Control-flow analysis

Lecture 22 Tuesday, April 21, 2015

1 Dataflow analysis for functional programs

One way of thinking about types is that they are static approximations of the behavior of a program. For example, if expression e has type **int** $\to$ **int**, then we know statically (i.e., without executing the expression) that the execution of e will result in a function from integers to integers (provided the execution terminates).

There are in fact many kinds of *static analyses* that approximate the behavior of programs without actually executing them. Today we will consider *control flow analysis*, a program analysis that approximates the order that individual expressions or commands are evaluated.

In a first-order language like IMP (i.e., a language where functions are not values), control flow analysis is straightforward: the lexical structure of the program tells us the control flow structure. But in a functional language, if we see a application $f\ y$, then control will jump to the function body of whatever function the variable f evaluates to. This makes determining the control flow of a functional language more complex than determining the control flow structure of a language like IMP, where functions are not first-class values. (Similar issues arise in object-oriented languages, where objects are first class and contain executable code.)

We will consider the 0-CFA analysis. (*CFA* stands for Control-Flow Analysis; it is an instance of k-CFA analysis, where the parameter k determines the precision of the analysis.) Like our type inference analysis, we will generate a set of constraints from a program, and then find a solution to the set of constraints.

1.1 Labeled lambda calculus

Let's consider a lambda calculus with integers. The syntax of the language is similar to what we've seen before, but every expression in our source program will have a unique label. Labels are used to uniquely identify program expressions. We use l to range over labels.

$$e ::= n^l \mid x^l \mid (\lambda x.\, e)^l \mid (e_1\ e_2)^l \mid (e_1 + e_2)^l$$

Every expression has a label, and we write $\mathit{labelof}(e)$ for the label of expression e. For convenience, we also write e^l to mean that l is a label such that $\mathit{labelof}(e) = l$.

Also, given a program e, we write $\mathit{exprof}(e, l)$ for the (unique) subexpression e' of e such that $\mathit{labelof}(e') = l$. That is $\mathit{exprof}(e, l)$ allows us to get the expression associated with label l.

We will use a large-step environment semantics for this analysis. Environments ρ are maps from variables to values, and that judgment $\langle e, \rho\rangle \Downarrow v$ means that expression e in environment ρ evaluates to value v. Note that in an environment semantics, a function $\lambda x.\, e$ evaluates to a *closure* $((\lambda x.\, e)^l, \rho_{lex})$, where ρ_{lex} is the environment that was current when $\lambda x.\, e$ was evaluated. That is, ρ_{lex} binds the free variables (except x) in the function body e. We define the label of a closure to be the label of the function: $\mathit{labelof}(((\lambda x.\, e)^l, \rho_{lex})) = l$.

Values in our language are thus the following.

$$v ::= n^l \mid ((\lambda x.\, e)^l, \rho)$$

$$\frac{}{\langle x^l, \rho\rangle \Downarrow \rho(x)} \qquad \frac{}{\langle n^l, \rho\rangle \Downarrow n^l} \qquad \frac{\langle e_1, \rho\rangle \Downarrow n_1^{l_1} \quad \langle e_2, \rho\rangle \Downarrow n_2^{l_1}}{\langle (e_1 + e_2)^l, \rho\rangle \Downarrow n^l}\, n = n_1 + n_2$$

$$\frac{}{\langle (\lambda x.\, e)^l, \rho\rangle \Downarrow ((\lambda x.\, e)^l, \rho)} \qquad \frac{\langle e_1, \rho\rangle \Downarrow ((\lambda x.\, e)^{l_1}, \rho_{lex}) \quad \langle e_2, \rho\rangle \Downarrow v_2 \quad \langle e, \rho_{lex}[x \mapsto v_2]\rangle \Downarrow v}{\langle (e_1\ e_2)^l, \rho\rangle \Downarrow v}$$

Notice that every value is labeled with the label of the expression that created it. (For integer values, the label is either the label of the integer literal that appeared in the source program, or the label of an addition.)

1.2 Analysis

The aim of our analysis is to approximate the values that expressions can evaluate to. Since functions are values, this will also tell us about control flow, since given an application $e_1\ e_2$, knowing which function values expression e_1 may evaluate to tells us about the possible control flow of the function application.

Our analysis will find a function $\mathsf{C} : \textbf{Label} \to \mathcal{P}(\textbf{Label})$ that approximates for each label l the set of values that the expression labeled l may evaluate to. Specifically, if $l' \in \mathsf{C}(l)$, then the expression labeled l may evaluate to a value labeled l'.

Our analysis will also find a function $\mathsf{r} : \textbf{Var} \to \mathcal{P}(\textbf{Label})$ that for each variable x will approximate the set of values that x may be bound to. That is, if $l \in \mathsf{r}(x)$ then the variable x may be bound to a value labeled l. We assume without loss of generality that variable names in a program are unique. (If we don't have this assumption, the analysis will still work, it will just be less precise.)

Given a program e, we produce a set of constraints on functions C and r by examining the program. Intuitively, if we can find functions C and r that satisfy the constraints, then C and r will give us correct information about what values expressions and variables may evaluate to.

We generate the set of constraints using function $\mathcal{C}[\![\cdot]\!]_e : \textbf{Expr} \to \mathcal{P}(\textbf{Constraint})$. Here, e is the program we are analyzing, and we use it in order to map labels to expressions. $\mathcal{C}[\![\cdot]\!]_e$ is defined as follows.

$$\begin{aligned}
\mathcal{C}[\![n^l]\!]_e &= \{l \in \mathsf{C}(l)\} \\
\mathcal{C}[\![(e_1 + e_2)^l]\!]_e &= \mathcal{C}[\![e_1]\!]_e \cup \mathcal{C}[\![e_2]\!]_e \cup \{l \in \mathsf{C}(l)\} \\
\mathcal{C}[\![x^l]\!]_e &= \{\mathsf{r}(x) \subseteq \mathsf{C}(l)\} \\
\mathcal{C}[\![(\lambda x.\, e_1)^l]\!]_e &= \{l \in \mathsf{C}(l)\} \cup \mathcal{C}[\![e_1]\!]_e \\
\mathcal{C}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]_e &= \mathcal{C}[\![e_1^{l_1}]\!]_e \cup \mathcal{C}[\![e_2^{l_2}]\!]_e \\
&\quad \cup \{l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_2) \subseteq \mathsf{r}(x) \mid \mathit{exprof}(e, l') = (\lambda x.\, e_0^{l_0})^{l'}\} \\
&\quad \cup \{l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_0) \subseteq \mathsf{C}(l) \mid \mathit{exprof}(e, l') = (\lambda x.\, e_0^{l_0})^{l'}\}
\end{aligned}$$

Let's consider what each of these constraints mean. For values n^l, $(\lambda x.\, e_1)^l$ and addition $(e_1 + e_2)^l$ we have constraint $l \in \mathsf{C}(l)$. This means that the expression labeled l may evaluate to a value labeled l, and, if we look at the semantics, it is indeed the case.

$\mathcal{C}[\![x^l]\!]_e$ produces the constraint $\mathsf{r}(x) \subseteq \mathsf{C}(l)$, meaning that if x may be bound to a value labeled l' (i.e., $l' \in \mathsf{r}(x)$), then the expression x^l may evaluate to that value (i.e., $l' \in \mathsf{C}(l)$.

The constraints for application $(e_1^{l_1}\ e_2^{l_2})^l$ are most interesting. The *conditional constraint* $l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_2) \subseteq \mathsf{r}(x)$ requires that if expression e_1 may evaluate to function value $(\lambda x.\, e_0^{l_0})^{l'}$ (i.e., $l' \in \mathsf{C}(l_1)$), then x, the argument of that function, may be bound to anything that e_2 can evaluate to (i.e., $\mathsf{C}(l_2) \subseteq \mathsf{r}(x)$).

Similarly, constraint $l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_0) \subseteq \mathsf{C}(l)$ requires that if expression e_1 may evaluate to function value $(\lambda x.\, e_0^{l_0})^{l'}$ (i.e., $l' \in \mathsf{C}(l_1)$), then the application expression may evaluate to anything that function body e_0 may evaluate to (i.e., $\mathsf{C}(l_0) \subseteq \mathsf{C}(l)$).

Solving these constraints is straightforward. We will start off with a very bad approximation to the functions: $\mathsf{C}_0(l) = \emptyset$ for all labels l, and $\mathsf{r}_0(x) = \emptyset$ for all variables x. We will then iteratively improve these approximations by adding labels to the sets to satisfy the constraints. We will keep iterating until we reach a fixed point. Since there are only a finite number of labels, and in each iteration we only add labels to the sets, we are guaranteed to reach a fixed point.

$$
\begin{aligned}
\mathsf{C}_0 &= \lambda l.\, \emptyset \\
\mathsf{r}_0 &= \lambda x.\, \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_{i+1} &= \lambda l.\, \mathsf{C}_i(l) \\
&\quad \cup \{l \mid (l \in \mathsf{C}(l)) \in \mathcal{C}[\![e]\!]_e\} \\
&\quad \cup \bigcup \{\mathsf{r}_i(x) \mid (\mathsf{r}(x) \subseteq \mathsf{C}(l)) \in \mathcal{C}[\![e]\!]_e\} \\
&\quad \cup \bigcup \{\mathsf{C}_i(l_0) \mid (l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_0) \subseteq \mathsf{C}(l)) \in \mathcal{C}[\![e]\!]_e \text{ and } l' \in \mathsf{C}_i(l_1)\} \\
\mathsf{r}_{i+1} &= \lambda x.\, \mathsf{r}_i(x) \\
&\quad \cup \bigcup \{\mathsf{C}_i(l_2) \mid (l' \in \mathsf{C}(l_1) \Rightarrow \mathsf{C}(l_2) \subseteq \mathsf{r}(x)) \in \mathcal{C}[\![e]\!]_e \text{ and } l' \in \mathsf{C}_i(l_1)\}
\end{aligned}
$$

The least fixed point, which we will denote C^* and r^* is the bound of all of the approximations.

$$
\begin{aligned}
\mathsf{C}^* &= \lambda l.\, \bigcup_{i \in \mathbb{N}} \mathsf{C}_i(l) \\
\mathsf{r}^* &= \lambda x.\, \bigcup_{i \in \mathbb{N}} \mathsf{r}_i(x)
\end{aligned}
$$

1.3 Example

Let's work through a simple example. Consider the following program.

$$e \;\equiv\; (((\lambda a.\, a^1)^2\ (\lambda b.\, b^3)^4)^5\ 99^6)^7$$

The set of constraints for this simple program is as follows. (Exercise: make sure you understand how these constraints were derived.)

$$
\begin{aligned}
\mathcal{C}[\![e]\!]_e = \{\ & 2 \in \mathsf{C}(2), 4 \in \mathsf{C}(4), 6 \in \mathsf{C}(6), \\
& \mathsf{r}(a) \subseteq \mathsf{C}(1), \mathsf{r}(b) \subseteq \mathsf{C}(3), \\
& 2 \in \mathsf{C}(5) \Rightarrow \mathsf{C}(6) \subseteq \mathsf{r}(a), \\
& 4 \in \mathsf{C}(5) \Rightarrow \mathsf{C}(6) \subseteq \mathsf{r}(b), \\
& 2 \in \mathsf{C}(5) \Rightarrow \mathsf{C}(1) \subseteq \mathsf{C}(7), \\
& 4 \in \mathsf{C}(5) \Rightarrow \mathsf{C}(3) \subseteq \mathsf{C}(7), \\
& 2 \in \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(a), \\
& 4 \in \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(b), \\
& 2 \in \mathsf{C}(2) \Rightarrow \mathsf{C}(1) \subseteq \mathsf{C}(5), \\
& 4 \in \mathsf{C}(2) \Rightarrow \mathsf{C}(3) \subseteq \mathsf{C}(5)\ \}
\end{aligned}
$$

Let's consider the result of solving it iteratively. In the following table, columns indicate the values of $\mathsf{C}_i(l)$ and $\mathsf{r}_i(x)$ over the various iterations.

i	$\mathsf{C}_i(1)$	$\mathsf{C}_i(2)$	$\mathsf{C}_i(3)$	$\mathsf{C}_i(4)$	$\mathsf{C}_i(5)$	$\mathsf{C}_i(6)$	$\mathsf{C}_i(7)$	$\mathsf{r}_i(a)$	$\mathsf{r}_i(b)$
0									
1		2		4		6			
2		2		4		6		4	
3	4	2		4		6		4	
4	4	2		4	4	6		4	
5	4	2		4	4	6		4	6
6	4	2	6	4	4	6		4	6
7	4	2	6	4	4	6	6	4	6
8	4	2	6	4	4	6	6	4	6

At the 8th iteration, we have $\mathsf{C}_7 = \mathsf{C}_8$ and $\mathsf{r}_7 = \mathsf{r}_8$, and we have reached a fixed point, and so $\mathsf{C}^* = \mathsf{C}_8$ and $\mathsf{r}^* = \mathsf{r}_8$.

Let's double check that this analysis returned reasonable results. For example, $\mathsf{C}^*(5) = \{4\}$, meaning that the expression $((\lambda a.\, a^1)^2\ (\lambda b.\, b^3)^4)^5$ may evaluate it a value labeled 4, i.e., to the value $(\lambda b.\, b^3)^4$. That is indeed consistent with the actual execution of the program. Another example: $\mathsf{C}^*(7) = \{6\}$, meaning that the whole program may evaluate to a value labeled 6, i.e., to the labeled integer 99^6.

Note that $2 \notin \mathsf{C}^*(5)$. That is, the analysis correctly says that expression $((\lambda a.\, a^1)^2\ (\lambda b.\, b^3)^4)^5$ can not evaluate to $(\lambda a.\, a^1)^2$.

1.4 Soundness

The analysis is sound, meaning that, given a program e_0, if r^* and C^* satisfy the set of constraints $\mathcal{C}[\![e_0]\!]_{e_0}$, then C^* conservatively describes what expressions may evaluate to, and r^* conservatively describes what variables may be bound to.

Theorem (Soundness)**.** *Let e_0 be a program. Let r^* and C^* satisfy the set of constraints $\mathcal{C}[\![e_0]\!]_{e_0}$. If $\langle e_0, \emptyset\rangle \Downarrow v_0$ and $\langle e, \rho\rangle \Downarrow v$ appears in the derivation of $\langle e_0, \emptyset\rangle \Downarrow v_0$, then $\mathit{labelof}(v) \in \mathsf{C}^*(\mathit{labelof}(e))$.*

In order to prove the soundness theorem, we need a stronger lemma. To state the lemma, we will extend the set-constraint generation function to environments and closures.

$$\mathcal{C}[\![\rho]\!]_e = \bigcup_{x \in \mathrm{dom}(\rho)} \{\mathsf{C}(\mathit{labelof}(\rho(x))) \subseteq \mathsf{r}(x)\} \cup \mathcal{C}[\![\rho(x)]\!]_e$$

$$\mathcal{C}[\![((\lambda x.\, e_1)^l, \rho)]\!]_e = \mathcal{C}[\![(\lambda x.\, e_1)^l]\!]_e \cup \mathcal{C}[\![\rho]\!]_e$$

With this extended definition in hand, we can state the lemma, which can then be proved by induction on derivations $\langle e, \rho\rangle \Downarrow v$.

Lemma. *Let e_0 be a program, e an expression and ρ an environment. Let r^* and C^* satisfy the constraints $\mathcal{C}[\![(e, \rho)]\!]_{e_0}$. If $\langle e, \rho\rangle \Downarrow v$ then $\mathit{labelof}(v) \in \mathsf{C}^*(\mathit{labelof}(e))$ and r^* and C^* satisfy the constraints $\mathcal{C}[\![v]\!]_{e_0}$.*

1.5 1-CFA

The analysis described above, 0-CFA, is *context insensitive*: for a given subexpression in a function, it computes the set of values the subexpression may evaluate to regardless of where the function is called from.

Consider the following program (where for clarity we use let-syntax and label only some of the subexpressions).

$$\begin{array}{l}\mathsf{let}\ id = \lambda y.\, y\ \mathsf{in}\\ \mathsf{let}\ a = (id\ 19^1)^3\ \mathsf{in}\\ (id\ 21^2)^4\end{array}$$

Note that if we performed the 0-CFA analysis on this program, then $\mathsf{r}(y) = \{1, 2\}$ (i.e., y can be bound to the integers 19 and 20), and so $\mathsf{C}(3) = \{1, 2\}$, even though a can evaluate only to the integer 19.

We can improve the precision of CFA by distinguishing the different uses of variables and expressions based on the *context*. Towards this end, we will modify the analysis so that it computes the sets $\mathsf{r}(x, c)$ and $\mathsf{C}(l, c)$, where x ranges over program variables, l ranges over expression labels, and c ranges over contexts. Also, instead of just tracking labels of values that can be computed, we will use pairs of labels and contexts. That is, $\mathsf{r}(x, c)$ and $\mathsf{C}(l, c)$ will be sets of pairs (l', c').

For our purposes, we will regard the context of an expression as being the label of the expression that called the current function. For example, in the program above, we would compute $\mathsf{r}(y, 3)$ and $\mathsf{r}(y, 4)$, i.e., the possible values that variable y can be bound to when the function is called from call site 3, and the the possible values that variable y can be bound to when the function is called from call site 4. By distinguishing these different invocations of the identity function, we will correctly compute that variable a can be bound only to the integer 19. (We also need to allow a special context, c_{init}, to be used for the "top-level" expressions, i.e., expressions that are evaluated without any function being invoked.)

Our choice of context defines 1-CFA. In general, k-CFA is a control-flow analysis where the context is the list of the k call-sites on the stack. There are also other kinds of context that can be considered, leading to other kinds of control-flow analyses.

Let's try an initial **incorrect** version of the constraints, to understand how the contexts work. (Don't worry if you don't understand all the details of how these 1-CFA constraints work; focus on the high-level ideas.)

$$
\begin{aligned}
\mathcal{C}[\![n^l]\!]_e &= \{\forall c.\ (l, c) \in \mathsf{C}(l, c)\} \\
\mathcal{C}[\![(e_1 + e_2)^l]\!]_e &= \mathcal{C}[\![e_1]\!]_e \cup \mathcal{C}[\![e_2]\!]_e \cup \{\forall c.\ \ (l, c) \in \mathsf{C}(l, c)\} \\
\mathcal{C}[\![x^l]\!]_e &= \{\forall c.\ \mathsf{r}(x, c) \subseteq \mathsf{C}(l, c)\} \\
\mathcal{C}[\![(\lambda x.\, e_1)^l]\!]_e &= \{\forall c.\ \ (l, c) \in \mathsf{C}(l, c)\} \cup \mathcal{C}[\![e_1]\!]_e \\
\mathcal{C}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]_e &= \mathcal{C}[\![e_1^{l_1}]\!]_e \cup \mathcal{C}[\![e_2^{l_2}]\!]_e \\
&\qquad \cup \{\forall c.\ (l', c') \in \mathsf{C}(l_1, c) \Rightarrow \mathsf{C}(l_2, c) \subseteq \mathsf{r}(x, l) \mid \mathit{exprof}(e, l') = (\lambda x.\, e_0^{l_0})^{l'}\} \\
&\qquad \cup \{\forall c.\ (l', c') \in \mathsf{C}(l_1, c) \Rightarrow \mathsf{C}(l_0, l) \subseteq \mathsf{C}(l, c) \mid \mathit{exprof}(e, l') = (\lambda x.\, e_0^{l_0})^{l'}\}
\end{aligned}
$$

Observe in the constraints for application $(e_1^{l_1}\ e_2^{l_2})^l$, if e_1 in context c might evaluate to $(\lambda x.\, e_0^{l_0})^{l'}$ (i.e., $(l', c') \in \mathsf{C}(l_1, c)$) then the variable x in the context l (i.e., when it is called from the application labeled l) may be bound to the result of evaluating $e_2^{l_2}$ in context c (i.e., $\mathsf{C}(l_2, c) \subseteq \mathsf{r}(x, l)$). Also, the evaluation of the application may result in any value that the function body might evaluate to (i.e., $\mathsf{C}(l_0, l) \subseteq \mathsf{C}(l, c)$).

These rules are, however, not quite correct. Consider the following program (where, again for clarity, we use let-expressions and label only some of the expressions).

$$
\begin{aligned}
&\mathsf{let}\ f = \lambda a.\, \lambda b.\, a^{l_0}\ \mathsf{in} \\
&\mathsf{let}\ g = (f\ 21)^{l_2}\ \mathsf{in} \\
&(g\ 99)^{l_1}
\end{aligned}
$$

When we analyze the call site l_1, we will find that g can evaluate to the function $\lambda b.\, a^{l_0}$, but when we consider $\mathsf{r}(a, l_1)$, we find that it is empty! The problem is that variable a was not defined in context l_1, but in context l_2, that is, at the call site $(f\ 21)^{l_2}$.

To address this issue, we will modify our sets and constraints to use a *context environment* that records in which context each variable was defined. We use γ to range over context environments, which are just maps from variables to the context in which the variable was defined. Also, in our set of values, instead of just the labels of functions, we need to track closures, i.e., a pair of a function label and a context environment. Our modified constraints are now the following.

$$
\begin{aligned}
\mathcal{C}[\![n^l]\!]_e &= \{\forall c,\gamma.\ (l,c) \in \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![(e_1 + e_2)^l]\!]_e &= \mathcal{C}[\![e_1]\!]_e \cup \mathcal{C}[\![e_2]\!]_e \cup \{\forall c,\gamma.\ (l,c) \in \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![x^l]\!]_e &= \{\forall c,\gamma.\ \mathsf{r}(x,\gamma(x)) \subseteq \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![(\lambda x.\, e_1)^l]\!]_e &= \{\forall c,\gamma.\ (l,c,\gamma) \in \mathsf{C}(l,c,\gamma)\} \cup \mathcal{C}[\![e_1]\!]_e \\
\mathcal{C}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]_e &= \mathcal{C}[\![e_1^{l_1}]\!]_e \cup \mathcal{C}[\![e_2^{l_2}]\!]_e \\
&\quad \cup \{\forall c,\gamma.\ (l',c',\gamma') \in \mathsf{C}(l_1,c,\gamma) \Rightarrow \mathsf{C}(l_2,c,\gamma) \subseteq \mathsf{r}(x,l) \mid \mathit{exprof}(e,l') = (\lambda x.\, e_0^{l_0})^{l'}\} \\
&\quad \cup \{\forall c,\gamma.\ (l',c',\gamma') \in \mathsf{C}(l_1,c,\gamma) \Rightarrow \mathsf{C}(l_0,l,\gamma'[x \mapsto l]) \subseteq \mathsf{C}(l,c,\gamma) \mid \mathit{exprof}(e,l') = (\lambda x.\, e_0^{l_0})^{l'}\}
\end{aligned}
$$

Note that in the constraint generated for x^l, we look up the possible values for x in the context in which x was defined (i.e., $\mathsf{r}(x,\gamma(x)) \subseteq \mathsf{C}(l,c,\gamma)$).

Note also that in the constratings for a function, we use a closure. That is, supposing that we evaluate a function $(\lambda x.\, e_1)^l$ in context c where the free variables of $\lambda x.\, e_1$ are defined in contexts as described by γ, then the expression may evaluate to a closure represented by (l,c,γ).

We use these closures in the rule for application, where given $(e_1^{l_1}\ e_2^{l_2})^l$, if e_1 can evaluate to a function $(\lambda x.\, e_0^{l_0})^{l'}$ with lexical context environment γ' (i.e., a closure represented by (l',c',γ')), we are interested in what the function body $e_0^{l_0}$ can evaluate to in context $\gamma'[x \mapsto l]$, i.e., the context that extends the lexical context with a mapping from variable x to context l, since variable x is defined in context l (via the constraint that requires $\mathsf{C}(l_2,c,\gamma) \subseteq \mathsf{r}(x,l)$).

There is an additional improvement we can make. We currently compute the set of values/labels that an expression can evaluate to for all contexts and environments, even though it is typically the case that most expressions will be evaluated only in a few contexts. We can see this inefficiency in, for example, the constraint for $(\lambda x.\, e_1)^l$ where for all contexts c and environments γ we have $(l,c,\gamma) \in \mathsf{C}(l,c,\gamma)$.

To address this, we can track which expressions are "reachable" in which contexts and environments. That is, we compute a predicate $\mathsf{reachable}(l,c,\gamma)$ which is true only if the expression labeled l may be evaluated in context c with environment γ. We need to "seed" the analysis by asserting that the program itself is reachable in the special initial context with an empty environment, i.e., $\mathsf{reachable}(\mathit{labelof}(e), c_{init}, \emptyset)$ holds, where e is the program itself.

$$
\begin{aligned}
\mathcal{C}[\![n^l]\!]_e &= \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \Rightarrow (l,c) \in \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![(e_1 + e_2)^l]\!]_e &= \mathcal{C}[\![e_1]\!]_e \cup \mathcal{C}[\![e_2]\!]_e \cup \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \Rightarrow (l,c) \in \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![x^l]\!]_e &= \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \Rightarrow \mathsf{r}(x,\gamma(x)) \subseteq \mathsf{C}(l,c,\gamma)\} \\
\mathcal{C}[\![(\lambda x.\, e_1)^l]\!]_e &= \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \Rightarrow (l,c,\gamma) \in \mathsf{C}(l,c,\gamma)\} \cup \mathcal{C}[\![e_1]\!]_e \\
\mathcal{C}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]_e &= \mathcal{C}[\![e_1^{l_1}]\!]_e \cup \mathcal{C}[\![e_2^{l_2}]\!]_e \\
&\quad \cup \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \wedge (l',c',\gamma') \in \mathsf{C}(l_1,c,\gamma) \Rightarrow \mathsf{C}(l_2,c,\gamma) \subseteq \mathsf{r}(x,l) \mid \mathit{exprof}(e,l') = (\lambda x.\, e_0^{l_0})^{l'}\} \\
&\quad \cup \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \wedge (l',c',\gamma') \in \mathsf{C}(l_1,c,\gamma) \Rightarrow \\
&\qquad\qquad \mathsf{C}(l_0,l,\gamma'[x \mapsto l]) \subseteq \mathsf{C}(l,c,\gamma) \mid \mathit{exprof}(e,l') = (\lambda x.\, e_0^{l_0})^{l'}\} \\
&\quad \cup \{\forall c,\gamma.\ \mathsf{reachable}(l,c,\gamma) \wedge (l',c',\gamma') \in \mathsf{C}(l_1,c,\gamma) \Rightarrow \\
&\qquad\qquad \mathsf{reachable}(l'',l,\gamma'[x \mapsto l]) \subseteq \mathsf{C}(l,c,\gamma) \mid \mathit{exprof}(e,l') = (\lambda x.\, e_0^{l_0})^{l'}, l'' \in \mathit{labelsin}(e^{l_0})\}
\end{aligned}
$$

where $labelsin(e)$ is the set of labels that appears in expression e, and is defined as follows.

$$
\begin{aligned}
labelsin(n^l) &= \{l\} \\
labelsin(x^l) &= \{l\} \\
labelsin((\lambda x.\, e)^l) &= \{l\} \cup labelsin(e) \\
labelsin((e_1\ e_2)^l) &= \{l\} \cup labelsin(e_1) \cup labelsin(e_2) \\
labelsin((e_1 + e_2)^l) &= \{l\} \cup labelsin(e_1) \cup labelsin(e_2)
\end{aligned}
$$