Demand-Driven Type Analysis: an Introduction

Danny Dubé Marc Feeley
DIRO
Université de Montréal
{dube, feeley}@iro.umontreal.ca

Abstract

We propose a new demand-driven approach to efficiently drive a powerful type analysis for a dynamically-typed functional language. The analyzer has the advantage of being controllable by a bound on the time that it can put into the analysis. When given enough time, it can provide results of very high quality. The analysis is based on a flexible analysis framework that allows the abstract modeling of the computation to be modified *while* the analysis is performed. The approach consists in generating initial demands from reliable hints in the program and processing these demands to purposefully guide the modifications of the abstract model. Our proposed approach has not been implemented fully, but we sketch a prototype implementation of demand-driven analysis which is based on simple pattern-matching.

1 Introduction

Program analyses are widely used in compilation. They range from common sub-expression detection analysis [2] to pointer analysis [10]. There are analyses intended more for low-level languages such as C and others more intended for high-level languages such as Scheme. The analyses have a tendency to become more essential and more complex as the languages they are intended for become advanced. Two reasons might help to explain that. First, a higher-level language offers more general services to the programmer, which often incur a penalty in code efficiency if a compilation is done without a certain effort in analysis and optimization. Second, the properties that must be discovered in order to do a good compilation are generally more complex. Unfortunately, more complex analyses usually imply more costly analyses.

When a compiler implementer is faced with the problem of gathering a certain kind of information, he often has to choose among a wide spectrum of approaches, especially when the problem is complex. The tradeoff is normally between the time (and/or space) taken by the analysis and the accuracy of the information to gather. Most of the time, the implementer chooses a certain approach and glues it to his compiler. But on what basis should a particular approach be chosen?

1.1 Choosing the "best" analysis

The choice is usually done considering the *average* needs of the target users. Most of the time, the chosen approach has a well-defined behavior in terms of its accuracy, running time, and required space. Of course, the choices made cannot satisfy every user in every situation: one user may find it too slow; another, too inaccurate. This is the case even if several optimizations levels are implemented in the compiler. Let us sketch the possibilities that are available to the implementer.

Traditional analyses

Fast analyses are popular. There are many: control-flow analysis [14, 15], numeric range analysis [8], abstract reference counting analysis [11], etc.

They manipulate a well-defined abstraction (or model) of the program and its computations. The size of the model is in direct relation with the size of the program and the time required to compute the analyses is always in $O(n^k)$ for a k rarely greater than 3. The amount of resources required is well under control. And they obtain results that are quite acceptable most of the time and provided that the program contains *typical* code.

Unfortunately, the polynomial time bound often causes serious limitations in the cleverness of these analyses. Sometimes, even very ordinary programming styles can mislead the analyses and make them produce poor results. As an example, Jagannathan and Weeks mention in [12] that control-flow analyses that use call-strings to disambiguate abstract evaluation environments (such as the k-cfa) get confused by the use of the `map` function called with different argument types. Such an example is showed in Figure 1. The code is straightforward and yet, the k-cfa or a similar analysis will fail to show that there is no type error, no matter which k is used. This is because, after k recursive calls of `map` to itself, the call-string is invariably the same. At that point, all the functions and all the pairs that are passed to `map` are merged together, which makes the analysis believe that the wrong operator may be applied to the wrong list.

In more general terms, we could say that the limitations of the k-cfa come primarily from the fact that it uses unreliable hints to distinguish the abstract evaluation environments; namely, the call-strings. For example, in Scheme, the body of a function has no means of computing the syntactic position where the call to the function occurred. Neither does there exist tests to determine where a particular pair was created. On the other hand, there exist type tests and primitives to inspect the contents of the objects. In the best of cases, call-strings and concrete computations are merely correlated, whereas types and values are directly involved in the computations. For these reasons, we consider call-strings to be unreliable hints for an analysis.

Many of the traditional analyses can be fooled by a pro-

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l))
            (map f (cdr l)))))

(map (lambda (n) (- n)) '(1 2 3 ...))

(map (lambda (p) (car p)) '((1) (2) ...))
```

Figure 1: Difficult code for the k-cfa

gramming style that is not convoluted. This can be frustrating for a user that has a program that he knows is correct but that is beyond the limited power of the available analyzer. He may be willing to give the analyzer plenty of resources in order to obtain better results but the analyzer will not take advantage of this to improve the analysis.

More accurate analyses

To avoid the limitations of the traditional analyses, one can instead choose an analysis that uses "the Right Hints" in order to distinguish various abstract environments. The right hints can be the type of the objects that are passed to the procedures, for example. This has a true correspondence with the concrete computations that occur in the program: an expression in caller position should return a function, the argument to `car` should be a pair, etc. We will expand on this later.

While we should expect better analysis results from such an analysis, we should expect catastrophic time and space consumption in certain cases. To see why, it suffices to consider an expression located inside a function of high arity or inside many nested λ-expressions (say, n variables in the lexical environment) and an analysis that distinguishes the abstract evaluation environments based on the type of the objects bound to the variables (say, k different types). This analysis immediately exhibits exponential behavior (k^n different abstract environments).

If a user has to use a compiler that features such a (potentially) costly analysis instead of a traditional one, it would be just as frustrating for the user as in the other case. He can only choose between disabling the analysis, if it is possible, and waiting for days for a single compilation.

Static model

It is clear that it is difficult to find the "right" balance between speed and accuracy when the time comes to choose an analysis model. Even when the "best" compromise has supposedly been chosen, when a individual program is compiled, it is tempting to believe that another compromise would have been "better". Having said that, we claim that this ambiguity comes from the fact that the model is *static*. Of course, it depends on the program, but in a very simple manner and it remains the same during the whole compilation of the program.

Since the analyzer is committed to an abstract model, it necessarily exposes itself to be either too simplistic or too heavy for particular programs. It results either in too poor accuracy or in good results that have been obtained with a vastly too great effort. It can even be both for the same program when some of its interesting properties are very easy to discover while the others are more challenging.

Dynamically changing model

What we believe to be more appropriate is to have an abstract model that can dynamically change. That is, it should adapt to the level of difficulty of analysis of the particular program to analyze.

Here is a sketch of an analysis using a dynamically changing model. At the start, the strategy is to choose an initial model that is coarse. Since coarse analyses do quite well *in the typical case*, a significant part of the interesting properties may already be found by this first analysis. Then, the model ought to be refined, in order to be better equipped to attack the remaining, more difficult properties. It may result in having some more properties to be found. Then the model is refined again. And so on...

Of course, this raises many questions: How do we identify the so-called "interesting" properties? What should a refinement of the model be? How can we automatically update an abstract model? And more importantly, what should a "better equipped model" be? Before we start to bring answers to these questions, we must describe our goal in more detail.

1.2 The objective

We intend to develop an *adaptable-power* type analysis for a purely functional, applicative, and dynamically typed mini-language. We assume that the entire program is available. The analysis must have the potential to be very precise. However, the user should have the control over the amount of *effort* that is put into the analysis of his program. This way, during development, he can request a fast and coarse analysis, and, at the final compilation, invest an appropriate amount of time to obtain a high-quality analysis.[1]

The analyzer has to be able to deal with a bound on the amount of work it can do. When given little time, it must terminate quickly, delivering results that are potentially of poor quality. When given a lot more time, it must either terminate prematurely if completely satisfying results are obtained or, in the usual case, continue to improve the quality of the results until the time is up. We do not want to rely on programmer annotations. These may be erroneous and, consequently, cannot be trusted. To trust any annotation would contradict the principle of safety that comes with a high-level language.[2] Only a safe analysis should provide results that are to be used for optimization purpose.

The abstract model used by the analyzer must be flexible. The crucial part of our objective is to find an "intelligent" driver that is able to coordinate the re-analysis and model-update cycle to try to obtain the best results within the time bound that is given. The driver must refine the model when it seems profitable, but refrain to do so when it seems useless. Note that, as intelligent as the driver might be, we do not want to do true AI, not even an expert system. We want a driver that proceeds in a more systematic way.

[1] What we consider as a fast and coarse analysis is something similar to the 0-cfa. A higher-quality analysis would necessarily be more costly. For very long programs, the cost may be prohibitive, even for a fast analysis, considering that 0-cfa has cubic complexity in worst case.

[2] Moreover, if the program contains an expression such as `(car x)`, it already means that the programmer believes that `x` can only be bound to pairs.

$$
\begin{array}{lcll}
\text{Exp} & := & e_l & e \in \text{Exp}', l \in \text{Lab} \\
\text{Exp}' & := & \texttt{\#f} & \\
 & & \text{x} & \text{x} \in \text{Var} \\
 & & (e_1\ e_2) & e_1, e_2 \in \text{Exp} \\
 & & (\lambda\text{x}.\ e_1) & \text{x} \in \text{Var}, e_1 \in \text{Exp} \\
 & & (\texttt{if}\ e_1\ e_2\ e_3) & e_1, e_2, e_3 \in \text{Exp} \\
 & & (\texttt{cons}\ e_1\ e_2) & e_1, e_2 \in \text{Exp} \\
 & & (\texttt{car}\ e_1) & e_1 \in \text{Exp} \\
 & & (\texttt{cdr}\ e_1) & e_1 \in \text{Exp} \\
 & & (\texttt{pair?}\ e_1) & e_1 \in \text{Exp} \\
\text{Lab} & := & \textit{Labels} & \\
\text{Var} & := & \textit{Variables} &
\end{array}
$$

Figure 2: Language syntax

In order to achieve our goal, we use a flexible analysis framework that is presented in Section 2 along with the mini-language. This framework can support very powerful analyses and, so, can help to prove interesting but difficult properties of the program. Section 3 presents an intuitive introduction to the *demand-driven analysis.* It is the demands that encompass the required "intelligent" driver for the analyzer. The idea is quite simple: interesting properties can be found with the help of *hints* present in the program; these properties are likely to be true and if they are, then may happen to be provable, mathematically speaking; it follows that they might be provable inside our framework and maybe in reasonable time. Section 4 sketches a basic demand-driven analysis implementation. It is based on *patterns.* Finally, Section 5 concludes with a brief mention of the research that is the closest to our own and with the next logical steps in our research.

2 Notation and definitions

2.1 A small language

The language we use in this paper is presented in Figure 2. It is a small subset of Scheme with a few modifications. It is purely functional, applicative, dynamically typed, and evaluation proceeds from left to right. The only types available are the booleans, with `#f` as the sole element, the pairs and the procedures having one parameter. The modifications are: all the pair-related primitives are syntactic forms and, when the `pair?` expression must evaluate to a true value, its evaluates to the same pair as its argument. All these details can be found in the semantics of the language in Figure 3.[3]

Despite the fact that the language is small, it is complex enough to allow the construction of programs that are as difficult to analyze as one can desire. A simple λ-calculus provides only one "type": the functions. In the present case, the variety of types combined to the fact that certain expressions require objects of a specific type creates the necessary complications. The call expression and the `car` and `cdr` expressions require the first sub-expression to be of a particular type (a simple implementation would perform a dynamic type test to guarantee safety).

Throughout the paper, we assume that a program in this language has no free variables, is α-converted[4], and properly labeled[5]. To keep things simple, we consider that the purpose of our type analysis is to collect information that allows the compiler to remove as many dynamic type tests as possible.

[3] The "$\dot\cup$" sign denotes the disjoint union. That is, $A = B \mathbin{\dot\cup} C$ if and only if $A = B \cup C$ and $B \cap C = \emptyset$.

[4] All variables in the program have a distinct name.

[5] Each expression in the program has a distinct label.

$$
\begin{array}{lll}
\text{Val}^{\uparrow} := \text{Err} \mathbin{\dot\cup} \text{Val} & & \\
\text{Err} := \textit{Errors} & & \\
\text{Val} := \text{ValB} \mathbin{\dot\cup} \text{ValC} \mathbin{\dot\cup} \text{ValP} & & \\
\text{ValB} := \{\#\text{f}\} & & \textit{Booleans} \\
\text{ValC} := \text{Val} \to \text{Val}^{\uparrow} & & \textit{Closures} \\
\text{ValP} := \text{Val} \times \text{Val} & & \textit{Pairs} \\
\text{Env} := \text{Var} \to \text{Val} & &
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E} : \text{Exp} \to \text{Env} \to \text{Val}^{\uparrow} & & \textit{Evaluation function} \\
\mathsf{E}\ [\![\texttt{\#f}]\!]\ \rho & = \#\text{f} & \\
\mathsf{E}\ [\![\text{x}]\!]\ \rho & = \rho\ \text{x} & \\
\mathsf{E}\ [\![(e_1\ e_2)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ [\![e_2]\!]\ \rho)\ (\mathsf{A}\ v_1)) & \\
\mathsf{E}\ [\![(\lambda\text{x}.\ e_1)]\!]\ \rho & = \lambda v.\ \mathsf{E}\ [\![e_1]\!]\ \rho[\text{x} \mapsto v] & \\
\mathsf{E}\ [\![(\texttt{if}\ e_1\ e_2\ e_3)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v \neq \#\text{f}\ ?\ \mathsf{E}\ [\![e_2]\!]\ \rho : \mathsf{E}\ [\![e_3]\!]\ \rho) & \\
\mathsf{E}\ [\![(\texttt{cons}\ e_1\ e_2)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ [\![e_2]\!]\ \rho)\ (\lambda v_2.\ (v_1,\ v_2))) & \\
\mathsf{E}\ [\![(\texttt{car}\ e_1)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v = (v_1,\ v_2)\ ?\ v_1 : \text{ERROR}) & \\
\mathsf{E}\ [\![(\texttt{cdr}\ e_1)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v = (v_1,\ v_2)\ ?\ v_2 : \text{ERROR}) & \\
\mathsf{E}\ [\![(\texttt{pair?}\ e_1)]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v \in \text{ValP}\ ?\ v : \#\text{f}) & \\
\mathsf{A} : \text{Val} \to \text{Val} \to \text{Val}^{\uparrow} & & \textit{Apply function} \\
\mathsf{A}\ f\ v & = f \in \text{ValC}\ ?\ f\ v : \text{ERROR} & \\
\mathsf{C} : \text{Val}^{\uparrow} \to (\text{Val} \to \text{Val}^{\uparrow}) \to \text{Val}^{\uparrow} & & \textit{Check function} \\
\mathsf{C}\ v\ k & = v \in \text{Err}\ ?\ v : k\ v &
\end{array}
$$

Figure 3: Language semantics

2.2 A generic analysis framework

In the introduction, we insisted on the fact that an analyzer should have the ability to modify the abstract model that it uses to analyze the program. This requires the introduction of a generic analysis *framework.* The framework by itself is not a complete analysis procedure; it requires many parameters to become an instanciation of an analysis. The parameters may be assimilated to the model itself. The framework imposes very few constraints on the model.

Instantiation parameters

Figure 4 presents the parameters and a brief description of each. First, the framework expects sets of abstract values. These are given by three *finite* non-empty disjoint sets. Second, another finite set provides the *contours.* Note that no other constraint exists on what these sets might be. Finally, the framework expects *abstract computation functions.* These mimic the concrete computations done by the program. There is one for the creation of closures, one for the creation of pairs and one to select contours associated with the abstract evaluation environments.

Function cc receives the label l of an expression and the current contour k and returns an abstract closure. Function pc receives the label l where a pair containing v_1 and v_2 is created in contour k, and returns an abstract pair. Function call receives a label l where a function f is applied to value

$\mathcal{V}al\mathcal{B} \neq \emptyset$	Abstract booleans
$\mathcal{V}al\mathcal{C} \neq \emptyset$	Abstract closures
$\mathcal{V}al\mathcal{P} \neq \emptyset$	Abstract pairs
$\mathcal{C}ont \neq \emptyset$	Contours
$k_0 \in \mathcal{C}ont$	Main contour
$\mathsf{cc} : \mathrm{Lab} \times \mathcal{C}ont \to \mathcal{V}al\mathcal{C}$	Abstract closure creation
$\mathsf{pc} : \mathrm{Lab} \times \mathcal{V}al \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{V}al\mathcal{P}$	Abstract pair creation
$\mathsf{call} : \mathrm{Lab} \times \mathcal{V}al\mathcal{C} \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{C}ont$	Contour selection

where $\mathcal{V}al := \mathcal{V}al\mathcal{B} \,\dot{\cup}\, \mathcal{V}al\mathcal{C} \,\dot{\cup}\, \mathcal{V}al\mathcal{P}$

Figure 4: Instantiation parameters of the analysis framework

v in contour k; it returns the contour in which the body of c has to be evaluated. These functions must be defined on all their domain and, of course, respect their type. On top of that, one of the contours must be identified as the *main* contour, that is, it is the contour in which the top-level expression e_{l_0} of the program is evaluated.

The case of the abstract booleans deserves a short explanation. It is obvious that the framework does not allow as much parameterization for the booleans as for the other types. There can be more than one abstract boolean, of course, but no boolean creation function is expected by the framework. There could be, since the `#f` and `pair?` expressions can evaluate to a boolean. However, since there is only one concrete boolean, we did not feel the need to provide the tools to manipulate distinct abstract booleans. In fact, we do not know if it would be useful at all. However, support for distinct boolean manipulation could be added in the framework with little effort.

Note that, although the abstract evaluation functions must be defined on all their domain, not all input combinations make sense. For example, the result of the cc function does not make sense when the label that it is passed is not the label of a λ-expression. However, the analysis will never use this result either, so cc can return any element of $\mathcal{V}al\mathcal{C}$ without consequences. This approach is simpler than having the set of labels partitioned into λ-expression labels, call labels, etc.

Analysis variables

Once the parameters are passed to the analysis framework, a complete analyzer is instanciated. Here we present the matrices of abstract variables that are used by this analyzer. Figure 5 briefly enumerates them.

The α matrix contains the abstract values to which each expression evaluates in each contour. A particular entry $\alpha_{l,k}$ may be empty. It occurs if the expression e_l does not get evaluated in the abstract environment represented by the contour k. The β matrix contains the values bound to each variable in each contour. When the body of the expression $(\lambda_l \mathtt{x}.\ e_{l'})$ is evaluated in a contour k, a reference to the variable x refers to the entry $\beta_{\mathrm{x},k}$. An entry $\beta_{\mathrm{x},k}$ may be empty, too, for similar reasons as with $\alpha_{l,k}$. An entry $\gamma_{c,k}$ of the matrix γ contains the values that are returned by the closure c when its body has been evaluated in the contour k. Once again, it may be empty. An entry $\delta_{l,k}$ is basically a flag. It indicates whether or not e_l gets evaluated in the contour k. Its contents is not important; only the fact that it is empty or not. Non-emptyness of $\delta_{l,k}$ implies evaluation.

Value of e_l in k:
$\alpha_{l,k} \subseteq \mathcal{V}al \qquad l \in \mathrm{Lab},\ k \in \mathcal{C}ont$
Contents of x in k:
$\beta_{\mathrm{x},k} \subseteq \mathcal{V}al \qquad \mathrm{x} \in \mathrm{Var},\ k \in \mathcal{C}ont$
Return value of c with its body in k:
$\gamma_{c,k} \subseteq \mathcal{V}al \qquad c \in \mathcal{V}al\mathcal{C},\ k \in \mathcal{C}ont$
Flag indicating evaluation of e_l in k:
$\delta_{l,k} \subseteq \mathcal{V}al \qquad l \in \mathrm{Lab},\ k \in \mathcal{C}ont$
Creation circumstances of c:
$\chi_c \subseteq \mathsf{cc}^{-1}(c) \qquad c \in \mathcal{V}al\mathcal{C}$
Creation circumstances of p:
$\pi_p \subseteq \mathsf{pc}^{-1}(p) \qquad p \in \mathcal{V}al\mathcal{P}$
Circumstances leading to k:
$\kappa_k \subseteq \mathsf{call}^{-1}(k) \qquad k \in \mathcal{C}ont$

Figure 5: Matrices containing the results of an analysis

The meaning of the remaining three matrices is less obvious. They provide a kind of log of the origins of the abstract values. As an example, let us consider an abstract pair $p \in \mathcal{V}al\mathcal{P}$. p could be created by any tuple in $\mathsf{pc}^{-1}(p) = \{(l, v_1, v_2, k) \mid \mathsf{pc}(l, v_1, v_2, k) = p\}$. However, the log entry π_p conserves only the tuples that the analyzer has *effectively* encountered during the (maybe numerous) creations of p. These logs allow the analyzer to avoid being too conservative.

The analysis is *sound*, in the sense that the analyzer acts conservatively with the abstract values. That is, every concrete evaluation environment in which an expression e_l truly evaluates is modeled by abstract values in a certain abstract contour. Every concrete value that exists in the concrete evaluation is represented by an abstract value in the analysis results. The concrete value that is returned by a certain closure at a certain step in the concrete evaluation has an abstract counter-part that is returned by an abstract closure in a certain abstract step (the contour). And so on. The soundness property can be formally proven, but we do not do so in this paper.

Evaluation and safety constraints

Given a program and the instantiation parameters, our framework performs the analysis of the program using the *evaluation* constraints presented in Figure 6. Basically, a set of constraints on the analysis variables is generated for the program. Any solution to this set of constraints provides a valid analysis result. Naturally, we are always interested in the *least* solution to the system of constraints. A solution always exists because, despite the variety of the generated constraints, they can all be decomposed into basic constraints of the form: $v_1 \in \nu_{i_{1,1},\ldots,i_{1,n_1}} \wedge \ldots \wedge v_m \in \nu_{i_{m,1},\ldots,i_{m,n_m}} \Rightarrow v \in \nu_{j_1,\ldots,j_k}$. So the saturation of all analysis variables gives a trivial valid solution.

The evaluation constraints are quite standard and do not deserve much more explanation. Except maybe the maintenance of the log matrices. For example, each time a pair p is created at a `cons` expression, the tuple (l, v_1, v_2, k) representing the label of the expression, both values to pack in the pair, and the current contour is logged in the variable π_p. The logged tuples are later used by various computations to discover the origins of the abstract values. For example, the `car` expression uses the log π_p (and not $\mathsf{pc}^{-1}(p)$) to enumerate to values that may be found in the CAR field of the

Evaluation constraints for program e_{l_0} are:

$$\bigcup_{k \in \mathcal{C}ont} \mathcal{E}\ \llbracket e_{l_0} \rrbracket\ k\ \cup\ \{\delta_{l_0,k_0} \supseteq \mathcal{V}al\mathcal{B}\}\ ,\ \text{where}$$

$$\begin{array}{l}
\mathcal{E}\ \llbracket \texttt{\#f}_l \rrbracket\ k = \\
\quad \{\delta_{l,k} \neq \emptyset \Rightarrow \alpha_{l,k} \supseteq \mathcal{V}al\mathcal{B}\} \\
\mathcal{E}\ \llbracket \mathrm{x}_l \rrbracket\ k = \\
\quad \{\delta_{l,k} \neq \emptyset \Rightarrow \alpha_{l,k} \supseteq \mathrm{ref}(\mathrm{x}, l, k)\} \\
\mathcal{E}\ \llbracket \texttt{(}_l e_{l_1}\ e_{l_2}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k}, \delta_{l_2,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \mathcal{E}\ \llbracket e_{l_2} \rrbracket\ k \cup \\
\quad \left\{ \begin{array}{l|l} \beta_{\mathrm{x},k'} \ni v, & c \in \alpha_{l_1,k} \cap \mathcal{V}al\mathcal{C},\ v \in \alpha_{l_2,k}, \\ \alpha_{l,k} \supseteq \gamma_{c,k'}, & k' = \mathsf{call}(l, c, v, k), \\ \kappa_{k'} \ni (l, c, v, k) & (l', k'') \in \chi_c,\ e_{l'} = \texttt{(}\lambda_{l'}\texttt{x. } e_{l''}\texttt{)} \end{array} \right\} \\
\mathcal{E}\ \llbracket \texttt{(}\lambda_l\texttt{x. } e_{l_1}\texttt{)} \rrbracket\ k = \\
\quad \left\{\delta_{l,k} \neq \emptyset \Rightarrow \alpha_{l,k} \ni \mathsf{cc}(l,k) \wedge \chi_{\mathsf{cc}(l,k)} \ni (l,k)\right\} \cup \\
\quad \{\delta_{l_1,k} \supseteq \beta_{\mathrm{x},k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \\
\quad \{\gamma_{c,k} \supseteq \alpha_{l_1,k} \mid c \in \mathcal{V}al\mathcal{C}, (l,k') \in \chi_c\} \\
\mathcal{E}\ \llbracket \texttt{(if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \\
\quad \{\delta_{l_2,k} \supseteq \alpha_{l_1,k} \cap (\mathcal{V}al\mathcal{C} \cup \mathcal{V}al\mathcal{P})\} \cup \\
\quad \{\delta_{l_3,k} \supseteq \alpha_{l_1,k} \cap \mathcal{V}al\mathcal{B}\} \cup \mathcal{E}\ \llbracket e_{l_2} \rrbracket\ k \cup \\
\quad \mathcal{E}\ \llbracket e_{l_3} \rrbracket\ k \cup \{\alpha_{l,k} \supseteq \alpha_{l_2,k} \cup \alpha_{l_3,k}\} \\
\mathcal{E}\ \llbracket \texttt{(cons}_l\ e_{l_1}\ e_{l_2}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k},\ \delta_{l_2,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \mathcal{E}\ \llbracket e_{l_2} \rrbracket\ k \cup \\
\quad \left\{ \begin{array}{l|l} \alpha_{l,k} \ni p, & v_1 \in \alpha_{l_1,k},\ v_2 \in \alpha_{l_2,k}, \\ \pi_p \ni (l, v_1, v_2, k) & p = \mathsf{pc}(l, v_1, v_2, k) \end{array} \right\} \\
\mathcal{E}\ \llbracket \texttt{(car}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \\
\quad \left\{\alpha_{l,k} \ni v_1 \mid p \in \alpha_{l_1,k} \cap \mathcal{V}al\mathcal{P},\ (l, v_1, v_2, k') \in \pi_p \right\} \\
\mathcal{E}\ \llbracket \texttt{(cdr}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \\
\quad \left\{\alpha_{l,k} \ni v_2 \mid p \in \alpha_{l_1,k} \cap \mathcal{V}al\mathcal{P},\ (l, v_1, v_2, k') \in \pi_p \right\} \\
\mathcal{E}\ \llbracket \texttt{(pair?}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \\
\quad \{\delta_{l_1,k} \supseteq \delta_{l,k}\} \cup \mathcal{E}\ \llbracket e_{l_1} \rrbracket\ k \cup \{\alpha_{l,k} \supseteq \alpha_{l_1,k} \cap \mathcal{V}al\mathcal{P}\} \cup \\
\quad \{\alpha_{l_1,k} \cap (\mathcal{V}al\mathcal{B} \cup \mathcal{V}al\mathcal{C}) \neq \emptyset \Rightarrow \alpha_{l,k} \supseteq \mathcal{V}al\mathcal{B}\}
\end{array}$$

$$\mathrm{ref}(\mathrm{x}, l, k) \quad = \quad \left\{ \begin{array}{ll} \mathrm{ref}(\mathrm{x}, l', k), & \text{if } e_{l'} \neq \texttt{(}\lambda_{l'}\texttt{y. } e_l\texttt{)} \\ \beta_{\mathrm{x},k}, & \text{if } e_{l'} = \texttt{(}\lambda_{l'}\texttt{x. } e_l\texttt{)} \\ \cup_{k'} \mathrm{ref}(\mathrm{x}, l', k'), & \text{if } e_{l'} = \texttt{(}\lambda_{l'}\texttt{y. } e_l\texttt{)}, \\ & (l'', c, v, k'') \in \kappa_k, \\ & (l', k') \in \chi_c \\ \text{where } l' = \mathrm{parent}(l) & \end{array} \right.$$

Figure 6: Evaluation constraints

Safety constraints for program e_{l_0} are:

$$\bigcup_{k \in \mathcal{C}ont} \mathcal{S}\ \llbracket e_{l_0} \rrbracket\ k,\ \text{where}$$

$$\begin{array}{l}
\mathcal{S}\ \llbracket \texttt{\#f}_l \rrbracket\ k = \emptyset \\
\mathcal{S}\ \llbracket \mathrm{x}_l \rrbracket\ k = \emptyset \\
\mathcal{S}\ \llbracket \texttt{(}_l e_{l_1}\ e_{l_2}\texttt{)} \rrbracket\ k = \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{C}\} \cup \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \cup \mathcal{S}\ \llbracket e_{l_2} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(}\lambda_l\texttt{x. } e_{l_1}\texttt{)} \rrbracket\ k = \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3}\texttt{)} \rrbracket\ k = \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \cup \mathcal{S}\ \llbracket e_{l_2} \rrbracket\ k \cup \mathcal{S}\ \llbracket e_{l_3} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(cons}_l\ e_{l_1}\ e_{l_2}\texttt{)} \rrbracket\ k = \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \cup \mathcal{S}\ \llbracket e_{l_2} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(car}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{P}\} \cup \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(cdr}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{P}\} \cup \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k \\
\mathcal{S}\ \llbracket \texttt{(pair?}_l\ e_{l_1}\texttt{)} \rrbracket\ k = \mathcal{S}\ \llbracket e_{l_1} \rrbracket\ k
\end{array}$$

Figure 7: Safety constraints

pair p. Finally, note that the extra constraint $\delta_{l_0,k_0} \supseteq \mathcal{V}al\mathcal{B}$ is added to ensure that the evaluation of the program *gets started*.

The reader may have noted that the evaluation constraints do not take errors into account and manipulate only the values that are legal. This is because we separate the evaluation constraints from the *safety* constraints. Figure 7 presents the safety constraints that are generated for a program e_{l_0}. These constraints are straightforward. The reason we keep these separated is that once we add the safety constraints to the set of evaluation constraints, there may be no solution to the system. If there is a solution to the joined sets of constraints, that means that the model (the parameters) provides a proof that the program is type-safe.

The usual way to analyze a program is to solve the system of evaluation constraints, which leaves the analysis results in the analysis variables, then confront the results to the safety constraints, and see which constraints are violated. The latter indicate where dynamic type tests are required. For example, the violation of the constraint $\alpha_{l',k} \not\subseteq \mathcal{V}al\mathcal{C}$ for a certain sub-expression $e_{l'}$ (whose parent is a call expression e_l) and contour k, indicates that there must be a dynamic test at e_l to ensure that the result of $e_{l'}$ is indeed a closure[6].

Power and genericity of the framework

The parameterization of the framework allows it to be a very powerful analysis tool. Here are some of its characteristics. We do not give proofs here, though.

- The parameters representing a model, as little constrained as they might be, are still finitely representable. One might ask whether it is possible to automatically decide whether there exists a model that allows the analyzer to demonstrate that a program is type-safe. Unfortunately, this problem is undecidable; it is possible to reduce the termination problem to this one.
- For every program that terminates normally, there exists a model that demonstrates that it is type-safe. A

[6]This explanation assumes that there is only one call expression e_l generated by the compiler in the executable code. This assumption may be too simplistic. A good optimizing compiler may generate more than one call expression instance of e_l, each corresponding to a contour (or to many). In this case, the instances associated to contours where no violation occurs do not require a dynamic type test. However, the topic of producing good executable code from analysis results is beyond the scope of this paper.

trivial model that does so consists in mimicking the concrete evaluation of the program. It introduces one abstract value for each concrete value. However, it is generally impossible to know that the program terminates normally, in the first place.

- For every program that terminates with an error, *all* models lead to a violation constraint. This is due to the soundness of the analysis. Unfortunately, an unsuccessful model attempt generally does not bring any information as to whether the program must *necessarily* terminate with an error.

- Among the programs that loop, some have a model proving they are type-safe, some do not. Note that they *are* type-safe. We believe that an important limitation to the power of the framework concerns program constructs where the safety depends on some mathematical invariant. Generally, this cannot be described by our kind of models.

The liberty in the choice of the framework parameters allows this one to simulate many traditional analyses. For example, call-string contours as in [15] can be easily imitated by a proper definition of call. Basic set-based analysis [9], being equivalent to the 0-cfa can be imitated, too.

The contours presented in [13] are based on *polymorphic splitting*. Values created in let-bindings can be specialized according to which variable *reference* accesses the values. Simply stated, an abstract value bound to a variable in a let-binding mutates differently depending on where the reference to the variable is located. Our framework does not allow such a thing. However, a trivial source-to-source transformation of the program and appropriate model selection make it possible to obtain a similar analysis.

3 Demand-driven analysis

Here is an informal introduction to demand-driven analysis. First, we illustrate the approach with an example. Then, an overview of what a complete demand-driven approach should include is presented. Next, the difficulty of dealing with the call and conditional expressions is exposed. Finally, many challenges to make a demand-driven approach work are mentioned.

3.1 An example

To illustrate what demands might be, where they come from, and how they can be processed, we use a small example. Suppose that this λ-expression appears somewhere in a program:

$$(\lambda_1\texttt{x. (if}_2\ \texttt{x}_3\ \texttt{(car}_4\ \texttt{(pair?}_5\ \texttt{x}_6\texttt{))}\ \texttt{\#f}_7\texttt{))}$$

Suppose also that a preliminary analysis has been done and that, according to its results, the λ-expression eventually gets evaluated, resulting in a closure, and that the closure is called many times with different pairs and with #f.

Note that a naïve compilation of λ-expression e_1 would immediately produce good code except for the (only) dynamic test coming from the `car` expression. It would be better if we could remove that test. Let us see how this can be done. We need to prove that e_5 returns nothing else than pairs. Now, as far as the preliminary analysis of the program can tell, e_5 can evaluate to pairs and #f (remember that, when e evaluates to a pair or to a non-pair, `(pair? e)` evaluates to that pair or to #f, respectively). So, for the moment, the dynamic test must stay there. In order to try to change this, we will emit and process demands. These, in turn, may lead to an update of the model such that it will create an instance of an analysis that can provide the desired proof.

Obviously, we need a first demand. Why not go for the simplest solution? That is, make the following request: "show that e_5 always evaluates to pairs". Or more precisely: "show that e_5 cannot evaluate to anything else than pairs". To show that it does not get evaluated at all would not be bad, too. Let us call this demand D_1.

Now, we have to process D_1 in some way. Note that D_1 concerns e_5, which is a not a simple expression. The value of e_5 strongly depends on the value of its sub-expression e_6. If we could rewrite D_1 into another demand related to the simpler e_6, we would have made some progress. This new demand D_2 could be: "show that e_6 cannot evaluate to anything else than pairs". Clearly, D_2, if it is positively answered, would have the same desirable consequences as D_1.

What can we do to respond to D_2? Note that the preliminary analysis says that x may be bound to #f (and suppose that it is truly the case). A reasonable approach is to process D_2 in two steps: first, we should separate the case where x is bound to a pair from the case where it is not; then, if x still evaluates to #f in the second case, we should request a demonstration that that evaluation cannot happen. What it means is that we emit a new demand D_3 and then, if necessary, another new demand D_4.

Let us first take care of D_3. In more precise terms, D_3 is: "split the current contour in order to separate the cases where x is bound to a pair from the cases where x is bound to #f". That is perfectly possible, as we explain shortly. So let us consider that the previous contour k has been effectively split into k' (pair case) and k'' (#f case). There is certainly no more problems with the evaluation in contour k' since x must be bound to a pair, so e_5 must return a pair, and so e_4 cannot go wrong. But what about evaluation in contour k''? Since x must be bound to #f in k'', the test in e_2 is always false, the then-branch is never executed and, consequently the CAR access is never made. Conclusion: in every case, there is no need to perform a dynamic type test in e_4. The initial demand has been positively answered. That is, we have emitted demands, processed them, and they have lead to an update of the model that was sufficient to demonstrate that the dynamic test is unnecessary.

Before we conclude this example, we need to explain why we said that it was easy to separate the cases where x is bound to a pair from the cases where x is bound to something else. This is because of our abstract model. The call parameter selects the contour in which the body of a closure evaluates. Let us refer to the closure generated by e_1 as c and to the contour in which the body evaluates as k in the old model. To keep things simple, we suppose that they are unique. That means that $\mathsf{call}(l, c, v, k^*) = k$ for any label l, argument v, and contour k^*. In other words, when c is called, its body is always evaluated in the contour k. Changing the model to make the required split simply means defining a new modeling function call' such that:

$$\forall l \in \mathrm{Lab}, v \in \mathcal{V}al, k^* \in \mathcal{C}ont,$$
$$\mathsf{call}'(l, c, v, k^*) = \begin{cases} k', & \text{if } v \in \mathcal{V}al\mathcal{P} \\ k'', & \text{otherwise} \end{cases}$$

3.2 Overview

As we mention in the introduction, our demand-driven analysis should be able to produce some results in a short time, if necessary, and be able to improve them continuously if it is allowed to continue longer. In order to behave this way, the demand-driven analyzer proceeds in two phases: the preliminary analysis, the demand-driven phase; as sketched in the introduction. The preliminary analysis is similar to a traditional analysis; its purpose is to collect initial information using a static model. Typically, this information is good enough to allow the removal of many dynamic type test but not all of them. During the demand-driven phase, demands are generated and processed in order to perform the model-update/re-analysis phase. This phase continues until all the demands have been positively answered or, usually, until the bound on the analysis time is reached.

The choice of the model in the preliminary analysis is what we discuss first. Next, we present a list of demands that seem vital to guide the demand-driven analysis. Finally, we present typical processing of many kinds of demands.

Initial model and initial demands

The choice of the initial model must be the result of a compromise between the time spent during the preliminary analysis and the quality of the preliminary analysis results. A model that is too complex will make the preliminary analysis costly, making even the fastest compilation *with* analysis too long. A model that is too coarse may render the preliminary analysis "blind", its results sometimes being overestimated to the point of being useless, thus leaving the whole task of real analysis to the demand-driven part, which is necessarily less efficient.

We believe that having an initial model with one abstract pair, one abstract closure per λ-expression, and one contour (or one contour per closure body) would be of a reasonable cost and provide preliminary analysis results of relatively good quality. Such a model instantiates a monovariant analysis that is comparable to the 0-cfa. Since, in the typical case, analyses like 0-cfa perform relatively well, much fewer demands are generated in the demand-driven part.

Choosing a coarser model having only *a single* abstract closure to represent all concrete closures would lead to excessively poor results. Except in the most trivial of cases, the abstract closure would be found to return everything, leaving all the analysis work to the demand-driven part. The only advantage of this choice would be a preliminary analysis with linear complexity.

Choosing a finer model would increase significantly the preliminary analysis time without any guarantee as to whether the *a priori* refinements would bring any help for the dynamic tests that would still remain after a 0-cfa-style analysis.

Once the preliminary analysis is done, formulating the initial demands is trivial. Expressed in terms of analysis variables, it takes the form of a list of "show $\alpha_{l,k} \subseteq \mathcal{V}al\mathcal{C}$" and "show $\alpha_{l',k'} \subseteq \mathcal{V}al\mathcal{P}$" demands.

Typical demands

The most natural demand type is like the initial demands. We shall call these *bound-demands.* Since they can so easily be reformulated in terms of other, more fundamental demands, bound-demands only involve an α matrix entry and a simple bound set.

One of these fundamental demand types is the *split-demand.* We mentioned that kind of demand in the example in Section 3.1. It says: "split *something* according to *something*". The thing to split may be an abstraction in the model or an analysis variable. As an example of the first case, the demand could be "split P according to the label where it is created", where P is the unique abstract pair. That would trigger a straightforward update of the function `pc`. As an example for the second case, the demand could be "split $\gamma_{c,k}$ according to the membership to the set of pairs". That means that the return values of closure c when its body is evaluated in contour k have to be split into pairs and non-pairs. The consequences in case of a successful response are that the abstract closure c, the contour k, and anything else if necessary will have to be split in such a way that for all c' specializing c and for all k' specializing k, $\gamma_{c',k'}$ will contain either only pairs or no pair at all.

Split-demands directly on the model or on α, β, or γ entries are reasonable demands. However, we believe that split-demands on δ entries should not exist since they make no sense. The only interesting concern with δ entries is whether they are empty or not. As for the "log" variables, it may make sense to *want* to split them, but maybe not to *try.* This is because they plainly describe how the abstraction functions have been used during the analysis. They have a very indirect (and passive) effect on the abstract evaluation of the program. In order to have an effect on these entries, a demand would certainly have to be reformulated in terms of demands concerning entities on which it is clear that we can have an effect.

A third type of demand consists in requesting a demonstration that some expression cannot get evaluated in some contour. We shall call these *never-demands* and obviously they can be formulated formally by "show $\delta_{l,k} = \emptyset$". Such demands typically arise when a certain evaluation *necessarily* leads to an error. To make a variation on the example of Section 3.1, if the sub-expression e_5 of the expression (`car`$_4$ e_5) in the non-pair contour k'' would have still returned some values, it would have been necessary to emit a never-demand on $\delta_{5,k''}$. Obviously, a split is not necessary since there are no good cases (pairs) to separate from the bad cases (non-pairs).

We touch a crucial issue, here: good cases and bad cases. When there are only good cases, everything is fine, nothing has to be done. When there are only bad cases, we have to emit a never-demand but at least everything is clear. When there are good and bad cases together, normally split-demands have to emitted before emitting never-demands. Otherwise, if we are asking a demonstration that such an evaluation cannot occur, we may ask the impossible since the good cases may reflect actual concrete evaluations in the program. This principle must be kept in mind when we propose processing techniques for the demands.

A fourth type of demand that seems vital is the no-call-demand. A no-call-demand basically means: "show that closure c cannot be called on argument v in call site l when the contour is k". It typically may be emitted due to the processing of a never-demand. To continue with our variation on the example, a never-demand on $\delta_{5,k''}$ may eventually require that we show that the closure c is never called with a non-pair argument. This translates into one or more no-call-demands.

Although an implementation of demand-driven analysis may formulate other types of demands, the ones that we just presented here form a core that must be present in one way or another in order to be able to perform demand-driven type analysis on a language such as ours.

Processing the demands

Demands originally express the need to demonstrate a "desirable" property. A demonstration takes the form of a model instantiating a particular analysis that brings the proof that the property is indeed true. If we want to go from the original demands to the appropriate model, these demands have to be processed in some way. Note that we have already implicitly described many cases of demand processing.

In general, processing a demand leads to immediate success, to immediate failure, or to emission of new, modified demands. Immediate success occurs when, for example, the demand is "split $\alpha_{l,k}$ according to its type" and the expression e_l is `#f`$_l$. In this case, the model trivially conforms to the demand: k itself is the only contour that is necessary in order to have that no two objects of different type result from the evaluation of e_l in the same contour k', for any k' specializing k.

Immediate failure, in our particular case, is most uncommon. One specific demand, however, can lead to immediate failure: "show $\delta_{l_0,k_0} = \emptyset$". That is, trying to show that the whole program does not get evaluated.

Most of the time, as was illustrated in the example of Section 3.1, processing a demand leads to the creation of new demands.

Even though particular demand-driven analyses may differ in the way their set of demands are processed, here we present processing schemas that, almost certainly, have to be similar in all cases.

- Original bound-demands, "show $\alpha_{l,k} \subseteq \langle\text{set}\rangle$", express properties that, if they are not trivially satisfied nor trivially contradicted, may first be re-expressed as a split-demand and, upon success of this first sub-demand, a never-demand ought to be emitted for each k_i specializing k such that $\alpha_{l,k_i} \not\subseteq \langle\text{set}\rangle$. Note that the split is intended to "separate the good cases from the bad ones". If the bound-demand property is trivially respected, immediate success occurs. If it is trivially contradicted, a single new demand is emitted: "show $\delta_{l,k} = \emptyset$".

- Split-demands on β entries result in an update of the model and in immediate success. Since only the cc and call functions determine which contour is selected depending, in particular, on the arguments to the closures, a model update is the only way to respond to such demands. Of course, any split-demand directly concerning the model causes an update of the model and an immediate success.

- A split-demand on a $\gamma_{c,k}$ analysis variable can trivially be reformulated in terms of a new split-demand on the $\alpha_{l,k}$ variable corresponding the result of the body e_l of the closure c.

- A split-demand on an $\alpha_{l,k}$ variable where e_l is `#f`$_l$, x_l, `(cons`$_l$ $e_{l'}$ $e_{l''}$`)`, `(car`$_l$ $e_{l'}$`)`, `(cdr`$_l$ $e_{l'}$`)`, or `(pair?`$_l$ $e_{l'}$`)` can normally be processed in a straightforward fashion. It becomes, in the first case, an immediate success, since it is clear that the sole #f value always falls into a single "split category" according to the split criterion. In the second case, it can trivially be reformulated as a split-demand on $\beta_{\text{x},k}$. In the third case, depending on the split criterion, the split may already be done (with a split-on-type criterion, for example) or it may easily be reformulated in terms of split-demands on the sub-expressions. The remaining cases are similar plus, maybe, a direct split-demand on the model to specialize abstract pairs. To make a simplistic observation, we would say that split-demands on α entries have a tendency to propagate from an expression towards its sub-expressions.

- A never-demand on a $\delta_{l,k}$ variable is processed accounting for the parent expression $e_{l'}$ of e_l. Most of the time, the demand is reformulated into a never-demand on $\delta_{l',k}$. However, if e_l is the consequent branch or the alternate branch of an `if` expression, the demand must be reformulated into a bound-demand onto the *test* sub-expression. The bound is the set of true values ($\mathcal{V}al\mathcal{C} \cup \mathcal{V}al\mathcal{P}$) or false values ($\mathcal{V}al\mathcal{B}$), respectively. Finally, if $e_{l'}$ is a λ-expression, the evaluation is the result of a call, and it is generally not a simple matter to process such a demand. Once again, to be simplistic, we could say that never-demands have a tendency to propagate from an expression towards its parent expression.

3.3 Difficult cases

In the preceding paragraphs, we presented some more or less precise descriptions of what the processing of various demands should be. However, we avoided certain demands deliberately because they are clearly difficult to process. The existence of difficult cases has to be expected since statically proving interesting properties about a program is uncomputable in general, and this uncomputability is not going to disappear simply because we are trying to make the analyzer smarter by using a demand-driven approach. The difficulties come mainly from the conditional expression and, to a greater extent, from the call expression. We illustrate the potential problems with two examples.

Let us suppose that we have the following expression: `(if`$_l$ e_{l_1} e_{l_2} e_{l_3}`)`. We must process a split-demand on $\alpha_{l,k}$ according to the type of the result. Let us suppose, also, that the analysis results under the current model indicate that e_{l_2} may evaluate to objects of all types, that e_{l_3} may evaluate only to pairs, and that e_{l_1} may evaluate to both true and false values. How can we process this demand?

Note that e_l evaluates to a set of values that is the union of the results of both its branches. Since e_{l_3} already has a pair-only result, we could emit a bound-demand on e_{l_2} to request a demonstration that in fact it evaluates only to pairs. Alternatively, we could emit a bound-demand on e_{l_1} to request a demonstration that it evaluates only to #f. Which strategy is the best?

Obviously, the example shows that the difficulty comes from the fact that there are more than one possible direction to continue processing. Moreover, note that neither of the two proposed demands is adequate because they may involve properties that flatly contradict what the concrete computations are. In such cases, there would be no hope of ever responding successfully to the demands.

The processing of demands concerning calls is even more difficult. Let us consider the following expression: `(`$_l e_{l_1}$ e_{l_2}`)`. Suppose that the demand is the same as in the `if` example. Also, suppose that the current analysis results tell us that: e_{l_1} evaluates to two closures c_1 and c_2, e_{l_2} evaluates to objects of more than one type, the closure c_1 returns objects of only one type, and c_2 returns objects of different types. How should we proceed?

The "poly-type" results of e_l may be explained by the fact that: c_2 returns objects of the same type as those that it receives, so we should split the value of e_{l_2}; the concrete closure corresponding to c_2 returns "mono-type" results, but its poor modeling suggests the contrary, so we should split its return value; or no concrete closure corresponding to c_2 is ever present at e_{l_1} during the concrete evaluation, so we should split the value of e_{l_1}, and demand that the case where e_{l_1} evaluates to c_2 be proved impossible.

Clearly, processing in such a case is far from obvious since the appropriate demands may concern e_{l_1}, e_{l_2}, the closures that are invoked, or a combination of the three.

3.4 Challenges

On top of the natural difficulty that comes with the processing of the demands, there are several others that make things more complex.

As we mentioned above, the processing of a demand and its sub-demands may last forever. This may happen in particular because the property that must be demonstrated is not based on legitimate reasons (such as in the conditional expression example) or simply because it is beyond the power of the framework to support the necessary proof. Clearly, there must be a mechanism that ensures that the analyzer does not get stuck in such processing.

Since the attempt to prove a property may last forever and there are generally more than one property to prove, the original demands cannot be processed one after the other. The amount of time available to the analysis may be exhausted by one of the first demands, possibly leaving unanswered many "easy" demands that would have been successfully processed in little time. So the processing of the demands must be made using some kind of concurrency.

Note that using a bound on the time available to the analyzer is clearly a necessity but it is also one of its feature. Although unusual in the field of program analysis, this concept is fairly natural when we think about it. In a way, it corresponds more to the human notion of work than to the algorithmic complexity notion of work. While the ideal parameter to an analyzer would be the quality of the results, a bound on the available time is probably the closest realistic equivalent.

The processing of an original demand naturally leads to a *tree* of sub-demands. Of course, these sub-demands cannot all be processed at the same time. Some have to be put into a waiting queue until it is their turn. However, during the time that a demand is in the queue, the model may have been refined due to the processing of other demands. In such a case, an "old" demand may refer to abstractions that have been broken down into more specialized abstractions. Consequently, there must be a mechanism to keep demands up to date.

Finally, an important question relates to the concurrent demand processing: should the various processing trees share the abstract model? Remark that they do not have to. Each original demand can be responded independently. This is because that what matters is which of the original demands are successfully answered. Two distinct dynamic tests may both be omitted from a program, even if each has been showed redundant using a distinct model.

The advantage of sharing the model is that a successful demonstration of property A may have uncovered many invariants of the program that would make the demonstration of property B easier. The inconvenience is that if all updates occur in the same model then almost every demand that goes in the waiting list has to be specialized to follow the numerous finer abstractions introduced by the updates, resulting in a proliferation of demands.

A compromise that may be interesting consists in sharing each model with the one used to successfully answer a demand. What is interesting with such a model is that it can be *reduced* prior to the sharing with the other models. The idea is the following: during the processing of a tree of demands, all kinds of updates are performed on the model; eventually, one last update causes the model to provide a proof for the original demand; however, only some of the refinements to the model are really necessary to provide the proof; undoing the unnecessary refinements produces a model that is as small as possible.

4 A basic analysis

We present a prototype of a demand-driven analysis that is based on *patterns*. We briefly describe this pattern-based modeling and some of the choices that we have made concerning the various problems that must be addressed.

4.1 Abstract model

The modeling of the abstractions is made using pattern matching. A pattern list must be exhaustive and, associated with each pattern, there is a particular abstraction instead of code to execute. For example, a very simple pattern matcher describing the abstract pairs might look like:

$$\begin{array}{lrlcl} (& \#\mathrm{f}, & \mathcal{V}al) & \Rightarrow & P_1 \\ (& \lambda_\forall, & \mathcal{V}al) & \Rightarrow & P_2 \\ (& (\mathcal{V}al, \mathcal{V}al), & \mathcal{V}al) & \Rightarrow & P_3 \end{array}$$

Obviously, it represents three abstract pairs, each being specialized with the type of the object that it contains in the CAR field.

One important characteristic of our pattern matching is that it does not require that a modeling of pairs, for example, has to be the *Cartesian product* of all the specializations found in the CAR with those found in the CDR. This is crucial for the patterns representing contours since these are kinds of "lists" that can be as long as the lexical environment in the program.

4.2 Demands

Figure 8 presents the syntax of the demands and that of the patterns they include. The set of demands corresponds basically to what we describe in Section 3 except for *split-call*, which is an auxiliary demand used in the processing of split-demands on call expressions, and *monitor-call*, which is another auxiliary demand that tries to prove that calls of certain closures on certain arguments cannot occur in certain contours.

The syntax of the patterns is described by ⟨pat⟩, which represents the *splitting* patterns, by ⟨sPat⟩, which are the *static* patterns, by ⟨ctPat⟩, which are splitting contour patterns, and by ⟨sCtPat⟩, which are static contour patterns. A splitting pattern contains one and only one splitting point, indicated by $\star$. When abstractions are split according to a pattern, only those that match the pattern are modified, and the modification consists in adding an "extra-level" of inspection at the splitting point. Static patterns are used to help describing the abstractions that are to be modified.

$$\begin{array}{lcl}
\langle\text{demand}\rangle & := & \textit{show}\ \alpha_{l,k} \subseteq \langle\text{bound}\rangle \\
 & & \textit{split}\ \alpha_{l,k}\ \langle\text{pat}\rangle \\
 & & \textit{split}\ \beta_{\text{x},k}\ l\ \langle\text{pat}\rangle \\
 & & \textit{split}\ \gamma_{c,k}\ \langle\text{pat}\rangle \\
 & & \textit{split}\ \mathcal{V}al\mathcal{P}\ \langle\text{pat}\rangle \\
 & & \textit{show}\ \delta_{l,k} = \emptyset \\
 & & \textit{split-call}\ l\ \langle\text{sCtPat}\rangle\ \langle\text{pat}\rangle \\
 & & \textit{monitor-call}\ l\ \langle\text{sCtPat}\rangle \\
\langle\text{bound}\rangle & := & \mathcal{V}al\mathcal{B} \mid \mathcal{V}al\mathcal{C} \mid \mathcal{V}al\mathcal{P} \mid \mathcal{V}al\mathcal{T}rues \\
\langle\text{pat}\rangle & := & \star \mid \lambda_\star \mid \lambda_l\ \langle\text{ctPat}\rangle \\
 & & (\langle\text{pat}\rangle,\ \langle\text{sPat}\rangle) \mid (\langle\text{sPat}\rangle,\ \langle\text{pat}\rangle) \\
\langle\text{sPat}\rangle & := & \mathcal{V}al \mid \#f \mid \lambda_\forall \mid \lambda_l\ \langle\text{sCtPat}\rangle \\
 & & (\langle\text{sPat}\rangle, \langle\text{sPat}\rangle) \\
\langle\text{ctPat}\rangle & := & (\langle\text{sPat}\rangle^*\ \langle\text{pat}\rangle\ \langle\text{sPat}\rangle^*) \\
\langle\text{sCtPat}\rangle & := & (\langle\text{sPat}\rangle^*)
\end{array}$$

Figure 8: Demand syntax

The contours used at an expression e_l are an abstract model of the lexical environment. So contour patterns are lists of patterns that are as long as the lexical environment is at the points of the program where they are used.

4.3 Back to the example

If we return to the example of Section 3.1, a pattern-based demand-driven analysis proceeds like this. The original demand is:

$$\textit{show}\ \alpha_{5,k} \subseteq \mathcal{V}al\mathcal{P}$$

where k represents $(\mathcal{V}al\ \ldots\ \mathcal{V}al)$. That demand is first reformulated into a split-demand according to the type:

$$\textit{split}\ \alpha_{5,k}\ \star$$

Processing this demand is trivial and it produces another split-demand. It concerns the sub-expression:

$$\textit{split}\ \alpha_{6,k}\ \star$$

This one becomes a split-demand on the variable:

$$\textit{split}\ \beta_{\text{x},k}\ 6\ \star$$

The label 6 is present in order to unambiguously indicate which program point requires an update. This is because k may be used in more than one function body. This demand finally causes an update in the model of the call function in such a way that a call to the closure can result in the contour

$$\begin{array}{lcl}
(& \#\text{f} & \mathcal{V}al\ \ldots\ \mathcal{V}al), \\
(& \lambda_\forall & \mathcal{V}al\ \ldots\ \mathcal{V}al),\ \text{or} \\
(& (\mathcal{V}al, \mathcal{V}al) & \mathcal{V}al\ \ldots\ \mathcal{V}al).
\end{array}$$

The rest of the explanations are similar.

4.4 The difficult cases

In Section 3, we showed that the difficult cases are the conditional expressions and the call expressions. Also, we list many other difficulties. We present some choices that we made in our pattern-based analysis.

A split-demand on the evaluation results of a conditional expression are dealt with in this way: split-demands with the same pattern are sent to both branches and another split-demand with the $\star$ pattern is sent to the test. With luck, all three sub-demands succeed, and the split-demand on the conditional is a success since each new contour necessarily leads to mono-type evaluation results of the conditional.

A split-demand on a call expression proceeds by: splitting the return value of each closure (that may be involved there) according to the same pattern; this processing indirectly creates an "association" between the output and the input of the closures; a *split-call* auxiliary demand then computes an "easiest" way to distinguish call situations that lead to different split categories; it finally emits a sequence of demands on the sub-expressions of the call expressions that, if successfully answered, would complete the split of the call expression.

These processing strategies are generally too aggressive in their generated sub-demands and a major difficulty is to deal with those that do not succeed. We have included a time-out feature to the processing of sub-demands that allows their parent to turn to a "backup plan" when the time-out is reached. The backup plans often resort to sub-demands that are often less legitimate than the ones that have expired and, so, maybe even more susceptible to be impossible to acknowledge or at least more difficult. But, as the name of these plans says, this is the last recourse.

4.5 Pros and cons

The pattern-based demand-driven analysis has the advantage of being of manageable complexity. That is why we have chosen it as a first attempt of demand-driven analysis. However, it has some weaknesses that may considerably reduce the power of the whole analysis. Its weaknesses come directly from its concept: patterns. Patterns can only distinguish object structures on the *surface* or not very deep. They are fundamentally incapable of distinguishing structures that start to differ at deep levels, such as, for example, lists of booleans ending with a boolean and lists of booleans ending with a function:

$$\begin{array}{c}
(\#\text{f}, (\#\text{f}, \ldots (\#\text{f}, \#\text{f}) \ldots)) \\
(\#\text{f}, (\#\text{f}, \ldots (\#\text{f}, \lambda_\forall) \ldots))
\end{array}$$

However, we cannot say that the pattern-based is just good enough to "show in greater detail that we still know nothing". If the program manipulates data structures that can be distinguished by looking only a few levels deep, then our analysis has the capability to find the characteristics of these data structures. Figure 9 shows such an example. Suppose that the program manipulates only lists of booleans and lists of functions. Then a simple split of the abstract pairs may lead to a perfect description of the lists. This is due to the log analysis variables, which record the circumstances that prevail when abstract objects are created. The figure shows two models, the coarse and the finer, and the information that is consigned in the logs.

5 Conclusion

5.1 Related work

As far as we know, there is no work with the same goal. The most closely related research is the work of Duesterwald et al. [7], Agrawal [1], and Heintze and Tardieu [10]. In [7], a framework to obtain a demand-driven analysis from a certain class of inter-procedural data-flow problems is described. However, as the authors of [10] mention, this class

$$\begin{array}{ccc}
\textsc{Model} & & \textsc{Observed results} \\
(\mathcal{V}al, \mathcal{V}al)_P & \overset{\text{analysis}}{\Rightarrow} & \left(\begin{array}{c}\#f \\ f \in \mathcal{V}al\mathcal{C}\end{array}, \begin{array}{c}\#f \\ P\end{array}\right)_P \\
\Downarrow_{split\ \mathcal{V}al\mathcal{P}\ (\star,\ \mathcal{V}al)} & & \\
\begin{array}{l}(\quad \#f \quad, \mathcal{V}al\ )_{P_1} \\ (\quad \lambda_\star \quad, \mathcal{V}al\ )_{P_2} \\ (\,(\mathcal{V}al, \mathcal{V}al)\,, \mathcal{V}al\ )_{P_3}\end{array} & \overset{\text{analysis}}{\Rightarrow} & \begin{array}{l}\left(\quad \#f \quad, \begin{array}{c}\#f \\ P_1\end{array}\right)_{P_1} \\ \left(f \in \mathcal{V}al\mathcal{C}, \begin{array}{c}\#f \\ P_2\end{array}\right)_{P_2} \\ (\quad \emptyset \quad, \quad \emptyset \quad)_{P_3}\end{array}
\end{array}$$

Figure 9: A simple split may uncover more complex structures

is restricted and does not even include the problem that they address: a flow-insensitive, context-insensitive pointer analysis; which is still elementary. In [1], a demand-driven data-flow analysis that does not require prior call graph information to be present is described.

What these proposals have in common with ours is the fact that demands are generated for some reasons and then propagated. That is all. Their goal is simply to take a well-known, traditional analysis and adapt it so that only a subset of the computations need to be performed in order to provide answers to certain requests. Only a few, when not only one, very simple demand types exist.

5.2 Future work

Investigation in this research should consider alternatives to patterns for abstract modeling and formulating demands. First, we should consider distinguishing pairs by their creation expression and contour rather than by their direct contents. We believe that this modeling may be more powerful than the pattern-based modeling. However, it is not clear how to express demands in this representation. Second, a representation using regular trees (see [4, 3, 5, 6]) systematically may prove to be very powerful. This representation could be used to express demands, too. It may be far from efficient, though. Third, we should explore an approach to systematically compute demands that is reminiscent of logic programming. The idea is to give a demand-driven analysis interpretation to the expressions. This interpretation is a function transforming demands (in the sense of bound-demands) to environment demands. The advantage of this approach seems to be the fact that it is systematic but it is not clear if it can be more powerful than pattern-based analysis.

The biggest problem with our approach is the processing of demands related to conditional and call expressions. Additional informations about facts that are known with *certainty*, might help to better decide what sub-demands to emit. For example, it could indicate that certain demands *have* to fail because a counter-example has been found. The certain facts would have to be discovered by an auxiliary analysis. The latter would concentrate on trying to prove facts that would help the most the demand-driven analyzer.

In a more complex application than our type analysis for a mini-language, original demands could come from a wider variety of hints. In consequence, it may be necessary to assign a reliability degree to the demands. For example, if we extend our problem to include detection of inlining opportunities, then it would be "desirable" to prove that a certain call expression can only invoke one particular closure. Since such a demand originates from a desire on our part and is not backed by any more solid evidence, then it should receive a lower reliability degree.

Another direction consists in extending the scope of the analysis to be able to deal with a language closer to Scheme, that is, including more algebraic types, higher- and variable-arity functions, continuations, I/O, and side-effects. Except for continuations, we do not expect any serious problems. Dealing with continuations probably requires that we introduce a new type of abstract objects since ordinary functions cannot mimic their behavior. Otherwise, a conversion to CPS may be required. Separate compilation of programs is not a standard part of Scheme, but it is common practice. Unfortunately, we do not see how our demand-driven approach could be adapted to deal with it. Not only does our analysis has to propagate abstract objects everywhere in the program, it also has to propagate demands everywhere (from callee to caller, for example, which may come from different modules).

Finally, other analyses than type analysis should be considered in order to verify how well our demand-driven approach applies outside of type analysis. One such analysis is range analysis for numerical values. A part of the goal of this analysis consists in removing bound checks in indexable data structure accesses and removing verifications before divisions and other unsafe numerical operations. Since these operations relate to safety issues, they can be seen as good hints from which we can generate initial demands.

5.3 Contributions

In this paper, we presented a proposal of how to perform a high-quality type analysis while trying to have a moderate time and space complexity. It is based on a demand-driven analysis that uses a very powerful analysis framework. The flexibility of the framework comes from the fact that the abstract model of the objects can be changed dynamically. With appropriate models, the framework can emulate the behavior of many traditional type analyses. Although the way to generate initial demands from hints present in the program is similar to what is done in other research, the purpose of the demands is radically different. Their generation and processing guides the successive updates of the analysis model that is used in the flexible framework, making successive analysis instances that are better equipped to analyze the program at hand. We also give a sketch of our implementation of a pattern-matching demand-driven analysis.

References

[1] G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proceedings of International Conference on Software Maintainance*, pages 453–462, sep 1999.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Functional Programming and Computer Architecture*, pages 427–447, aug 1991.

[4] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In ACM, editor, *POPL*

'91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL, pages 279–290, 1991.

[5] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, jun 1993.

[6] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, mar 1983.

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Symposium of Principles of Programming Languages*, pages 37–48, jan 1995.

[8] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, 1993.

[9] N. Heintze. Set based analysis of ML programs (extended abstract). Technical Report CS-93-193, Carnegie Mellon University, School of Computer Science, jul 1993.

[10] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, jun 2001.

[11] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363, 1986.

[12] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages*, pages 392–401, jan 1995.

[13] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. *Lecture Notes in Computer Science*, 854:207–224, 1995.

[14] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, jun 1988.

[15] O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 190–198, jun 1991.