\, \rho_0'\rangle \\
x_i \in \mathcal{D}(l_b) \Rightarrow \begin{cases} \mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_i,\, \rho'\rangle \Downarrow v_i' \\ \mathcal{D} \models v_i \cong v_i' \end{cases} \\
\quad i = 1, \ldots, n
\end{array}$$

Let

$$\begin{array}{l}
\rho_1 = \rho_0[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \\
\rho_1' = \rho_0'[x_{i_1} \mapsto v_{i_1}'] \ldots [x_{i_n} \mapsto v_{i_m}'] \\
\quad i_j \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f)
\end{array}$$

We know

$$\begin{array}{l}
\mathcal{D} \models \rho_0 \cong_{l_f} \rho_0' \\
\mathcal{D} \models v_{i_j} \cong v_{i_j}' \quad i_j \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f) \\
i \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f) \Leftrightarrow x_i \in \mathcal{D}(l_b) \cap \{x_1, \ldots, x_n\} \\
\mathcal{D}(l_b) - \{x_1, \ldots, x_n\} \subseteq \mathcal{D}(l_f)
\end{array}$$

We need to prove $\mathcal{D} \models \rho_1 \cong_{l_b} \rho_1'$. Consider $x \in \mathcal{D}(l_b)$:

- if $x \notin \{x_1, \ldots, x_n\}$
  Then $\rho_1(x) = \rho_0(x)$, $\rho_1'(x) = \rho_0'(x)$, and $x \in \mathcal{D}(l_f)$, which means $\mathcal{D} \models \rho_0(x) \cong \rho_0'(x)$. Therefore $\mathcal{D} \models \rho_1(x) \cong \rho_1'(x)$.
- if $x \in \{x_1, \ldots, x_n\}$
  Let $x = x_i$, then $i \in \mathit{DFormals}_{(\mathcal{D},\Sigma)}(l_f)$ and

$$\begin{array}{l}
\rho_1(x) = v_i \\
\rho_1'(x) = v_i' \\
\mathcal{D} \models v_i \cong v_i'
\end{array}$$

  So $\mathcal{D} \models \rho_1(x) \cong \rho_1'(x)$.

Therefore $\mathcal{D} \models \rho_1 \cong_{l_b} \rho_1'$, as claimed, and $\mathcal{D} \models \langle l_b,\, \rho_1\rangle \cong \langle l_b,\, \rho_1'\rangle$. We know that $(\Sigma, \langle l_f,\, \rho_0\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ and $(\Sigma, v_i) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. From the control-flow analysis, we also know that $v_i = \langle l_{v_i},\, \rho_i\rangle$, $l_{v_i} \in \hat{C}(l_i)$, and $\hat{C}(l_i) \subseteq \hat{\rho}(x_i)$. From these we can conclude that $(\Sigma, \langle l_b,\, \rho_1\rangle) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. Then we can apply the inductive hypothesis at $\langle l_b,\, \rho_1\rangle$ to derive

$$\begin{array}{l}
\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_b,\, \rho_1'\rangle \Downarrow v' \\
\mathcal{D} \models v \cong v'
\end{array}$$

We have

$$\begin{array}{l}
\mathcal{T}_{\mathcal{A}}(\Sigma)(l) = (\mathcal{T}_{\mathcal{A}}(l_0)\ \mathcal{T}_{\mathcal{A}}(l_{i_1}) \ldots \mathcal{T}_{\mathcal{A}}(l_{i_m})) \\
\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_{i_j},\, \rho'\rangle \Downarrow v_{i_j}' \\
\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_0,\, \rho'\rangle \Downarrow \langle l_f,\, \rho_0'\rangle \wedge \\
\mathcal{T}_{\mathcal{A}}(\Sigma)(l_f) = (\mathtt{fn}\, x_{i_1} \ldots x_{i_m} \Rightarrow \mathcal{T}_{\mathcal{A}}(l_b)) \wedge \\
\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l_b,\, \rho_1'\rangle \Downarrow v'
\end{array}$$

hence we can apply rule $[\mathbf{app_{fn}}]$ and derive $\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \langle l,\, \rho'\rangle \Downarrow v'$, where $\mathcal{D} \models v \cong v'$.

[QED]

Lemma 7.2 *If* $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *then* $\mathcal{D} \models C \cong \mathcal{T}_{\mathcal{A}}(C)$.

Corollary 7.1
If $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *and* $\Sigma \vdash C \Downarrow v$ *then* $\exists v'$ *s.t.* $\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \mathcal{T}_{\mathcal{A}}(C) \Downarrow v'$ *and* $\mathcal{D} \models v \cong v'$.

As usual, since $\mathcal{T}_{\mathcal{A}}(c) = c$, we deduce that if the original program evaluates to a constant, then the transformed program evaluates to the same constant:

Corollary 7.2
If $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *and* $\Sigma \vdash C \Downarrow c$, *then* $\mathcal{T}_{\mathcal{A}}(\Sigma) \vdash \mathcal{T}_{\mathcal{A}}(C) \Downarrow c$.

Furthermore, the flow analysis still holds for the transformed program. This is the counterpart in the untyped analysis world to the idea that transformations preserve type:

Theorem 7.2 (The Transformation preserves the analysis)
If $(\Sigma, C) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ *then* $(\mathcal{T}_{\mathcal{A}}(\Sigma), \mathcal{T}_{\mathcal{A}}(C)) \models (\hat{C}, \hat{\rho}, \mathcal{D})$.

This is a desirable property in a "compilation by transformation" approach, where a series of transformations are applied one after the other. Control flow information can be re-used without affecting the correctness of the transformation though each transformation may be working with "less precise" values.

8 Complexity

The analyses of Figures 3 and 4 are all straightforward constraint systems, generating constraints of the forms

$$x_1 \in X_1 \wedge \ldots \wedge x_n \in X_n \Rightarrow y \in Y$$

$$x_1 \in X_1 \wedge \ldots \wedge x_n \in X_n \Rightarrow X \subseteq Y$$

constraining the $O(n)$ sets $\hat{C}(l)$, $\hat{\rho}(x)$, and $\mathcal{D}(l)$. There are only $O(n^2)$ of these, since the number of argument constraints such as

$$l' \in \hat{C}(l_0) \Rightarrow \hat{C}(l_i) \subseteq \hat{\rho}(x_i) \quad i = 1, \ldots, n$$

is bounded by the number of actual parameters l_i in the program. Hence these can be solved by the standard method [PS94] in $O(n^3)$ time.

The only potential difficulty is the $O(n^3)$ constraints of the form

$$l', l'' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_b) \Rightarrow y_i \in \mathcal{D}(l_c)$$

But each of these has a consequent of the form $y \in Y$, which costs only constant time, rather than a consequent of the form $X \subseteq Y$, which would take $O(n)$ time. So the total cost of the analysis is still $O(n^3)$.

9 Adding Effects and Separate Compilation

This transformation does not preserve effects such as non-termination. Consider

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

This program will not terminate, of course, but the transformed version will eliminate the second argument, and hence it will terminate.

We can extend the analysis to preserve such effects, by using a separate analysis to detect such expressions. The dependency analysis can use the information provided by the effect analysis in this way: If the i-th actual parameter could not be proved to be free of the effect being preserved, then mark that position as useful and for each function that flows to the application, include its i-th formal parameter in the dependency set of its body. The new constraint will look like:

$$\begin{aligned}&\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})^l \Rightarrow \\ &\quad \forall l' : (\Sigma(l') = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) \vee \\ &\qquad \Sigma(l') = (\texttt{fun}\, y\, x_1 \ldots x_n \Rightarrow t_b^{l_b})) \Rightarrow \\ &\qquad l' \in \hat{C}(l_0) \wedge \neg(\mathit{Effect\text{-}Free}(l_i)) \Rightarrow x_i \in \mathcal{D}(l_b)\end{aligned}$$

In the case that external functions need to be considered *i.e.* references to a function in another module, we can extend the set of abstract values of the flow analysis to include a value *External*. We declare all the formal parameters of an external function as useful by adding the following constraint to the application case:

$$\begin{aligned}&\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})^l \ \wedge\ \mathit{External} \in \hat{C}(l_0) \\ &\quad \Rightarrow \mathcal{D}(l_i) \subseteq \mathcal{D}(l) \quad i = 1, \ldots, n\end{aligned}$$

and by extending the definition of $\mathit{DActuals}_{(\hat{C},\mathcal{D},\Sigma)}$ to:

$$\begin{aligned}&\Sigma(l) = (t_0^{l_0}\ t_1^{l_1} \ldots t_n^{l_n})^l \ \wedge\ \mathit{External} \in \hat{C}(l_0) \\ &\quad \Rightarrow \mathit{DActuals}_{(\hat{C},\mathcal{D},\Sigma)}(l) = \{1, \ldots, n\}\end{aligned}$$

Similarly, if a function escapes the program, we must assume that all its formal parameters are useful. Furthermore, its formal parameters could be bound to any function after it has escaped the program, and therefore we must declare them as *External* as well:

$$\begin{aligned}&\Sigma(l) = (\texttt{fn}\, x_1 \ldots x_n \Rightarrow t_b^{l_b}) \wedge \mathit{Escapes}(l) \\ &\quad \Rightarrow \begin{cases}\{x_1, \ldots, x_n\} \subseteq \mathcal{D}(l_b) \\ \mathit{External} \in \hat{\rho}(x_i) \quad i = 1, \ldots, n\end{cases}\end{aligned}$$

The rest of the analysis proceeds as before.

10 Conclusions

We have completed a proof of correctness of useless variable elimination. This transformation is the extension to higher-order programming of a variety of dead-code elimination optimizations that are important in compilers for first-order imperative languages.

Our proof highlights the difference between proving the soundness of an analysis and proving the correctness of a transformation. We were able to formulate and prove a very plausible soundness theorem for our analysis, but it turned out that this soundness property was essentially useless for the transformation: indeed, we had to strengthen the analysis itself in order to obtain a transformation.

The choice of semantics was crucial: the proof would have been much more difficult in a small-step framework, because it would have been quite difficult to formalize just when a particular reduction step in the original computation was relevant to the reduction of the transformed program. Because the big-step semantics is organized top-down, we can choose just which subtrees to explore. The usual advantage of small-step semantics in dealing with non-terminating computations [NN97] was not relevant because our transformation does not preserve non-termination.

The proof also introduces a few technical tricks that may be useful to others who wish to prove the correctness of analysis-based transformations. In particular, the use of a universe Σ of labeled terms provides a clean way of managing the link between labels and terms, and of re-using labels in different universes.

We hope to extend this work to consider transformations that duplicate or otherwise move code, and also to consider how our proofs change as we consider finer analyses, such as replacing 0CFA by k-CFA.

References

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the ACM '96 International Conference on Functional Programming*, pages 83–91, 1996.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Hei92] Nevin Heintze. *Set-Based Program Analysis.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1992.

[HRB84] Susan Horwitz, Thomas Reps, and David Binkley. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, August 1984.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and progrms with recursive data structures. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Jon81] Neil D. Jones. Flow analysis of lambda expressions. In *International Colloquium on Automata, Languages, and Programming*, 1981.

[Mic68] Donald Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[Muc97] Steven S. Muchnick. *Advanced Comiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[MW77] James H. Morris, Jr. and Ben Wegbreit. Subgoal induction. *Communications of the ACM*, 20:209–222, 1977.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings 24th Annual ACM Symposium on Principles of Programming Languages*, pages 332–345. ACM, January 1997.

[PJHH+93] Simon Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93.

[PJS98] S Peyton Jones and A Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* Wiley Professional Computing, Chichester, 1994.

[Ses88] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, October 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, May 1991.

[ST98] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 324–332. ACM, June 1998.

[SW97] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, pages 48–86, January 1997. Original version appeared in Proceedings 21st Annual ACM Symposium on Principles of Programming Languages, 1994.

[Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993. preliminary version appeared in *Conf. Rec. 20th ACM Symp. on Principles of Prog. Lang.* (1993), 137–143.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193. IEEE, April 1998.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, August 1984.