UNIVERSITÉ LAVAL

FACULTÉ DES SCIENCES ET DE GÉNIE
Département d'Informatique

Cité universitaire
Québec, Canada G1K 7P4

CONTROL-FLOW ANALYSIS REQUIRES THE REFLEXIVE TRANSITIVE CLOSURE OF A GRAPH
Or Does It?

PAR

DANNY DUBÉ

RAPPORT DE RECHERCHE
DIUL-RR-0402

DÉPARTEMENT D'INFORMATIQUE
FACULTÉ DES SCIENCES ET DE GÉNIE

Pavillon Adrien-Pouliot
Université Laval
Sainte-Foy, Québec, Canada
G1K 7P4

AVRIL 2004

Control-Flow Analysis Requires the Reflexive Transitive Closure of a Graph
Or Does It?

Danny Dubé *

Université Laval
Danny.Dube@ift.ulaval.ca

Abstract. This article present a novel analysis technique that produces results strictly identical to those produced by Shivers's Standard Control-Flow Analysis. The analysis uses constraints between infinite binary trees of sets. An efficient variant of tree based analysis, namely graph based analysis, is also presented. Although graph based analysis uses a completely different algorithm than S-CFA, it produces the same results as S-CFA and has the same complexity. The presented techniques may eventually lead to variants that provide partial analysis results in less than cubic time or allow a compiler to perform separate compilation and whole-program control-flow analysis.

1 Introduction

Optimizing compilers for functional programming languages face the problem of having to perform mutually dependent data-flow and control-flow analyses. Indeed, traditional data-flow analysis requires the prior execution of a control-flow analysis in order to know which paths are taken at execution time. However, in functional languages, functions are ordinary data and one cannot compute the flow of control without first determining where the functions propagate.

Fortunately, control-flow analyses adapted to functional languages have been proposed. The most widely known is the one from Shivers [1]. We will refer to it as the Standard Control-Flow Analysis (S-CFA). It produces relatively good analysis results on typical programs and its cost is moderate: it has cubic time complexity and quadratic space complexity. Compilers for functional languages that include a sophisticated control-flow analysis often use S-CFA or a derivative.

S-CFA is often criticized, perhaps with good reasons. By definition, S-CFA has to be applied to a complete program at once. Despite its relative cheapness, it is usually considered too expensive to be applied to very large programs. Moreover, these programs are often distributed in many modules. Consequently, one has to decide whether to abandon separate compilation or to use an alternative, less precise analysis.

In this paper, we present a novel analysis that, while it does not solve these problems in the immediate, may lead to techniques that would be more practical.

* This work is funded by NSERC and Université Laval.

The technique is based on the use of infinite binary trees whose nodes are sets of labels and the use of constraints between these trees. While the heart of the usual implementation of S-CFA is an algorithm called reflexive transitive closure of a graph, our analysis generates a fixed graph (represented by sets and constraints) for a given program and analysis results are obtained by finding the minimal solution of the system of constraints.

Our tree based analysis has many interesting properties. In particular, there is the fact that the constraints are established in a compositional way. Also, it leads naturally to an efficient variant. We refer to the variant as the graph based analysis. Its complexity is the same as that of S-CFA.

The article is organized as follows. First, we come back to the definition of S-CFA. Next, we present our infinite tree based analysis. Then, the efficient variant is derived. Finally, we briefly mention related work and give directions for future work.

2 Reminder of Standard-CFA

The language that we consider is the pure λ-calculus. Here is the syntax of programs. Programs are simple λ-terms decorated with labels. A λ-term or expression e is one of:

$$\mathrm{x}_l \mid (\lambda_l \mathrm{x}.\ e') \mid ({}_l e'\ e'')$$

where l is a label, x is a variable, and e', e'' are labeled λ-terms. The cases represent a reference to a variable x, a λ-expression with formal parameter x and body e', and an application where the function produced by e' is applied to the value (another function, naturally) produced by e''. Labels are added to expressions for analysis purpose. In this presentation, we assume that programs do not reuse the same variables and labels twice. We use integers as labels.

The Standard Control-Flow Analysis is a well-known analysis for functional languages introduced by Shivers [1]. It is intended to conservatively determine at compile-time where the different functions may appear. Many different formulations of S-CFA have been presented and we introduce our own here. It is relatively similar to that used by Heintze and McAllester [2].

Our version of S-CFA takes the form of logical rules that control the propagation of λ-expressions in the program that we intend to analyze. The results of the analysis are "collected" in sets that indicate the value of expressions and the contents of variables. That is, the set S_l contains the λ-expressions to which expression e_l may evaluate and the set S_{x} contains the λ-expressions that variable x may contain.

The analysis works by adding entries in two relations. The first relation indicates whether a particular λ-expression $(\lambda_{l'} \mathrm{y}.\ e_{l''})$ may reach a particular program point α, which is a label l or a variable 'x', and is denoted by $(\lambda_{l'} \mathrm{y}.\ e_{l''}) \in S_\alpha$. The second relation indicates whether all λ-expressions that flow to a program point α also have to flow to another program point β and is denoted $S_\alpha \longrightarrow S_\beta$.

For $e_l = \mathrm{x}_l$: $$\frac{}{S_{\mathrm{x}} \longrightarrow S_l} \text{ (ref)}$$

For $e_l = (\lambda_l \mathrm{x}.\ e_{l'})$: $$\frac{}{(\lambda_l \mathrm{x}.\ e_{l'}) \in S_l} \text{ (λ)}$$

For $e_l = ({}_l e_{l'}\ e_{l''})$: $$\frac{(\lambda_{l'''} \mathrm{x}.\ e_{l^{(4)}}) \in S_{l'}}{S_{l''} \longrightarrow S_{\mathrm{x}}} \text{ (call-1)}$$

$$\frac{(\lambda_{l'''} \mathrm{x}.\ e_{l^{(4)}}) \in S_{l'}}{S_{l^{(4)}} \longrightarrow S_l} \text{ (call-2)}$$

For any program points α, β: $$\frac{(\lambda_l \mathrm{x}.\ e_{l'}) \in S_\alpha \quad S_\alpha \longrightarrow S_\beta}{(\lambda_l \mathrm{x}.\ e_{l'}) \in S_\beta} \text{ (prop)}$$

Fig. 1. Rules for S-CFA

The rules that define S-CFA are presented in Figure 1.[1] Corresponding to a given program E, a set of rules come to existence. The analysis proceeds by starting with empty '$\in$' and '$\longrightarrow$' relations and then by deriving all the assertions that can be possibly obtained using the rules. When no new assertions can be obtained anymore, the analysis terminates. The results of the analysis are expressed by the '$\in$' relation. Since both relations start out empty and that only rule-generated assertions can be added to these relations, we obtain the smallest solution for the analysis.

Although if, strictly speaking, this formulation of the analysis does nothing else than manipulating assertions, it offers a set-theoretic point of view of the results. That is, even if S_α is only a *name* that appears in the generated assertions, we can refer to it as a genuine set. It is reputed to contain all λ-expressions $(\lambda_l \mathrm{x}.\ e_{l'})$ such that assertion $(\lambda_l \mathrm{x}.\ e_{l'}) \in S_\alpha$ is generated by the rules.

Performing S-CFA using these rules or any equivalent formulation leads to important difficulties. Indeed, because of the (call-n) and (prop) rules, the '$\longrightarrow$' and '$\in$' relations are built incrementally, based on each other, and the '$\in$' relation may ultimately relate *any* λ-expression to *any* program point. These characteristics of the analysis force S-CFA to be performed on whole programs at once and makes this process equivalent to a dynamic reflexive transitive closure on a graph, which has cubic complexity (see [2]). Whole-program processing and cubic complexity limit the usefulness of S-CFA. In the next section, we gradually elaborate another method for obtaining the same results as those of S-CFA without requiring a dynamic closure on a graph.

3 Towards a Closureless Analysis

Here, using a rather informal approach, we derive a completely different way to compute the same results as those of S-CFA using infinite trees of sets. The

[1] We write $l^{(4)}$ instead of l'''' to avoid clutter when there are too many prime signs.

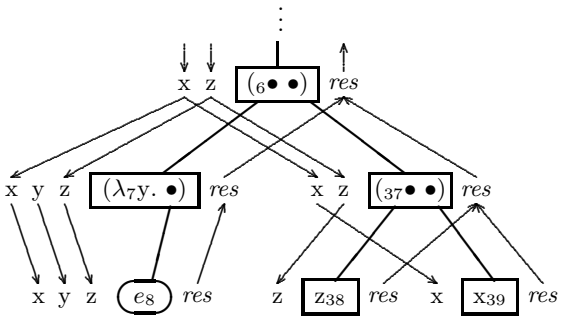

Fig. 2. Computing S-CFA using inherited and synthesized attributes

definitive formulation of infinite tree based analysis is presented in the next section with all necessary details.

3.1 A Compositional Static Analysis

Before we present the way our analysis gathers its results, we first state a principle that the gathering process should obey. The principle is that of *compositionality*. We want the analysis to be able to produce analysis results for an expression as a function of the analysis results for its sub-expressions only. For example, the results for an application $({}_l e_{l'}\ e_{l''})$ should be a function of the results for expression '$e_{l'}$' and those for expression '$e_{l''}$'.

Even if this principle looks nice as it is stated, it is too simplistic as in both dynamic interpretation and static interpretation, the value (or analysis results) of an expression depends on the value (or static contents, respectively) of its free variables. So the right principle that should be obeyed is that the analysis should produce the analysis results for an expression by using only the analysis results for its sub-expressions *and* the contents of its free variables. In turn, the analysis results for the sub-expressions are computed by using the contents of their free variables.

The principle of compositionality may be best explained using terminology from the field of attributed grammars. From the point of view of a particular expression in the program, the contents of its free variables can be seen as *inherited* attributes and the analysis results for the expression can be seen as a *synthesized* attribute. For example, the illustration in Figure 2 shows the flow of information in a fragment of some program's abstract syntax tree (AST). Here, expression e_6 is the sub-expression of another, surrounding expression and expression e_8 is some arbitrary expression. For the sake of the example, let us suppose that the free variables of e_8 are 'x', 'y', and 'z' (denoted as $\mathrm{FV}(e_8) = \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$). The free variables for the other expressions in Figure 2 can easily be identified. For each individual expression, its inherited attributes are enumerated on its left as the names of its free variables and its synthesized attribute (the analysis results) is depicted on its right as '*res*'. The contents of the variables flow downwards

to all expressions that depend on them. Note that the contents of a particular variable either comes from the parent expression or *appears* as a new variable because of the presence of a formal parameter (variable 'y' in e_7, for instance). The contents of the '*res*' attributes are computed using the '*res*' attributes of the sub-expressions and the inherited attributes. This is especially clear in the case of references to variables (expressions e_{38} and e_{39}, for instance).

Until now, we have remained vague about the actual contents of the attributes and the way the synthesized attributes of sub-expressions are combined in order to produce the synthesized attribute of an expression. We next describe the contents of the attributes and give a sketch of the way their contents flow in order to compute the equivalent of S-CFA.

3.2 Trees of Sets

Now, we have to determine what the contents of the attributes should be. A temptingly simple answer prescribes each attribute to be a set containing labels of λ-expressions. That is, an inherited attribute ought to contain the labels of the λ-expressions that the corresponding variable may contain and a synthesized attribute ought to contain the labels of the λ-expressions that the corresponding expression may evaluate to.[2]

While the contents of an attribute include *at least* such a set, it is easy to realize that more data are required. In order to do so, let us consider the following code fragment:

$$\ldots\ ({}_{l}e_{l'}\ e_{l''})\ \ldots$$

Let us suppose that the '*res*' attributes for $e_{l'}$ and $e_{l''}$ are already computed. It means that we can determine the sets of λ-expressions that flow to $e_{l'}$ and $e_{l''}$. Now, how can the '*res*' attribute for e_l be computed? In other words, how can we determine the set of λ-expressions that e_l may evaluate to? Maybe that, by inspection of the sub-expressions and free variables of e_l, it is possible to compute the '*res*' attribute for e_l. Even if it could be done, we prefer to take another approach. We decree that the '*res*' attribute of $e_{l'}$ includes another set which contains the labels of the λ-expressions that may be *returned* by the functions to which $e_{l'}$ may evaluate. Using this additional set, it is trivial to determine which λ-expressions e_l may evaluate to. Note that the additional set can be seen as describing the *range* of the functions to which $e_{l'}$ evaluates.

While reading the reasoning about the previous code fragment, one may have asked himself where the informations in the '*res*' attribute for $e_{l''}$ are used. This attribute describes the arguments that are passed to the functions produced by $e_{l'}$ and, certainly, it should not be forgotten. It is especially clear if our code fragment is:

$$\ldots\ ({}_{l}(\lambda_{l'}\mathrm{x}.\ \mathrm{x}_{l'''})\ e_{l''})\ \ldots$$

[2] More precisely, the labels are those of the λ-expressions that generated the functions to which the variables may be bound or to which the expressions may evaluate. However, for the sake of brevity, we will keep this slight imprecision.

as the informations in the '*res*' attribute for $e_{l''}$ directly flow into the '*res*' attributes for $e_{l'''}$ and e_l. Consequently, there must be a third set in the '*res*' attribute for $e_{l'}$ that contains the labels of the λ-expressions that may be *passed to* the functions to which $e_{l'}$ may evaluate. This new set can be seen as describing the *domain* of the functions to which $e_{l'}$ evaluates.

Now, before the explanations become too verbose, we will give names to our sets instead of referring to them as the original, the second, and the third sets. First, we will refer to the '*res*' attribute for some expression e_l as T_l. The set containing the labels of the λ-expressions to which e_l evaluates is denoted by $\texttt{set}(T_l)$. Also, for now, we will refer to the set of labels of the λ-expressions that are returned by and passed to the functions to which e_l evaluates as $\texttt{rg}(T_l)$ and $\texttt{dom}(T_l)$, respectively. Similarly, the inherited attribute 'x' of some expression e_l will be denoted by $T_{\mathrm{x,l}}$. An inherited attribute also comprises the same three sets that we will denote as $\texttt{set}(T_{\mathrm{x,l}})$, $\texttt{rg}(T_{\mathrm{x,l}})$, and $\texttt{dom}(T_{\mathrm{x,l}})$.

We have argued that there should be three sets in each attribute. However, it is not clear that these are sufficient. In fact, it is relatively easy to see that more sets are required. For instance, in order to properly analyze this code fragment:

$$\ldots\ (_l(_{l'} e_{l''}\ e_{l'''})\ e_{l^{(4)}})\ \ldots$$

there must be some way to compute the set of λ-expressions to which e_l may evaluate, that is, $\texttt{set}(T_l)$. Naturally, $\texttt{set}(T_l)$ contains the labels of the λ-expressions that are returned by the functions to which $e_{l'}$ evaluates. That is, $\texttt{set}(T_l) = \texttt{rg}(T_{l'})$. But how can the set $\texttt{rg}(T_{l'})$ be computed? Clearly, there should be a way to obtain it from $T_{l''}$. The sets $\texttt{set}(T_{l''})$, $\texttt{rg}(T_{l''})$, and $\texttt{dom}(T_{l''})$ are of no help since what we are really interested in are the λ-expressions *returned* by the λ-expressions in $\texttt{rg}(T_{l''})$. By doing a slight abuse of notation, it is the set $\texttt{rg}(\texttt{rg}(T_{l''}))$ that would be necessary to compute $\texttt{set}(T_l)$ and $\texttt{rg}(T_{l'})$. Using similar arguments, we could show that the sets $\texttt{dom}(\texttt{rg}(T_{l''}))$, $\texttt{rg}(\texttt{dom}(T_{l''}))$, and $\texttt{dom}(\texttt{dom}(T_{l''}))$ are necessary also.

In fact, the analysis of arbitrary code fragments may require the existence of the set $f_1(\ldots f_n(T_l)\ldots)$, for any n with f_i being one of '$\texttt{rg}$' and '$\texttt{dom}$', for $1 \leq i \leq n$. This means that any attribute has to comprise an infinite number of sets. These sets are organized in a binary tree fashion. Figure 3 illustrates the shape of such an infinite binary tree of sets.

An infinite tree of sets t is made of three components: a set of labels at the root, an infinite tree of sets for the range, and an infinite tree of sets for the domain. We can now get rid of the abuse of notation that we had to make previously and give appropriate names to trees and sets. The three components of an infinite tree of sets t are referred to as $\texttt{set}(t)$, $\texttt{rg}(t)$, and $\texttt{dom}(t)$, respectively. Only the first component turns out to be a set of labels. Consequently, the sets comprised in a tree t are referred to as $\texttt{set}(t)$, $\texttt{set}(\texttt{dom}(t))$, $\texttt{set}(\texttt{rg}(t))$, $\texttt{set}(\texttt{dom}(\texttt{dom}(t)))$, $\texttt{set}(\texttt{rg}(\texttt{dom}(t)))$, $\texttt{set}(\texttt{dom}(\texttt{rg}(t)))$, $\texttt{set}(\texttt{rg}(\texttt{rg}(t)))$, $\texttt{set}(\texttt{dom}(\texttt{dom}(\texttt{dom}(t))))$, $\texttt{set}(\texttt{rg}(\texttt{dom}(\texttt{dom}(t))))$, $\texttt{set}(\texttt{dom}(\texttt{rg}(\texttt{dom}(t))))$, ...

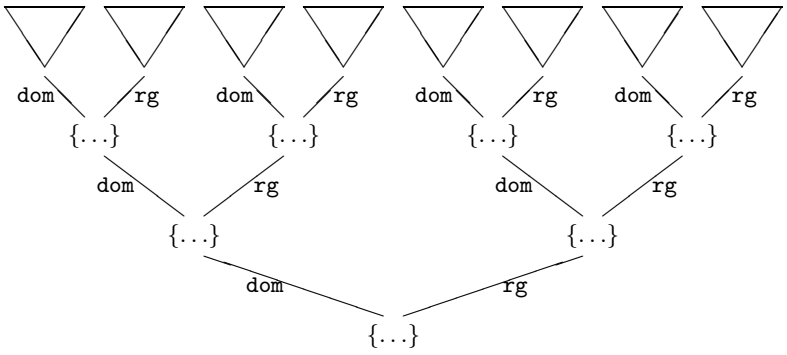

Fig. 3. Infinite tree of sets

3.3 Separating the Sets

Up to this point, we have considered T_l, the analysis results for an expression e_l (the old '*res*' attribute), as being synthesized and, $T_{\mathrm{x,l}}$, the contents of variable 'x' at expression e_l, as being inherited. We are about to see that, while this is true for the trees themselves, it is not necessarily the case for all the sets they comprise.

Let us consider T_l, which corresponds to expression e_l. This tree is synthesized as it represents the analysis results for e_l. But what about its sets? The set $\mathtt{set}(T_l)$ is certainly synthesized since it contains the labels of the λ-expressions to which e_l may evaluate. It makes sense to say that e_l *produces* $\mathtt{set}(T_l)$ or that it *decides* which labels go into $\mathtt{set}(T_l)$. By a similar reasoning, the set $\mathtt{set}(\mathtt{rg}(T_l))$ is also synthesized because e_l decides which functions it evaluates to and these functions decide which functions they return when called. So e_l is indirectly responsible for the contents of $\mathtt{set}(\mathtt{rg}(T_l))$.

Now, let us see whether the same reasoning could be used for the set $\mathtt{set}(\mathtt{dom}(T_l))$. As for $\mathtt{set}(\mathtt{rg}(T_l))$, the beginning of the reasoning holds since e_l decides which functions it evaluates to. However, these functions *cannot decide* which functions they are passed. It is the context surrounding e_l that decides whether the functions in $\mathtt{set}(T_l)$ are called and with which arguments. For this reason, we say that the set $\mathtt{set}(\mathtt{dom}(T_l))$ is not synthesized but inherited.

For example, let us examine T_6 in the following code fragment:

$$\ldots\ ({}_5(\lambda_6\mathrm{x}.\ (\lambda_7\mathrm{y}.\ e_8))\ (\lambda_{13}\mathrm{z}.\ e_{14}))\ \ldots$$

Clearly, e_6 decides of the contents of $\mathtt{set}(T_6)$ and indeed the latter is $\{6\}$. Also, it decides of the contents of $\mathtt{set}(\mathtt{rg}(T_6))$ and the latter is $\{7\}$. That is, even if it is only indirectly, e_6 still is responsible for the contents of $\mathtt{set}(\mathtt{rg}(T_6))$ as the contents is directly determined by one of its own sub-expressions, namely e_7. On the other hand, e_6 does not decide of the contents of $\mathtt{set}(\mathtt{dom}(T_6))$, not even indirectly. In our code fragment, it is the application e_5 that feeds the

functions produced by e_{13} (only $\{13\}$ in this case) as arguments to e_6. So the set $\texttt{set}(\texttt{dom}(T_6))$ has contents $\{13\}$ but it is not by a decision by e_6.

What happens with the sets two levels up in the tree T_l? The set $\texttt{set}(\texttt{rg}(\texttt{rg}(T_l)))$ is indirectly produced by e_l as it contains the labels of the functions that are returned by the functions that are themselves returned by the functions to which e_l evaluates. So $\texttt{set}(\texttt{rg}(\texttt{rg}(T_l)))$ in synthesized. The sets $\texttt{set}(\texttt{dom}(\texttt{rg}(T_l)))$ and $\texttt{set}(\texttt{rg}(\texttt{dom}(T_l)))$ are considered to be inherited as their contents is provided by the context surrounding e_l. Indeed, the context decides what is passed to the functions in $\texttt{set}(\texttt{rg}(T_l))$ and the functions in $\texttt{set}(\texttt{dom}(T_l))$ decide by themselves what they may return. Curiously, the set $\texttt{set}(\texttt{dom}(\texttt{dom}(T_l)))$ turns out to be synthesized. This may seem counter-intuitive at first since domain branches intervene. However, one has to realize that while the functions in $\texttt{set}(\texttt{dom}(T_l))$ have not been chosen by e_l, it is e_l (or one of its sub-expressions) that may call these functions and pass them arguments. These arguments are exactly the functions found in $\texttt{set}(\texttt{dom}(\texttt{dom}(T_l)))$. The arguments are chosen by e_l, directly or indirectly, and consequently $\texttt{set}(\texttt{dom}(\texttt{dom}(T_l)))$ is considered to be synthesized.

We could make similar reasonings for every set in T_l. Moreover, we could do the same for an inherited tree such as $T_{\text{x,l}}$, the contents of variable 'x' at expression e_l. However, the "synthesizedness" and "inheritedness" of the sets are best summarized using the following rules. If an infinite binary tree of sets t is synthesized, its root set, $\texttt{set}(t)$, is synthesized, its range sub-tree, $\texttt{rg}(t)$, is synthesized, and its domain sub-tree, $\texttt{dom}(t)$, is inherited. *Vice versa*, if a tree t is inherited, its root set is inherited, its range sub-tree is inherited, and its domain sub-tree is synthesized. The reader has certainly recognized the principles of covariance and contravariance here. A set s is in a *covariant* position in a tree t if we have to traverse an even number of "dom" branches when going from the root of t to s. A set s is in a *contravariant* position in a tree t if we have to traverse an odd number of "dom" branches when going from the root of t to s. The sets of t that are in covariant position flow in the same direction (synthesized or inherited) as t and the sets of t that are in contravariant position flow in the opposite direction.

From this point, we are going to separate the infinite binary trees in two parts: the covariant part and the contravariant part. Each part is a partial infinite binary tree that contains only the sets that are in covariant or in contravariant position, respectively. We will refer to the two parts of a tree simply as the covariant tree and the contravariant tree. This separation will help to make the presentation in the next section simpler. From now on, every attribute comprises a covariant tree and a contravariant tree. That is, the analysis results for an expression e_l will be expressed using T_l^+ and T_l^-. Naturally, T_l^+ and all the sets it comprises are synthesized. T_l^- and all the sets it comprises are inherited. The contents of a variable 'x' at some expression e_l will be expressed using $T_{\text{x,l}}^+$ (the inherited part) and $T_{\text{x,l}}^-$ (the synthesized part).

3.4 Dropping Trees for the Environment

Using the infinite binary trees, we are now almost ready to present the formulation of the control-flow analysis based on infinite trees. We only need to make a little modification to our organization of trees. The modification concerns the trees that represent the contents of the free variables, that is, environment trees.

In our attribute-inspired presentation, we associate to each expression e_l of the program a synthesized attribute for its result and an inherited attribute for each of its free variables. However, it is clear that the contents of the variables does not change as they flow downwards to the expressions that use them. At least, the covariant part of the contents ($T^+_{\mathrm{x,l}}$) certainly does not change.[3] Anyway, environment trees are intended for connecting formal parameters to references to variables. These connections are prescribed by the program's syntax tree and there are only a finite number of them.

In order to simplify the presentation of the new analysis and to reduce the number of trees that are required by the analysis, the trees for the free variables will be dropped except for those that are directly associated to the formal parameters of the program. A consequence of this decision is that the number of trees required by the analysis is proportional to the size of the program. Indeed, there are now two trees per expression for the corresponding result (T^+_l and T^-_l) and two additional trees per λ-expression for the formal parameter (T^+_{x} and T^-_{x}). There is no need for the additional l subscript in the name of the environment trees anymore as the name of the variables uniquely identifies the trees.[4]

4 Infinite Tree Based Analysis

Now that the infinite binary trees of sets of labels are defined, the static control-flow analysis based on them can be presented. We briefly come back to the infinite trees and give a few definitions, then we describe the constraints that specify the analysis, and explain how these constraints may be solved. The section ends with the statement of a few important properties of infinite tree based analysis.

4.1 Defining the Trees

The analysis works with two kinds of trees: result trees and environment trees. The result trees correspond intuitively to the result of the static evaluation of expressions. The environment trees correspond intuitively to the value of the formal parameters. For each expression e_l of the program, there are the covariant result tree T^+_l, which is synthesized, and the contravariant result tree T^-_l, which is inherited. For each formal parameter 'x', there are the covariant environment tree T^+_{x}, which is inherited, and the contravariant environment tree T^-_{x}, which

[3] We will see in the next section that the contravariant part may change as it flows upwards.

[4] Remember that we assume that the program uses a different variable name for each variable.

is synthesized. These four kinds of trees are referred to as the *main* trees as they are not part of "bigger" trees themselves. We denote these trees using a capital "T". When referring to a main tree or a sub-tree without distinction, we use t instead. Also, we use T^{σ}_{α} when referring to the main tree for some program point α; that is, T^{σ}_{α} is either a result tree ($\alpha = l$) or an environment tree ($\alpha = \mathrm{x}$) and σ is one of '+' and '−'.

Apart from their intended meaning, result trees and environment trees are similar. It is between covariant trees and contravariant trees that there is a real difference. A covariant tree t is made of three components: a set of labels, $\mathtt{set}(t)$, a covariant sub-tree for the range, $\mathtt{rg}(t)$, and a contravariant sub-tree for the domain, $\mathtt{dom}(t)$. A contravariant tree t is made of only two components: a contravariant sub-tree for the range, $\mathtt{rg}(t)$, and a covariant sub-tree for the domain, $\mathtt{dom}(t)$. Strictly speaking, there exist only the sets of labels. The trees are simply a useful concept that helps us refer to a whole collection of sets at once. Since the trees are not data structures, there are no construction or elimination operations that allow us to "build" and "deconstruct" the trees. Nonetheless, we use the notations $\mathtt{tr}^{+}(t_1,\, s,\, t_2)$ and $\mathtt{tr}^{-}(t_1,\, t_2)$ as shorthands to mention many trees and/or sets at once. These notations help to keep the constraints concise.

Infinite tree based analysis use constraints and each of these is a containment constraint between trees that is denoted using '$\sqsubseteq$'. This inclusion constraint requires the contents of each set in the first tree to be contained into the corresponding set in the second tree. A containment constraint can only be put between two covariant trees or between two contravariant trees. Even if the two kinds of constraints are not defined exactly the same way, we use the same symbol ($\sqsubseteq$) for both kinds. The trees under constraint indicate which kind of constraint is being used. As the trees themselves do not really exist, strictly speaking, the constraints between trees do not either. The containment constraint between two covariant trees t_1^+ and t_2^+ can be seen as a shorthand for three other containment constraints:

$$t_1^+ \sqsubseteq t_2^+ \qquad \text{means} \qquad \begin{array}{r} \mathtt{dom}(t_1^+) \sqsubseteq \mathtt{dom}(t_2^+) \\ \mathtt{set}(t_1^+) \subseteq \mathtt{set}(t_2^+) \\ \mathtt{rg}(t_1^+) \sqsubseteq \mathtt{rg}(t_2^+) \end{array}$$

The second of the three constraints is a containment constraint between sets of labels. On the other hand, the first and third constraints are themselves constraints between trees and, as such, they are shorthands. Note that the first constraint is set between contravariant trees while the third is set between covariant trees. Similarly, the containment constraint between two contravariant trees t_1^- and t_2^- can be seen as a shorthand for two other containment constraints:

$$t_1^- \sqsubseteq t_2^- \qquad \text{means} \qquad \begin{array}{r} \mathtt{dom}(t_1^-) \sqsubseteq \mathtt{dom}(t_2^-) \\ \mathtt{rg}(t_1^-) \sqsubseteq \mathtt{rg}(t_2^-) \end{array}$$

The expansion for this kind of constraint does not (directly) mention a constraint between sets as there are no sets at the roots of t_1^- and t_2^-. Given these two equivalences, it is clear that containment constraints between trees are only a notational convenience.

$$
\begin{array}{lll}
\text{For } e_l = \mathrm{x}_l\text{:} & & T^+_{\mathrm{x}} \sqsubseteq T^+_l \\
& & T^-_l \sqsubseteq T^-_{\mathrm{x}} \\
\\
\text{For } e_l = (\lambda_l \mathrm{x}.\ e_{l'})\text{:} & \mathtt{tr}^+(T^-_{\mathrm{x}},\ \{l\},\ T^+_{l'}) \sqsubseteq T^+_l & \\
& T^-_l \sqsubseteq \mathtt{tr}^-(T^+_{\mathrm{x}},\ T^-_{l'}) & \\
\\
\text{For } e_l = ({}_l e_{l'}\ e_{l''})\text{:} & \mathtt{dom}(T^+_{l'}) \sqsubseteq T^-_{l''} & \\
& \mathtt{rg}(T^+_{l'}) \sqsubseteq T^+_l & \\
& \mathtt{tr}^-(T^+_{l''},\ T^-_l) \sqsubseteq T^-_{l'} &
\end{array}
$$

Fig. 4. Constraints generated by the infinite tree based analysis

An important remark has to be made about the containment constraints. The constraints, between either trees or sets, must be considered as formal assertions and not as containment relations. This means that we are not interested in checking whether two sets s_1 and s_2 are in the relation '$\subseteq$' or not. In fact, there is no *relation* involved, except if we say otherwise. What we are interested in, is to determine whether the constraints emitted by the analysis state, directly or indirectly, that s_2 *has* to contain all the labels contained in s_1. The difference is subtle but it is important for the validity of the proofs of correctness of the analysis.

Among the constraints that are emitted directly by the analysis, there are some that mention the fictitious tree constructors. The meaning of these constraints becomes clear when we replace the shorthand notations by their equivalents. For example, a constraint of the form:

$$
\mathtt{tr}^+(t^-_1,\ s,\ t^+_2) \sqsubseteq t^+ \qquad \text{simply means:} \qquad
\begin{array}{r}
t^-_1 \sqsubseteq \mathtt{dom}(t^+) \\
s \subseteq \mathtt{set}(t^+) \\
t^+_2 \sqsubseteq \mathtt{rg}(t^+)
\end{array}
$$

The constraints that are emitted by the analysis are highly regular and, once fully expanded into constraints between sets, they are of one of only two forms. The first form is set between two sets located somewhere in the trees. That is, it looks like $\mathtt{set}(\mathtt{dom}(\mathtt{rg}(\ldots T_\alpha \ldots))) \subseteq \mathtt{set}(\mathtt{rg}(\ldots T_\beta \ldots))$. The second form is set between a *literal* set of labels and a set located in a tree. That is, it looks like $\{l, \ldots\} \subseteq \mathtt{set}(T_\alpha)$. In fact, the way the analysis is defined guarantees that only literal *singletons* may appear in the constraints.

4.2 Generating the Constraints

The constraints that are generated for the analysis of a program E are relatively straightforward. For each expression e_l in E, a few constraints are emitted. The constraints for e_l depend only on the kind of e_l. Figure 4 presents the constraints that are generated for each kind of expression.

Let us explain briefly the meaning of the generated constraints. For a reference to a variable x_l, the constraints simply connect the expression e_l with the formal parameter 'x'. The first constraint expresses the fact that the formal parameter provides the (covariant) contents of the variable. The second constraint ensures that the contribution by the context surrounding the expression propagates to the formal parameter. Note that, in general, a single formal parameter may collect the contributions from more than one references to it. The constraints for the reference to a variable are very simple as they can be seen as two connections between the ends of giant cables. Moreover, the "type" of the cables (i.e. covariant vs. contravariant) ensures that no incorrect connection can be made.

The constraints for the λ-expression $(\lambda_l \mathrm{x}.\ e_{l'})$ are not as simple. The constraints make the connection between the exterior, the context surrounding e_l, and the interior, sub-expression $e_{l'}$ and formal parameter 'x'. The first constraint summarizes in one tree what is synthesized by e_l using what is synthesized by $e_{l'}$ and 'x'. Of course, the root set of T_l^+ is the singleton as all the functions that may result from the evaluation of e_l must have label l. The range of T_l^+ is the same tree as the covariant tree for the result of e_l. It makes sense as "what the functions produced by e_l will return" is "what is produced by $e_{l'}$". The domain of T_l^+ indicates what will happen to the eventual arguments of the functions produced by e_l. The second constraint splits "what happens to e_l" into the part that "happens to the body" and the part that "make the contents of the formal parameter". For instance, if some function with label l'' is passed to e_l (that is, $l'' \in \mathtt{set}(\mathtt{dom}(T_l))$), then this function flows down in 'x' (that is, $l'' \in \mathtt{set}(T_{\mathrm{x}})$).

Finally, the constraints for the application $({}_l e_{l'}\ e_{l''})$, although they are three, are relatively intuitive. The second one expresses the fact that the result when evaluating e_l is the range of the result when evaluating $e_{l'}$. The first one extracts a description of "what happens to the arguments" and propagates this to the argument expression. The third constraint both takes care of passing the argument's value to the callee and propagating "what happens to e_l" to the range of $e_{l'}$.

Note how the connections between the trees propagate almost losslessly all the information produced by the sub-expressions or the context. There are two exceptions however. First, in an application $({}_l e_{l'}\ e_{l''})$, the root set of the result for the evaluation of $e_{l'}$ $(\mathtt{set}(T_{l'}^+))$ is not channeled anywhere. This may seem odd at first as this set contains valuable information, that is, the labels of the functions that are invoked. However, from the point of view of the analysis, an application is the final destination of the functions that get invoked. The application "consumes" the functions that it invokes. Only the arguments to the functions and the returned values flow through the application. The second place where information gets lost is at the root expression of program E, that is, at E itself. Indeed, the evaluation results for E (T_1^+, if we suppose that $E = e_1$) do not appear in the left member of any constraint. That makes sense however as T_1^+ represent the result of the whole program's evaluation and this result falls out of the reach of the program itself. A final remark concerns the first

Program:

$$E = ({}_1(\lambda_2 \mathrm{x}.\ ({}_3\mathrm{x}_4\ \mathrm{x}_5))\ (\lambda_6 \mathrm{y}.\ ({}_7\mathrm{y}_8\ \mathrm{y}_9)))$$

Constraints:

$$\begin{aligned}
\mathtt{dom}(T_2^+) &\sqsubseteq T_6^- \\
\mathtt{rg}(T_2^+) &\sqsubseteq T_1^+ \\
\mathtt{tr}^-(T_6^+,\, T_1^-) &\sqsubseteq T_2^- \\
\mathtt{tr}^+(T_{\mathrm{x}}^-,\, \{2\},\, T_3^+) &\sqsubseteq T_2^+ \\
T_2^- &\sqsubseteq \mathtt{tr}^-(T_{\mathrm{x}}^+,\, T_3^-) \\
\mathtt{dom}(T_4^+) &\sqsubseteq T_5^- \\
\mathtt{rg}(T_4^+) &\sqsubseteq T_3^+ \\
\mathtt{tr}^-(T_5^+,\, T_3^-) &\sqsubseteq T_4^- \\
T_{\mathrm{x}}^+ &\sqsubseteq T_4^+ \\
T_4^- &\sqsubseteq T_{\mathrm{x}}^- \\
T_{\mathrm{x}}^+ &\sqsubseteq T_5^+
\end{aligned}
\qquad
\begin{aligned}
T_5^- &\sqsubseteq T_{\mathrm{x}}^- \\
\mathtt{tr}^+(T_{\mathrm{y}}^-,\, \{6\},\, T_7^+) &\sqsubseteq T_6^+ \\
T_6^- &\sqsubseteq \mathtt{tr}^-(T_{\mathrm{y}}^+,\, T_7^-) \\
\mathtt{dom}(T_8^+) &\sqsubseteq T_9^- \\
\mathtt{rg}(T_8^+) &\sqsubseteq T_7^+ \\
\mathtt{tr}^-(T_9^+,\, T_7^-) &\sqsubseteq T_8^- \\
T_{\mathrm{y}}^+ &\sqsubseteq T_8^+ \\
T_8^- &\sqsubseteq T_{\mathrm{y}}^- \\
T_{\mathrm{y}}^+ &\sqsubseteq T_9^+ \\
T_9^- &\sqsubseteq T_{\mathrm{y}}^-
\end{aligned}$$

Fig. 5. An example of infinite tree based analysis

constraint generated for λ-expressions. This is the only constraint that denotes a constraint between a literal set and a set located in a tree. All the other (set-to-set) constraints are established between sets located in trees. These literal-to-tree constraints are the ones that forces labels to flow in the system of constraints. Otherwise, the minimal solution of the constraints would simply let every set be empty.

4.3 An example

Figure 5 presents an example of tree based analysis. A small program and the constraints generated for its analysis are given. The program consists in an infinite loop. The effect is that λ-expression e_6 is the sole value that is non-trivially manipulated by the program. If we were to minimally solve the constraints, we would find that $2 \in \mathtt{set}(T_2^+)$, which is obvious, and that label 6 flows to the following sets:

$$\begin{array}{ll}
\mathtt{set}(\mathtt{dom}^{(2n+2)}(T_2^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_2^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_{\mathrm{x}}^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_{\mathrm{x}}^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_4^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_4^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_5^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_5^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_6^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_6^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_{\mathrm{y}}^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_{\mathrm{y}}^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_8^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_8^-)) \\
\mathtt{set}(\mathtt{dom}^{(2n)}(T_9^+)) & \mathtt{set}(\mathtt{dom}^{(2n+1)}(T_9^-))
\end{array}$$

where $n \in \mathbb{N}$ and $\mathtt{dom}^{(i)}(\cdot)$ denotes i consecutive accesses to the '$\mathtt{dom}$' branch of a tree. All other sets are left empty. We skip the reasoning that leads to this solution. Note that the part of the results in which an optimizing compiler

would probably be interested are the root set of each covariant main tree. Indeed, expressions e_1, e_3, and e_7 do not evaluate to any function as they represent an infinite computation. As expected, the function labeled with 2 reaches e_2 and the function labeled with 6 reaches e_4, e_5, e_6, e_8, and e_9 and in variables 'x' and 'y'.

4.4 Solving the Constraints

Once the constraints for a program have been generated, the remaining step consists in computing the minimal solution for these constraints. However, simply "computing the minimal solution" does not qualify as an algorithm since an infinite number of sets are involved. A question that naturally arises is whether it is possible to solve the constraints and, if so, how. We will only say that the answer is affirmative. The reason is that trees that contain the minimal solution in their sets happen to be regular. In the following exposition of the properties of tree based analysis, a short explanation outlines the reasoning behind that. We do not elaborate more on the resolution of the constraints as it does not form an algorithm whose efficiency compares favorably with that of S-CFA. Section 5 presents a derivative of infinite tree based analysis that is efficient. Still, from a theoretical point of view, it is important to know that it is possible to obtain a finite description of the minimal solution.

4.5 Properties

The most important property of our infinite tree based analysis is that it produces analysis results that are similar to those of S-CFA. Although it is clear that the intent in the design of tree based analysis is to perform a control-flow analysis, it is not that easy to convince oneself that it does produce the same results. Indeed, S-CFA uses a constructive process to compute its results while tree based analysis generates a fixed set of constraints whose minimal solution contains the appropriate results. The equivalence of the results of the two analyses is established by the following theorems.

Theorem 1. *Infinite tree based analysis causes the propagation of at least the same labels as S-CFA. Formally,*

$$(\lambda_l \mathrm{x}.\ e_{l'}) \in S_\alpha \quad \Longrightarrow \quad l \in \mathtt{set}(T^+_\alpha) \ \textit{(in any solution of the constraints)}$$

Proof. We only give a sketch of the proof here. The idea consists in sorting the assertions produced by S-CFA in topological order. Indeed, the conditions-conclusion organization of the rules imposes a partial order on the assertions. These can be topologically sorted based on that partial order. It is then simple to prove by induction that, each time a new assertion is derived, these exists a corresponding chain of constraints in tree based analysis. That is, for each new assertion, we show that:

$$\begin{aligned} (\lambda_l \mathrm{x}.\ e_{l'}) \in S_\alpha &\quad\Longrightarrow\quad T^+_l \sqsubseteq \ldots \sqsubseteq T^+_\alpha \\ S_\alpha \longrightarrow S_\beta &\quad\Longrightarrow\quad T^+_\alpha \sqsubseteq \ldots \sqsubseteq T^+_\beta \end{aligned}$$

□

The reversed proposition is as simple to state but it is a little bit harder to prove.

Theorem 2. *S-CFA causes the propagation of at least the same labels as infinite tree based analysis. Formally,*

$$l \in \mathtt{set}(T_\alpha^+) \text{ (in the minimal solution of the constraints)} \quad \Longrightarrow \quad (\lambda_l \mathrm{x}.\ e_{l'}) \in S_\alpha$$

Proof. Here is a sketch of the proof. Once again, the trick is to obtain a convenient ordering. Let us explain what the trick is. For any label l that has to appear in the root set $\mathtt{set}(T_\alpha^+)$ of some main tree in the minimal solution of the constraints, there exist chains of set-to-set constraints like this one:

$$\{l\} \subseteq \ldots \subseteq \mathtt{set}(T_\alpha^+)$$

Consequently, there exists a chain of minimal length among these. An ordering of the pairs $(l, \mathtt{set}(T_\alpha^+))$ is made according to the length of the corresponding minimal chain lengths. Using this ordering, there remains to prove by induction on this ordering that for any propagated label in tree based analysis, the corresponding λ-expression has to go through the same propagation. In fact, the pairs that we sort are pairs of covariant main trees (T_α^+, T_β^+) for which $T_\alpha^+ \sqsubseteq \ldots \sqsubseteq T_\beta^+$. For each such pair, we have to show that:

$$T_\alpha^+ \sqsubseteq \ldots \sqsubseteq T_\beta^+ \quad \Longrightarrow \quad S_\alpha \longrightarrow S_\beta$$

□

We now state rigorously the property of the infinite trees with the minimal solution in their sets being regular. We give only hints on how this result can be obtained. The statement uses finite state automata to express the property. These automata consume an access path and output a set of labels when the input is exhausted. An access path is a sequence of '`dom`' and '`rg`' tokens. Of course, the set of labels is a subset of the set of the program's labels. We denote by $A_t(P)$ the set of labels returned by automaton A_t for tree t when presented the access path P. For example, if we are interested in set $\mathtt{set}(\mathtt{dom}(\mathtt{dom}(\mathtt{rg}(T_5^+))))$, then we can obtain it by computing $A_{T_5^+}(\mathtt{dom} \cdot \mathtt{dom} \cdot \mathtt{rg})$.

Theorem 3. *Let E be a program for which tree based analysis has been performed and whose constraints have been solved minimally. Then, corresponding to each covariant or contravariant main tree T_α^σ, there exists a finite state automaton, $A_{T_\alpha^\sigma}$, that computes the contents of each set in T_α^σ given its access path. That is, for each b_i being either '`dom`' or '`rg`', we have that:*

$$A_{T_\alpha^\sigma}(b_1 \cdot \ldots \cdot b_n) \quad = \quad \mathtt{set}(b_1(\ldots b_n(T_\alpha^\sigma)\ldots))$$

provided that the access path is legal for T_α^σ (i.e. that it contains a valid number of accesses to the '`dom`' branch).

Proof. First note that the problem of creating the desired finite state automata reduces to the sub-problem of creating automata that track the propagation of one arbitrary label l. The original problem is then solved by solving a sub-problem for each λ-expression of the program and collecting the results together. Next, the solution of the sub-problem for label l consists essentially in computing the languages L_α^σ that contain the access paths of the sets to which l must propagate. We simply state without proof that these languages are regular. So a solution for a sub-problem consists in creating ordinary (i.e. accept vs. reject input) finite state automata for these languages. □

5 Graph Based Analysis

Infinite tree based analysis, as simple and elegant as it can be, does not provide an efficient algorithm. However, solving the constraints as they stand may seem overkill as an optimizing compiler does not need to know the contents of all the sets in the infinite trees but only the root sets of the main trees. This section presents a lighter approach that seeks to compute only the root sets.

The lighter approach is based on graphs but, at the same time, its presentation is relatively similar to that of S-CFA. That is, the approach consists in building a graph where each vertex corresponds to a main tree of tree based analysis and where arcs are added according to a set of rules until no arcs can be added anymore. There are two vertices S_α^+ and S_α^- per program point α. Alternatively, graph based analysis can be approached from the point of view of a '$\longrightarrow$' relation to which we add entries. In fact, graph based analysis uses many kinds of arcs.

The following subsections introduce the different kinds of arcs, the rules that allow the introduction of new arcs, and a few properties of graph based analysis.

5.1 Defining the Arcs

Graph based analysis uses five kinds of arcs. One of the kinds of arcs simply represents ordinary propagation of labels but the other kinds denote special forms of propagation where labels may flow from a non-root set in a tree to a non-root set in another tree.

The five kinds of arcs are described in Figure 6. The three columns indicate to what each end of the arcs connects and how the arcs are denoted. The first kind of arc denote the ordinary propagation of labels while the other four kinds denote the special kinds of propagations. The signs on top of the arrows try to be meaningful visual indicators of the way labels propagate. For example, arrow '$\overset{\uparrow\mathtt{dom}}{\longrightarrow}$' indicates that labels propagate from the tree on its left *up* to the *domain* branch of the tree on its right.

5.2 Computing the Reflexive Transitive Closure

Graph based analysis essentially consists in the reflexive transitive closure of a directed graph. Rules are used to introduce arcs in the graph. The closure is

Notation:	From:	To:
$p \longrightarrow q$	vertex p	vertex q
$p \xrightarrow{\uparrow \mathtt{dom}} q$	vertex p	domain of vertex q
$p \xrightarrow{\downarrow \mathtt{dom}} q$	domain of vertex p	vertex q
$p \xrightarrow{\uparrow \mathtt{rg}} q$	vertex p	range of vertex q
$p \xrightarrow{\downarrow \mathtt{rg}} q$	range of vertex p	vertex q

Fig. 6. Kinds of arcs in graph based analysis

For any vertex p:	$\dfrac{}{p \longrightarrow p}$ (refl)
For a tree based constraint	
... of the form $T^\sigma_\alpha \sqsubseteq T^\sigma_\beta$:	$\dfrac{}{S^\sigma_\alpha \longrightarrow S^\sigma_\beta}$ (cnstr-1)
... of the form $T^\sigma_\alpha \sqsubseteq \mathtt{dom}(T^\sigma_\beta)$:	$\dfrac{}{S^\sigma_\alpha \xrightarrow{\uparrow \mathtt{dom}} S^\sigma_\beta}$ (cnstr-2)
... of the form $\mathtt{dom}(T^\sigma_\alpha) \sqsubseteq T^\sigma_\beta$:	$\dfrac{}{S^\sigma_\alpha \xrightarrow{\downarrow \mathtt{dom}} S^\sigma_\beta}$ (cnstr-3)
... of the form $T^\sigma_\alpha \sqsubseteq \mathtt{rg}(T^\sigma_\beta)$:	$\dfrac{}{S^\sigma_\alpha \xrightarrow{\uparrow \mathtt{rg}} S^\sigma_\beta}$ (cnstr-4)
... of the form $\mathtt{rg}(T^\sigma_\alpha) \sqsubseteq T^\sigma_\beta$:	$\dfrac{}{S^\sigma_\alpha \xrightarrow{\downarrow \mathtt{rg}} S^\sigma_\beta}$ (cnstr-5)

Fig. 7. Rules for the static phase of graph based analysis

done in two phases. The first phase introduces arcs that mimic the constraints of tree based analysis and also takes care of the reflexive part of the closure. The rules that govern the first phase are given in Figure 7. We will also refer to the first phase as the *static* phase as it only uses rules that can fire once. The second phase takes care of the transitive part of the closure and mimics the resolution process of tree based analysis. The rules that govern the second phase are given in Figure 8. We will also refer to the second phase as the *dynamic* phase.

The meaning of the rules for the static phase is immediate. The meaning of the rules for the dynamic phase is quite simple too. For instance, rule (arg) simply says that labels that are pushed up in the domain sub-tree propagate to a root set where they are pulled down from the domain sub-tree.

Graph based analysis proceeds first by triggering the rules of the static phase and then by triggering the rules of the dynamic phase until no new arcs can be added anymore. Once the dynamic phase has completed, analysis results are available under the form of ordinary arcs $S^+_l \longrightarrow S^+_\alpha$ that connect the vertex corresponding to a λ-expression e_l to the vertex of some program point α. From

$$\frac{p \xrightarrow{\uparrow \mathtt{dom}} q \qquad q \longrightarrow r \qquad r \xrightarrow{\downarrow \mathtt{dom}} s}{p \longrightarrow s} \text{ (arg)}$$

$$\frac{p \xrightarrow{\uparrow \mathtt{rg}} q \qquad q \longrightarrow r \qquad r \xrightarrow{\downarrow \mathtt{rg}} s}{p \longrightarrow s} \text{ (ret)}$$

$$\frac{p \longrightarrow q \qquad q \longrightarrow r}{p \longrightarrow r} \text{ (trans)}$$

Fig. 8. Rules for the dynamic phase of graph based analysis

the analysis results, one can easily recover either the set of program points to which a particular λ-expression flows or the set of λ-expressions that flow to a particular program point.

5.3 Properties

Graph based analysis produces the same analysis results as infinite tree based analysis and S-CFA. This is no surprise as graph based analysis mimics tree based analysis with fidelity. An interesting property of graph based analysis is its cubic complexity. This is the same worst-case complexity as that of S-CFA. This is quite fascinating as the two techniques are designed from very different principles at the start.

The cubic complexity of graph based analysis is caused by rule (trans). The two statements used as conditions in the rule mention three variables that range over the vertices of the graph. As a consequence, the rule may be triggered up to $O(n^3)$ times, where n is the size of the program. Ironically, rules (arg) and (ret) which use three statements as conditions can only be triggered up to $O(n^2)$ times. This is caused by the relative rareness of special arcs. Indeed, given a vertex p and one of the four special kinds of arcs '$\xrightarrow{\updownarrow \mathtt{branch}}$', there is at most one vertex q such that $p \xrightarrow{\updownarrow \mathtt{branch}} q$.

6 Related Work

Infinite tree based analysis and its efficient graph based variant presented in this paper have been inspired by the work of Heintze and McAllester [2] and, especially, the work of Mossin [3]. In both of the latter, the presented techniques are linear complexity analyses that produce analysis results under an implicit form. Individual requests for explicit results such as "the set of expressions to which a particular λ-expression propagates" and "the set of λ-expressions that reach a particular expression" can be answered in linear time. Consequently, the user has the option of obtaining explicit answers for a bounded number of requests in $O(n)$ time or obtaining complete explicit analysis results in $O(n^2)$ time, where n is the size of the program. This is asymptotically faster than S-CFA. However, these techniques can only be applied to well-typed programs

for which the type of each expression has size smaller than a fixed bound. The applicability of the techniques is limited by this restriction.

Techniques intended to perform separate analysis of programs by abstract interpretation are presented in [4]. The techniques can apply to an analysis such as S-CFA. However, they cause a degradation of the quality of the analysis results, they cause an increase of the analysis times, they are not applicable in general, or they require annotations from the programmer.

7 Future Work

The faster analysis techniques by Heintze, McAllester, and Mossin are interesting and we intend to investigate the possibility of developing a similar implicit-form analysis from which individual requests can be answered and with complexity less than cubic. Our graph based analysis allows the creation of a kind of implicit form for the analysis results (first phase in linear time) but any subsequent request for explicit results may trigger cubic time computations. However, the author has not yet convinced himself that there is no way to reformulate the analysis so that partial explicit results could be obtained in less than cubic time.

Also, since one of the main principles that lead to tree based analysis is compositionality, we intend to see whether our techniques could be used to conciliate whole program control-flow analysis and separate compilation. Indeed, the cubic complexity of S-CFA has made whole-program analysis prohibitive for large, modular programs. However, since our tree based analysis sets its constraints in a compositional fashion, it has the potential to allow for the partial analysis of a module by keeping the behavior of the rest of the program abstract. The inherited trees that should be provided to the module by the rest of the program may simply be considered as unknowns. The results of such a partial analysis would be a function taking the module's inherited trees and returning the module's synthesized trees. Hopefully, using this function at link time would be cheaper than completely re-analyzing the module. Similarly, the partial analysis of a program with one or many modules missing (holes, so to speak) might be possible.

Finally, we expect tree based analysis to be relatively adaptable to a richer set of data structures and expressions. Here, we only considered the λ-calculus in which the only possible values are functions. Handling lists and algebraic data structures should be possible while still obtaining results of S-CFA-like quality. We also expect expressions found in typical functional programming languages to be quite easy to handle too.

8 Conclusion

This article present a novel analysis technique that produces results strictly identical to those produced by the Standard Control-Flow Analysis. The analysis uses constraints between infinite binary trees of sets. The analysis results are obtained by finding the minimal solution to these constraints. To the opposite

of what is done by S-CFA, the list of constraints for a given program is fixed and can be established in a straightforward manner. We demonstrate that tree based analysis produces the same results as S-CFA and that, although the analysis results are computed using an infinite number of sets, these have a regular structure.

An efficient variant of tree based analysis, namely graph based analysis, is also presented. This variant mimics just the part of the resolution process that is needed to obtain the same results as S-CFA. Although graph based analysis uses a completely different algorithm than S-CFA, it has the same complexity as S-CFA, that is, cubic complexity.

The techniques presented in this article are interesting from a theoretical point of view but do not bring improvements in practice. However, there is potential to develop variants that provide partial analysis results in less than cubic time. Also, extensions of the techniques could allow a compiler to perform separate compilation *and* whole program control-flow analysis at an affordable cost.

References

1. Shivers, O.: Control flow analysis in Scheme. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. (1988) 164–174
2. Heintze, N., McAllester, D.: Control-flow analysis for ML in linear time. In: 1997 ACM Conference on Programming Language Design and Implementation. Volume 32 of ACM SIGPLAN Notices., New York, ACM Press (1997) 261–272
3. Mossin, C.: Higher-order value flow graphs. In: Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP), Southampton, UK, Springer-Verlag (1997) 159–174
4. Cousot, P., Cousot, R.: Compositional separate modular static analysis of programs by abstract interpretation. In: Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, Scuola Superiore G. Reiss Romoli (2001)