A Demand-Driven Adaptive Type Analysis

Danny Dubé and Marc Feeley
Département d'Informatique et Recherche Opérationelle
Université de Montréal
{dube,feeley}@IRO.UMontreal.CA

Abstract

Compilers for dynamically and statically typed languages ensure safe execution by verifying that all operations are performed on appropriate values. An operation as simple as `car` in Scheme and `hd` in SML will include a run time check unless the compiler can prove that the argument is always a non-empty list using some type analysis. We present a demand-driven type analysis that can adapt the precision of the analysis to various parts of the program being compiled. This approach has the advantage that the analysis effort can be spent where it is justified by the possibility of removing a run time check, and where added precision is needed to accurately analyze complex parts of the program. Like the *k*-cfa our approach is based on abstract interpretation but it can analyze some important programs more accurately than the *k*-cfa for any value of *k*. We have built a prototype of our type analysis and tested it on various programs with higher order functions. It can remove all run time type checks in some nontrivial programs which use `map` and the Y combinator.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Languages

Keywords

Demand-driven analysis, static analysis, type analysis

1 Introduction

Optimizing compilers typically consist of two components: a program analyzer and a program transformer. The goal of the analyzer is to determine various attributes of the program so that the transformer can decide which optimizations are possible and worthwhile. To avoid missing optimization opportunities the analyzer typically computes a very large set of attributes to a predetermined level of detail. This wastes time because the transformer only uses a small subset of these attributes and some attributes are more detailed than required. Moreover the transformer may require a level of detail for some attributes which is higher than what was determined by the analyzer.

```
(let ((f (lambda (a b) (cons₁ (car₂ a) b)))
      (i (lambda (c) c)))
  (let ((j (lambda (d) (₃i d))))
    (car₄ (f (f (cons₅ 5 '())
                (cons₆ 6 '()))
             (₇i (cons₈ 8 (₉i '())))))))
```

Figure 1. A Scheme program under analysis

Consider a compiler for Scheme that optimizes calls to `car` by removing the run time type check when the argument is known to be a pair. The compiler could use the 0-cfa analysis [8, 9] to compute for every variable of a program the (conservative) set of allocation points in the program that create a value (pair, function, number, etc) that can be bound to an instance of that variable. In the program fragment shown in Figure 1 the 0-cfa analysis computes that only pairs, created by $\texttt{cons}_1$ and $\texttt{cons}_5$, can be bound to instances of the variable `a` and consequently the transformer can safely remove the run time type check in the call to $\texttt{car}_2$.

Note that the 0-cfa analysis wasted time computing the properties of variable `b` which are not needed by the transformer. Had there been a call `(car b)` in `f`'s body it would take the more complex 1-cfa analysis to discover that only a pair created by $\texttt{cons}_6$ and $\texttt{cons}_8$ can be bound to an instance of variable `b`; the 0-cfa does not exclude that the empty list can be bound to `b` because the empty list can be bound to `c` and returned by function `i`. The 1-cfa analysis achieves this higher precision by using an abstract execution model which partitions the instances of a particular variable on the basis of the call sites that create these instances. Consequently it distinguishes the instances of variable `c` created by the call $(_7\texttt{i (cons ...)})$ and those created by the call $(_9\texttt{i '()})$, allowing it to narrow the type returned by $(_7\texttt{i (cons ...)})$ to pairs only. If these two calls to `i` are replaced by calls to `j` then the 2-cfa analysis would be needed to fully remove all type checks on calls to `car`. By using an abstract execution model that keeps track of call chains up to a length of 2 the 2-cfa analysis distinguishes the instances of variable `c` created by the call chain $(_7\texttt{j (cons ...)}) \rightarrow (_3\texttt{i d})$ and the call chain $(_9\texttt{j '()}) \rightarrow (_3\texttt{i d})$. The compiler implementer (or user) is faced with the difficult task of finding for each program

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

$$
\begin{array}{lll}
\mathrm{Exp} := & \texttt{\#f}_l & l \in \mathrm{Lab} \\
\quad\mid & \mathrm{x}_l & \mathrm{x} \in \mathrm{Var},\ l \in \mathrm{Lab} \\
\quad\mid & \texttt{(}_l e_1\ e_2\texttt{)} & l \in \mathrm{Lab},\ e_1, e_2 \in \mathrm{Exp} \\
\quad\mid & \texttt{(}\lambda_l \mathrm{x}\texttt{.}\ e_1\texttt{)} & l \in \mathrm{Lab},\ \mathrm{x} \in \mathrm{Var}, e_1 \in \mathrm{Exp} \\
\quad\mid & \texttt{(if}_l\ e_1\ e_2\ e_3\texttt{)} & l \in \mathrm{Lab},\ e_1, e_2, e_3 \in \mathrm{Exp} \\
\quad\mid & \texttt{(cons}_l\ e_1\ e_2\texttt{)} & l \in \mathrm{Lab},\ e_1, e_2 \in \mathrm{Exp} \\
\quad\mid & \texttt{(car}_l\ e_1\texttt{)} & l \in \mathrm{Lab},\ e_1 \in \mathrm{Exp} \\
\quad\mid & \texttt{(cdr}_l\ e_1\texttt{)} & l \in \mathrm{Lab},\ e_1 \in \mathrm{Exp} \\
\quad\mid & \texttt{(pair?}_l\ e_1\texttt{)} & l \in \mathrm{Lab},\ e_1 \in \mathrm{Exp} \\
\mathrm{Lab} := & \textit{Labels} & \\
\mathrm{Var} := & \textit{Variables} &
\end{array}
$$

Figure 2. Syntax of the Source Language

an acceptable trade-off between the extent of optimization and the value of k and compile time.

The analysis approach presented in this paper is a *demand-driven* type analysis that adapts the analysis to the source program. The work performed by the analyzer is driven by the need to determine which run time type checks can be safely removed. By being demand-driven the analyzer avoids performing useless analysis work and performs deeper analysis for specific parts of the program when it may result in the removal of a run time type check. This is achieved by changing the abstract execution model dynamically to increase the precision where it appears to be beneficial. Like the k-cfa our analysis is based on abstract interpretation. As explained in Section 4, our models use lexical contours instead of call chains. Some important programs analyzed with our approach are more accurately analyzed than with the k-cfa for any value of k (see Section 6). In particular, some programs with higher order functions, including uses of `map` and the Y combinator, are analyzed precisely.

Our demand-driven analysis does not place *a priori* limits on the precision of the analysis. This has the advantage that the analysis effort can be varied according to the complexity of the source program and in different parts of the same program. On the other hand, the analysis may not terminate for programs where it is difficult or impossible to prove that a particular type check can be removed. We take the pragmatic point of view that it is up to the user to decide what is the maximal optimization effort (limit on the time or on some other resource) the compiler should expend. The type checks that could not be removed within this time are simply kept in the generated code. We think this is better than giving the user the choice of an "optimization level" (such as the k to use in a k-cfa) because there is a more direct link with compilation time.

Although our motivation is the efficient compilation of Scheme, the analysis is also applicable to languages such as SML and Haskell for the removal of run time pattern-matching checks. Indeed the previous example can be translated directly in these statically typed languages, where the run time type checks are in the calls to `hd`.

After a brief description of the source language we explain the analyzer, the abstract execution models and the processing of demands. Experimental results obtained with a prototype of our analyzer are then presented.

2 Source Language

The source language of the analysis is a purely functional language similar to Scheme and with only three data types: the false value, pairs and one argument functions. Each expression is uniquely labeled to allow easy identification in the source program. The syntax is given in Figure 2.

$$
\begin{array}{lll}
\mathrm{Val}^{\uparrow} := \mathrm{Err} \dot\cup \mathrm{Val}\ [1] & & \\
\mathrm{Err} := \textit{Errors} & & \\
\mathrm{Val} := \mathrm{ValB} \dot\cup \mathrm{ValC} \dot\cup \mathrm{ValP} & & \\
\mathrm{ValB} := \{\#\mathrm{f}\} & & \textit{Booleans} \\
\mathrm{ValC} := \mathrm{Val} \to \mathrm{Val}^{\uparrow} & & \textit{Closures} \\
\mathrm{ValP} := \mathrm{Val} \times \mathrm{Val} & & \textit{Pairs} \\
\mathrm{Env} := \mathrm{Var} \to \mathrm{Val} & & \\
\\
\mathsf{E} : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Val}^{\uparrow} & & \textit{Evaluation function} \\
\mathsf{E}\ [\![\texttt{\#f}_l]\!]\ \rho & = \#\mathrm{f} & \\
\mathsf{E}\ [\![\mathrm{x}_l]\!]\ \rho & = \rho\ \mathrm{x} & \\
\mathsf{E}\ [\![\texttt{(}_l e_1\ e_2\texttt{)}]\!]\ \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}\ [\![\texttt{(}\lambda_l \mathrm{x}\texttt{.}\ e_1\texttt{)}]\!]\ \rho & = \lambda v.\ \mathsf{E}\ [\![e_1]\!]\ \rho[\mathrm{x} \mapsto v] & \\
\mathsf{E}\ [\![\texttt{(if}_l\ e_1\ e_2\ e_3\texttt{)}]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v \neq \#\mathrm{f}\ ?\ \mathsf{E}\ [\![e_2]\!]\ \rho : \mathsf{E}\ [\![e_3]\!]\ \rho) & \\
\mathsf{E}\ [\![\texttt{(cons}_l\ e_1\ e_2\texttt{)}]\!]\ \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}_l\ e_1\texttt{)}]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v = (v_1, v_2)\ ?\ v_1 : \mathrm{ERROR}) & \\
\mathsf{E}\ [\![\texttt{(cdr}_l\ e_1\texttt{)}]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v = (v_1, v_2)\ ?\ v_2 : \mathrm{ERROR}) & \\
\mathsf{E}\ [\![\texttt{(pair?}_l\ e_1\texttt{)}]\!]\ \rho & = \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho) & \\
 & \quad (\lambda v.\ v \in \mathrm{ValP}\ ?\ v : \#\mathrm{f}) & \\
\\
\mathsf{A} : \mathrm{Val} \to \mathrm{Val} \to \mathrm{Val}^{\uparrow} & & \textit{Apply function} \\
\mathsf{A}\ f\ v & = f \in \mathrm{ValC}\ ?\ f\ v : \mathrm{ERROR} & \\
\\
\mathsf{C} : \mathrm{Val}^{\uparrow} \to (\mathrm{Val} \to \mathrm{Val}^{\uparrow}) \to \mathrm{Val}^{\uparrow} & & \textit{Check function} \\
\mathsf{C}\ v\ k & = v \in \mathrm{Err}\ ?\ v : k\ v &
\end{array}
$$

Figure 3. Semantics of the Source Language

There is no built-in `letrec` special form. The Y combinator must be written explicitly when defining recursive functions. Note also that `cons`, `car`, `cdr` and `pair?` are treated as special forms.

The semantics of the language is given in Figure 3. A notable departure from the Scheme semantics is that `pair?` returns its argument when it is a pair. The only operations that may require a run time type check are `car` and `cdr` (the argument must be a pair) and function call (the function position must be a function).

3 Analysis Framework

To be able to modify the abstract evaluation model during the analysis of the program we use an *analysis framework*. The framework is a parameterized analysis general enough to be used for type analysis, as we do here, as well as a variety of other program analyses. When the specifications of an abstract evaluation model are fed to the framework an analysis instance is obtained which can then be used to analyze the program.

The analysis instance is composed of a set of *evaluation constraints* that is produced from the framework parameters and the program. These constraints represent an abstract interpretation of the program. The analysis of the program amounts to solving the set of constraints. The solution is the *analysis results*. From the program and the framework parameters can also be produced the *safety constraints* which indicate at which program points run time type checks may be needed. It is by confronting the analysis results with the safety constraints that redundant type checks are identified. If all the safety constraints are satisfied, all the type checks can be removed by the optimizer. A detailed description of the analysis

[1]The $\dot\cup$ operator is the *disjoint union*, i.e. the sets to combine must be disjoint.

$\mathcal{V}al\mathcal{B} \neq \emptyset$	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
$\mathsf{cc} : \text{Lab} \times \mathcal{C}ont \to \mathcal{V}al\mathcal{C}$	Abstract closure creation
$\mathsf{pc} : \text{Lab} \times \mathcal{V}al \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{V}al\mathcal{P}$	Abstract pair creation
$\mathsf{call} : \text{Lab} \times \mathcal{V}al\mathcal{C} \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{C}ont$	Contour selection

where $\mathcal{V}al := \mathcal{V}al\mathcal{B} \,\dot\cup\, \mathcal{V}al\mathcal{C} \,\dot\cup\, \mathcal{V}al\mathcal{P}$
subject to $|\mathcal{V}al| < \infty$ and $|\mathcal{C}ont| < \infty$

Figure 4. Instantiation parameters of the analysis framework

Value of e_l in k:
$\alpha_{l,k} \subseteq \mathcal{V}al \qquad l \in \text{Lab}, \quad k \in \mathcal{C}ont$
Contents of x in k:
$\beta_{\mathrm{x},k} \subseteq \mathcal{V}al \qquad \mathrm{x} \in \text{Var}, \quad k \in \mathcal{C}ont$
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:
$\delta_{l,k} \subseteq \mathcal{V}al \qquad l \in \text{Lab}, \quad k \in \mathcal{C}ont$
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

framework and its implementation is given in [4]. Here we only give an overview of the framework.

3.1 Framework Parameters

Figure 4 presents the framework parameters that specify the abstract evaluation model. The interface is simple and flexible. Four abstract domains, the main contour, and three abstract evaluation functions have to be provided to the framework.

$\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, and $\mathcal{V}al\mathcal{P}$ are the abstract domains for the Booleans, closures, and pairs. They must be non-empty and mutually disjoint. $\mathcal{V}al$ is the union of these three domains. $\mathcal{C}ont$ is the abstract domain of contours. Contours are abstract versions of the evaluation contexts in which expressions of the program get concretely evaluated. The part of the evaluation contexts that is abstractly modeled by the contours may be the lexical environment, the continuation, or a combination of both. The main contour k_0 indicates in which abstract contour the main expression of the program is to be evaluated.

The abstract evaluation functions cc, pc, and call specify closure creation, pair creation, and how the contour is selected when a function call occurs. $\mathsf{cc}(l,k)$ returns the abstract closure created when the λ-expression e_l is evaluated in contour k. $\mathsf{pc}(l,v_1,v_2,k)$ returns the abstract pair created by the `cons`-expression labeled l evaluated in contour k with arguments v_1 and v_2. Finally, $\mathsf{call}(l,c,v,k)$ indicates the contour in which the body of closure c is evaluated when c is called from the call-expression e_l in contour k and with argument v.

Any group of modeling parameters that satisfies the constraints given in Figure 4 is a valid abstract evaluation model for the framework.

$$
\begin{array}{lll}
\langle\text{mPat}\rangle & := & \forall \mid \mathbf{\#f} \mid \lambda_\forall \mid \lambda_l\ k \mid (P, P') \\
& & \text{where } l \in \text{Lab}, k \in \langle \text{m}k\text{Pat}\rangle, P, P' \in \langle\text{mPat}\rangle \\
\langle\text{sPat}\rangle & := & \star \mid \lambda_\star \mid \lambda_l\ k \mid (P, P') \mid (P', P) \\
& & \text{where } l \in \text{Lab}, k \in \langle \text{s}k\text{Pat}\rangle, \\
& & \qquad P \in \langle\text{sPat}\rangle, P' \in \langle\text{mPat}\rangle \\
\langle\text{m}k\text{Pat}\rangle & := & (P_1 \ldots P_n) \\
& & \text{where } \forall_{1 \leq j \leq n}.\ P_j \in \langle\text{mPat}\rangle \\
\langle\text{s}k\text{Pat}\rangle & := & (P_1 \ldots P_i \ldots P_n) \\
& & \text{where } P_i \in \langle\text{sPat}\rangle, \forall_{\substack{1 \leq j \leq n \\ j \neq i}}.\ P_j \in \langle\text{mPat}\rangle
\end{array}
$$

Figure 6. Syntax of patterns

3.2 Analysis Results

The analysis results are returned in the seven abstract matrices shown in Figure 5. Matrices α, β, and γ indicate respectively the value of the expressions, the value of the variables, and the return value of closures. The value $\beta_{\mathrm{x},k}$ is defined as follows. Assume that closure c was created by λ-expression $(\lambda_l \mathrm{x}.\ e_{l'})$. Then if c is called and the call function prescribes contour k for the evaluation of c's body, then parameter x will be bound to the abstract value $\beta_{\mathrm{x},k}$.

$\delta_{l,k}$ indicates whether or not expression e_l is evaluated in contour k. e_l is evaluated in contour k if and only if $\delta_{l,k} \neq \emptyset$. Apparently, $\delta_{l,k}$ should have been defined as a Boolean instead of a set. However, the use of sets makes the implementation of the analysis framework simpler (see [4]).

Matrices χ, π, and κ are logs keeping the circumstances prevailing when the different closures, pairs, and contours, respectively, are created. For example, if during the abstract interpretation of the program a pair p is created at expression e_l with values v_1 and v_2 and in contour k, then this creation of p is logged into π_p. That is, $(l,v_1,v_2,k) \in \pi_p$. Most of the time, the circumstances logged into the log variables are much fewer than what they can theoretically be. In other words, π_p usually contains fewer values than $\mathsf{pc}^{-1}(p)$. Similarly, when closure $c = \mathsf{cc}(l,k)$ is created by the evaluation of e_l in k, (l,k) is inserted in χ_c. And when contour $k' = \mathsf{call}(l,f,v,k)$ is selected to be the contour in which the body of f is to be evaluated when f gets invoked on v at e_l in k, $(l,\ f,\ v,\ k)$ is inserted in $\kappa_{k'}$.

4 Pattern-Based Models

In the demand-driven type analysis we use patterns and pattern-matchers to implement abstract evaluation models. Patterns constitute the abstract values ($\mathcal{V}al$) and the abstract contours ($\mathcal{C}ont$). Abstract values are *shallow* versions of the concrete values and abstract contours are shallow versions of the lexical environments. These lexical contours are one of the features distinguishing our analysis from most type and control-flow analyses which typically use call chains. A call chain is a string of the labels of the k nearest enclosing dynamic calls. Although the use of call chains guarantees polynomial-time analyses, it can also be fooled easily. We believe that lexical contours provide a much more robust way to abstract concrete evaluation contexts.

Figure 6 gives the syntax of patterns. There are two kinds of patterns: *modeling patterns* ($\langle\text{mPat}\rangle$ and $\langle\text{m}k\text{Pat}\rangle$) and *split patterns* ($\langle\text{sPat}\rangle$ and $\langle\text{s}k\text{Pat}\rangle$). For both modeling patterns and split patterns, there is a *value* variant ($\langle\text{mPat}\rangle$ and $\langle\text{sPat}\rangle$) and a *contour* variant ($\langle\text{m}k\text{Pat}\rangle$ and $\langle\text{s}k\text{Pat}\rangle$). Split patterns contain a single *split point* that is designated by $\star$. They are used in the demands that drive the analysis (in *split* demands, more precisely). Modeling patterns contain no split point. They form the representation of the abstract values and contours.

$\nearrow \subseteq \text{Val} \times \langle\text{mPat}\rangle$
$v \nearrow \forall$, if $v \in \text{Val}$
$\#\text{f} \nearrow \mathbf{\#f}$
$c \nearrow \lambda_\forall$, if $c \in \text{ValC}$
$c \nearrow \lambda_l\ k$, if $c \in \text{ValC}$, $c = \mathsf{E}\ [\![(\lambda_l \mathtt{x}.\ e)]\!]\ \rho$, and $\rho \nearrow_l k$
$(v, w) \nearrow (P_1, P_2)$, if $v \nearrow P_1$ and $w \nearrow P_2$
$\rho \nearrow_l ()$, if ρ is valid at label l [2] and $\text{Dom}(\rho) = \emptyset$ [3]
$\rho \nearrow_l (P_1\ P_2\ \ldots\ P_n)$, if ρ is valid at label l, x is the innermost variable in $\text{Dom}(\rho)$, $\rho\ \text{x} \nearrow P_1$, $e_{l'} = (\lambda_{l'}\text{x}.\ e)$, and $\rho\,[\text{x} \mapsto \bot] \nearrow_{l'} (P_2\ \ldots\ P_n)$ [4]

Figure 7. Formal definition of relation "is abstracted by"

4.1 Meaning of Patterns

Modeling patterns represent abstract values, which in turn can be seen as sets of concrete values. Pattern $\forall$ abstracts any value, pattern **#f** abstracts the Boolean value #f, pattern $\lambda_\forall$ abstracts any closure, pattern $\lambda_l\ k$ abstracts any closure coming from λ-expression labeled l and having a definition environment that can be abstracted by $k \in \langle\text{m}k\text{Pat}\rangle$, and pattern (P_1, P_2) abstracts any pair whose components can be abstracted by P_1 and P_2, respectively. The difference between abstract values and concrete values is that an abstract value can be made imprecise by having parts of it *cut off* using $\forall$ and $\lambda_\forall$.

Modeling *contour* patterns appear in the modeling patterns of closures. To simplify, we use the term contour to mean modeling contour pattern. Contours abstract lexical environments. A contour is a list with an abstract value for each variable visible from a certain label (from the innermost variable to the outermost). For example, the contour $(\lambda_\forall\ (\forall, \forall))$ indicates that the innermost variable (say y) is a closure and the other (say x), is a pair. It could abstract the following concrete environment:[5]

$$\cdot\ [\text{x} \mapsto (\#\text{f}, \#\text{f})]\ [\text{y} \mapsto \mathsf{E}\ [\![(\lambda_l \mathtt{z}.\ \mathtt{\#f}_{l'})]\!]\ \cdot]$$

A formal definition of what concrete values are abstracted by what abstract values is given in Figure 7. The relation $\nearrow \subseteq \text{Val} \times \langle\text{mPat}\rangle$ relates concrete and abstract values such that $v \nearrow P$ means that v is abstracted by P. We mention (without proof) that any concrete value obtained during execution of the program can be abstracted by a modeling pattern that is perfectly accurate. That is, the latter abstracts only one concrete value, which is the former.

The split patterns and split contour patterns are used to express *split* demands that increase the precision of the abstract evaluation model. Their structure is similar to that of the modeling patterns but they include one and only one split point ($\star$) that indicates exactly where in an abstract value an improvement in the precision of the model is requested. Their utility will be made clearer in Section 5. Operations on split patterns are explained next.

[2] ρ is valid at label l if its domain is exactly the set of variables that are visible from e_l.

[3] $\text{Dom}(f)$ denotes the *domain* of function f.

[4] '$\bot$' denotes an undefined value. Consequently, $\rho\,[\text{x} \mapsto \bot]$ is the same environment as ρ but without the binding to x.

[5] The empty concrete environment, $\cdot$, contains no bindings.

$\cap : (\langle\text{sPat}\rangle \cup \langle\text{mPat}\rangle) \times (\langle\text{sPat}\rangle \cup \langle\text{mPat}\rangle) \rightarrow (\langle\text{sPat}\rangle \cup \langle\text{mPat}\rangle)$
$P_1 \cap P_2$ is undefined if $P_1, P_2 \in \langle\text{sPat}\rangle$
$\forall \cap P_2 = P_2$
$P_1 \cap \forall = P_1$
$\star \cap P_2 = \star$
$P_1 \cap \star = \star$
$\mathbf{\#f} \cap \mathbf{\#f} = \mathbf{\#f}$
$\lambda_\forall \cap P_2 = P_2$, if $P_2 = \lambda_\forall$ or $P_2 = \lambda_l\ (P'_1 \ldots P'_n)$
$P_1 \cap \lambda_\forall = P_1$, if $P_1 = \lambda_\forall$ or $P_1 = \lambda_l\ (P'_1 \ldots P'_n)$
$\lambda_\star \cap P_2 = \lambda_\star$, if $P_2 = \lambda_\forall$ or $P_2 = \lambda_l\ (P'_1 \ldots P'_n)$
$P_1 \cap \lambda_\star = \lambda_\star$, if $P_1 = \lambda_\forall$ or $P_1 = \lambda_l\ (P'_1 \ldots P'_n)$
$\lambda_l\ (P_1 \ldots P_n) \cap \lambda_l\ (P'_1 \ldots P'_n) = \lambda_l\ (P''_1 \ldots P''_n)$, if $P''_i = P_i \cap P'_i, \forall 1 \le i \le n$
$(P_1, P_2) \cap (P'_1, P'_2) = (P''_1, P''_2)$, if $P''_1 = P_1 \cap P'_1$ and $P''_2 = P_2 \cap P'_2$

Figure 8. Algorithm for computing the intersection between two patterns

4.2 Pattern Intersection

Although the $\nearrow$ relation provides a formal definition of when a concrete value is abstracted by an abstract value, and, by extension, when an abstract value is abstracted by another, it is not necessarily expressed as an algorithm. Moreover, the demand-driven analysis does not manipulate concrete values, only patterns of all kinds. So we present a method to test whether an abstract value is abstracted by another. More generally, we want to be able to test whether a (modeling or split) pattern *intersects* with another. Similarly for both kinds of contour patterns.

The intersection between patterns is defined in Figure 8. It is partially defined because two patterns may be *incompatible*, in the sense that they do not have an intersection and as such, their empty intersection cannot be represented using patterns, or as the intersection of two split patterns may create something having two split points. The equations in the figure should be seen as cases to try in order from the first to the last until, possibly, a case applies.

A pattern P intersects with another pattern P' if the intersection function is defined when applied to P and P'. Moreover, when P intersects with P', the resulting intersection $P'' = P \cap P'$ is characterized by:[6]

$$\{v \in \text{Val} \mid v \nearrow P''\} = \{v \in \text{Val} \mid v \nearrow P \ \wedge\ v \nearrow P'\}$$

4.3 Spreading on Split Patterns

Another relation that is needed to perform the demand-driven analysis is the spreading test. It is useful in determining if a given split pattern will increase the precision of the model if it is used in a *split* demand. Spreading can occur between a set of abstract values (modeling patterns) and a split pattern. A split pattern can be thought of as denoting a sub-division: the set of its abstracted concrete value is partitioned into a number of sets corresponding to the different possibilities seen at the split point. Each of those sets is called a *bucket*. For example, the pattern $\star$ abstracts all values, that is, Val. It sub-divides Val into three buckets: ValB, ValC, and ValP. Spreading occurs between the set of abstract values V and the split

[6] Provided that we consider '$\star$' and '$\lambda_\star$' to abstract all concrete values and all concrete closures, respectively.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\divideontimes \subseteq 2^{\langle \text{mPat} \rangle} \times \langle \text{sPat} \rangle} \\
S \divideontimes P, & \text{if } \forall \in S \\
S \divideontimes \star, & \text{if } \mathbf{\#f} \in S \text{ and } S \setminus \{\mathbf{\#f}\} \neq \emptyset \text{, or} \\
 & \quad S \cap T \neq \emptyset \neq S \setminus T \\
 & \text{where } T = \{(P, P') \mid P, P' \in \langle \text{mPat} \rangle\} \\
S \divideontimes P, & \text{if } S \setminus T \neq \emptyset \text{ and } T \divideontimes P \\
 & \text{where } T = \{P' \in S \mid P' \text{ intersects with } P\} \\
S \divideontimes P, & \text{if } \lambda_\forall \in S \\
S \divideontimes \lambda_\star, & \text{if } \lambda_l\ (P_1 \ldots P_m), \lambda_{l'}\ (P'_1 \ldots P'_n) \in S \\
 & \text{and } l \neq l' \\
\multicolumn{2}{l}{S \divideontimes \lambda_l\ (P_1 \ldots P_i \ldots P_n),} \\
 & \text{if } P_i \in \langle \text{sPat} \rangle \text{ and } T \divideontimes P_i \\
 & \text{where } T = \{P'_i \mid \lambda_l\ (P'_1 \ldots P'_i \ldots P'_n) \in S\} \\
S \divideontimes (P_1, P_2), & \text{if } P_1 \in \langle \text{sPat} \rangle \text{ and } T \divideontimes P_1 \\
 & \text{where } T = \{P'_1 \mid (P'_1, P'_2) \in S\} \\
S \divideontimes (P_1, P_2), & \text{if } P_2 \in \langle \text{sPat} \rangle \text{ and } T \divideontimes P_2 \\
 & \text{where } T = \{P'_2 \mid (P'_1, P'_2) \in S\}
\end{array}
$$

Figure 9. Algorithm for the relation "is spread on"

pattern P if some two values (or refinements of values) in V that are abstracted by P fall into different buckets. We say that V is spread on split pattern P and denote it with $V \divideontimes P$. Figure 9 gives a formal definition of $\divideontimes$. As with the $\cap$ operator, cases should be tried in order.

Mathematically, the relation $\divideontimes$ has the following meaning. The set of abstract values S is spread on the split pattern P, denoted $S \divideontimes P$, if:

$$
\begin{array}{l}
\exists P_1, P_2 \in S.\ \exists v_1, v_2 \in \text{Val}.\ \forall P' \in \{P_{\mathcal{B}}, P_{\mathcal{C}}, P_{\mathcal{P}}\}. \\
\quad v_1 \nearrow P_1 \wedge v_2 \nearrow P_2 \wedge \\
\quad v_1 \nearrow P \wedge v_2 \nearrow P \wedge \\
\quad (v_1 \nearrow P' \Rightarrow \neg(v_2 \nearrow P'))
\end{array}
$$

where $P_{\mathcal{B}}$, $P_{\mathcal{C}}$, and $P_{\mathcal{P}}$ are modeling patterns obtained by replacing '$\star$' in P by $\mathbf{\#f}$, $\lambda_\forall$, and $(\forall, \forall)$, respectively.

4.4 Model Implementation

An abstract value can be viewed as a concrete value that has gone through a *projection*. Similarly, a contour can be viewed as a lexical environment that has gone through a projection. If one arranges for the image of the projection to be finite, then one obtains the desired abstract domains $\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, $\mathcal{V}al\mathcal{P}$, and $\mathcal{C}ont$.

But which projection should be used? The $\nearrow$ relation is not of much help since, generally, for a concrete value v, there may be more than one abstract value $\hat{v}$ such that $v \nearrow \hat{v}$. So a projection based on $\nearrow$ would be ill-defined.

The projection we use is based on an exhaustive non-redundant pattern-matcher. That is, the pattern-matcher implementing the projection of the values is a finite set of modeling patterns. For any concrete value v, there will exist one and only one modeling pattern $\hat{v}$ in the set such that $v \nearrow \hat{v}$. Such a pattern-matcher describes a finite partition of Val.

For example, the simplest projection for the values is:[7]

$$\{\mathbf{\#f},\ \lambda_\forall,\ (\forall, \forall)\}$$

It is finite, exhaustive and non-redundant.

[7] This is not exactly true. The simplest pattern-matcher would be the trivial one, $\{\forall\}$, but it would not implement a legal model for the framework since an abstract model must at least distinguish the Booleans, the closures, and the pairs.

As for the projection of contours, we use one pattern-matcher per λ-expression. For a given λ-expression e_l, the lexical environment in which its body is evaluated can be projected by the pattern-matcher M_l. The empty lexical environment is always projected onto the list of length 0, as the empty list is the only contour that abstracts the empty environment.

The simplest contour pattern-matcher M_l for expression $(\lambda_l \texttt{x}.\ e_{l'})$ is $\{(\forall\ \ldots\ \forall)\}$, it is a single list having as many entries as there are visible variables in the environment in which $e_{l'}$ is evaluated.

Having a pattern-matcher M_v that projects values and a family of pattern-matchers $\{M_i \mid \ldots\}$ that project lexical environments, and assuming that M_v projects closures coming from different λ-expressions to different abstract closures, it is easy to create an abstract model, i.e. to define the parameters of the analysis framework, as follows.

- $\mathcal{V}al\mathcal{B}$ is $\{\mathbf{\#f}\}$
- $\mathcal{V}al\mathcal{C}$ is $\{\lambda_l\ k \in M_v\}$
- $\mathcal{V}al\mathcal{P}$ is $\{(v_1, v_2) \in M_v\}$
- $\mathcal{C}ont$ is $()\ \cup\ \bigcup_i M_i$
- k_0 is $()$
- $\mathsf{cc}(l, k)$ is the projection of $\lambda_l\ k$ by M_v
- $\mathsf{pc}(l, v_1, v_2, k)$ is the projection of (v_1, v_2) by M_v
- $\mathsf{call}(l, \lambda_l\ (w_1\ \ldots\ w_n), v, k)$ is the projection of $(v\ w_1\ \ldots\ w_n)$ by M_l

4.5 Maintaining Model Consistency

One remaining problem that requires special attention is *consistency*. During the demand-driven analysis, pattern-matchers are not used to project concrete values, but abstract values. If one of the abstract values is not precise enough the projection operation may become ill-defined. In general, abstract values abstract a *set* of concrete values. Suppose that $\hat{v}_1$ is such an imprecise abstract value. Now, let $\hat{v}_2$ be a modeling pattern that contains $\hat{v}_1$ as a sub-pattern. We want to project $\hat{v}_2$ in order to obtain the resulting abstract value. A sensible definition for the projection of $\hat{v}_2$ consists in choosing a modeling pattern $\hat{w}$ in the pattern-matcher M such that all concrete values abstracted by $\hat{v}_2$ are abstracted by $\hat{w}$. Unfortunately, such a $\hat{w}$ may not exist as it may take the union of many modeling patterns of M to properly abstract all the concrete values abstracted by $\hat{v}_2$.

Here is an example to help clarifying this notion. The following pattern-matcher M, intended for the projection of values, is inconsistent:

$$
\left\{
\begin{array}{lll}
\mathbf{\#f}, & (\forall, \mathbf{\#f}), & (\forall, (\mathbf{\#f}, \forall)), \\
\lambda_\forall, & (\forall, \lambda_\forall), & (\forall, (\lambda_\forall, \forall)), \\
 & & (\forall, ((\forall, \forall), \forall))
\end{array}
\right\}
$$

Note that the pattern-matcher is finite, exhaustive, and non-redundant but nevertheless inconsistent. Before explaining why, let us see how it models the values. First, it distinguishes the values by their (top-level) type. Second, it distinguishes the pairs by the type of the value in the CDR-field. Finally, the pairs containing a sub-pair in the CDR-field are distinguished by the type of the value in the CAR-field of the sub-pair. Note that the CAR-field of the sub-pairs is more precisely described than the CAR-field of the pairs themselves. This is the inconsistency. Problems occur when we try to make a pair with another pair in the CDR-field. Let us try to make

$$
\begin{array}{lcl}
\mathrm{PM} & := & \mathrm{PM_O} \mid \mathrm{PM_C} \mid \mathrm{PM_L} \\
\mathrm{PM_O} & := & \texttt{Onode}\ [\mathcal{V}al \Rightarrow M_1] \ \mid \\
& & \texttt{Onode}\ [\mathcal{V}al\,\mathcal{B} \Rightarrow M_1,\ \mathcal{V}al\,\mathcal{C} \Rightarrow M_2,\ \mathcal{V}al\,\mathcal{P} \Rightarrow M_3] \\
& & \quad \text{where } M_1, M_2, M_3 \in \mathrm{PM} \\
\mathrm{PM_C} & := & \texttt{Cnode}\ [\mathrm{Lab} \Rightarrow M_1] \ \mid \\
& & \texttt{Cnode}\ [l_1 \Rightarrow M_1,\ \ldots,\ l_n \Rightarrow M_n] \\
& & \quad \text{where } M_1, \ldots, M_n \in \mathrm{PM} \\
& & \quad \text{and } \{l_1, \ldots, l_n\} = \{l \in \mathrm{Lab} \mid e_l \text{ is a } \lambda\text{-expr.}\} \\
\mathrm{PM_L} & := & \texttt{Leaf}\ \langle \mathrm{mPat} \rangle \ \mid \texttt{Leaf}\ \langle \mathrm{m}k\mathrm{Pat} \rangle
\end{array}
$$

Figure 10. Implementation of the pattern-matchers

To project $P \in \langle \mathrm{mPat} \rangle$ with $M \in \mathrm{PM}$,
compute $\mathrm{pm}(M, [\,] \triangleleft P)$, and
to project $(P_1 \ldots P_n) \in \langle \mathrm{m}k\mathrm{Pat} \rangle$ with $M \in \mathrm{PM}$,
compute $\mathrm{pm}(M, [\,] \triangleleft P_1 \triangleleft \ldots \triangleleft P_n)$, where

$$
\begin{array}{l}
\mathrm{pm} : \mathrm{PM} \times [\text{queue of } \langle \mathrm{mPat} \rangle] \rightarrow \langle \mathrm{mPat} \rangle \cup \langle \mathrm{m}k\mathrm{Pat} \rangle \\
\mathrm{pm}(\texttt{Onode}\ [\mathcal{V}al \Rightarrow M_1],\ P \triangleleft q) \\
\quad = \mathrm{pm}(M_1, q) \\
\mathrm{pm}(\texttt{Onode}\ [\mathcal{V}al\,\mathcal{B} \Rightarrow M_1, \ldots],\ \mathbf{\#f} \triangleleft q) \\
\quad = \mathrm{pm}(M_1, q) \\
\mathrm{pm}(\texttt{Onode}\ [\ldots, \mathcal{V}al\,\mathcal{C} \Rightarrow M_2, \ldots],\ P \triangleleft q) \\
\quad = \mathrm{pm}(M_2, q \triangleleft P) \\
\quad \text{if } P = \lambda_{\forall} \text{ or } P = \lambda_l\ (P_1 \ldots P_n) \\
\mathrm{pm}(\texttt{Onode}\ [\ldots, \mathcal{V}al\,\mathcal{P} \Rightarrow M_3],\ (P_1, P_2) \triangleleft q) \\
\quad = \mathrm{pm}(M_3, q \triangleleft P_1 \triangleleft P_2) \\
\mathrm{pm}(\texttt{Cnode}\ [\mathrm{Lab} \Rightarrow M_1],\ P \triangleleft q) \\
\quad = \mathrm{pm}(M_1, q) \\
\mathrm{pm}(\texttt{Cnode}\ [\ldots, l_i \Rightarrow M_i, \ldots],\ \lambda_{l_i}\ (P_1 \ldots P_n) \triangleleft q) \\
\quad = \mathrm{pm}(M_i, q \triangleleft P_1 \triangleleft \ldots \triangleleft P_n) \\
\mathrm{pm}(\texttt{Leaf}\ P, [\,]) \\
\quad = P
\end{array}
$$

Figure 11. Pattern-matching algorithm

a pair with the values **#f** and $(\forall, (\mathbf{\#f}, \forall))$. We obtain the modeling pattern $\hat{v} = (\mathbf{\#f}, (\forall, (\mathbf{\#f}, \forall)))$ and we have to project it using M. It is clear that we cannot non-ambiguously choose one of the modeling patterns of M as an abstraction of all the values abstracted by $\hat{v}$.

In order to avoid inconsistencies, each time an entity is refined in one of the pattern-matchers, we must ensure that the abstract values and the contours on which the refined entity depends are sufficiently precise. If not, cascaded refinements are propagated to the dependencies of the entity. This cascade terminates since, for each propagation, the depth at which the extra details are required decreases.

4.6 Pattern-Matcher Implementation

Our implementation of the pattern-matchers is quite simple. A pattern-matcher is basically a decision tree doing a breadth-first inspection of the modeling pattern or modeling contour pattern to project. An internal node of the decision tree is either an *O-node* (object) or a *C-node* (closure). A leaf contains an abstract value or a contour which is the result of the projection. Each O-node is either a three-way switch that depends on the type of the object to inspect or is a one-way catch-all that ignores the object and continues with its single child. Each C-node is either a multi-way switch that depends on the label of the closure to inspect or is a one-way catch-all that ignores the closure and continues with its single child. Figure 10 presents the data structures used to implement the pattern-matchers.

$$
\begin{array}{lll}
\langle \text{demand} \rangle := & \multicolumn{2}{l}{\textbf{show}\ a \subseteq B} \\
& \multicolumn{2}{l}{\text{where } a \in \langle \alpha\text{-var} \rangle, B \in \langle \text{bound} \rangle} \\
& \multicolumn{2}{l}{\mid\ \textbf{split}\ s\ P} \\
& \multicolumn{2}{l}{\text{where } s \in \langle \text{splittee} \rangle, P \in \langle \text{sPat} \rangle} \\
& \multicolumn{2}{l}{\mid\ \textbf{show}\ d = \emptyset} \\
& \multicolumn{2}{l}{\text{where } d \in \langle \delta\text{-var} \rangle} \\
& \multicolumn{2}{l}{\mid\ \textbf{bad-call}\ l\ P'\ P''\ k} \\
& \multicolumn{2}{l}{\text{where } l \in \mathrm{Lab}, P', P'' \in \langle \mathrm{mPat} \rangle, k \in \langle \mathrm{m}k\mathrm{Pat} \rangle} \\
\langle \text{bound} \rangle & \multicolumn{2}{l}{:= \mathcal{V}al\,\mathcal{B} \mid \mathcal{V}al\,\mathcal{C} \mid \mathcal{V}al\,\mathcal{P} \mid \mathcal{V}al\,\mathcal{T}rues} \\
\langle \text{splittee} \rangle & \multicolumn{2}{l}{:= \mathcal{V}al\,\mathcal{C} \mid \mathcal{V}al\,\mathcal{P} \mid a \mid b \mid c} \\
& \multicolumn{2}{l}{\text{where } a \in \langle \alpha\text{-var} \rangle, b \in \langle \beta\text{-var} \rangle, c \in \langle \gamma\text{-var} \rangle} \\
\langle \alpha\text{-var} \rangle & := \alpha_{l,k} & \text{where } l \in \mathrm{Lab}, k \in \langle \mathrm{m}k\mathrm{Pat} \rangle \\
\langle \beta\text{-var} \rangle & := \beta_{\mathrm{x},k,l} & \text{where } \mathrm{x} \in \mathrm{Var}, k \in \langle \mathrm{m}k\mathrm{Pat} \rangle, l \in \mathrm{Lab} \\
\langle \gamma\text{-var} \rangle & := \gamma_{c,k} & \text{where } c \in \langle \mathrm{mPat} \rangle, k \in \langle \mathrm{m}k\mathrm{Pat} \rangle \\
\langle \delta\text{-var} \rangle & := \delta_{l,k} & \text{where } l \in \mathrm{Lab}, k \in \langle \mathrm{m}k\mathrm{Pat} \rangle
\end{array}
$$

Figure 12. Syntax of demands

The pattern-matching algorithm is presented in Figure 11. The breadth-first traversal is done using a queue. The contents of the queue always remain synchronized with the position in the decision tree. That is, when a C-node is reached, a closure is next on the queue, and when a leaf is reached, the queue is empty. The initial queue for an abstract value projection contains only the abstract value itself. The initial queue for a contour projection contains all the abstract values contained in the contour, with the first abstract value of the contour being the first to be extracted from the queue. To keep the notation terse, we use the view operation $\triangleleft$ both to enqueue and dequeue values. When enqueuing, the queue is on the left of $\triangleleft$. When dequeuing, the queue is on the right of $\triangleleft$. The empty queue is denoted by $[\,]$.

The pattern-matchers used in the initial abstract model are the following. Note that we describe them in terms of set theory and not in terms of the actual data structures. The value pattern-matcher contains one abstract Boolean, one abstract pair, and one abstract closure for each λ-expression. For each λ-expression, its corresponding contour pattern-matcher is the trivial one. Note that they are consistent as the pattern-matchers are almost blind to any detail. The only inspection that is performed is the switch on the label when projecting a closure. However, the projection of closures always involves closures with explicit labels since it only occurs through the use of the abstract model function cc.

We do not give a detailed description of the process of refining a pattern-matcher because it would be lengthy and it is not conceptually difficult.

5 Demand Processing

Figure 12 presents the syntax of demands. The syntax of the demands builds on the syntax of the patterns. There are *show* demands, *split* demands, and *bad call* demands.

5.1 Meaning of Demands

A show demand asks for the demonstration of a certain property. For example, it might ask for demonstration that a particular abstract variable must only contain pairs, meaning that a certain expression, in a certain evaluation context, must only evaluate to pairs. Or it might ask for the demonstration that a particular abstract variable must be empty, meaning that a certain expression, in a certain evaluation context, must not get evaluated. Note that the bound $\mathcal{V}al\,\mathcal{T}rues$ represents the values acting as *true* in the conditionals. That is, $\mathcal{V}al\,\mathcal{T}rues = \mathcal{V}al\,\mathcal{C} \,\dot{\cup}\, \mathcal{V}al\,\mathcal{P}$.

A bad call demand asks for the demonstration that a particular function call cannot happen. It specifies where and in which contour the bad call currently happens, which function is called, and which value is passed as an argument. Of course, except for the label, the parameters of the demand are abstract.

A split demand asks that proper modifications be done on the model in such a way that the *splittee* is no longer *spread* on the pattern. Take this demand for example: **split** $\alpha_{l,k}\ \star$. It asks that the abstract values contained in $\alpha_{l,k}$ be distinguished by their type (because of the pattern $\star$). If the variable $\alpha_{l,k}$ currently contains abstract values of different types, then these values are said to be spread on the pattern $\star$. Then the model ought to be modified in such a way that the contour k has been subdivided into a number of sub-contours $k_1, \ldots, k_n$, such that α_{l,k_i} contains only abstract values of a single type, for $1 \leq i \leq n$. In case of success, one might observe that α_{l,k_1} contains only pairs, α_{l,k_2}, only closures, α_{l,k_3}, nothing, α_{l,k_4}, only **#f**, etc. That is, the value of expression e_l in contour k would have been split according to the type.

In a split demand, the splittee can be an aspect of the abstract model (when it is $\mathcal{V}al\mathcal{C}$ or $\mathcal{V}al\mathcal{P}$) or an abstract variable from one of the α, β, or γ matrices. A splittee in $\langle\beta\text{-var}\rangle$ does not denote an ordinary entry in the β matrix. It does indicate the name of the source variable but it also gives a label and a contour where this variable is *referenced* (not bound).

Only the values that intersect with the pattern are concerned by the split. For example, if the demand is **split** $\alpha_{l,k}\ (\forall,\star)$ and $\alpha_{l,k} = \{\mathbf{\#f}, (\mathbf{\#f},\mathbf{\#f}), (\mathbf{\#f},\lambda_\forall)\}$, the only thing that matters is that the two abstract pairs must be separated. What happens with the Boolean is not important because it does not intersect with the pattern $(\forall,\star)$.

Normally, a show demand is emitted because the analysis has determined that, if the specified property was false, then a type error will most plausibly happen in the real program. Similarly for a bad call demand. Unfortunately, split demands do not have such a natural interpretation. They are a purely artificial creation necessary for the demand-driven analysis to perform its task. Moreover, during the concrete evaluation of the program, an expression, in a particular evaluation context, evaluates to exactly one value. So splitting in the concrete evaluation is meaningless.

5.2 Demand-Driven Analysis Algorithm

The main algorithm of the demand-driven analysis is relatively simple. It is sketched in Figure 13. Basically, it is an analysis/model-update cycle. The analysis phase analyses the program using the framework parameterized by the current abstract model. The model-update phase computes, when possible, a model-updating demand based on the current analysis results and applies it to the model. Note that the successive updates of the abstract model make it increasingly refined and the analysis results that it helps to produce improve monotonically. Consequently, any run time type check that is proved to be redundant at some point remains as such for the rest of the cycle.

The steps performed during the model-update phase are: the initial demands are gathered; demand processing (of the demands that do not modify the model) and call monitoring occur until no new demands can be generated; if there are model-updating demands, the *best* one is selected and applied on the model. The model-modifying demands are the split demands in which the splittee is $\mathcal{V}al\mathcal{C}$, $\mathcal{V}al\mathcal{P}$, or a member of $\langle\beta\text{-var}\rangle$.

create initial model
analyze program with model
while there is time left
    set demand pool to initial demands
    make the set of modifying demands empty
    repeat
        monitor call sites (l,k) that are marked
        while there is time left **and**
                there are new demands in the pool **do**
            pick a new demand D in the pool
            if D is a modifying demand **then**
                insert D in the modifying demands set
            else
                process D
                add the returned demands to the pool
    until there is no time left **or**
            there are no call sites to monitor
    if modifying demands set empty **then**
        exit
    else
        pick the best modifying demand D
        modify model with D
        re-analyze program with new model

Figure 13. Main demand-driven analysis algorithm

The initial demands are those that we obtain by responding to the needs of the optimizer and not by demand processing. That is, if non-closures may be called or non-pairs may go through a strictly “pairwise” operation, bound demands asking a demonstration that these violations do not really occur are generated. More precisely, for a call $({}_l e_{l'}\ e_{l''})$ and for $k \in \mathcal{C}ont$, if $\alpha_{l',k} \not\subseteq \mathcal{V}al\mathcal{C}$, then the initial demand **show** $\alpha_{l',k} \subseteq \mathcal{V}al\mathcal{C}$ is generated. And for a pair-access expression $(\texttt{car}_l\ e_{l'})$ or $(\texttt{cdr}_l\ e_{l'})$ and for $k \in \mathcal{C}ont$, if $\alpha_{l',k} \not\subseteq \mathcal{V}al\mathcal{P}$, then the initial demand **show** $\alpha_{l',k} \subseteq \mathcal{V}al\mathcal{P}$ is generated.

The criterion used to select a good model-updating demand in our implementation is described in Section 6.

The analysis/model-update cycle continues until there is no more time left or no model updates have been proposed in the model-update phase. Indeed, it is the user of a compiler including our demand-driven analysis who determines the bound on the computational effort invested in the analysis of the program. The time is not necessarily wall clock time. It may be any measure. In our implementation, a unit of time allows the algorithm to process a demand. Two reasons may cause the algorithm to stop by lack of model-updating demands. One is that there are no more initial demands. That means that all the run time type checks of the program have been shown to be redundant. The other is that there remain initial demands but the current analysis results are mixed in such a way that the demand processing does not lead to the generation of a model-updating demand.

5.3 Demand Processing

5.3.1 Show In Demands

Now, let us present the processing of demands. We begin with the processing of **show** $\langle\alpha\text{-var}\rangle \subseteq \langle\text{bound}\rangle$ demands. Let us consider the demand **show** $\alpha_{l,k} \subseteq B$. There are 3 cases. First case, if the values in $\alpha_{l,k}$ all lie inside of the bound B, then the demand is trivially successful. Nothing has to be done in order to obtain the desired demonstration.

if $\alpha_{l,k} \subseteq B$:
$\Rightarrow$ (SUCCESS)

Second case, if the values in $\alpha_{l,k}$ all lie outside of the bound B, then it must be shown that the expression e_l does not get evaluated in the abstract contour k. This is a sufficient and necessary condition because, if e_l is evaluated in contour k, any value it returns is outside of the bound, causing the original demand to fail. And if e_l does not get evaluated in contour k, then we can conclude that any value in $\alpha_{l,k}$ lies inside the bound.

if $\alpha_{l,k} \cap B = \emptyset$:
$\Rightarrow$ **show** $\delta_{l,k} = \emptyset$

Last case, some values in $\alpha_{l,k}$ lie inside of B and some do not. The only sensible thing to do is to first split the contour k into sub-contours in such a way that it becomes clear whether the values all lie inside of B or they all lie outside of B. Since the bounds are all simple, splitting on the type of the objects is sufficient. Once (we would better say "if") the split demand is successful, the original demand can be processed again.

otherwise:
$\Rightarrow$ **split** $\alpha_{l,k}$ $\star$

5.3.2 Show Empty Demands

We continue with the processing of **show** $\langle\delta\text{-var}\rangle = \emptyset$ demands. Let us consider the demand **show** $\delta_{l,k} = \emptyset$. There are many cases in its processing. First, if the variable $\delta_{l,k}$ is already empty, then the demand is trivially successful.

if $\delta_{l,k} = \emptyset$:
$\Rightarrow$ (SUCCESS)

Otherwise, the fact that e_l does get evaluated or not in contour k depends a lot on its parent expression, if it has one at all. If it does not have a parent expression, it means that e_l is the main expression of the program and, consequently, there is no possibility to prove that e_l does not get evaluated in contour k.[8]

if e_l is the main expression:
$\Rightarrow$ (FAILURE)

In case e_l does have a parent expression, let $e_{l'}$ be that expression. Let us consider the case where $e_{l'}$ is a λ-expression. It implies that e_l is the body of $e_{l'}$. Note that the evaluation of e_l in contour k has no direct connection with the evaluation of $e_{l'}$ in contour k. In fact, e_l gets evaluated in contour k if a closure c, resulting from the evaluation of $e_{l'}$ in some contour, gets called somewhere (at expression $e_{l''}$) in some (other) contour k' on a certain argument v in such a way that the resulting contour $\mathsf{call}(l'',c,v,k')$ in which the body of c must be evaluated is k. So the processing of the demand consists in emitting a bad call demand for each such abstract call. Note how the log matrices κ and χ are used to recover the circumstances under which the contours and closures were created.

[8]In fact, it is a little more complicated than that. We suppose here that the abstract variables contain the *minimal solution* for the evaluation constraints generated by the analysis framework. In these conditions, for l being the label of the program main expression, $\delta_{l,k}$ is non-empty if and only if k is the main abstract contour. For any other contour k', $\delta_{l,k'} = \emptyset$.

if $e_{l'} = (\lambda_{l'}\mathtt{x}.\ e_l)$:
$$\Rightarrow \left\{ \ \textbf{bad-call}\ l''\ c\ v\ k' \ \middle|\ \begin{array}{l} (l'',c,v,k') \in \kappa_k \ \wedge \\ \exists k'' \in \mathcal{C}ont.\ (l',k'') \in \chi_c \end{array} \right\}$$

Now, let us consider the case where $e_{l'}$ is a conditional. A conditional has three sub-expressions, so we first consider the case where e_l is the then-branch of $e_{l'}$. Clearly, it is sufficient to show that $e_{l'}$ is not evaluated at all in contour k. However, such a requirement is abusive. The sufficient *and* necessary condition for a then-branch to be evaluated (or not to be evaluated) is for the test to return (not to return, resp.) some true values.

if $e_{l'} = (\mathtt{if}_{l'}\ e_{l''}\ e_l\ e_{l'''})$:
$\Rightarrow$ **show** $\alpha_{l'',k} \subseteq \mathcal{V}al\mathcal{B}$

The case where e_l is the else-branch of the conditional is analogous. The else-branch cannot get evaluated if the test always returns true values.

if $e_{l'} = (\mathtt{if}_{l'}\ e_{l''}\ e_{l'''}\ e_l)$:
$\Rightarrow$ **show** $\alpha_{l'',k} \subseteq \mathcal{V}al\mathcal{T}rues$

The case where e_l is the test of the conditional can be treated as a default case. The default case concerns all situations not explicitly treated above. In the default case, to prove that e_l does not get evaluated in contour k requires a demonstration that $e_{l'}$ does not get evaluated in contour k either. This is obvious since the evaluation of a call, `cons`, `car`, `cdr`, or `pair?` expression necessarily involves the evaluation of all its sub-expressions. Similarly for the test sub-expression in a conditional.

otherwise:
$\Rightarrow$ **show** $\delta_{l',k} = \emptyset$

5.3.3 Bad Call Demands

We next describe how the bad call demands are processed. Let us consider this demand: **bad-call** $l\ f\ v\ k$. The expression e_l is necessarily a call and let $e_l = (_l e_{l'}\ e_{l''})$. There are two cases: either the specified call does not occur, or it does. If the call does not occur, then the demand is trivially successful.[9]

if $f \notin \alpha_{l',k}$ or $v \notin \alpha_{l'',k}$:
$\Rightarrow$ (SUCCESS)

In the other case, the specified call is noted into the *bad call log*. Another note is kept in order to later take care of all the bad calls at e_l in contour k. We call this operation *monitoring e_l in contour k*. More than one bad call may concern the same expression and the same contour. Because the monitoring is a crucial operation, it should have access to bad call informations that are as accurate as possible. So, it is preferable to postpone the monitoring as much as possible.

otherwise:
$\Rightarrow$ Insert (l,f,v,k) in the bad call log.
Flag (l,k) as a candidate for monitoring.

[9]Actually, in the current implementation, this case cannot occur. The demand is generated precisely because the specified call was found in the κ matrix. However, previous implementations differed in the way demands were generated and bad call demands could be emitted that were later proved to be trivially successful.

5.3.4 Split Demands

Direct Model Split

Let us now present the processing of the split demands. The processing differs considerably depending on the splittee. We start by describing the processing of the following demands: **split** $\mathcal{V}al\,\mathcal{C}\ P$ and **split** $\mathcal{V}al\,\mathcal{P}\ P$. These are easy to process because they explicitly prescribe a modification to the abstract model. The modification can always be accomplished successfully.

$\Rightarrow$ Update M_v with P
(SUCCESS)

Split α*-variables*

The most involving part of the demand processing is the processing of the **split** $\langle\alpha\text{-var}\rangle$ $\langle\text{sPat}\rangle$ demands. Such a demand asks for a splitting of the value of an expression in a certain contour, so that there is no more spreading of the values on the specified pattern. Let us consider the demand **split** $\alpha_{l,k}\ P$. The first possibility is that there is actually no spreading. Then the demand is trivially successful.

if $\neg\,(\alpha_{l,k} \boxtimes P)$:
$\Rightarrow$ (SUCCESS)

However, if there is spreading, then expression e_l has to be inspected, as the nature of the computations for the different expressions vary greatly. Let us examine each kind of expression, one by one. First, we consider the false constant. Note that this expression can only evaluate to #f. So its value cannot be spread on P, no matter which split pattern P is. For completeness, we mention the processing of the demand nevertheless.

if $e_l = \#\mathtt{f}_l$:
$\Rightarrow$ (SUCCESS)

Second, e_l may be a variable reference. Processing this demand is straightforward and it translates into a split demand onto a $\langle\beta\text{-var}\rangle$.

if $e_l = \mathrm{x}_l$:
$\Rightarrow$ **split** $\beta_{\mathrm{x},k,l}\ P$

Third, e_l may be a call. Clearly, this case is the most difficult to deal with. This is because of the way a call expression is abstractly evaluated. Potentially many closures are present in the caller position and many values are present in the argument position. It follows that a Cartesian product of all possible invocations must be done. In turn, each invocation produces a set that potentially contains many return values. So, in order to succeed with the split, each set of return values that is spread on the pattern must be split. And the sub-expressions of the call must be split in such a way that no invocation producing non-spread return values can occur in the same contour than another invocation producing incompatible non-spread return values. This second task is done with the help of the function SC (Split Couples) that prescribes split patterns that separate all the incompatible couples. An example follows the formal description of the processing of the split demand on a call.

if $e_l = (_l e_{l'}\ e_{l''})$:

$$
\begin{aligned}
\Rightarrow & \left\{ \textbf{split}\ \gamma_{c,k'}\ P \;\middle|\; \begin{array}{l} c \in \alpha_{l',k} \cap \mathcal{V}al\,\mathcal{C} \wedge \\ v \in \alpha_{l'',k} \wedge \\ k' = \mathsf{call}(l,c,v,k) \wedge \\ \gamma_{c,k'} \boxtimes P \end{array} \right\} \\
& \cup \left\{ \textbf{split}\ \alpha_{l',k}\ P_1 \;\middle|\; P_1 \in B \right\} \\
& \cup \left\{ \textbf{split}\ \alpha_{l'',k}\ P_2 \;\middle|\; P_2 \in C \right\} \\
& \text{where } A \quad = \left\{ ((c,v),\gamma_{c,k'}) \;\middle|\; \begin{array}{l} c \in \alpha_{l',k} \cap \mathcal{V}al\,\mathcal{C} \wedge \\ v \in \alpha_{l'',k} \wedge \\ k' = \mathsf{call}(l,c,v,k) \wedge \\ \neg\,(\gamma_{c,k'} \boxtimes P) \end{array} \right\} \\
& \qquad (B,C) = \mathrm{SC}(A,P)
\end{aligned}
$$

The following example illustrates the processing of the demand. Suppose that we want to process the demand **split** $\alpha_{l,k}\ \star$; that two closures may result from the evaluation of $e_{l'}$, say, $\alpha_{l',k} = \{c_1,c_2\}$; and that two values may be passed as arguments, say, $\alpha_{l'',k} = \{v_1,v_2\}$. Define k_{ij}, for $i,j \in \{1,2\}$, as $\mathsf{call}(l,c_i,v_j,k)$. Also suppose that

$$
\begin{array}{ll}
\neg\,(\gamma_{c_1,k_{11}} \boxtimes \star), & \gamma_{c_1,k_{12}} \boxtimes \star, \\
\gamma_{c_2,k_{21}} \boxtimes \star, \text{ and } & \neg\,(\gamma_{c_2,k_{22}} \boxtimes \star),
\end{array}
$$

that $\gamma_{c_1,k_{11}} \subseteq \mathcal{V}al\,\mathcal{B}$, and that $\gamma_{c_2,k_{22}} \subseteq \mathcal{V}al\,\mathcal{P}$. Closure c_1, when called on v_2, and closure c_2, when called on v_1, both return values that are spread on $\star$. It follows that their return values in those circumstances must be split. So, $\gamma_{c_1,k_{12}}$ and $\gamma_{c_2,k_{21}}$ must be split by the pattern $\star$. It is necessary for these two splits to succeed in order to make our original demand succeed. It is not sufficient, however. We cannot allow c_1 to be called on v_1 and c_2 to be called on v_2 under the same contour k. It is because the union of their return values is spread on $\star$. They are *incompatible*. This is where the SC function comes into play and its use:

$$
\mathrm{SC}(\{((c_1,v_1),\gamma_{c_1,k_{11}}),((c_2,v_2),\gamma_{c_2,k_{22}})\},\star)
$$

returns either $(\{\lambda_\star\},\emptyset)$ or $(\emptyset,\{\star\})$. In either case, a split according to the prescribed pattern, if successful, would make the two incompatible calls occur in different contours. If we suppose that the first case happens, the result of processing the original demand is:

$\Rightarrow$ **split** $\gamma_{c_1,k_{12}}\ \star$
split $\gamma_{c_2,k_{21}}\ \star$
split $\alpha_{l',k}\ \lambda_\star$

Fourth, e_l may be a λ-expression. The processing of this demand is simple as it reduces to a split on the abstract model of closures.

if $e_l = (\lambda_l \mathtt{x}.\ e_{l'})$:
$\Rightarrow$ **split** $\mathcal{V}al\,\mathcal{C}\ P$

Fifth, let us consider the case where e_l is a conditional. Two cases are possible: the first case is that at least one of the branches is spread on the pattern; the second is that each branch causes no spreading on the pattern but they are incompatible and the test sub-expression evaluates to both true and false values. In the first case, a conservative approach consists in splitting the branches that cause the spreading.

if $e_l = (\mathtt{if}_l\ e_{l'}\ e_{l''}\ e_{l'''}) \wedge (\alpha_{l'',k} \boxtimes P \vee \alpha_{l''',k} \boxtimes P)$:

$$
\Rightarrow \left\{ \textbf{split}\ \alpha_{l^{(n)},k}\ P \;\middle|\; l^{(n)} \in \{l'',l'''\} \wedge \alpha_{l^{(n)},k} \boxtimes P \right\}
$$

In the second case, it is sufficient to split on the type of the test sub-expression, as determining the type of the test sub-expression allows one to determine which of the two branches is taken and consequently knowing that the value of the conditional is equal to one of the two branches.

if $e_l = (\texttt{if}_l\ e_{l'}\ e_{l''}\ e_{l'''})$:
$\Rightarrow$ **split** $\alpha_{l',k}\ \star$

Sixth, our expression e_l may be a pair construction. The fact that the value of e_l is spread on the pattern implies first that the pattern has the form (P', P'') and second that the value of one of the two sub-expressions of e_l is spread on its corresponding sub-pattern (P' or P''). In either case, the demand is processed by splitting the appropriate sub-expression by the appropriate sub-pattern.

if $e_l = (\texttt{cons}_l\ e_{l'}\ e_{l''}) \wedge P = (P', P'') \wedge P' \in \langle\text{sPat}\rangle$:
$\Rightarrow$ **split** $\alpha_{l',k}\ P'$

if $e_l = (\texttt{cons}_l\ e_{l'}\ e_{l''}) \wedge P = (P', P'')$:
$\Rightarrow$ **split** $\alpha_{l'',k}\ P''$

Seventh, e_l may be a `car`-expression. In order to split the value of e_l on P, the sub-expression has to be split on $(P, \forall)$. However, there is the possibility that the abstract model of the pairs is not precise enough to abstract the pairs up the level of details required by $(P, \forall)$. If not, the model of the pairs has to be split first. If it is, the split on the sub-expression can proceed as planned.

if $e_l = (\texttt{car}_l\ e_{l'}) \wedge \mathcal{V}al\mathcal{P}$ is precise enough for $(P, \forall)$:
$\Rightarrow$ **split** $\alpha_{l',k}\ (P, \forall)$

if $e_l = (\texttt{car}_l\ e_{l'})$:
$\Rightarrow$ **split** $\mathcal{V}al\mathcal{P}\ (P, \forall)$

Eighth, if e_l is a `cdr`-expression, the processing is similar to that of a `car`-expression.

if $e_l = (\texttt{cdr}_l\ e_{l'}) \wedge \mathcal{V}al\mathcal{P}$ is precise enough for $(\forall, P)$:
$\Rightarrow$ **split** $\alpha_{l',k}\ (\forall, P)$

if $e_l = (\texttt{cdr}_l\ e_{l'})$:
$\Rightarrow$ **split** $\mathcal{V}al\mathcal{P}\ (\forall, P)$

Ninth, e_l must be a `pair?`-expression. Processing the demand simply consists in doing the same split on the sub-expression. To see why, it is important to recall that, if this case is currently being considered, it is because $\alpha_{l,k} \ltimes\!\!\!\rtimes P$. If $P = \star$, the type of the sub-expression must be found in order to find the type of the expression. If $P = (P', P'')$, the same split is required on the sub-expression since all the pairs of the `pair?`-expression come from its sub-expression. P cannot be $\lambda_\star$ or $\lambda_{l''}\ k'$, for $l'' \in \text{Lab}$, $k' \in \langle\text{m}k\text{Pat}\rangle$, because e_l can only evaluate to Booleans and pairs.

otherwise $e_l = (\texttt{pair?}_l\ e_{l'})$:
$\Rightarrow$ **split** $\alpha_{l',k}\ P$

Split β*-variables*

The next kind of split demands have a $\langle\beta\text{-var}\rangle$ as a splittee. Recall that a $\langle\beta\text{-var}\rangle$ indicates the name of a program variable and the label and contour where a reference to that variable occurs. Let us consider this particular demand: **split** $\beta_{\text{x},k,l}\ P$. Recall also that the contour k is a modeling contour pattern which consists in a list of modeling patterns, one per variable in the lexical environment visible from the expression e_l. Each modeling pattern represents a kind of bound in which the value of the corresponding is guaranteed to lie. The first modeling pattern corresponds to the innermost variable. The last corresponds to the outermost.

Note that the analysis framework does not compute the value of variable references using these bounds. As far as the framework is concerned, the whole contour is just a name for a particular evaluation context. In the framework, a reference to a variable x is computed by either inspecting the abstract variable $\beta_{\text{x},k}$ if x is the innermost variable or by translating it into a reference to x from the label l' of the λ-expression immediately surrounding e_l and contour k' in which λ-expression $e_{l'}$ got evaluated, creating a closure that later got invoked, leading to the evaluation of its body in contour k. For the details on variable references in the analysis framework, see [4]. Nonetheless, because of the way we implement the abstract model, a reference to a variable x from a label l, and in a contour k always produces values that lie inside of the bound corresponding to x in k.

Consequently, a split on a program variable involves a certain number of splits on the abstract models of call and cc. Moreover, consistency between abstract values also prescribes multiple splits on the abstract model. For example, if contour k results from the call of closure $\lambda_{l'}\ k'$ on a value v at label l'', and in contour k'', that is, $k = \text{call}(l'', \lambda_{l'}\ k', v, k'')$, then contour k cannot be more precise than k' about the program variable bounds it shares with contour k'. In turn, if closure $\lambda_{l'}\ k'$ results from the evaluation of $e_{l'}$ in contour k''', that is, $\lambda_{l'}\ k' = \text{cc}(l', k''')$, then contour k' cannot be more precise than k''' about the program variable bounds it shares with contour k'''. It follows that a split on a program variable, which can be seen as a refining of its bound in the local contour, requires the refining of a chain of contours and closure environments until a point is reached where the contour to refine does not share the variable with the closure leading to its creation.

Now, if we come back to the processing of **split** $\beta_{\text{x},k,l}\ P$, the first thing that must be verified is whether a reference to x from e_l in contour k produces values that are spread on pattern P. We denote such a variable reference by $\text{ref}(\text{x}, k, l)$. If no spreading occurs,[10] the demand is trivially successful, otherwise modifications to the model must be done.

if $\neg\ (\text{ref}(\text{x}, k, l) \ltimes\!\!\!\rtimes P)$:
$\Rightarrow$ (SUCCESS)

otherwise:
$\Rightarrow$ Update M_{l_m} with $(P'_m\ P_{m-1}\ \ldots\ P_0)$
  Update M_v with $\lambda_{l_{m+1}}\ (P'_m\ P_{m-1}\ \ldots\ P_0)$
  Update $M_{l_{m+1}}$ with $(P_{m+1}\ P'_m\ P_{m-1}\ \ldots\ P_0)$
  $\vdots$
  Update M_v with $\lambda_{l_n}\ (P_{n-1}\ \ldots\ P_{m+1}\ P'_m\ P_{m-1}\ \ldots\ P_0)$
  Update M_{l_n} with $(P_n\ \ldots\ P_{m+1}\ P'_m\ P_{m-1}\ \ldots\ P_0)$
  where
    $(\lambda_{l_m}\text{x}.\ \ldots(\lambda_{l_{m+1}}\text{y}_{m+1}.\ \ldots(\lambda_{l_n}\text{y}_n.\ \ldots\text{x}_l\ldots)\ldots)\ldots)$
    is the λ-expression binding x
    $k = (P_n\ \ldots\ P_{m+1}\ P_m\ P_{m-1}\ \ldots\ P_0)$
    $P'_m = P_m \cap P$

Split γ*-variables*

The last kind of demands is the split demand with a $\langle\gamma\text{-var}\rangle$ as a splittee. The processing of such a demand is straightforward since

[10] Once again, this case cannot occur in the current implementation.

the return value of a closure is the result of the evaluation of its body. Let us consider this particular demand: **split** $\gamma_{c,k}$ P. In case the return value is not spread on the pattern, the demand in trivially successful.

if $\neg\,(\gamma_{c,k} \bowtie P)$:
$\Rightarrow$ (SUCCESS)

otherwise:
$\Rightarrow$ **split** $\alpha_{l',k}$ P
where $c = \lambda_l\ k' \wedge e_l = (\lambda_l x.\ e_{l'})$

5.3.5 Call Site Monitoring

The processing rules have been given for all the demands. However, we add here the description of the monitoring of call sites. The monitoring of call sites is pretty similar to the processing of the demand **split** $\alpha_{l,k}$ P where e_l is a call. The difference comes from the fact that, with the monitoring, effort is made in order to prove that the bad calls do not occur. Let us consider the monitoring of call expression $({}_l e_{l'}\ e_{l''})$ in contour k. Let L_{BC} denote the bad call log. Potentially many closures may result from the evaluation of $e_{l'}$ and potentially many values may result from the evaluation of $e_{l''}$. Among all the possible closure-argument pairs, a certain number may be marked as bad in the bad call log and the others not. If no pair is marked as bad, then the monitoring of e_l in k is trivially successful.

if $\big((\alpha_{l',k} \cap \mathcal{V}al\,\mathcal{C}) \times \alpha_{l'',k}\big) \cap L_{\mathrm{BC}}(l,k) = \emptyset$:
$\Rightarrow$ (SUCCESS)

On the contrary, if all the pairs are marked as bad calls, then a demand is emitted asking to show that the call does not get evaluated at all.

if $\big((\alpha_{l',k} \cap \mathcal{V}al\,\mathcal{C}) \times \alpha_{l'',k}\big) \subseteq L_{\mathrm{BC}}(l,k)$:
$\Rightarrow$ **show** $\delta_{l,k} = \emptyset$

But in the general case, there are marked pairs and non-marked pairs occurring at the call site. It is tempting to emit a demand D asking a proof that the call does not get evaluated at all. It would be simple but it would not be a good idea. The non-marked pairs may abstract actual computations in the concrete evaluation of the program and, consequently, there would be no hope of ever making D successful.[11] What has to be done is to separate, using splits, the pairs that are marked and the pairs that are not. The (overloaded) SC function is used once again.

otherwise:
$\Rightarrow \{\textbf{split}\ \alpha_{l',k}\ P_1 \mid P_1 \in A\} \cup \{\textbf{split}\ \alpha_{l'',k}\ P_2 \mid P_2 \in B\}$
where $A = (\alpha_{l',k} \cap \mathcal{V}al\,\mathcal{C}) \times \alpha_{l'',k}$
$(B,C) = \mathrm{SC}(A, L_{\mathrm{BC}}(l,k))$

5.3.6 The Split Couples Function

We conclude this section with a short description of the SC function. SC is used for two different tasks: splitting closure-argument pairs according to the bucket in which the return values fall relatively to a split pattern P; splitting closure-argument pairs depending on the criterion that they are considered bad calls or not. In fact, those two tasks are very similar. In both cases, the set of pairs is partitioned into equivalence classes that are given either by the split pattern bucket or by the badness of the call. In order to separate two pairs (v_1, v_2) and (w_1, w_2) belonging to different classes, it is sufficient to provide a split that separates v_1 from w_1 or a split that separates v_2 from w_2. So, what SC has to do is to prescribe a set of splits to perform only on the first component of the pairs and another set of splits to perform only on the second component such that any two pairs from different classes would be separated. This is clearly possible since prescribing splits intended to separate any first component from any other is a simple task. Similarly for the second components. This way, *any* pair would be separated from *all* the others. Doing so would be overly aggressive, however, as there are usually much smaller sets of splits that are sufficient to separate the pairs.

Our implementation of SC proceeds this way. It first computes the equivalence classes. Next, each pair is converted into a genuine abstract pair (a modeling pattern). Then, by doing a breadth-first traversal of all the pairs simultaneously, splitting strategies are elaborated and compared. At the end, the strategy requiring the smallest number of splits is obtained. Being as little aggressive as possible is important because each of the proposed splits will have to be applied on one of the two sub-expressions of a call expression. And these sub-expressions may be themselves expressions that are hard to split (such as calls).

6 Experimental Results

6.1 Current Implementation

Our current implementation of the demand-driven analysis is merely a prototype written in Scheme to experiment with the analysis approach. No effort has been put into making it fast or space-efficient. For instance, abstract values are implemented with lists and symbols and closely resemble the syntax we gave for the modeling patterns. Each re-analysis phase uses these data without converting them into numbers nor into bit-vectors. And a projection using the pattern-matchers is done for each use of the cc, pc, and call functions.

Aside from the way demands are processed, many variants of the main algorithm have been tried. The variant that we present in Section 5 is the first method that provided interesting results. Previous variants were trying to be more clever by doing model changes concurrently with demand processing. This lead to many complications: demands could contain values and contours expressed in terms of an older model; a re-analysis was periodically done but not necessarily following each model update, which caused some demands to not see the benefits of a split on the model that had just been done; a complex system of success and failure propagation, sequencing of processing, and periodic processing resuming was necessary; etc. The strength of the current variant is that, after each model update, a re-analysis is done and the whole demand-propagation is restarted from scratch, greatly benefitting from the new analysis results.

In the current variant, we tried different approaches in the way the *best* model-updating demand is selected to be applied on the model. At first, we applied *all* the model-updating demands that were proposed by the demand processing phase. This lead to exaggerate

[11] This is because an analysis done using the framework is *conservative* (see [4]). That is, the computations made in the abstract interpretation abstract *at least* all the computations made in the concrete interpretation. So, it is impossible to prove that an abstract invocation does not occur if it has a concrete counterpart occurring in the concrete interpretation.

```
(letrec_1 map =
  (λ_2 op. (λ_3 l. (if_4 l_5 (cons_6 (_7 op_8 (car_9 l_10))
                                     (_11 (_12 map_13 op_14)
                                          (cdr_15 l_16)))
                        l_17)))
 (let_18 op1 = (λ_19 x. (car_20 x_21))
  (let_22 op2 = (λ_23 y. (_24 y_25 #f_26))
   (letrec_27 loop =
     (λ_28 data.
       (let_29 res1 = (_30 (_31 map_32 op1_33) (car_34 data_35))
        (let_36 res2 = (_37 (_38 map_39 op2_40) (cdr_41 data_42))
         (_43 loop_44 (cons_45 (cons_46 (cons_47 #f_48 #f_49)
                                        (car_50 data_51))
                               (cons_52 (λ_53 w. #f_54)
                                        (cdr_55 data_56))))))))
    (_57 loop_58 (cons_59 #f_60 #f_61))))))
```

Figure 14. Source of the `map-hard` benchmark

refining of the model, leading to massive space use. So we decided to make a selection of one of the demands according to a certain criterion. The first criterion was to measure how much the abstract model increases in size if a particular demand is selected. While it helped in controlling the increase in size of the model, it was not choosing very wisely as for obtaining very *informative* analysis results. That is, the new results were expressed with finer values but the knowledge about the program data flow was not always increased. Moreover, it did not necessarily help in controlling the increase in size of the analysis results. The second criterion, which we use now, measures how much the abstract model plus the analysis results increase in size. This criterion really makes a difference, although the demand selection step involves re-analyzing the program for all candidate demands.

6.2 Benchmarks

We experimented with a few small benchmark programs. Most of the benchmarks involve numeric computations using naturals. Two important remarks must be made. First, our mini-language does not include `letrec`-expressions. This means that recursive functions must be created using the Y combinator. Note that we wrote our benchmarks in an extended language with `let`- and `letrec`-expressions, and used a translator to reduce them into the base language. We included two kinds of `letrec` translations: one in which Y is defined once globally and all recursive functions are created using it; one in which a private Y combinator is generated for each `letrec`-expression. The first kind of translation really makes the programs more intricate as all recursive functions are closures created by Y. The second kind of translation loosely corresponds to making the analysis able to handle `letrec`-expressions as a special form. We made tests using both translation modes. Our second remark concerns numbers. Our mini-language does not include integers. Another translation step replaces integers and simple numeric operators by lists of Booleans and functions, respectively. Thus, integers are represented in unary as Peano numbers and operations on the numbers proceed accordingly. This adds another level of difficulty on top of the `letrec`-expression translation. For an example of translation, see Appendix A.

Our benchmarks are the following. `Cdr-safe` contains the definition of a function which checks its argument to verify that it is a pair before doing the access. It can be analyzed perfectly well by a 1-cfa, but not by a 0-cfa. `Loop` is an infinite loop. `2-1` computes the value of `(- 2 1)`. `Map-easy` uses the 'map' function on a short list of pairs using two different operators. `Map-hard` repetitively uses the 'map' function on two different lists using two different operators. The lists that are passed are growing longer and longer. This use of 'map' is mentioned in [7] as being impossible to analyze perfectly well by any *k*-cfa. The source code of this benchmark is shown in Figure 14. `Fib`, `gcd`, `tak`, and `ack` are classical numerical computations. `N-queens` counts the number of solutions for 4 queens. `SKI` is an interpreter of expressions written with the well known S, K, and I combinators. The interpreter runs an SKI program doing an infinite loop. The combinators and the calls are encoded using pairs and Booleans.

6.3 Results

Figure 15 presents the results of running our analysis on the benchmarks. Each benchmark was analyzed when reduced with each translation method (global and private Y). A time limit of 10000 "work units" has been allowed for the analysis of each benchmark. The machine running the benchmarks is a PC with a 1.2 GHz Athlon CPU, 1 GByte RAM, and running RH Linux kernel 2.4.2. Gambit-C 4.0 was used to compile the demand-driven analysis.

The column labeled "Y" indicates whether the Y combinator is Global or Private. The next column indicates the size of the translated benchmark in terms of the number of basic expressions. The columns labeled "total", "pre", "during", and "post" indicate the number of run time type checks still required in the program at those moments, respectively: before any analysis is done, after the analysis with the initial model is done, during, and after the demand-driven analysis. Finally, the computation effort invested in the analysis is measured both in terms of work units and CPU time.

The measure in column "total" is a purely syntactic one, it basically counts the number of call-, `car`-, and `cdr`-expressions in the program. The measure in "pre" is useful as a comparison between the 0-cfa and our analysis. Indeed, the initial abstract model used in our approach is quite similar to that implicitly used in the 0-cfa. An entry like **2**@23 in column "during" indicates that 2 run time type checks are still required after having invested 23 work units in the demand-driven analysis (this gives an idea of the convergence rate of the analysis).

When we look at Figure 15, the aspect of the results that is the most striking is the small improvements that the full demand-driven analysis obtains over the results obtained by the 0-cfa. Two reasons explain this fact. First, many run time type checks are completely trivial to remove. For instance, every `let`-expression, once translated, introduces an expression of the form `((λx. ...) ...)`. In turn, the translation of each `letrec`-expression introduces 2 or 3 `let`-expressions, depending on the translation method. It is so easy to optimize such an expression that even a purely syntactic detection would suffice. Second, type checks are not all equally difficult to remove. The checks that are removed by the 0-cfa are removed because it is "easy" to do so. The additional checks that are removed by the demand-driven phase are more difficult ones. In fact, the difficulty of the type checks seems to grow very rapidly as we come close to the 100% mark. This statement is supported by the numbers presented in [2] where a linear-time analysis, the *sub-0-cfa*, obtains analysis results that are almost as useful to the optimizer than those from the 0-cfa, despite its patent negligence in the manipulation of the abstract values.

Note how translating with a private Y per `letrec` helps both the 0-cfa and the demand-driven analysis. In fact, except for the `n-queens` benchmark, the demand-driven analysis is able to remove all type checks when private Y combinators are used. The success of the analysis varies considerably between benchmarks.

	Y	size	total	pre	during	post	units	time(s)
`cdr-safe`	G	17	4	1		0	5	0.02
	P	17	4	1		0	5	0.02
`loop`	G	32	11	0		0	1	0.02
	P	26	9	0		0	1	0.02
`2-1`	G	48	15	2	**1**@7	0	47	0.30
	P	42	13	2	**1**@7	0	48	0.26
`map-easy`	G	82	26	6	**4**@19	0	134	1.95
	P	76	24	6	**4**@19	0	134	1.76
`map-hard`	G	96	33	9	**6**@38 **5**@254 **3**@305 **1**@520	0	1399	76.26
	P	101	35	4	**2**@118	0	284	5.22
`fib`	G	141	40	12		12	10000	1480.57
	P	168	50	5	**4**@16 **3**@29 **2**@39 **1**@46	0	358	8.77
`gcd`	G	257	77	8	**7**@25 **6**@47 **5**@66 **4**@82 **3**@95 **2**@105 **1**@112	1	10000	7958.30
	P	328	103	6	**5**@19 **4**@35 **3**@48 **2**@58 **1**@65	0	8509	1088.58
`tak`	G	202	46	9		9	10000	1996.30
	P	218	52	4	**3**@13 **2**@23 **1**@30	0	240	11.63
`n-queens`	G	372	121	51		51	10000	15899.39
	P	454	151	11	**10**@34 **9**@65 **8**@93 **7**@118 **6**@140 **5**@1750	5	10000	1816.00
`ack`	G	162	49	5	**4**@16 **3**@29 **2**@39 **1**@46	1	10000	3948.32
	P	189	59	3	**2**@10 **1**@17	0	200	7.97
`SKI`	G	285	46	19	**15**@91 **13**@173 **11**@323 **9**@397 **7**@473 **6**@543 **5**@1474 **4**@3584	4	10000	841.13
	P	290	48	17	**13**@52 **11**@98 **9**@138 **8**@212 **5**@249 **4**@358 **3**@567 **1**@673	0	899	64.37

Figure 15. Experimental results

unrolling	1	2	4	8	16
units	176	280	532	1276	3724
time(s)	6.77	15.53	51.76	248.59	1592.93

Figure 16. The effect of the size of a program on the analysis work

Moreover, it is not closely related to the size of the program. It is more influenced by the style of the code. In order to evaluate the performance of the analysis on similar programs, we conducted experiments on a family of such programs. We modified the `ack` benchmark by unrolling the recursion a certain number of times. Translation with private Y is used. Figure 16 shows the results for a range of unrolling levels. For each unrolling level i, the total number of type checks in the resulting program is $43+19i$ if no optimization is done, 3 checks are still required after the program is analyzed with the initial model, and all the checks are eliminated when the demand-driven analysis finishes. We observe a somewhat quadratic increase in the analysis times. This is certainly better than the exponential behavior expected for a type analysis using lexical-environment contours.

7 Conclusions

The type analysis presented in this paper produces high quality results through the use of an adaptable abstract model. During the analysis, the abstract model can be updated in response to the specifics of the program while considering the needs of the optimizer. This adaptivity is obtained by the processing of demands that express, directly or indirectly, the needs of the optimizer. That is, the model updates are demand-driven by the optimizer. Moreover, the processing rules for the demands make our approach more robust to differences in coding style.

The approach includes a flexible analysis framework that generates analyses when provided with modeling parameters. We proposed a modeling of the data that is based on patterns and described a method to automatically compute useful modifications on the abstract model. We gave a set of demands and processing rules for them to compute useful model updates. Finally, we demonstrated the power of the approach with some experiments, showing that it analyzes precisely (and in relatively short time) a program that is known to be impossible to analyze with the k-cfa. A complete presentation of our contribution can be found in [3]. An in-depth presentation of all the concepts and algorithms along with the proofs behind the most important theoretical results are also found there.

Except for the ideas of abstract interpretation and *flexible* analyses, the remainder of the presented work is, to the best of our knowledge, original. Abstract interpretation is frequently used in the field of static analysis (see [2, 7, 8, 9]). The k-cfa family of analyses (see [8, 9]) can, to some extent, be considered as flexible. The configurable analysis presented in [2] by Ashley and Dybvig can produce an extended family of analyses, but *at compiler implementation time*. Our analysis framework (see [4]) allows for more subtlety and can be modified *during the analysis*.

We can think of many ways to continue research on this subject: extended experiments on our approach in comparison to many other analyses; the speed and memory consumption of the analysis; incremental re-analysis (that is, if analysis results R_1 were obtained by using model M_1, and model M_2 is a refinement of model M_1, then compute new results R_2 efficiently), better selection of the model-updating demands. Moreover, language extensions should be considered to handle a larger part of Scheme and extending our demand-driven approach to other analyses. There are also more theoretical questions. We know that analyzing with the analysis framework and adequate modeling parameters is always at least as powerful as the k-cfa (or many other analyses). However, it requires the parameters to be given by an oracle. What we do not know is whether our current demand-driven approach is always at least as powerful as the k-cfa family. We think it is not, but do not yet have a proof.

```
(letrec_1 ack =
   (λ_2 m. (λ_3 n. (if_4 (=_5 m_6 0_7)
                   (+_8 n_9 1_10)
                   (if_11 (=_12 n_13 0_14)
                     (_15 (_16 ack_17 (−_18 m_19 1_20)) 1_21)
                     (_22 (_23 ack_24 (−_25 m_26 1_27))
                        (_28 (_29 ack_30 m_31) (−_32 n_33 1_34)))))))
 (_35 (_36 ack_37 4_38) 0_39))
```

Figure 17. The `ack` benchmark, before expansion

```
(_1 (λ_2 Y. (_3 (λ_4 +p.
              (_5 (λ_6 +.
                 (_7 (λ_8 -p.
                    (_9 (λ_10 -. (_11 (λ_12 =p.
                                   (_13 (λ_14 =.
                                      (_15 (λ_16 ackp.
                                         (_17 (λ_18 ack. (_19 (_20 ack_21 (cons_22 #f_23 (cons_24 #f_25 (cons_26 #f_27 (cons_28 #f_29 #f_30))))) #f_31))
                                            (_32 Y_33 ackp_34)))
                                       (λ_35 ackf. (λ_36 m. (λ_37 n. (if_38 (_39 (_40 =_41 m_42) #f_43)
                                                                  (_44 (_45 +_46 n_47) (cons_48 #f_49 #f_50))
                                                                  (if_51 (_52 (_53 =_54 n_55) #f_56)
                                                                   (_57 (_58 ackf_59 (_60 (_61 -_62 m_63) (cons_64 #f_65 #f_66))) (cons_67 #f_68 #f_69))
                                                                   (_70 (_71 ackf_72 (_73 (_74 -_75 m_76) (cons_77 #f_78 #f_79)))
                                                                     (_80 (_81 ackf_82 m_83) (_84 (_85 -_86 n_87) (cons_88 #f_89 #f_90)))))))))))
                                  (_91 Y_92 =p_93)))
                              (λ_94 =f. (λ_95 x. (λ_96 y. (if_97 x_98 (if_99 y_100 (_101 (_102 =f_103 (cdr_104 x_105)) (cdr_106 y_107)) #f_108)
                                                                   (if_109 y_110 #f_111 (cons_112 #f_113 #f_114))))))))
                   (_115 Y_116 -p_117)))
                (λ_118 -f. (λ_119 x2. (λ_120 y2. (if_121 y2_122 (_123 (_124 -f_125 (cdr_126 x2_127)) (cdr_128 y2_129)) x2_130))))))
             (_131 Y_132 +p_133)))
          (λ_134 +f. (λ_135 x3. (λ_136 y3. (if_137 x3_138 (cons_139 #f_140 (_141 (_142 +f_143 (cdr_144 x3_145)) y3_146)) y3_147))))))
 (λ_148 f. (_149 (λ_150 g. (_151 g_152 g_153)) (λ_154 h. (λ_155 z. (_156 (_157 f_158 (_159 h_160 h_161)) z_162))))))
```

Figure 18. The `ack` benchmark, after expansion

Other researchers have worked on *demand-driven analysis* but in a substantially different way (see the work of Duesterwald et al. [5], Agrawal [1], and Heintze and Tardieu [6]). These approaches do not have an abstract execution model that changes to suit the program. Their goal is to adapt well-known analysis algorithms into variants with which one can perform what amounts to a lazy evaluation of the analysis results.

8 Acknowledgments

The authors thank the anonymous referees for their careful review and the numerous constructive comments.

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

9 References

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

[2] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, jul 1998.

[3] D. Dubé. *Demand-Driven Type Analysis for Dynamically-Typed Functional Languages*. PhD thesis, Université de Montréal, 2002. Available at: `http://www.iro.umontreal.ca/˜dube`.

[4] D. Dubé and M. Feeley. Demand-driven type analysis: an introduction. In *Proceedings of the Workshop on Scheme and Functional Programming 2001*, pages 21–32, sep 2001.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

A Translation of a Benchmark

In Section 6, we mention that `letrec`-expressions and numerical operations in the benchmarks have to be translated into the base language. We illustrate this process using the `ack` benchmark. Figure 17 shows the source code of the benchmark and Figure 18 shows the program after translation. Note that numbers have been replaced by `cons`-expressions and that the necessary arithmetic operators have been introduced as functions. Also note that the Y combinator is bound first and that the recursive functions ('ack' and three arithmetic operators) are created using it.