Université de Montréal

# Demand-Driven Type Analysis
# for Dynamically-Typed Functional Languages

par

Danny Dubé

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de Ph.D.

en Informatique

Août, 2002

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Demand-Driven Type Analysis
for Dynamically-Typed Functional Languages

présentée par:

Danny Dubé

a été évaluée par un jury composé des personnes suivantes:

Gilles Brassard
président-rapporteur

Marc Feeley
directeur de recherche

Alain Tapp
membre du jury

Matthias Felleisen
examinateur externe

Gilles Brassard (par interim)
représentant du doyen de la FES

# Résumé

Nous présentons une nouvelle analyse de types destinée aux langages typés dynamiquement qui produit des résultats de grande qualité à un coût qui la rend utilisable en pratique. Bien que statique, l'analyse est capable de s'adapter aux besoins de l'optimiseur et aux caractéristiques du programme à compiler. Le résultat est un analyseur qui se modifie rapidement pour être en mesure de mieux effectuer son travail sur le programme. Des tests démontrent que notre approche peut user de passablement d'intelligence pour permettre la réalisation de certaines optimisations.

L'analyse est adaptable parce qu'elle est effectuée à l'aide d'un cadre d'analyse paramétrisable qui peut produire des instances d'analyses à partir de modèles abstraits. Ces modèles abstraits peuvent être remplacés au cours de l'analyse du programme. Plusieurs propriétés du cadre d'analyse sont présentées et démontrées dans ce document. Parmi celles-ci, on retrouve la garantie de terminaison associée à toute instance d'analyse produite à l'aide du cadre, la capacité d'analyser parfaitement tout programme qui se termine sans erreur et la capacité d'imiter plusieurs analyses conventionnelles.

Les modifications apportées au modèle abstrait en fonction des besoins de l'optimiseur le sont grâce à l'utilisation de demandes et de règles de traitement des demandes. Les demandes décrivent des requêtes pour la démonstration de propriétés jugées utiles à l'optimiseur. Les règles de traitement permettent la traduction de demandes décrivant les besoins de l'optimiseur en des directives précises de modifications au modèle abstrait. Chaque directive de modification du modèle peut apporter une aide directe à l'optimiseur parce que les règles de traitement font en sorte que des demandes justifiées sont transformées en d'autres demandes justifiées.

Une approche d'analyse sur demande complète basée sur le *pattern-matching* est décrite et a été implantée. Le prototype implantant cette approche a démontré le potentiel considérable de nos travaux. Il faudra encore effectuer d'autres recherches avant qu'on puisse utiliser couramment notre approche dans les compilateurs. C'est toutefois compréhensible si on considère que tous nos travaux, outre les idées liées aux analyses statiques conventionnelles, sont une contribution originale.

**Mots-clés** : analyse sur demande — analyse adaptable — analyse statique — analyse de types — techniques de compilation — optimisation de programmes

# Abstract

We present a new static type analysis for dynamically-typed languages that produces high quality results at a cost that remains practicable. The analysis has the ability to adapt to the needs of the optimiser and to the characteristics of the program at hand. The result is an analyser that quickly transforms itself to be better equipped to attack the program. Experiments show that our approach can be pretty clever in the optimisations that it enables.

The analysis is adaptable because it is accomplished using a parametric analysis framework that can instantiate analyses by building them from abstract models. The abstract models can be changed during the analysis of the program. Many properties of the analysis framework are presented and proved in the dissertation. Among which there is the guarantee of termination of any analysis instance it produces, the capacity to analyse perfectly well error-free terminating programs, and the ability to mimic many conventional static analyses.

Modifications to the abstract model in response to the needs of the optimiser are realised through the use of demands and demand processing rules. Demands express a request for the demonstration of a property deemed useful to the optimiser. The processing rules allow demands that directly express the needs of the optimiser to be translated into precise proposals of modifications to the abstract model. Each modification to the model that is proposed is potentially directly helpful to the optimiser because the processing rules ensure that pertinent demands are translated into other pertinent demands.

A complete approach of demand-driven analysis based on pattern-matching is exposed and has been implemented. The prototype implementing the approach has demonstrated that our work has great potential. Further research has to be conducted to make the method usable in everyday compilers. Still, this is understandable, considering that our whole work, except the notions related to conventional static analysis, is original material.

**Key-words**: demand-driven analysis — adaptable analysis — static analysis — type analysis — compilation techniques — program optimisation

# Contents

# List of Tables

# List of Figures

# Remerciements

Je tiens à remercier ma copine, Marie-Lisa. Elle a toujours été encourageante et a su m’apporter la motivation nécessaire. Merci à mes parents, ma soeur et son mari. Tous m’ont accompagné dans mon cheminement et ont constitué un milieu réconfortant durant tous les moments, heureux et pénibles.

Je tiens par-dessus tout à remercier mon directeur de thèse, Marc Feeley, pour son support, tant moral que financier et technique. Il a su être aussi patient qu’il fallait l’être avec moi. Il a toujours cru en moi, plus que je ne pouvais croire en moi-même. Il m’a toujours témoigné un grand respect, même quand j’étais une peste d’entêtement.

Merci à toute la grande famille au complet, à mes amis et à mes camarades à l’école. Tout particulièrement : Odi pour m’avoir aidé à garder la forme et dont la compagnie est toujours agréable ; Sébastien pour être M. Divertissement en personne ; Mohamad qui m’a apporté une authentique aide et avec qui j’allais noyer ma déprime dans la caféine ; Diane qui a insisté avec tellement d’énergie pour que je complète mon doctorat ; le très constant (et comique) Mario ; Dominique ; Martin ; Wissam ; Fernanda ; Éric ; Étienne et Jean-François qui ont été si tannants et qui ont subi mes foudres tellement souvent (“Danny, t’es pas parlable !”) ; et François qui m’a aidé à découvrir ma vraie nature de dictateur.

L’aide financière des organismes subventionnaires CRSNG et FCAR m’a été précieuse. Elle n’a pas été vaine, après tout.

# Chapter 1

# Introduction

## 1.1 A Gentle Introduction

Very high quality type analysis can be performed on programs written in a dynamically-typed functional language while maintaining control over the analysis time. A quality type analysis is achieved by using a "clever" adaptive analysis method called demand-driven analysis. Although the method does not come from traditional artificial intelligence, it allows the analysis to adapt to the characteristics of the program at hand in ways that seem rather intelligent. But all this is quite vague, so let us proceed from the beginning.

Program analyses that are used for optimisation purposes are always stretched between two contradictory goals: quality and efficiency. Indeed, the user of a compiler wants the compiler to produce the best possible code while taking the least possible time to do so. Unfortunately, these desires are incompatible.

Roughly speaking, in the case of type analysis, two kinds of analyses exist, depending on which goal is considered to have priority. *Fast analyses* aim the efficiency of the analysis while *heavy analyses* aim the quality of the generated code. The fast ones feature reasonable analysis times and obtain results of a fair quality. The heavy ones inspect the program very closely and do not feature reasonable analysis times. Commonly used compilers that perform some type analysis use a fast one because the heavy ones are too costly in practice.

Of course, the user's desire to have his cake and eat it too is unrealistic but a relaxed version is still interesting. What we are interested in is a type analysis of very high quality that can be performed within times that remain practical. In our opinion, an analysis featuring practical times is crucial if we want our type analysis or a derivative to eventually be applied in some routinely used compiler. Despite the fact that our goal is relaxed, it still seems to be a naïve, "spoiled child's" desire. It seems to disregard the apparently strong relation between quality and efficiency that years of research in type analysis have outlined. Until now, this empirical relation has brought the user to expect a certain cost for a certain quality. Our childish desire lies in the high-quality part of the spectrum while incurring a cost that is well under the one that the quality-efficiency relation suggests. Is it reasonable to aim at such a goal?

We believe it is reasonable because a small amount of cleverness often pays off more than a lot of brute force. This is so in many aspects of real life and in computer science, too. For example, during a war, the army with the greatest number of soldiers and the best equipment does not necessarily defeat its enemy if it is poorly directed. In computer science, an $O(n \log n)$ algorithm can outperform an $O(n^2)$ algorithm, even if the latter is run on a computer that is faster by orders of magnitude. However, discovering the better algorithm requires careful thought.

But what clever thing could be done about type analysis? This thesis has its origins in an innocent sounding remark by my supervisor, roughly paraphrased as: "It would be nice to have an analysis that is very powerful but that uses its power only as much as needed by the optimiser to perform its job." We all know something that has this kind of behaviour; that is, something powerful but always trying to do as little as possible: a human. Let us imagine an optimising compiler where the type analysis would be done by a human; say, Mr. D. Let us describe the way Mr. D would proceed in analysing a program.

Mr. D would use his *intelligence* to perform the analysis. And he would perform a very good one. Indeed, he wants to help the compiler to produce highly optimised code. But he would use his intelligence mostly *where* it would really help the optimiser. That is, Mr. D is *lazy*. If an easy check allows the optimiser to improve a particular piece of code, Mr. D will not waste his time by making a complex proof involving the full extent of his mathematical knowledge.

The mental work performed by Mr. D can be divided in two parts: *raw* program *analysis* and *reasoning* about the task of analysing the program. The raw analysis part is essentially similar to what conventional analyses do. On the other hand, he does the reasoning part by inspecting the program, by looking at the analysis results, by inventing new raw analyses, by searching for the right invariants, etc. The raw analysis part can be done by hand or by writing an algorithm and running it on a machine. It does not matter how it is done. It is mechanical work, anyway. But Mr. D is able to do the reasoning part only because he is intelligent and *understands* what he is doing.

To summarise Mr. D's work, we would say that he is intelligent, he is lazy, and he knows what he is doing. He is able to analyse the program while he is also able to elaborate strategies about the way he should analyse the program. The demand-driven analysis approach that we introduce in this dissertation is inspired by the clever behaviour of Mr. D. Our approach features the same division of the work into a raw analysis part and a reasoning part. The reasoning part is able to modify the way raw analysis is done. The approach features laziness as the reasoning part is goal-driven: it takes care of the needs of the optimiser and only of these; any modification to the way raw analysis is done derives from those needs. Up to this point, our approach seems to act exactly as Mr. D. But, as expected, there is a difference and it lies in the fact that our approach does not understand what it is doing. It is only a combination of numerous deterministic algorithms and it does not exhibit any sign of learning or understanding whatsoever. Nevertheless, experiments have shown that it exhibits considerable cleverness in the execution of its task. This is satisfying as only intelligence is required, not consciousness.

## 1.2 Some More Precisions

The purpose of our type analysis is to help the optimiser to remove unnecessary dynamic *safety* type tests from the executable. The code resulting from the compilation of a program written in a dynamically-typed language includes dynamic safety type tests in the code of many primitive operations. For example, let us consider the following Scheme[1] expression: `(car x)`. This expression extracts the object in the first field of the pair contained in '`x`', provided '`x`' really contains a pair. Since Scheme is dynamically-typed, '`x`' could potentially contain objects of any type, depending on the computations done by the program. Consequently, the '`car`' function must perform a safety type tests before it can extract the

[1] For a reference to the Scheme language, see [51].

contents of the first field. Safety tests guarantee that the execution of the program proceeds safely.

If no test were performed before '`car`' did the extraction, the extraction could trigger an illegal operation at the hardware or operating system level and an abnormal termination of the program would occur. Or the illegal extraction could go undetected and cause a corruption of the data of the program, leading to potentially disastrous consequences. High-level languages such as Scheme are designed with safe execution in mind. Consequently, it is natural to include such dynamic safety tests in the executable.

Of course, these safety tests incur a penalty in the efficiency of the executable program. So it is perfectly understandable to want to avoid the added inefficiency. A common way to do so consists in telling the compiler to omit the inclusion of those tests. All potentially illegal operations made by the executable program then go unchecked and result in low-level crashes or program misbehaviour in case of an error. In this work, we choose not to consider this "solution". We prefer to insist on keeping the safety of the execution and turn to another option: safe optimisations. For some operations made by the program, the compiler may be able to determine that they can never go wrong. Safe optimisations can be enabled only in these cases. For example, the `(car x)` expression need not include a dynamic safety test if the compiler is able to determine that '`x`' cannot contain anything else than pairs. The demonstrations needed to trigger safe optimisations are obtained through the use of *static analysis*.

But what is a static analysis? It is the gathering of informations about the execution of the program. The nature of the informations that are gathered depends on what the optimiser needs to perform its task. They may relate to the heap-space usage, the liveliness of the objects at run-time, the may-alias information, or something else. In this work, the informations that interest us is the type of the values involved in the computations done by the program. What is particular to static analyses is the method that is used to gather the informations: a phony execution of the program or some other process that *does not* involve its real execution. It is mandatory to avoid the real execution because its duration is unknown (and possibly infinite). On the other hand, the phony execution requires the manipulation of phony values only for a bounded number of steps. So it is fast (and predictable) enough to be a part of a compilation. The reader may find numerous examples of static analyses in [3].

Despite the fact that the static execution is phony, it is designed in such a way that it has a mathematical connection to the real execution. Consequently, the results of the phony execution constitute the desired informations about the real execution. In general, the informations are only approximations of what could be observed if the program were really run. Moreover, in order to be useful to the optimiser, these informations have to be *conservative*.

When we say that the gathered informations have to be conservative, it means that they must take into account *at least* all possible behaviours of the program. But why is it “at least” and not “at most” or “the best approximation of”? Because of the nature of typical optimisations. Optimisations, such as the removal of safety type tests, require a particular property to be true for all possible executions of the program. Consequently, if the property is true for all behaviours listed in the description, then it has to be true for all concrete behaviours of the program. For example, if the analysis says that ‘`x`’ can contain nothing else than pairs, then, during the concrete execution of the program, it is certain that ‘`x`’ contains a pair (at least, if ‘`x`’ ever comes to existence) and `(car x)` can be optimised.

In opposition to the conservativeness of the analysis results, there is the need of the optimiser for results that are as useful as possible. It is clear that obtaining a conservative analysis is easy. We only need to write an analysis that pretends that anything *may* happen during the execution. Note that these analysis results are certainly conservative. However, the analysis results thus produced would not be useful as the property allowing optimisations to be performed would not be true in general (according to the results). For example, if the type analysis blindly determines that ‘`x`’ potentially contains objects of any type (which is true), then the optimisation of `(car x)` cannot be performed.

Essentially, the best interest of the analyser is to overestimate the description as little as possible while it must imperatively avoid underestimating the description. Reducing the overestimation as much as possible requires increasing the computational effort put into the analysis. However, an increase in the computational effort means that the compilation time increases, too. It is clear that choosing a compromise between the quality of the description (the smallness of its overestimation) and the compilation times is a difficult choice.

This difficulty in the choice of an analysis, in particular in the choice of a type analysis, makes the conventional type analyses inappropriate almost all the time. Let us explain ourselves. When a particular program is analysed, the analysis may be too coarse, producing results that are too overestimated to be really useful to the optimiser. Or it may be too strong and time would be wasted because sufficiently accurate results could have been produced by a much more efficient analysis. Surprisingly, the analysis may sometimes be both too coarse *and* too strong at the same time for the program at hand. This is the case when parts of the programs are easier to analyse than others. That is, the difficulty of producing analysis results accurate enough to trigger the optimisation of certain expressions may be much greater than for other expressions. This fact is made obvious by an example. Suppose that our `(car x)` expression occurs in two places in the program. The first occurrence is in expression `(if (pair? x) (car x) ...)`[2] and the second is in `(let ((x (get-lost foo bar))) (car x))`. Suppose that the `get-lost` function is extremely complex. Then a heavy conventional analysis could well be both too strong and too coarse for the program at the same time.

The fundamental reason behind the inappropriateness is that conventional analyses use a fixed *abstract model*. We need to introduce the meaning of the "abstract model" term. We mention just above that an analysis is done by performing a phony execution of the program. This phony execution is often performed using abstract interpretation.[3] During the abstract interpretation of a program, phony values are manipulated, instead of concrete values as in *concrete* interpretation. These phony values are called *abstract values*. Also, during abstract interpretation, expressions are evaluated in phony contexts, not in concrete contexts (lexical environment, current continuation, etc). An important difference between concrete values and phony values is that, while concrete values are defined by the language, the definition of the phony values has to be chosen by the implementer of the analysis. A short introduction to abstract interpretation is given in [19].

Taken together, the values and contexts that are to be used by an analysis, constitute the abstract model. Roughly speaking, the abstract model indicates under which simplistic point of view the execution of the program is going to be modelled. Since conventional analyses use a fixed abstract model, this point of view cannot change and it leads immediately to the inappropriateness of the analyses. Since the inappropriateness of the conventional analyses comes from the fixedness of the abstract model behind them, then clearly the solution is to use some adaptive abstract model.

[2] The `pair?` function is a predicate that tests whether its argument is a pair or not.

[3] Not all static analyses are done using abstract interpretation. There are other kinds of static analyses. Nevertheless, in all cases, there is an abstract model behind the analysis.

A direct consequence of the goal-driven nature of our type analysis is that the analyser and the optimiser must collaborate. This collaboration necessarily comprises two elements. First, the needs of the optimiser have to be expressed in some way. Second, the analyser has to react in a positive way to the needs expressed by the optimiser. The first element is quite simple. The property required for a particular optimisation to get enabled is well-defined and, usually, relatively easy to formalise. It is sufficient to choose some formalism in which the needs can be expressed. We illustrate the elements with our running example. In the hope of removing the safety type test in expression `(car x)`, the optimiser expresses its need by emitting a request like: "I would like to see a demonstration that '`x`' can only contain pairs." The analyser then has to do its best to fulfil the need of the optimiser.

The second element, however, is difficult and it is the core of our work. For the analyser, to be able to take care of the needs of the optimiser means that it must be able to detect when the analysis currently performed does not allow an optimisation to be enabled and, if it is the case, to adapt the analysis with the intention to enable the optimisation. In order to have an analyser capable of doing so, two new elements have to be provided. First, the analyser must have the ability to change the analysis it performs *while* the compiler is processing the program. That is, the analyser has to be able to change the abstract model behind the analysis at will. Second, a decision procedure has to be included in the analyser to let it determine *how* the abstract model ought to be modified. Indeed, a lot of freedom is granted to the analyser by the adaptivity of the abstract model and this freedom must not be used mindlessly. The first element can be realised without too much difficulty but the second remains quite a challenge. Clearly, the second element is the one that seems to require *understanding* and *intelligent reasoning.* Nonetheless, the demand-driven analysis that we propose possesses the desired flexibility and adaptivity.

Now that we have a more precise description of the requirements for the analyser, especially those concerning its adaptivity, we come back to our goal for the quality expected from the analyser. We do not simply expect a high-quality analysis at a practical cost, where the quality is comparable to that of the heavy analyses. We expect an even higher quality. Our expectations are justified by the adaptivity of the analyser. By its adaptivity, the analyser ought to spend the minimum of effort to enable the easy optimisations and invest more time on harder optimisations. Each optimisation ought to be taken care of using an effort corresponding to its difficulty. Since the spectrum of the difficulty of optimising the various expressions of the programs is typically very wide, the analyser is able to trigger the optimisation of a maximum of expressions for the time it consumes. On the other hand, fast conventional analyses only trigger the optimisation of the easy expressions. Heavy ones may waste huge amounts of resources on easy and intermediate expressions by applying an ill-adapted tedious procedure that is nevertheless too weak for the slightly more difficult expressions. The demand-driven analyser, by its reasoning about the needs of the optimiser and the strategies it elaborates, ought to find the *specific* modifications to the analysis that are necessary to trigger the optimisation of the more difficult expressions. In other words, it ought to trigger the optimisation of more difficult expressions because it is able to produce a "well-tailored" analysis.

## 1.3 Sketch of a Solution

Before we present a quick overview of the solution, we need to come back to the optimisation that interests us. We concentrate on the elimination of dynamic *safety* type tests. The other type tests do not interest us. Those include the explicit tests written by the programmer himself, such as the one in expression `(if (pair? x) ... ...)`, and the implicit ones that are not related to safety, such as the type tests performed by the garbage collector when it traverses the heap-allocated objects. The difference between the safety tests and the others is that the outcome of the safety tests is highly predictable. In fact, during the execution of a bug-free program, all safety type tests have a positive outcome. On the other hand, the explicit tests are precisely inserted by the programmer because he *wishes* these tests to be performed. Then it is reasonable to assume that these tests have an active purpose and that they result in both outcomes. Consequently, these tests are rarely redundant. The safety tests, on the contrary, are in most cases redundant and can be removed (if identified as such). They are a more valuable target for the analyser.

Not only are the safety type tests a valuable target, but their high predictability forms the basis of the reasoning made by the demand-driven analysis. The analyser concentrates on the needs of the optimiser that it considers to be plausibly realisable. The other needs of the optimiser are not less legitimate but there is no evidence that they have a reasonable chance of being realisable and they provide no clue on how to elaborate an analysis strategy to enable them. For example, let us consider call `(f x)` and suppose that the optimiser is able to improve the code produced for a call when only one function can possibly be invoked there (e.g. by replacing the generic invocation sequence by a direct call). The need of the optimiser consists in obtaining the confirmation that '`f`' contains only one particular function. It is a noble request as it would be profitable to the code if the confirmation could be obtained. However, there is no indication that '`f`' contains only one particular function. In fact, functional languages are notable for using higher-order functions. So it would be perfectly normal to see '`f`' contain different functions during the execution of the program. On top of the low plausibility of this need, there is the technical problem that this need provides no cue to the demand-driven analyser on how to answer it successfully.

Now we give an overview of the way we obtain an analyser that is adaptive and that is able to reason about the way to modify the analysis it performs. The analyser is adaptive because it uses an *analysis framework* instead of a fixed analysis. Roughly speaking, the analysis framework is the *shell* of an analyser. It contains all the usual mechanisms needed by an analyser. However, it does not include an abstract model. The framework has a parameter through which it receives an abstract model. When passed an abstract model and a program, it performs the type analysis prescribed by the model on the received program. The output of the framework is the analysis results. The latter are exactly those that would be obtained if a true analyser incorporating the given abstract model would have been used on the program.

The analysis framework has many useful properties. Any analysis that it instantiates (through the reception of an abstract model) is guaranteed to terminate and is conservative. The framework is able to mimic the behaviour of many conventional analyses. It is very powerful: given a bug-free program and an appropriate model, it produces analysis results that allows the optimiser to remove *all* safety type tests. Unfortunately, it is generally unfeasible to decide if an "appropriate" model exists.

As to the reasoning procedure that elaborates new analysis strategies according to the needs of the optimiser, we have two options. Either we create a (good old) AI program, or we create a heuristic based on a limited set of simple and mechanical rules. In all cases, the best that can be done is to obtain a heuristic since the optimisation problem toys with undecidable properties. We choose the mechanical rules. We give the reasons behind this choice in the next chapter.

The abstract models that we use are based on *patterns*. The patterns are similar to those found in languages that include pattern-matching, such as Haskell,[4] ML,[5] and Prolog.[6] At the heart of the reasoning procedure used in the demand-driven analysis, there are... the demands. Broadly speaking, demands are requests for the demonstration of facts that are deemed useful to the optimiser. The demands directly constitute the formalism in which the needs of the optimiser are expressed. But they also express other, indirect requests which are produced through the reasoning process. For example, apart from the syntax, the request of the optimiser "I would like to see a demonstration that '`x`' can only contain pairs." that we mention above is in fact a demand.

The demands by themselves are not an active component of our approach. *Demand processing rules* form the engine of the reasoning process. They translate existing demands into new ones with the intent to elaborate a strategy on how to modify the analysis. The reasoning obtained through the processing of demands is reminiscent of the resolution algorithm used by Prolog. Our demand processing rules come, shall we say, from the top of our hat. They are not perfectly arbitrary, however. They are relatively simple rules that make a lot of sense and they obey two principles that we only mention here: *sufficiency* and *necessity*. These principles are responsible for the cleverness shown by the analyser and for keeping the analyser from letting the analysis degenerate to a heavy, impractical one.

Globally, the demand-driven analysis is a cycle made of two phases. One phase consists in analysing the program using the current abstract model (raw analysis). The other consists in modifying the abstract model through demand processing (reasoning). If all the safety tests are eventually removed, the cycle ends. In the other case, the cycle would not end were it not for a time limit placed by the user on the computational resources allotted to the analysis. This unusual approach is consistent with our view that more precise results are expected from the analyser if it is given more time. At least, is makes as much sense to let an analyser work for a specified amount of time as it does to let an analyser work for an *a priori* unknown amount of time up to the completion of its algorithm. In either case, the user has no guarantee on the extent of the optimisations. Having a limit on the time taken by the analysis is even more user-friendly. Moreover, the limit on the resources need not necessarily be wall-clock time. It may be space or the number of logical steps performed by the demand-driven analysis. Interestingly, this last measure has some kind of deterministic

[4] For a reference to the Haskell language, see [28].

[5] For a reference to the ML language, see [44].

[6] For a reference to the Prolog language, see [50].

relation with the quality of the executable code that results from the compilation (this is discussed in Section 6.1).

## 1.4 Plan

In Chapter 2, we explain in detail the problem that we attack. We precisely describe the optimisation for which the type analysis shall gather information. We introduce a mini-language similar to a kind of Scheme that is simplified almost down to a $\lambda$-calculus. We present its syntax and semantics. We bring justification for the selection of our goal.

Chapter 3 presents the analysis framework. It first gives a description of the use of the framework. That is, the parameters (the abstract model) and the analysis results that it produces. It then gives a precise description of its implementation. Finally, many properties of the analysis framework are demonstrated. Namely: that any analysis it instantiates always terminates; that the analysis is conservative; that it is powerful, as any error-free terminating program can be analysed perfectly well using an appropriate model, i.e. all safety tests can be removed from the program; that, unfortunately, it is generally impossible to find such a model when it exists; that, in practice, it is very flexible since it can mimic many conventional static analyses.

In Chapter 4, we give a sketch of what a demand-driven analysis should be, but without giving a precise specification. We propose a cyclic approach where the program is first analysed, then the static analysis is improved, then the program is analysed again, etc. An imprecise definition of demands and processing rules is given. Some notions that help to create a reasonable demand-driven analysis are presented. Namely, the necessity and the sufficiency principles. An extensive example is used to better explain the principles behind the approach.

In Chapter 5, we propose a concrete implementation of a demand-driven analysis that is based on patterns. The chapter includes a complete description of pattern-based modelling, from the representation of abstract values to the elaboration of an abstract model to be fed to the analysis framework, of the syntax and meaning of the demands, of the demand processing rules, and of the main algorithm controlling the analysis. An example illustrates the working of the whole process. A brief history of the development of our current prototype implementing the demand-driven analysis is presented.

Chapter 6 evaluates our prototype through many experiments. A brief description of each of the benchmarks used in the experiments is given. The methodology used is presented and justified.

Chapter 7 summarises our contributions, makes a quick survey of the (not so) related work in demand-driven analysis, and, most importantly, presents some future work.

# Chapter 2

# Definition of the Problem

## 2.1 Objective

We intend to develop an adaptable type analysis for a dynamically-typed language. The language is presented below. Basically, it is a minimalist applicative functional language that includes three types: closures, pairs and the Boolean false (#f). To keep things simple, the programs should be closed. That is, they should have no free variables. Also, compilation is done on whole programs at once.

Some operations of the language require dynamic safety type tests. For example, before performing the extraction of the CAR-field of an object, a check must be made to ensure that it is truly a pair. At least, it is the case if safe execution of the program is desired. Indeed, we work under the context of safe execution. Under the context of non-safe execution, the problem of eliminating safety dynamic type tests would no longer exist. Additionally, if the optimiser were to trust annotations given by the programmer, the context would also be that of non-safe execution. We are interested in safe execution, so no external source of information is trusted.

A naïve compilation of the programs would require the inclusion of code to perform safety tests at run time everywhere a hazardous operations is made. However, optimising compilers try to generate more efficient code by performing a *static analysis* on the programs to discover evidence that some or all of the dynamic tests can be safely removed. Our analysis intends to achieve this task.

The following sections first present the functional language to analyse. A detailed presentation of both the syntax and the semantics of the language is given. Then there is a discussion about the generality of the quite specific analysis task that we have chosen.

## 2.2 Language

Figure 2.1 presents the syntax of our small applicative functional language. It does not have a name but we will often refer to it as the mini-language. Expressions in the mini-language are labelled. The labels are used to give a unique "name" to the expressions. For example, it allows us to refer to a particular expression as $e_{12}$ instead of having to write it verbatim everywhere. We use numerical labels throughout this text.

The mini-language provides functions, pairs, and the Boolean '#f'. As in Scheme, anything except '#f' is considered to be a true Boolean value when the '`if`' expression tests its first sub-expression. The '`pair`?' expression provides a way to distinguish between pairs and the other objects. Depending on whether its argument is a pair or not, it returns either the pair itself or '#f', respectively. Finally, evaluation of sub-expressions generally proceeds from left to right. This particularity could make a difference if one of the sub-expressions loops and the other leads to an error, but it cannot when the program eventually terminates. The rest of the semantics of the language is fairly standard: the '`if`' expression first evaluates the test and then only one of its two branches; the body of the $\lambda$-expression is evaluated only when the function is eventually called; the other expressions evaluate all of their sub-expressions.

Only three of the nine kinds of expressions require a dynamic safety test. We do not include `pair`?-expressions in these three as their purpose is not safety and there is no reason to expect their result to always be true (or false). Expressions accessing pairs, namely '`car`' and '`cdr`', must ensure that the objects that they are about to access are truly pairs. Calls must ensure that the objects returned by the evaluation of the first sub-expression are truly functions. The task of our type analysis is to give the optimiser the opportunity to remove as many safety checks as possible among those introduced by these expressions.

The detailed semantics of the language are presented in Figure 2.2.[1] Semantic domain

[1] The $\dot{\cup}$ operator is the *disjoint union*. Its results is the union of its two argument sets but it is defined

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

Figure 2.1: Mini-language syntax

$$
\begin{array}{lll}
\mathrm{Val}^{\uparrow} & := \mathrm{Err} \mathbin{\dot\cup} \mathrm{Val} & \\
\mathrm{Err} & := \textit{Errors} & \\
\mathrm{Val} & := \mathrm{ValB} \mathbin{\dot\cup} \mathrm{ValC} \mathbin{\dot\cup} \mathrm{ValP} & \\
\mathrm{ValB} & := \{\#\mathrm{f}\} & \textit{Booleans} \\
\mathrm{ValC} & := \{\mathrm{clos}((\lambda_l \mathrm{x}.\ e_1),\ \rho) \mid (\lambda_l \mathrm{x}.\ e_1) \in \mathrm{Exp},\ \rho \in \mathrm{Env}\} & \textit{Closures} \\
\mathrm{ValP} & := \{\mathrm{pair}(v_1,\ v_2) \mid v_1, v_2 \in \mathrm{Val}\} & \textit{Pairs} \\
\mathrm{Env} & := \mathrm{Var} \to \mathrm{Val} &
\end{array}
$$

$$
\begin{array}{lcll}
\multicolumn{3}{l}{\mathsf{E} : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Val}^{\uparrow}} & \textit{Evaluation function} \\
\mathsf{E}\ \llbracket \#\mathtt{f}_l \rrbracket\ \rho & = & \#\mathrm{f} & \\
\mathsf{E}\ \llbracket \mathrm{x}_l \rrbracket\ \rho & = & \rho\ \mathrm{x} & \\
\mathsf{E}\ \llbracket ({}_l e_1\ e_2) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ \llbracket e_2 \rrbracket\ \rho)\ (\mathsf{A}\ v_1)) & \\
\mathsf{E}\ \llbracket (\lambda_l \mathrm{x}.\ e_1) \rrbracket\ \rho & = & \mathrm{clos}((\lambda_l \mathrm{x}.\ e_1),\ \rho) & \\
\mathsf{E}\ \llbracket (\mathtt{if}_l\ e_1\ e_2\ e_3) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v.\ v \neq \#\mathrm{f}\ ?\ \mathsf{E}\ \llbracket e_2 \rrbracket\ \rho : \mathsf{E}\ \llbracket e_3 \rrbracket\ \rho) & \\
\mathsf{E}\ \llbracket (\mathtt{cons}_l\ e_1\ e_2) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ \llbracket e_2 \rrbracket\ \rho)\ (\lambda v_2.\ \mathrm{pair}(v_1,\ v_2))) & \\
\mathsf{E}\ \llbracket (\mathtt{car}_l\ e_1) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v.\ v = \mathrm{pair}(v_1,\ v_2)\ ?\ v_1 : \textsc{error}) & \\
\mathsf{E}\ \llbracket (\mathtt{cdr}_l\ e_1) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v.\ v = \mathrm{pair}(v_1,\ v_2)\ ?\ v_2 : \textsc{error}) & \\
\mathsf{E}\ \llbracket (\mathtt{pair?}_l\ e_1) \rrbracket\ \rho & = & \mathsf{C}\ (\mathsf{E}\ \llbracket e_1 \rrbracket\ \rho)\ (\lambda v.\ v \in \mathrm{ValP}\ ?\ v : \#\mathrm{f}) & \\
\multicolumn{3}{l}{\mathsf{A} : \mathrm{Val} \to \mathrm{Val} \to \mathrm{Val}^{\uparrow}} & \textit{Apply function} \\
\mathsf{A}\ f\ v & = & f = \mathrm{clos}((\lambda_l \mathrm{x}.\ e_1),\ \rho)\ ?\ \mathsf{E}\ \llbracket e_1 \rrbracket\ \rho[\mathrm{x} \mapsto v] : \textsc{error} & \\
\multicolumn{3}{l}{\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 2.2: Mini-language semantics[1]

$\mathrm{Val}^{\uparrow}$ contain evaluation results, which are either normal values or error values. We do not explicitly define the error values. Normal values (or simply, values) are the Boolean, from ValB, closures, from ValC, or pairs, from ValP. A closure is a constructor containing a $\lambda$-expression and the definition lexical environment. Note that pairs and environments can only contain values, not error values.

The evaluation function computes the value of an expression in a certain lexical environment. It makes extensive use of the check function $\mathsf{C}$ to verify whether the values obtained during the evaluation of sub-expressions are normal. $\mathsf{C}$ takes an evaluation result and a continuation. It immediately returns the evaluation result if it is an error, otherwise it passes it to the continuation, which does the rest of the computation. The apply function $\mathsf{A}$ takes care of the details of the invocation of a closure on an argument. The specification of the evaluation function $\mathsf{E}$ itself is quite straightforward.

Note the situations in which an error can occur: in the access to the CAR- or CDR-field and in a call. Evaluation of the other expressions is always safe, barring the occurrence of an error in the evaluation of a sub-expression.

## 2.3 Generality of the Objective

Despite the fact that the objective of our research is done on type analysis, namely the removal of dynamic safety type tests, we expect the research to have a much broader impact. We present a few reasons to support our belief.

The mini-language is applicative; that is, the argument expression is completely evaluated before the closure is invoked with the result. However, that does not mean that the scope of our research is limited to applicative languages. We could aim at the same objective while using a lazy language. The task of type analysis would be similar in such a language.

The choice of a *type* analysis is a reasonable one, too, as performing a good type analysis in a dynamically-typed language is not less difficult than performing some other analysis. Instances of analyses include escape analyses [53], reference counting analyses [35], numerical range analyses [26, 27, 41, 48], and representation analyses [54, 32, 33]. In all cases, relatively simple analysis methods can lead to relatively good analysis results. However, doing an

only it the two sets are disjoint.

optimal job, that is, obtaining results that allow the optimiser to do the best job possible, is uncomputable as all the desired properties depend on the actual computations done be the program.

Note that our real goal is not necessarily to obtain the best possible method to remove dynamic type tests in the code generated by compilers. We also want to study the efficiency of a demand-driven approach as a mean to drive an adaptive analysis intelligently. Non-adaptive methods clearly have intrinsic limitations that are more or less easily encountered. On the contrary, adaptive methods can push these limitations much farther. However, there has to be some mechanism to guide the adaptations. As will be presented in the following chapters, type analysis of the programs is performed using an adaptable analysis framework and a demand-driven approach provides the means to translate the needs of the optimiser (the task of removing safety tests) into precise directives on how to adapt the analysis of the program to obtain analysis results that are more useful to the optimiser. Although the demand-driven approach that we develop in this research is quite specific and the idea of being demand-driven is quite general, success in our particular project would bring evidence that the general idea can be useful.

The restriction to whole program compilation is not a mandatory one. In a concrete implementation, our type analysis could be adapted to support separate compilation while guaranteeing complete safety. However, a certain cooperation from the programmer would be required. First, the program would have to be separated in module. This way, no mutation of a variable could be done from another module (if the language includes side-effects). Second, the programmer would have to give type annotations for all variables that are exported out of a module. The importation of a module into a module under compilation would make these annotations available to the compiler. The more precise these annotations, the higher the quality of the analysis results for the module, and the higher the quality of the executable code. In order to ensure safety of the evaluation of the program, the compilation of each module would include a verification that the module conforms to the given annotations and, at run time, before the start of the normal evaluation of the program, the executable would perform a verification to ensure that each importing module has seen the same annotations than those truly declared in each imported module.

The restriction to a language without input/output is not mandatory either. We chose not to consider I/O because it does not add any interesting problem from the point of view of the type analysis. It is clear that the ability to write data does not change what the programs compute and it would not interfere with the type analysis. So output is not interesting. It is less clear that the ability to read data is also uninteresting. Indeed, the data that are read have an impact on the computations that programs perform. They introduce an uncertainty factor in the computations. However, this uncertainty is quite easy to manage: a (`read`) expression may return *any value* that the language's specification allows as a valid input value. For example, the specification could say that (`read`) returns a value made of pairs and Booleans every time it is evaluated. Consequently, any attempt by the type analysis to obtain precise type information about the possible value of (`read`) plainly fails.

Clearly, a type analysis is useful in the compilation of dynamically-typed languages. But it may seem useless for statically-typed languages such as ML or Haskell. However, it is not the case. The main reason is that these languages both provide algebraic types. An algebraic type may include many constructors. For example, in Haskell, list types are algebraic types including two constructors: '`[]`' of arity 0 for the empty list and '`:`' of arity 2 for the pairs. The programmer can define a function taking lists as an argument and use pattern-matching with a pattern for only one of the two constructors. If the function is passed a list built using the other constructor, an error occurs. For example, an error actually occurs if the head or the tail is extracted from an empty list. The inspection of the argument is a kind of safety dynamic test as the typing of the program cannot guarantee that only the *expected* constructor(s) will be passed. A type analysis such as ours would be required in order to remove as many of those tests as possible. If we reverse the point of view, programs in our mini-language can be considered to be statically typable using a unique type that includes three constructors. The uniqueness of this hypothetical type makes the static typing trivial and leaves all verifications relative to the *constructors* to the run time.

Object-oriented languages could also benefit from an adaptation of our type analysis. The exact instantiation class of an object can be seen as a constructor. The class of a declared variable can be seen as an algebraic type including all the constructors corresponding to its sub-classes. Moreover, the case where a variable does not reference any object, that is, when its value is NULL, can be seen as corresponding to an additional '`null`' constructor.

Despite the fact that our type analysis could be applied to a variety of languages, we decided to use this particular applicative dynamically-typed functional mini-language because it is the kind of language that needs and stresses type analysis the most. First, programs written in dynamically-typed languages typically need more safety type tests than those in statically-typed languages. Second, functional programs have a tendency to have a more complex control-flow because of the use of higher-order functions. So our mini-language (which is similar to Scheme) is particularly challenging for a type analyser.

Finally, demand-driven analysis could be useful in the field of dynamic compilation, or just-in-time compilation. Of course, it would have to operate within relatively limited resources, especially in time. But the advantage is that analysis would operate while the program runs and profiling statistics about the real execution would be available.

# Chapter 3

# Analysis Framework

This chapter presents the analysis framework and numerous properties related to it. The analysis framework, by itself, is not a complete static analysis for programs drawn from the syntactic domain Exp. An *abstract model* has to be provided to the framework in order to create an instance of analysis. Recall that the abstract model specifies what the phony values and phony evaluation contexts are when a phony execution of the program is performed. From now on, we designate phony values as *abstract values* and phony contexts as *contours*. The abstract model takes the form of a few *framework parameters*. This parameterisation of the analysis framework brings the mutability of the analysis that we need. Indeed, the framework has a great flexibility as will be made apparent by results in this chapter.

We start the presentation of the analysis framework by describing its external behaviour, that is, the description of its parameters and that of the results of an analysis instance. Next, we present the functioning of the framework. The rest presents different properties of the framework. The first one is the fact that any analysis instantiated from the framework always terminates. Next, a collecting machine is introduced. The machine computes the same result as the standard semantics for the mini-language but it also produces a *cache* containing the details of the computation. With the help of the collecting machine, we demonstrate that the analysis instances are *conservative*, that is, the results they produce represent *at least* all the concrete computations made during the concrete evaluation. Next, we show that for any program that terminates without error, there exists an abstract model showing that all dynamic type tests can safely be removed. We also show that, unfortunately, it is undecidable to determine if such a model actually exists for an arbitrary program. We end the chapter by illustrating the flexibility of the framework by giving abstract models with which it is possible to imitate many known analyses.

A kind of analysis framework was previously presented by Ashley and Dybvig in [11]. It is parameterised by two modelling functions: one that controls the accuracy of the analysis by splitting abstract evaluation contexts and one that controls the speed of the analysis by performing *widening* on stores. In simple words, widening is some sort of "exaggeration" of the abstract values to help the analysis results to reach a stable state faster. Their analysis framework does not offer the subtlety that ours does. Both parameters have a global effect on the analysis. We consider them to be too coarse for our application. Also their framework handles mutable variables and data structures. This adds unnecessary complexity since our language is purely functional.

## 3.1 Instantiation of an Analysis

Before we present the process of instantiating an analysis for a program, we need to mention the existence of a few restrictions imposed on the program itself. Let $e_{l_0} \in \mathrm{Exp}$ be the program to analyse. First, the framework requires the program to be *$\alpha$-converted*. That is, each variable in the program must have a distinct name. This restriction poses no big problem since, for a program having variables with the same name, a simple renaming remedy to the situation. Second, the program must include *proper labelling*, that is, all labels have to be distinct. It is vital to uniquely identify each expression in the program in order to analyse it properly. Once again, there is no problem there since labels are an artificial creation, anyway. They are introduced for analysis purpose only. Third, the program has to be *closed*, that is, it must not have free variables. This restriction is closely related to our choice not to provide input/output operations in the mini-language (see Section 2.3).

Now, if we suppose we have an appropriate program $e_{l_0}$, the analysis of $e_{l_0}$ using an abstract model $\mathcal{M}$ is denoted by

$$\mathcal{R} = \mathrm{FW}(e_{l_0}, \mathcal{M})$$

where FW is the analysis framework receiving a program and a model, and returning analysis results $\mathcal{R}$. We first describe the abstract model. Then the analysis results are presented.

$$\mathcal{M} = (\mathcal{V}al\mathcal{B},\, \mathcal{V}al\mathcal{C},\, \mathcal{V}al\mathcal{P},\, \mathcal{C}ont,\, \hat{k}_0,\, \mathsf{cc},\, \mathsf{pc},\, \mathsf{call})$$

| | |
|---|---|
| $\mathcal{V}al\mathcal{B} \neq \emptyset$ | 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 |
| $\hat{k}_0 \in \mathcal{C}ont$ | Main contour |
| $\mathsf{cc} : \mathrm{Lab} \times \mathcal{C}ont \to \mathcal{V}al\mathcal{C}$ | Abstract closure creation |
| $\mathsf{pc} : \mathrm{Lab} \times \mathcal{V}al \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{V}al\mathcal{P}$ | Abstract pair creation |
| $\mathsf{call} : \mathrm{Lab} \times \mathcal{V}al\mathcal{C} \times \mathcal{V}al \times \mathcal{C}ont \to \mathcal{C}ont$ | Contour selection |

$$\begin{array}{l} \text{where } \mathcal{V}al := \mathcal{V}al\mathcal{B} \mathbin{\dot\cup} \mathcal{V}al\mathcal{C} \mathbin{\dot\cup} \mathcal{V}al\mathcal{P} \\ \text{subject to } |\mathcal{V}al| + |\mathcal{C}ont| < \infty \end{array}$$

Figure 3.1: Instantiation parameters of the analysis framework

### 3.1.1 Framework Parameters

The abstract model, formed by framework parameters, is presented in Figure 3.1.[1] The model includes abstract values, abstract contours, and abstract evaluation functions.

The abstract values include Booleans ($\mathcal{V}al\mathcal{B}$), closures ($\mathcal{V}al\mathcal{C}$), and pairs ($\mathcal{V}al\mathcal{P}$). $\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, and $\mathcal{V}al\mathcal{P}$ are finite, non-empty sets. That is, these abstract domains must be finite in order to guarantee that the abstract evaluation of the program always uses a finite amount of resources. And they must be non-empty in order to have at least one abstract representative for the concrete values of each type. The three sets must be mutually disjoint, as it is expressed by the use of the disjoint union operator ($\dot\cup$). The set of abstract values $\mathcal{V}al$ is the union of the three sets. As soon as three sets conform to the mentioned constraints, they can be considered as legal abstract value domains. Nothing special is required of the abstract values themselves. Their type comes from the fact that they belong to one (and only one) of the three sets.

The abstract contours are given by the set $\mathcal{C}ont$. It must be a finite, non-empty set. No other restriction applies to the abstract contours. Contours are abstract representatives for concrete evaluation contexts. A concrete evaluation context describes the circumstances in which an expression gets evaluated. It includes the current lexical environment that is visible by the expression. It also includes the identity of the caller to the closure which led

[1] In this chapter, we put a hat (ˆ) on the abstract entities to distinguish them from the concrete ones.

to the current evaluation, the caller of the caller, etc. The context usually has an impact on the value of an expression. For instance, an expression may produce different values when evaluated in different lexical environments during concrete interpretation. Similarly, this expression may produce different abstract values when evaluated in different contours during abstract interpretation.

Each abstract contour represents a certain fraction of all possible evaluation contexts. The abstract evaluation of an expression $e_l$ in a contour $\hat{k}$ must summarise everything that could happen during the concrete evaluation of $e_l$ in any evaluation context that is represented by $\hat{k}$. For example, if $e_l$ evaluates to a pair in a certain evaluation context and to a closure in another context, and that both evaluation contexts are abstracted by $\hat{k}$, then, during abstract evaluation in contour $\hat{k}$, $e_l$ will evaluate to at least an abstract pair and an abstract closure, the last two being abstract counterparts of the concrete values returned by $e_l$.

Parameter $\hat{k}_0$ is the contour in which the program (the top-level expression $e_{l_0}$) is to be abstractly evaluated. Except for that special use, $\hat{k}_0$ is an ordinary contour.

When a $\lambda$-expression is abstractly evaluated, an abstract closure must be produced. Similarly for a cons-expression. However, the analysis framework does not decide by itself which closure or which pair should be returned. This is where the closure creation function (cc) and the pair creation function (pc) come into play. Function cc chooses the abstract closure from $\mathcal{V}al\mathcal{C}$ that should be returned based on the $\lambda$-expression and the current contour. Function pc does the same but has also the possibility to base its decision on the two values that *go into* the abstract pair. We explain in the next sections what it means to produce a value that *contains* other values. pc may choose the abstract pair in function of the label of the cons-expression, or in function of the contour, or in function of the type of the value that goes in the CDR-field of the pair, or, in general, according to a combination of strategies. As long as cc returns an element of $\mathcal{V}al\mathcal{C}$ and pc returns an element of $\mathcal{V}al\mathcal{P}$, everything works.

The possibility of specifying $\mathcal{V}al\mathcal{C}$ and $\mathcal{V}al\mathcal{P}$ contributes to the flexibility of the framework but it is especially because of the existence of the cc and pc functions that the framework is very flexible. It is also because of the call function that we describe below.

One might worry about the fact that there is no bc function (no Boolean creation function). Indeed, Booleans are produced by the evaluation of the false constant and sometimes by `pair?`-expressions. There could have been a bc function. However, we do not see the utility of such a function as there is just one concrete Boolean. What would be the benefit of choosing one abstract Boolean over another one since they all represent the same concrete Boolean? We believe there is none. But why do we allow $\mathcal{V}al\mathcal{B}$ to have more than one element in the first place? In fact, there is no advantage, but there is no problem in doing so, either. The decision of having no bc function could be changed in the future if something indicates that it would be beneficial. The current treatment of Boolean creation by the framework is that each time an abstract Boolean is to be produced, the whole $\mathcal{V}al\mathcal{B}$ set is returned.

The last framework parameter is the call function. This function selects contours in which expressions are evaluated. It is not used before the evaluation of each individual expression but only before the whole body of a closure. A (possibly) new contour is selected each time a closure is called. Indeed, when an abstract closure $\hat{c}$ is invoked on argument $\hat{v}$ in call expression $({}_{l}e_{l_1}\ e_{l_2})$ and in contour $\hat{k}$, the body of $\hat{c}$ gets evaluated in contour $\mathsf{call}(l, \hat{c}, \hat{v}, \hat{k})$. Hence, the call function contributes greatly to the flexibility of the analysis framework as different contours can be selected, depending, of course, on the invoked closure but also on the argument, on the label of the call expression where the invocation occurs, and on the contour in which this invocation occurs. The resulting flexibility allows our framework to have contours that may be call-chains or that may be abstract representatives of the lexical environment, etc. Examples of various uses of the call function can be found in Section 3.7.

In order to be a legal model for the analysis of a program $e_{l_0}$, $\mathcal{M}$ has to obey to a last constraint. The three creation (or selection) functions have to be defined on the part of their from-set that covers at least every possible argument passed by the analysis framework. That is, their domain must cover at least every possible argument. The functions are not required to be defined on their whole from-set as the label argument poses a problem. Presumably, Lab is an infinite set and the rest of the specification of models manipulates only finite sets. So now we present the part of the from-set that must be covered by each function. Let us denote by $\triangle(e_{l_0})$ the set of labels in program $e_{l_0}$.[2] Closure creation function cc has to be defined at least on $\triangle(e_{l_0}) \times \mathcal{C}ont$. Pair creation function pc has to be defined at least on

[2] The $\triangle$ function is formally defined in Section 3.4.1.

$$\mathcal{R} = (\alpha,\, \beta,\, \gamma,\, \delta,\, \chi,\, \pi,\, \kappa)$$

| | | | |
|---|---|---|---|
| Value of $e_l$ in $k$: | $\alpha_{l,\hat{k}} \subseteq \mathcal{V}al$ | $l \in \text{Lab}$, | $\hat{k} \in \mathcal{C}ont$ |
| Contents of x when bound in $\hat{k}$: | $\beta_{\text{x},\hat{k}} \subseteq \mathcal{V}al$ | $\text{x} \in \text{Var}$, | $\hat{k} \in \mathcal{C}ont$ |
| Return value of $\hat{c}$ with body in $\hat{k}$: | $\gamma_{\hat{c},\hat{k}} \subseteq \mathcal{V}al$ | $\hat{c} \in \mathcal{V}al\mathcal{C}$, | $\hat{k} \in \mathcal{C}ont$ |
| Flag indicating evaluation of $e_l$ in $\hat{k}$: | $\delta_{l,\hat{k}} \subseteq \mathcal{V}al$ | $l \in \text{Lab}$, | $\hat{k} \in \mathcal{C}ont$ |
| Creation circumstances of $\hat{c}$: | $\chi_{\hat{c}} \subseteq \text{Lab} \times \mathcal{C}ont$ | | $\hat{c} \in \mathcal{V}al\mathcal{C}$ |
| Creation circumstances of $\hat{p}$: | $\pi_{\hat{p}} \subseteq \text{Lab} \times \mathcal{V}al \times \mathcal{V}al \times \mathcal{C}ont$ | | $\hat{p} \in \mathcal{V}al\mathcal{P}$ |
| Selection circumstances of $\hat{k}$: | $\kappa_{\hat{k}} \subseteq \text{Lab} \times \mathcal{V}al\mathcal{C} \times \mathcal{V}al \times \mathcal{C}ont$ | | $\hat{k} \in \mathcal{C}ont$ |

Figure 3.2: Analysis results of the framework

$\triangle(e_0) \times \mathcal{V}al \times \mathcal{V}al \times \mathcal{C}ont$. And contour selection function call has to be defined at least on $\triangle(e_0) \times \mathcal{V}al\mathcal{C} \times \mathcal{V}al \times \mathcal{C}ont$.

We could relax this last constraint on the domain of the abstract creation functions a little more. For instance, the label passed to cc can only be that of a $\lambda$-expression. For pc and call, the label can only be that of a cons-expression and a call expression, respectively. However, specifying the minimal domains that way would be unnecessarily heavy. Anyway, the given specification does not pose a real problem as, for example, cc may return any element of $\mathcal{V}al\mathcal{C}$ it wishes if the argument label is not one of a $\lambda$-expression; it does not matter.

### 3.1.2 Analysis Results

The analysis results $\mathcal{R}$ of the analysis of program $e_{l_0}$ using model $\mathcal{M}$ are described in Figure 3.2. $\mathcal{R}$ takes the form of seven matrices of abstract variables. Each matrix contains a certain kind of information. In fact, it is directly with these matrices that the framework does the analysis of programs.

We describe the contents of each matrix. Essentially, the first four matrices are the analysis results that are normally considered as the most interesting, especially the first. The last three are rather intended for internal purpose.

The $\alpha$ matrix indicates the set of values to which each expression evaluates to in each contour. Typically, there are many entries that remain empty after the analysis, because, for example, there is some dead code in the program or, by the way the model is built, some expressions simply do not get evaluated in certain contours.

The $\beta$ matrix indicates the values that each variable of $e_{l_0}$, in each contour, may contain. Note how the entries in this matrix require $e_{l_0}$ to be $\alpha$-converted. Identical names for different variables would produce pollution in the results as the values of all variables sharing a certain name would also share their contents. The meaning of an abstract variable like $\beta_{\mathrm{x},\hat{k}}$ is quite subtle. It is not necessarily equivalent to the result of a reference to x in contour $\hat{k}$. This would be ill-defined as there is no direct relation between the contour that prevails when x is (abstractly) bound to a value and the contour that prevails when x is referenced. The reference may occur inside of the body of a closure originating from a $\lambda$-expression that is in the scope of x. Remember that the contour possibly changes during each invocation. The abstract variable $\beta_{\mathrm{x},\hat{k}}$ represents the value of variable x if x is the parameter of some closure $\hat{c}$ and if, for every invocation where $\hat{c}$ gets called on a certain value, contour $\hat{k}$ is the one that is prescribed by call for the given situation. For example, consider the following program excerpt:

$$\begin{array}{l} \ldots \\ ({}_1e_2\ e_3) \\ \ldots \\ (\lambda_4\mathrm{x}.\ (\lambda_5\mathrm{y}.\ \mathrm{x}_6)) \\ \ldots \end{array}$$

Suppose that during evaluation of call $e_1$ in contour $\hat{k}$, a closure $\hat{c}$, coming from $\lambda$-expression $e_4$, gets called on some value $\hat{v}$, and that $\mathsf{call}(1, \hat{c}, \hat{v}, \hat{k}) = \hat{k}'$. Then, it follows that $\hat{v} \in \beta_{\mathrm{x},\hat{k}'}$. Now, suppose that a closure originating from $\lambda$-expression $e_5$ gets called and that its body is evaluated in contour $\hat{k}''$. Then, the reference to x in $e_6$ in contour $\hat{k}''$ will include the contents of $\beta_{\mathrm{x},\hat{k}'}$ (and not of $\beta_{\mathrm{x},\hat{k}''}$) because $\hat{k}'$ is the contour in which x was bound.

The $\gamma$ matrix indicates the values returned by the closures. Abstract variable $\gamma_{\hat{c},\hat{k}}$ contains the values returned by closure $\hat{c}$ when its body has been evaluated in contour $\hat{k}$.

The $\delta$ matrix indicates in which contours each expression gets evaluated. Each entry of the matrix acts as a flag. If $\delta_{l,\hat{k}}$ is non-empty, then expression $e_l$ gets evaluated in contour $\hat{k}$, otherwise, it is not. The actual contents of these abstract variables are not important. The role of the $\delta$ matrix is to help the framework to generate analyses that are not *too* conservative. Analyses should always be conservative, but it should avoid unnecessary pollution of the results as much as possible. This is particularly true in the case of our framework. Arbitrary contour definition through parameters typically causes the instantiation of analyses that include very discriminating contours. Discriminating contours can mimic concrete evaluation contexts with high fidelity and most expressions may get evaluated in only a small fraction of the contours. So it is important to avoid propagation of values from the expressions that are not supposed to be evaluated.

The $\chi$, $\pi$, and $\kappa$ matrices are *logs* of the creation and selection of abstract values and contours by the $\mathsf{cc}$, $\mathsf{pc}$, and $\mathsf{call}$ functions. They record the circumstances under which values and contours are created and selected. Let us illustrate their usage with the case of the $\pi$ matrix. For each abstract pair, the $\pi$ matrix logs which quadruples were effectively used in the creation of the pair. Note that a pair $\hat{p}$ could be created when any quadruple from $\mathsf{pc}^{-1}(\hat{p})$ is passed to $\mathsf{pc}$. But that does not mean that, during the analysis, pair $\hat{p}$ really got created under all the circumstances present in $\mathsf{pc}^{-1}(\hat{p})$. The exact set of circumstances that were prevailing when $\hat{p}$ was created during the analysis are logged in $\pi_{\hat{p}}$. The three log matrices are very helpful in helping to reduce the propagation of superfluous values throughout the analysis results.

- Abstract variable $\chi_{\hat{c}}$ contains all couples that lead to the creation of $\hat{c}$, each being formed by a label and a contour.
- Abstract variable $\pi_{\hat{p}}$ contains all quadruples that lead to the creation of $\hat{p}$, each begin formed by the label of the $\mathsf{cons}$-expression, the two values to *cons* together, and the contour that was prevailing during that creation.
- Abstract variable $\kappa_{\hat{k}}$ contains all quadruples that lead to the selection of $\hat{k}$ as a context for the evaluation of the body of a closure, each being formed by the label of the call expression, the closure that was invoked, the value that was passed, and the contour that was prevailing during the call.

Note that, from the point of view of the framework, the fact that $\hat{p}$ has some contents comes from the fact that $\pi_{\hat{p}}$ contains some quadruples, and not from the fact that $\hat{p}$ is actually denoted in $\mathcal{V}al\mathcal{P}$ by $P$, pair$(v_1, v_2)$, or even $\clubsuit$. The presentation of the internal functioning of the framework in the next section show the intensive use of the log variables.

### 3.1.3 An Example of Use of the Analysis Framework

To illustrate the use of the analysis framework, we present the analysis of a little program using a simple model. Here is the program:

$$e_0 = (\texttt{car}_0\ (\texttt{cdr}_1\ (\texttt{cons}_2\ \#\texttt{f}_3\ (\texttt{cons}_4\ \#\texttt{f}_5\ \#\texttt{f}_6))))$$

Note that we avoid calls in the example as the mechanics for analysing functions and calls is quite involving. We choose the simplest legal model for the analysis of $e_0$:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathcal{M} = (\mathcal{V}al\mathcal{B},\ \mathcal{V}al\mathcal{C},\ \mathcal{V}al\mathcal{P},\ \mathcal{C}ont,\ K,\ \mathsf{cc},\ \mathsf{pc},\ \mathsf{call})\ \text{where}} \\ \mathcal{V}al\mathcal{B} & = & \{\#\mathrm{f}\} \\ \mathcal{V}al\mathcal{C} & = & \{C\} \\ \mathcal{V}al\mathcal{P} & = & \{P\} \\ \mathcal{C}ont & = & \{K\} \\ \mathsf{cc}(l, \hat{k}) & = & C \\ \mathsf{pc}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & P \\ \mathsf{call}(l, \hat{f}, \hat{v}, \hat{k}) & = & K \end{array}$$

The model contains a single abstract value of each type and a single abstract contour. Naturally, there is no freedom left in the choice of the three creation functions. Here are the analysis results that we obtain from the analysis $\mathrm{FW}(e_0, \mathcal{M})$:

$$\mathcal{R} = (\alpha,\ \beta,\ \gamma,\ \delta,\ \chi,\ \pi,\ \kappa)\ \text{where}$$

| | | | |
|---|---|---|---|
| $\alpha_{0,K} = \{\#\mathrm{f}\}$ | $\alpha_{1,K} = \{\#\mathrm{f}, P\}$ | $\alpha_{2,K} = \{P\}$ | $\alpha_{3,K} = \{\#\mathrm{f}\}$ |
| $\alpha_{4,K} = \{P\}$ | $\alpha_{5,K} = \{\#\mathrm{f}\}$ | $\alpha_{6,K} = \{\#\mathrm{f}\}$ | |
| $\gamma_{C,K} = \emptyset$ | | | |
| $\delta_{0,K} = \{\#\mathrm{f}\}$ | $\delta_{1,K} = \{\#\mathrm{f}\}$ | $\delta_{2,K} = \{\#\mathrm{f}\}$ | $\delta_{3,K} = \{\#\mathrm{f}\}$ |
| $\delta_{4,K} = \{\#\mathrm{f}\}$ | $\delta_{5,K} = \{\#\mathrm{f}\}$ | $\delta_{6,K} = \{\#\mathrm{f}\}$ | |
| $\chi_C = \emptyset$ | | | |
| $\pi_P = \{(2,\ \#\mathrm{f},\ P,\ K), (4,\ \#\mathrm{f},\ \#\mathrm{f},\ K)\}$ | | | |
| $\kappa_K = \emptyset$ | | | |

Here is the signification of the results. We keep the description of the $\alpha$ matrix for the end. Note that the $\beta$ matrix is degenerated as there is no variable in $e_0$. The $\gamma$ matrix has only one entry. It says that closure $C$ returns nothing when its body is evaluated in contour $K$. It is because $C$ never gets created in the first place, as is expressed by the $\chi$ matrix. The $\delta$ matrix contains an entry per expression and it shows that all the expressions are evaluated in contour $K$. The fact that their content is $\{\#\mathrm{f}\}$ is not important, only that it is not empty. The $\kappa$ matrix indicates that $K$ gets selected in no circumstance. There are no calls, of course. The (necessary) use of $K$ as the main contour is not considered by the framework to be a contour selection.

Now, we come to the interesting part of the results. Let us first comment the contents of the $\pi$ matrix. It indicates that pair $P$ got created under two circumstances. One by $e_4$ in contour $K$ and using two Booleans. The other one by $e_2$ in contour $K$ and using a Boolean and $P$ itself. Intuitively, this is credible. But, as will be made clear in the next section, these circumstances originate from the interaction between entries of the $\pi$ and $\alpha$ matrices.

We complete the example by describing the contents of the $\alpha$-matrix. Entries for $e_3$, $e_5$, and $e_6$ only contain #f. This is the only possible result for the evaluation of the constant false. Also, entries for $e_2$ and $e_4$ contain pair $P$. A pair is the only result a `cons`-expression could provide and there is only one abstract pair. Abstract variable $\alpha_{1,\hat{k}}$ contains *two* values: a Boolean and a pair. Note that, during a concrete evaluation, only a pair could be the result for the evaluation of $e_1$. This is an example where analysis results contains superfluous values that do not correspond to anything in the concrete evaluation. This is caused by the conservativeness of the analysis. The values in $\alpha_{1,\hat{k}}$ are the result of the extraction of the CDR-field of $P$. The extraction proceeds by taking the third field of all quadruples in $\pi_P$. This explains the presence of the two values in $\alpha_{1,\hat{k}}$. The value in $\alpha_{0,\hat{k}}$ is the result of the extraction of the CAR-field and also by the filtering of non-pairs among the values returned by the sub-expression $e_1$. The framework does not try to perform some kind of CAR-field extraction on #f, but only on $P$.

Note, however, that the presence of a non-pair in $\alpha_{1,\hat{k}}$ would force a compiler to include a dynamic type test in the generated code for $e_0$ in order to keep the operations safe. That has to be so, unless it did another analysis with a more precise model and managed to show that only pairs can result from the evaluation of $e_1$.

Note also how the log variable $\pi$ helped us in obtaining more precise results. If $\mathsf{pc}^{-1}(P)$ were to be used instead of $\pi_P$, the values of expressions $e_0$ and $e_1$ would include $\mathcal{V}al$ entirely.

## 3.2 Internal Functioning of the Framework

Essentially, the analysis framework works by performing an abstract interpretation of the program. The analysis is done in two steps. First, a set of constraints is generated. These constraints involve the abstract variables mentioned above ($\alpha_{l,\hat{k}}$, etc.). The constraints that are generated in order to perform the analysis are the *evaluation constraints.* Their name comes from the fact that their goal is to simulate the evaluation of the program. The second step consists in solving these constraints. Contrarily to what is done in [29], no transformation or creation is performed on the constraints themselves but, instead, abstract values are propagated in the abstract variables until all the constraints are satisfied.

In the rest of the section, we first present the generation of the evaluation constraints. We do not present an algorithm for solving the constraints as it is a simple, mechanical process. As is common with the resolution of systems of constraints between sets, there are typically many solutions to the system. The one that is interesting is the *least solution* since the analysis ought to avoid the propagation of superfluous values as much as possible.

Then we present the generation of *safety constraints.* These constraints are not a part of the analysis. However, their purpose is to provide a systematic way to verify which optimisation's are enabled by the analysis results. That is, if all safety constraints are satisfied for a particular expression, then the code generated by the compiler for this expression need not include any dynamic safety type test.

### 3.2.1 Evaluation Constraints

The set of evaluation constraints that the analysis framework generates for a program $e_{l_0} \in$ Exp and abstract model $\mathcal{M}$, where

$$\mathcal{M} = (\mathcal{V}al\mathcal{B}, \mathcal{V}al\mathcal{C}, \mathcal{V}al\mathcal{P}, \mathcal{C}ont, k_0, \mathsf{cc}, \mathsf{pc}, \mathsf{call}),$$

is presented in Figure 3.3. Note that, exceptionally for this figure, we omit putting a hat (ˆ) on the abstract values and contours. The equations are already loaded enough without it. And no concrete value is manipulated by the framework, anyway. The set of constraints includes a special constraint '$\delta_{l_0,k_0} \supseteq \mathcal{V}al\mathcal{B}$', used to *start* the abstract interpretation, and, for each contour $k$ and each expression $e_l$ in the program, a set of constraints simulating the (eventual) evaluation of $e_l$ in $k$. This constraint generator may seem very complex *a priori*, so we explain the meaning of the constraints generated for each kind of expression. The complexity of the constraints generated for each kind of expression vary wildly and so we try to order the presentation from that of the simplest kind to that of the most difficult.

Let us start with the case of the constant false expression; i.e. let $e_l = \#\mathtt{f}_l$. Anytime $e_l$ is evaluated, its value is #f. The constraint that is generated expresses just that. If $\delta_{l,k} \neq \emptyset$, that is, if $e_l$ gets evaluated, then $\alpha_{l,k} \supseteq \mathcal{V}al\mathcal{B}$. During the description of the abstract model, we mentioned that we did not include a creation function for the Booleans. This is apparent here as the whole set of abstract Booleans is poured into the value of the expression, that is, in $\alpha_{l,k}$. Note that, for expression $\#\mathtt{f}_l$ and for all subsequent expressions, great care has been taken to ensure that they do not produce values if they do not get evaluated.

We continue with the `pair?`-expression; i.e. let $e_l = (\mathtt{pair?}_l\ e_{l_1})$. Here, $e_l$ has a sub-expression and some "pipes" have to be installed in order to coordinate the evaluation of $e_{l_1}$ with that of its parent. Let us sketch the *concrete* evaluation of $e_l$ step by step and compare it with the generated constraints. The first thing $e_l$ does is to trigger the evaluation of its sub-expression. This is expressed by the constraint $\delta_{l_1,k} \supseteq \delta_{l,k}$. When the evaluation of $e_{l_1}$ is completed, the type of the resulting value is checked. If the value is a pair, $e_l$ returns it directly. So the next constraint does the equivalent operation. The idea is that, if an abstract pair represents a concrete pair returned by $e_{l_1}$, then the same abstract pair also represents the concrete pair returned by $e_l$. If the value is not a pair, then #f is returned by $e_l$. This is expressed by the last constraint. So, during the abstract evaluation of $e_l$ in a particular contour $k$, both the pair and the non-pair cases can occur concurrently. This is typical in abstract interpretation.

We remain in pair-related cases and consider the `cons`-expression: i.e. let $e_l = (\mathtt{cons}_l\ e_{l_1}\ e_{l_2})$. The first constraints express the fact that both sub-expressions have to be evaluated when $e_l$ is. Instead, of creating one pair as during concrete interpretation, possibly many abstract pairs may have to be created as each sub-expression may produce more than one value. The last constraints create pairs for each combination of values. The pair is created with the help of the `pc` function. Moreover, the circumstances prevailing when each pair is created are logged in the appropriate $\pi$ matrix entry. The logging of these informations is required for the access to the fields of the pairs.

Let us consider the `car`-expression; i.e. let $e_l = (\mathtt{car}_l\ e_{l_1})$. Basically, the evaluation steps

Evaluation constraints for program $e_{l_0}$ are:

$$\bigcup_{k\in\mathcal{C}ont} \mathcal{E}\ [\![e_{l_0}]\!]\ k\ \cup\ \{\delta_{l_0,k_0} \supseteq \mathcal{V}al\mathcal{B}\},$$

where

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

$$\mathrm{ref}(\mathrm{x}, l, k) \quad = \quad \begin{cases} \mathrm{ref}(\mathrm{x}, l', k), & \text{if } e_{l'} \neq (\lambda_{l'}\mathrm{y}.\ e_l) \\ \beta_{\mathrm{x},k}, & \text{if } e_{l'} = (\lambda_{l'}\mathrm{x}.\ e_l) \\ \cup_{k'}\mathrm{ref}(\mathrm{x}, l', k'), & \text{otherwise } /*\ e_{l'} = (\lambda_{l'}\mathrm{y}.\ e_l), \text{ where } \mathrm{y} \neq \mathrm{x}\ */ \\ \quad \text{for } (l'', c, v, k'') \in \kappa_k, & \\ \qquad (l', k') \in \chi_c & \\ \text{where } l' = \mathrm{parent}(l) & \end{cases}$$

Figure 3.3: Evaluation constraints

that are simulated by the constraints are the triggering of the evaluation of $e_{l_1}$ and then, for each pair thus obtained, the extraction of the CAR-field. The contents of the CAR-field is computed by looking into the pair log ($\pi$) to recover the circumstances leading to the creation of the pairs. The second component of each quadruple contains the value intended for the CAR-field of a pair. Aside from the extraction issue, a point worth mentioning is the treatment of the non-pair results coming from the sub-evaluation. The constraints simply ignore the non-pair values. This may seem strange as, in the concrete interpretation, a non-pair value would cause an error. However, in the design of the framework, we have chosen to simulate only the non-erroneous computations by the evaluation constraints. But, as will be made clear below, there are safety constraints that precisely have the verification of the "pairness" of the results coming from $e_{l_1}$ as a task. Moreover, propagating abstract error values would be a waste of resources as there frequently are errors occurring somewhere during the abstract evaluation. An error value appearing as result from the evaluation of an expression would not be very meaningful, anyway: "An error possibly occurred during the evaluation of $e_l$."

The explanations for the `cdr`-expression are similar.

Now we turn to the conditional; i.e. let $e_l = (\texttt{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$. The interesting characteristic of the conditional is the fact that the last two sub-expressions get evaluated or not depends on the value of the test. The abstract interpretation of $e_l$ goes like this. The evaluation of $e_{l_1}$ is triggered. Then the evaluation of $e_{l_2}$ is triggered if some true value comes from $e_{l_1}$ and the evaluation of $e_{l_3}$ is triggered if some false value comes from $e_{l_1}$. The evaluation of both (or even none) may be triggered. Finally, the value of $e_l$ is the union of the values of $e_{l_2}$ and $e_{l_3}$. These constraints are an example that the value of a $\delta$ entry may depend on the value of an $\alpha$ entry.

The three kinds of expression that remain are all related to closures and invocations. Let us consider the $\lambda$-expression; i.e. let $e_l = (\lambda_l \mathrm{x}.\ e_{l_1})$. The constraints generated for $e_l$ are divided in two groups: the ones related to the evaluation of the $\lambda$-expression itself and the ones related to the invocation of closures originating from $e_l$. The constraints of the first group simply verify whether $e_l$ gets evaluated in contour $k$ and, if so, create a closure using `cc` and log its creation. Note that different $\lambda$-expressions could produce the same abstract closure. However, the $\chi$ matrix logs the origins of all closures. The constraints of the second group direct the evaluation of the body when a closure originating from $e_l$ is invoked. First, the evaluation of $e_{l_1}$ in $k$ is triggered if the parameter is bound to any value in contour $k$. After $e_{l_1}$ is evaluated, the values it produces are copied as the return value of the closures ($\gamma_{c,k}$) originating from $e_l$. These closures may have been created in any contour, but the thing that matters is that their body gets evaluated in $k$. Note that there is no logical connection between the contour in which $e_l$ is evaluated and the contour in which $e_{l_1}$ is evaluated. A contour selection using $\mathsf{call}$ occurs during each invocation. Nevertheless, both groups of constraints are generated together as the constraint generator produces the constraints for the evaluation in contour $k$ for the whole program at once.

We continue by describing the constraints generated for a call; i.e. let $e_l = ({}_l e_{l_1}\ e_{l_2})$. The triggering of the evaluation of the sub-expressions is routine, now. However, the invocation is more interesting. An invocation occurs for all combinations of a closure $c$ coming from $e_{l_1}$ and an argument $v$ coming from $e_{l_2}$. Note that non-closures coming from $e_{l_1}$ are ignored. The contour $k'$ in which the body ought to be evaluated is selected using $\mathsf{call}$. Then, the parameter that has to be bound to the argument is located by searching for the origins of $c$ in the closure log $\chi$. Note that there could be more than one parameter for abstract closure $c$ since different $\lambda$-expressions may produce $c$. The constraints then simulate the binding of the parameter to $v$ in contour $k'$, the contribution of the return value of $c$ to the value of $e_l$, and the logging in $\kappa$ of the circumstances in which $k'$ got selected.

The last kind of expression is the variable reference; i.e. let $e_l = \mathrm{x}_l$. It may seem surprising that we describe the constraints related to this innocent-looking expression at the end, but the reference is really not a trivial matter. Note that the framework does not maintain an explicit abstract representative for the lexical environment. Also, remember that the abstract variable $\beta_{\mathrm{x},k}$ *does not* represent the value of a reference to x in contour $k$. So the constraint generated for $e_l$ involves the use of the 'ref' function. This function does the necessary work to gather the values to which x could be bound to when the reference is made at label $l$ in contour $k$. Essentially, 'ref' searches for the binging site of x by climbing in the syntax tree of the program. This is why it computes the label $l'$ of the parent expression.[3] Most of the steps made during the climb are simple, except when it goes through a $\lambda$-expression. Remember that there is no simple connection between the contour in which a closure body executes and the contour in which its native $\lambda$-expression was evaluated. The value of $\mathrm{ref}(\mathrm{x}, l, k)$ depends on $e_{l'}$. There are three cases.

[3] The parent expression always exists because the program is closed. The main expression $e_{l_0}$ has no parent, but it is in the scope of no variable either, so a reference cannot occur there.

1. If $e_{l'}$ is not a $\lambda$-expression, then a reference to x from $e_l$ in contour $k$ has to give the same results as one from $e_{l'}$.

2. If $e_{l'}$ is a $\lambda$-expression and its parameter is x, then the climb has come to an end. The value of $\mathrm{ref}(\mathrm{x}, l, k)$ is exactly $\beta_{\mathrm{x},k}$.

3. Otherwise, $e_{l'}$ is a $\lambda$-expression and its parameter is not x. Let us suppose that the parameter is y. The value of $\mathrm{ref}(\mathrm{x}, l, k)$ is the value of a reference to x from $e_{l'}$ in the context in which it was evaluated. The contour in which $e_{l'}$ was evaluated is not necessarily $k$. More than that, it may not be unique. In fact, any contour $k'$ in which $e_{l'}$ has got evaluated, having resulted in a closure $c$, which has in turn been invoked in some circumstances, leading to the evaluation of $e_l$ in contour $k$, should be considered. This is exactly what is expressed in the third case of the definition of 'ref'. Closures involved in the selection of contour $k$ are first searched for in the $\kappa$ matrix. Only those originating from $e_{l'}$ are considered, since 'ref' performs a climb in the syntax tree. Each of those closures has been created in some contour $k'$, according to log $\chi$. So the reference to x continues at $e_{l'}$ in each such contour $k'$ and then the union of their result is taken.

Now that the constraints for each kind of expression have been described, there remains the '$\delta_{l_0,k_0} \supseteq \mathcal{V}al\mathcal{B}$' constraint. This constraint ensures that the abstract interpretation of $e_{l_0}$ effectively happens. Otherwise, the minimal solution to the evaluation constraints would consist in leaving all abstract variables empty.

### Example of Evaluation Constraints

We come back on the example of Section 3.1.3 and give the evaluation constraints for the same program

$$e_0 = (\texttt{car}_0\ (\texttt{cdr}_1\ (\texttt{cons}_2\ \texttt{\#f}_3\ (\texttt{cons}_4\ \texttt{\#f}_5\ \texttt{\#f}_6))))$$

and the same model $\mathcal{M}$. Fortunately, the use of a single contour helps in keeping the size of the constraints moderate. Here they are:

$$\begin{array}{ll} & \{\delta_{1,K} \supseteq \delta_{0,K}\} \\ \cup & \{\delta_{2,K} \supseteq \delta_{1,K}\} \\ \cup & \{\delta_{3,K} \supseteq \delta_{2,K}, \delta_{4,K} \supseteq \delta_{2,K}\} \end{array}$$

$$
\begin{array}{ll}
\cup & \{\delta_{3,K} \neq \emptyset \Rightarrow \alpha_{3,K} \supseteq \mathcal{V}al\mathcal{B}\} \\
\cup & \{\delta_{5,K} \supseteq \delta_{4,K}, \delta_{6,K} \supseteq \delta_{4,K}\} \\
\cup & \{\delta_{5,K} \neq \emptyset \Rightarrow \alpha_{5,K} \supseteq \mathcal{V}al\mathcal{B}\} \\
\cup & \{\delta_{6,K} \neq \emptyset \Rightarrow \alpha_{6,K} \supseteq \mathcal{V}al\mathcal{B}\} \\
\cup & \left\{ \begin{array}{l|l} \alpha_{4,K} \ni p & v_1 \in \alpha_{5,K},\ v_2 \in \alpha_{6,K} \\ \pi_p \ni (4,\, v_1,\, v_2,\, K) & p = \mathsf{pc}(4,\, v_1,\, v_2,\, K) \end{array} \right\} \\
\cup & \left\{ \begin{array}{l|l} \alpha_{2,K} \ni p & v_1 \in \alpha_{3,K},\ v_2 \in \alpha_{4,K} \\ \pi_p \ni (2,\, v_1,\, v_2,\, K) & p = \mathsf{pc}(2,\, v_1,\, v_2,\, K) \end{array} \right\} \\
\cup & \left\{ \begin{array}{l|l} \alpha_{1,K} \ni v_2 & p \in \alpha_{2,K} \cap \mathcal{V}al\mathcal{P},\ (l,\, v_1,\, v_2,\, k') \in \pi_p \end{array} \right\} \\
\cup & \left\{ \begin{array}{l|l} \alpha_{0,K} \ni v_1 & p \in \alpha_{1,K} \cap \mathcal{V}al\mathcal{P},\ (l,\, v_1,\, v_2,\, k') \in \pi_p \end{array} \right\} \\
\cup & \{\delta_{0,K} \supseteq \mathcal{V}al\mathcal{B}\}
\end{array}
$$

### 3.2.2 Safety Constraints

To verify which dynamic type tests are still required once the analysis results are computed, one can confront the latter to the safety constraints. Three kinds of expression may require dynamic type tests: calls and `car`- and `cdr`-expressions. A dynamic test may have to be included to check the value returned by their first sub-expression. Figure 3.4 presents the safety constraints generated for a program $e_{l_0}$ using a model $\mathcal{M}$. These constraints are very simple and we do not give more details on their meaning.

An expression is *safe* and does not have to comprise a dynamic type test if the safety constraints on the value of its first sub-expression (if there are any) are satisfied for all contours $k \in \mathcal{C}ont$.

A program is *analysed perfectly well* using model $\mathcal{M}$ if all safety constraints are satisfied. In other words, if the system of constraints obtained by joining both evaluation and safety constraints has a solution. A program is *analysable perfectly well* if there exists a model $\mathcal{M}$ such that $e_{l_0}$ is analysed perfectly well using $\mathcal{M}$.

If we come back to our running example, generating the constraints and confronting them to the analysis results would reveal that $e_0$ must include a dynamic type test to ensure that it always operates on pairs, but $e_1$ does not have to.

Safety constraints for program $e_{l_0}$ are:

$$\bigcup_{k \in \mathcal{C}ont} \mathcal{S}\ [\![e_{l_0}]\!]\ k,$$

where

$$\begin{aligned}
\mathcal{S}\ [\![\#\mathtt{f}_l]\!]\ k &= \emptyset\\
\mathcal{S}\ [\![\mathrm{x}_l]\!]\ k &= \emptyset\\
\mathcal{S}\ [\![({}_l e_{l_1}\ e_{l_2})]\!]\ k &= \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{C}\} \cup \mathcal{S}\ [\![e_{l_1}]\!]\ k \cup \mathcal{S}\ [\![e_{l_2}]\!]\ k\\
\mathcal{S}\ [\![(\lambda_l \mathrm{x}.\ e_{l_1})]\!]\ k &= \mathcal{S}\ [\![e_{l_1}]\!]\ k\\
\mathcal{S}\ [\![(\mathtt{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})]\!]\ k &= \mathcal{S}\ [\![e_{l_1}]\!]\ k \cup \mathcal{S}\ [\![e_{l_2}]\!]\ k \cup \mathcal{S}\ [\![e_{l_3}]\!]\ k\\
\mathcal{S}\ [\![(\mathtt{cons}_l\ e_{l_1}\ e_{l_2})]\!]\ k &= \mathcal{S}\ [\![e_{l_1}]\!]\ k \cup \mathcal{S}\ [\![e_{l_2}]\!]\ k\\
\mathcal{S}\ [\![(\mathtt{car}_l\ e_{l_1})]\!]\ k &= \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{P}\} \cup \mathcal{S}\ [\![e_{l_1}]\!]\ k\\
\mathcal{S}\ [\![(\mathtt{cdr}_l\ e_{l_1})]\!]\ k &= \{\alpha_{l_1,k} \subseteq \mathcal{V}al\mathcal{P}\} \cup \mathcal{S}\ [\![e_{l_1}]\!]\ k\\
\mathcal{S}\ [\![(\mathtt{pair?}_l\ e_{l_1})]\!]\ k &= \mathcal{S}\ [\![e_{l_1}]\!]\ k
\end{aligned}$$

Figure 3.4: Safety constraints

## 3.3 Termination of the Analysis

The following theorem establishes that an analysis instance obtained from the analysis framework (using a legal model) always terminates.

**Theorem 3.1** *An analysis performed by the evaluation constraints always finishes.*

**Proof 3.1** First, observe that each evaluation constraint can be rewritten as a set of constraints, each constraint having the form:

$$I_1\ \wedge\ \ldots\ \wedge\ I_n \Rightarrow I_0$$

where each $I_i$, $0 \leq i \leq n$, is a simple membership condition (for example, $p_3 \in \alpha_{l,k}$). It follows that the saturation of all abstract variables (for example, $\alpha_{l,k} = \mathcal{V}al$) constitutes a trivial solution to the evaluation constraints. So finding the minimum solution to the constraints is guaranteed to finish since there is only a finite number of values that can be put in each abstract variable. □

As an example, the evaluation constraints for expression $\#\mathtt{f}_l$ in contour $k$ can be transformed in the following way:

$$\begin{array}{l}\mathcal{E}\ [\![\#\mathtt{f}_l]\!]\ k = \{\delta_{l,k} \neq \emptyset \Rightarrow \alpha_{l,k} \supseteq \mathcal{V}al\mathcal{B}\} \\ \quad \mapsto \\ \mathcal{E}\ [\![\#\mathtt{f}_l]\!]\ k = \{v_1 \in \delta_{l,k} \Rightarrow v_2 \in \alpha_{l,k} \mid v_1 \in \mathcal{V}al,\ v_2 \in \mathcal{V}al\mathcal{B}\}\end{array}$$

## 3.4 A Collecting Machine

The establishment of many properties of the framework requires us to introduce a collecting machine for the mini-language. So Figure 3.5 presents the semantics of a collecting machine. The collecting machine essentially does the same computations as those performed during an ordinary evaluation except that it also builds a *cache* containing a detailed description of every step of the computations. For each evaluation of an expression in a particular evaluation context, a *pre-entry* and a *post-entry* are logged into the cache. *Concrete contours* are used by the collecting machine in order to designate each evaluation context met during the evaluation of the program.

Let us comment on Figure 3.5. First, the contours are represented by finite strings of labels. The labels in a particular contour are those of the call expressions through which invocations were done that led to the evaluation context designated by the contour. For example, the main expression of the program is evaluated in contour $\epsilon$. If closure $c_1$ is invoked from call expression $e_{l_1}$ during the evaluation of the main expression, its body is evaluated in contour $l_1$. In turn, if (another) closure $c_2$ is invoked from call expression $e_{l_2}$ during evaluation of the body of $c_1$, the body of $c_2$ is evaluated in contour $l_1 l_2$. And so on. We show below that this definition of concrete contours is sufficient to unambiguously designate each evaluation context.

Second, a cache (of type Cache) is a set of entries. Each entry is either a pre-entry, i.e. a member of PreEnt, or a post-entry, i.e. a member of PostEnt. Pre-entry pre$(l,\ k,\ \rho)$ indicates what lexical environment $\rho$ was present when expression $e_l$ got evaluated in contour $k$. Post-entry post$(l,\ k,\ v)$ indicates the value (or error value) $v$ to which expression $e_l$ has evaluated to in contour $k$.

The semantics of the collecting machine is very similar to the standard semantics of the

$$
\begin{array}{lll}
\mathrm{Val}^{\uparrow} & := \mathrm{Err} \mathbin{\dot\cup} \mathrm{Val} & \\
\mathrm{Err} & := \mathit{Errors} & \\
\mathrm{Val} & := \mathrm{ValB} \mathbin{\dot\cup} \mathrm{ValC} \mathbin{\dot\cup} \mathrm{ValP} & \\
\mathrm{ValB} & := \{\#\mathrm{f}\} & \mathit{Booleans} \\
\mathrm{ValC} & := \{\mathrm{clos}((\lambda_l \mathrm{x}.\ e),\ \rho) \mid (\lambda_l \mathrm{x}.\ e) \in \mathrm{Exp},\ \rho \in \mathrm{Env}\} & \mathit{Closures} \\
\mathrm{ValP} & := \{\mathrm{pair}(v_1,\ v_2) \mid v_1,\ v_2 \in \mathrm{Val}\} & \mathit{Pairs} \\
\mathrm{Env} & := \mathrm{Var} \to \mathrm{Val} & \\
\mathrm{Cont} & := \mathrm{Lab}^* & \mathit{Contours} \\
\mathrm{Cache} & := 2^{\mathrm{Entry}} & \\
\mathrm{Entry} & := \mathrm{PreEnt} \mathbin{\dot\cup} \mathrm{PostEnt} & \\
\mathrm{PreEnt} & := \{\mathrm{pre}(l,\ k,\ \rho) \mid l \in \mathrm{Lab},\ k \in \mathrm{Cont},\ \rho \in \mathrm{Env}\} & \\
\mathrm{PostEnt} & := \{\mathrm{post}(l,\ k,\ v) \mid l \in \mathrm{Lab},\ k \in \mathrm{Cont},\ v \in \mathrm{Val}^{\uparrow}\} &
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E} : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Cont} \to \mathrm{Val}^{\uparrow} \times \mathrm{Cache} & & \mathit{Main\ evaluation\ function} \\
\mathsf{E}\ [\![e_l]\!]\ \rho\ k & = & \mathrm{let}\ (v,\ \Xi) = \mathsf{E}'\ [\![e_l]\!]\ \rho\ k\ \mathrm{in} \\
 & & \quad (v,\ \Xi \cup \{\mathrm{pre}(l,\ k,\ \rho),\ \mathrm{post}(l,\ k,\ v)\})
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E}' : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Cont} \to \mathrm{Val}^{\uparrow} \times \mathrm{Cache} & & \mathit{Auxiliary\ eval.\ function} \\
\mathsf{E}'\ [\![\#\mathtt{f}_l]\!]\ \rho\ k & = & (\#\mathrm{f},\ \emptyset) \\
\mathsf{E}'\ [\![\mathrm{x}_l]\!]\ \rho\ k & = & (\rho\ \mathrm{x},\ \emptyset) \\
\mathsf{E}'\ [\![(_l e_1\ e_2)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ [\![e_2]\!]\ \rho\ k)\ (\mathsf{A}\ l\ k\ v_1)) \\
\mathsf{E}'\ [\![(\lambda_l \mathrm{x}.\ e_1)]\!]\ \rho\ k & = & (\mathrm{clos}((\lambda_l \mathrm{x}.\ e_1),\ \rho),\ \emptyset) \\
\mathsf{E}'\ [\![(\mathtt{if}_l\ e_1\ e_2\ e_3)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v.\ v \neq \#\mathrm{f}\ ?\ \mathsf{E}\ [\![e_2]\!]\ \rho\ k : \mathsf{E}\ [\![e_3]\!]\ \rho\ k) \\
\mathsf{E}'\ [\![(\mathtt{cons}_l\ e_1\ e_2)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v_1.\ \mathsf{C}\ (\mathsf{E}\ [\![e_2]\!]\ \rho\ k)\ (\lambda v_2.\ (\mathrm{pair}(v_1,\ v_2),\ \emptyset))) \\
\mathsf{E}'\ [\![(\mathtt{car}_l\ e_1)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v.\ v = \mathrm{pair}(v_1,\ v_2)\ ?\ (v_1,\ \emptyset)\ :\ (\textsc{error},\ \emptyset)) \\
\mathsf{E}'\ [\![(\mathtt{cdr}_l\ e_1)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v.\ v = \mathrm{pair}(v_1,\ v_2)\ ?\ (v_2,\ \emptyset)\ :\ (\textsc{error},\ \emptyset)) \\
\mathsf{E}'\ [\![(\mathtt{pair?}_l\ e_1)]\!]\ \rho\ k & = & \mathsf{C}\ (\mathsf{E}\ [\![e_1]\!]\ \rho\ k) \\
 & & \quad (\lambda v.\ v \in \mathrm{ValP}\ ?\ (v,\ \emptyset)\ :\ (\#\mathrm{f},\ \emptyset))
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{A} : \mathrm{Lab} \to \mathrm{Cont} \to \mathrm{Val} \to \mathrm{Val} \to \mathrm{Val}^{\uparrow} \times \mathrm{Cache} & & \mathit{Apply\ function} \\
\mathsf{A}\ l\ k\ v_1\ v_2 & = & (v_1 = \mathrm{clos}((\lambda_{l'} \mathrm{x}.\ e_1),\ \rho)) \\
 & & \quad ?\ \mathsf{E}\ [\![e_1]\!]\ \rho[\mathrm{x} \mapsto v_2]\ kl \\
 & & \quad :\ (\textsc{error},\ \emptyset)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{C} : \mathrm{Val}^{\uparrow} \times \mathrm{Cache} \to (\mathrm{Val} \to \mathrm{Val}^{\uparrow} \times \mathrm{Cache}) \to \mathrm{Val}^{\uparrow} \times \mathrm{Cache} & & \mathit{Check\ function} \\
\mathsf{C}\ (v_1,\ \Xi_1)\ k & = & v_1 \in \mathrm{Err}\ ?\ (v_1,\ \Xi_1) \\
 & & \qquad\qquad :\ \mathrm{let}\ (v_2,\ \Xi_2) = k\ v_1\ \mathrm{in} \\
 & & \qquad\qquad\quad (v_2,\ \Xi_1 \cup \Xi_2)
\end{array}
$$

Figure 3.5: Semantics of the collecting machine

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\triangle : \mathrm{Exp} \rightarrow 2^{\mathrm{Exp}}} \\
\triangle(\#\mathtt{f}_l) & = & \{\#\mathtt{f}_l\} \\
\triangle(\mathrm{x}_l) & = & \{\mathrm{x}_l\} \\
\triangle(({}_l e_1\ e_2)) & = & \{({}_l e_1\ e_2)\} \cup \triangle(e_1) \cup \triangle(e_2) \\
\triangle((\lambda_l \mathrm{x}.\ e_1)) & = & \{(\lambda_l \mathrm{x}.\ e_1)\} \cup \triangle(e_1) \\
\triangle((\mathtt{if}_l\ e_1\ e_2\ e_3)) & = & \{(\mathtt{if}_l\ e_1\ e_2\ e_3)\} \cup \triangle(e_1) \cup \triangle(e_2) \cup \triangle(e_3) \\
\triangle((\mathtt{cons}_l\ e_1\ e_2)) & = & \{(\mathtt{cons}_l\ e_1\ e_2)\} \cup \triangle(e_1) \cup \triangle(e_2) \\
\triangle((\mathtt{car}_l\ e_1)) & = & \{(\mathtt{car}_l\ e_1)\} \cup \triangle(e_1) \\
\triangle((\mathtt{cdr}_l\ e_1)) & = & \{(\mathtt{cdr}_l\ e_1)\} \cup \triangle(e_1) \\
\triangle((\mathtt{pair?}_l\ e_1)) & = & \{(\mathtt{pair?}_l\ e_1)\} \cup \triangle(e_1)
\end{array}
$$

Figure 3.6: Function computing the set of sub-expressions

mini-language. The major difference lies in the instrumentation that insert entries in the cache. The semantic equations are divided in the definition of the main evaluation function $\mathsf{E}$ and that of the auxiliary function $\mathsf{E}'$. $\mathsf{E}'$ is essentially similar to the standard semantic function. $\mathsf{E}$ provides the instrumentation for recording the evaluation steps and leaves the actual computations to $\mathsf{E}'$. Note also how the apply function $\mathsf{A}$ updates the contour when the invocation of a closure occurs. The label of the current call expression is appended at the end of the current contour. The body of the closure is evaluated in this extended contour.

### 3.4.1 Well-Definedness of Cache Entries

Now, we need to demonstrate that cache entries are properly recorded in the cache. In particular, that there is no ambiguity or conflict between entries. The fact that pre- and post-entries are added in the cache for each evaluation of an expression is obvious. The fact that *at most* one pre-entry and one post-entry are added in the cache for the evaluation of an expression under a certain contour is less obvious. Precisely, there should be at most one pre-entry (post-) for each expression and contour pair. In order to show this fact, we first introduce some notation, then characterise the contents of the cache returned by a call to $\mathsf{E}$, and finally show that there cannot be a conflict between entries.

Figures 3.6 and 3.7 define functions $\triangle$ and $\underline{\triangle}$, respectively. Function $\triangle$ returns the set of sub-expressions of a particular expression. Function $\underline{\triangle}$ returns the set of *immediate* sub-expressions. The immediate sub-expressions of $e_l$ are the ones that could be evaluated

$$
\begin{array}{lcl}
\underline{\triangle} : \mathrm{Exp} \rightarrow 2^{\mathrm{Exp}} & & \\
\underline{\triangle}(\texttt{\#f}_l) & = & \{\texttt{\#f}_l\} \\
\underline{\triangle}(\mathrm{x}_l) & = & \{\mathrm{x}_l\} \\
\underline{\triangle}((_l e_1\ e_2)) & = & \{(_l e_1\ e_2)\} \cup \underline{\triangle}(e_1) \cup \underline{\triangle}(e_2) \\
\underline{\triangle}((\lambda_l \mathrm{x}.\ e_1)) & = & \{(\lambda_l \mathrm{x}.\ e_1)\} \\
\underline{\triangle}((\texttt{if}_l\ e_1\ e_2\ e_3)) & = & \{(\texttt{if}_l\ e_1\ e_2\ e_3)\} \cup \underline{\triangle}(e_1) \cup \underline{\triangle}(e_2) \cup \underline{\triangle}(e_3) \\
\underline{\triangle}((\texttt{cons}_l\ e_1\ e_2)) & = & \{(\texttt{cons}_l\ e_1\ e_2)\} \cup \underline{\triangle}(e_1) \cup \underline{\triangle}(e_2) \\
\underline{\triangle}((\texttt{car}_l\ e_1)) & = & \{(\texttt{car}_l\ e_1)\} \cup \underline{\triangle}(e_1) \\
\underline{\triangle}((\texttt{cdr}_l\ e_1)) & = & \{(\texttt{cdr}_l\ e_1)\} \cup \underline{\triangle}(e_1) \\
\underline{\triangle}((\texttt{pair?}_l\ e_1)) & = & \{(\texttt{pair?}_l\ e_1)\} \cup \underline{\triangle}(e_1)
\end{array}
$$

Figure 3.7: Function computing the set of immediate sub-expressions

if $e_l$ were evaluated, but without going through a closure invocation. For example, if

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

then if $e_{l'''}$ is evaluated while $e_l$ is evaluated, it is necessarily through a closure invocation. That implies that $e_{l'''}$ is not an immediate sub-expression of $e_l$. The difference between the implementation of $\triangle$ and $\underline{\triangle}$ only lies in the treatment of $\lambda$-expressions. The definition of $\underline{\triangle}$ is purely syntactic and does not try to determine if a sub-expression may *really* be evaluated.

We will not distinguish between the expressions and their labels in the use of $\triangle$ and $\underline{\triangle}$. They could as well have type Lab $\rightarrow 2^{\mathrm{Lab}}$, or Exp $\rightarrow 2^{\mathrm{Lab}}$, etc.

The following theorem characterises the entries that may appear in a cache returned by the collecting machine.

**Theorem 3.2** *Let* $e_0 \in \mathrm{Exp}$ *be a program and* $e_l \in \triangle(e_0)$*, a sub-expression. Also, let* $(v, \Xi) = \mathsf{E}\ [\![e_l]\!]\ \rho\ k$*. All entries in cache* $\Xi$ *have the form* $\mathrm{pre}(l', k', \rho')$ *or* $\mathrm{post}(l', k', v')$ *where*

$$(k' = k\ \wedge\ l' \in \underline{\triangle}(l))\ \vee\ (k' = kl''k''\ \wedge\ l'' \in \underline{\triangle}(l))$$

What the theorem means is that all contours met during evaluation of $e_l$ have $k$ as a prefix. Cases where precisely $k$ was met involve immediate sub-expressions of $e_l$. And in cases where an extension of $k$ was met, the label used to extend $k$ for the first time is one belonging to an immediate sub-expression of $e_l$.

**Proof 3.2** We prove the property by induction on the number of uses of the function $\mathsf{E}$ in the computation of $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$. The proof is easy and a complete one would be too lengthy. We only cover a few cases.

*Basis.* $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed with one use of $\mathsf{E}$. Necessarily, $e_l = \#\mathtt{f}_l$, $e_l = \mathrm{x}_l$, or $e_l = (\lambda_l \mathrm{x}.\ e_{l'})$. Then:

$$\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v)\}$$

Clearly, both entries in $\Xi$ have the desired form as they contain the contour $k$ and $l \in \underline{\triangle}(l)$.

*Induction hypothesis.* Let us suppose that entries in $\Xi$ have the desired form if $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed in at most $n_0$ uses of $\mathsf{E}$.

*Induction step.* $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed in $n_0 + 1$ uses of $\mathsf{E}$. Necessarily, $e_l$ is one of:

$$\left\{ \begin{array}{lll} ({}_l e_{l_1}\ e_{l_2}), & (\mathtt{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3}), & (\mathtt{cons}_l\ e_{l_1}\ e_{l_2}), \\ (\mathtt{car}_l\ e_{l_1}), & (\mathtt{cdr}_l\ e_{l_1}), & (\mathtt{pair?}_l\ e_{l_1}) \end{array} \right\}$$

As it is the most complex and, consequently, a good representative, we present the case where $e_l = ({}_l e_{l_1}\ e_{l_2})$.

The first sub-case occurs when $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1,\ \Xi_1)$ where $v_1 \in \mathrm{Err}$. It follows that

$$\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v_1)\} \cup \Xi_1$$

Note that $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ is computed with $n_0$ uses of $\mathsf{E}$. So, by induction hypothesis, each entry in $\Xi_1$ is of the form $\mathrm{pre}(l',\ k',\ \rho')$ or $\mathrm{post}(l',\ k',\ v')$ where

$$(k' = k\ \wedge\ l' \in \underline{\triangle}(l_1))\ \vee\ (k' = kl''k''\ \wedge\ l'' \in \underline{\triangle}(l_1))$$

Since $\underline{\triangle}(l_1) \subseteq \underline{\triangle}(l)$, we can conclude that each entry in $\Xi$ has the desired form.

The second sub-case occurs when $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1,\ \Xi_1)$ where $v_1 \in \mathrm{Val}$, $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k = (v_2,\ \Xi_2)$, and either $v_2 \in \mathrm{Err}$ or $v_1 \notin \mathrm{ValC}$. It follows that

$$\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ \textsc{error})\} \cup \Xi_1 \cup \Xi_2$$

Again, by induction hypothesis, entries in $\Xi_1$ and $\Xi_2$ have the desired form (relatively to $l_1$ and $l_2$, respectively), and we can conclude that entries in $\Xi$ all have the desired form.

The last sub-case occurs when

$$\begin{array}{lcl}\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k & = & (\mathrm{clos}((\lambda_{l_4}\mathrm{x}.\ e_{l_3}),\ \rho_1),\ \Xi_1), \\ \mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k & = & (v_2,\ \Xi_2)\ \text{where}\ v_2 \in \mathrm{Val},\ \text{and} \\ \mathsf{E}\ [\![e_{l_3}]\!]\ \rho_1[\mathrm{x} \mapsto v_2]\ kl & = & (v_3,\ \Xi_3).\end{array}$$

It follows that

$$\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v_3)\} \cup \Xi_1 \cup \Xi_2 \cup \Xi_3$$

Once again, the induction hypothesis can be used to determine that entries in $\Xi_1$ have the desired form relatively to $l_1$ and $k$, entries in $\Xi_2$ have the desired form relatively to $l_2$ and $k$, and entries in $\Xi_3$ have the desired form relatively to $l_3$ and $kl$. Note that all contours found in entries of $\Xi_3$ have $k$ as a *strict* prefix. We can conclude that all entries in $\Xi$ have the desired form. □

With the help of this theorem, we can show that the contours unambiguously designate the various evaluation contexts in which expressions are evaluated in the collecting machine. In other words, that each distinct evaluation of a particular expression occurs in a distinct contour.

**Theorem 3.3** *Let* $e_0 \in \mathrm{Exp}$ *be a program, and let* $(v_0,\ \Xi_0) = \mathsf{E}\ [\![e_0]\!]\ \cdot\ \epsilon$. *We have that*

$$\begin{array}{l}\forall l' \in \mathrm{Lab},\ \ k' \in \mathrm{Cont}. \\ \qquad |\{\mathrm{pre}(l',\ k',\ \rho) \in \Xi_0 \mid \rho \in \mathrm{Env}\}| = |\{\mathrm{post}(l',\ k',\ v) \in \Xi_0 \mid v \in \mathrm{Val}^{\uparrow}\}| \leq 1\end{array}$$

**Proof 3.3** We make the demonstration by induction on the number of uses of $\mathsf{E}$ necessary to compute $(v,\ \Xi) = \mathsf{E}\ [\![e_l]\!]\ \rho\ k$ where $e_l \in \triangle e_0$, $\rho \in \mathrm{Env}$, and $k \in \mathrm{Cont}$. For brevity, we consider only a few cases.

*Basis.* If $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed with one use of $\mathsf{E}$, verifying the property is trivial.

*Induction hypothesis.* Suppose that the desired property is true for $\Xi$ when the number of uses of $\mathsf{E}$ is at most $n_0$.

*Induction step.* $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed in $n_0 + 1$ uses of $\mathsf{E}$. Then $e_l$ has to be one of six kinds of expressions. As it is the most complex, we choose the case where $e_l = ({}_l e_{l_1}\ e_{l_2})$ as a representative. Also, we restrict ourselves to the sub-case where

$$\begin{array}{lcl}\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k & = & (\mathrm{clos}((\lambda_{l_4}\mathrm{x}.\ e_{l_3}),\ \rho_1),\ \Xi_1), \\ \mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k & = & (v_2,\ \Xi_2)\ \text{where}\ v_2 \in \mathrm{Val},\ \text{and} \\ \mathsf{E}\ [\![e_{l_3}]\!]\ \rho_1[\mathrm{x} \mapsto v_2]\ kl & = & (v_3,\ \Xi_3).\end{array}$$

It follows that

$$\Xi = \Xi_+ \cup \Xi_1 \cup \Xi_2 \cup \Xi_3\ \text{where}\ \Xi_+ = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v)\}$$

The desired property holds for all cache parts $\Xi_1$, $\Xi_2$, and $\Xi_3$ as each of the three sub-evaluations uses $\mathsf{E}$ less than $n_0$ times and consequently the induction hypothesis applies. So it is easy to first convince oneself that

$$\begin{array}{l}\forall l' \in \mathrm{Lab},\ k' \in \mathrm{Cont}. \\ \quad \Big[\exists \rho' \in \mathrm{Env}.\ \mathrm{pre}(l',\ k',\ \rho') \in \Xi\ \ \text{if and only if}\ \ \exists v' \in \mathrm{Val}.\ \mathrm{post}(l',\ k',\ v') \in \Xi\Big]\end{array}$$

What remains to be shown is either the non-existence or the uniqueness of the pre-entry for a particular expression $e_{l'}$ and contour $k'$. Similarly for the post-entries. As the arguments for both kinds of entries are almost the same, the rest of the demonstration considers only pre-entries.

Now, to make the theorem false, we would have to find two conflicting pre-entries in $\Xi$. That is, $\mathrm{pre}(l',\ k',\ \rho')$, $\mathrm{pre}(l',\ k',\ \rho'') \in \Xi$ such that $\rho' \neq \rho''$. The two pre-entries cannot come from only one of the cache parts $\Xi_+$, $\Xi_1$, $\Xi_2$, and $\Xi_3$ as $\Xi_+$ introduces only one pre-entry and the others have been given to us by the induction hypothesis. Let us enumerate the different possibilities for the source of the two pre-entries and show that each possibility leads to a contradiction.

If $\mathrm{pre}(l',\ k',\ \rho') \in \Xi_+$ and $\mathrm{pre}(l',\ k',\ \rho'') \in \Xi_1$, then $l' = l$, $k' = k$, and it implies that $l \in \underline{\triangle}(l_1)$. Contradiction.

If $\mathrm{pre}(l',\ k',\ \rho') \in \Xi_+$ and $\mathrm{pre}(l',\ k',\ \rho'') \in \Xi_2$, then, similarly, it implies that $l \in \underline{\triangle}(l_2)$. Contradiction.

If $\mathrm{pre}(l',\ k',\ \rho') \in \Xi_+$ and $\mathrm{pre}(l',\ k',\ \rho'') \in \Xi_3$, then $k'$ would have to be equal to $k$ and have $k$ as a strict prefix at the same time. Contradiction.

If $\mathrm{pre}(l', k', \rho') \in \Xi_1$ and $\mathrm{pre}(l', k', \rho'') \in \Xi_2$, then there are two cases. Either $k' = k$ and $l' \in \underline{\triangle}(l_1) \cap \underline{\triangle}(l_2) = \emptyset$. Contradiction. Or $k' = kl''k''$ where $l'' \in \underline{\triangle}(l_1) \cap \underline{\triangle}(l_2) = \emptyset$. Contradiction.

If $\mathrm{pre}(l', k', \rho') \in \Xi_1$ and $\mathrm{pre}(l', k', \rho'') \in \Xi_3$, then $k' = klk''$ and it implies that $l \in \underline{\triangle}(l_1)$. Contradiction.

Finally, if $\mathrm{pre}(l', k', \rho') \in \Xi_2$ and $\mathrm{pre}(l', k', \rho'') \in \Xi_3$, then, similarly, it implies that $l \in \underline{\triangle}(l_2)$. Contradiction. □

## 3.5 Conservativeness of the Analysis

An essential property about our analysis framework is that any analysis instance that it produces is *conservative*. In short, the analysis results always force the optimiser to include at least all the truly required dynamic type tests, and so, no matter what the abstract model is. This property is to be established as the final result of this section and it is derived from the main theorem saying that an analysis instance mimics conservatively the concrete evaluation of the program. Before we present both, we first introduce many definitions and notations helping in the next proofs.

### 3.5.1 Accessory Definitions

Let $e_0 \in \mathrm{Exp}$ be the program to analyse. Let $\mathcal{M} = (\mathcal{V}al\mathcal{B}, \mathcal{V}al\mathcal{C}, \mathcal{V}al\mathcal{P}, \mathcal{C}ont, \hat{k}_0, \mathsf{cc}, \mathsf{pc}, \mathsf{call})$ be the abstract model. We will denote the analysis results by $\mathcal{R}$. Formally,

$$\mathcal{R} = (\alpha, \beta, \gamma, \delta, \chi, \pi, \kappa) = \mathrm{FW}(e_0, \mathcal{M})$$

As the proof of conservativeness mentions both concrete and abstract values, a hat marks the abstract values.

We define the abstract environment function $\hat{\rho}$ this way:

$$\begin{aligned} &\hat{\rho} : \mathrm{Lab} \times \mathcal{C}ont \rightarrow \mathrm{Var} \rightarrow 2^{\mathcal{V}al} \\ &\hat{\rho}(l, \hat{k})(\mathrm{x}) = \mathrm{ref}(\mathrm{x}, l, \hat{k}) \end{aligned}$$

That is, it returns the abstract lexical environment visible from expression $e_l$ in contour $\hat{k}$.

Next, we define the "*is abstracted by*" relation. We denote the relation by the $\nearrow$ glyph. This relation is defined in terms of the abstract model and parts of the analysis results. These equations define when a concrete value is considered to be abstracted by an abstract value:

$$\begin{array}{ll}
\#\mathrm{f} \nearrow \hat{v}, & \text{if } \hat{v} \in \mathcal{V}al\mathcal{B} \\
\mathrm{clos}((\lambda_l \mathrm{x}.\ e),\, \rho) \nearrow \hat{v}, & \text{if } \hat{v} \in \mathcal{V}al\mathcal{C} \text{ and } \exists (l,\, \hat{k}) \in \chi_{\hat{v}}.\ \rho \nearrow \hat{\rho}(l, \hat{k}) \\
\mathrm{pair}(v_1,\, v_2) \nearrow \hat{v}, & \text{if } \hat{v} \in \mathcal{V}al\mathcal{P} \text{ and } \exists (l,\, \hat{v}_1,\, \hat{v}_2,\, \hat{k}) \in \pi_{\hat{v}}.\ v_1 \nearrow \hat{v}_1 \ \wedge\ v_2 \nearrow \hat{v}_2
\end{array}$$

The relation $\nearrow$ on values basically verifies that an abstract value has at least the same *behaviour* as the concrete one. There are no special conditions for Booleans. The conditions for pairs verify that appropriate values can be extracted from the CAR- and CDR-fields of the abstract pair. The conditions for closures verify that the right $\lambda$-expression can be recovered with an appropriate lexical environment. This last test consists in testing if an abstract lexical environment abstracts a concrete lexical environment. We define the $\nearrow$ relation on environments as:

$$\rho \nearrow \hat{\rho}(l, \hat{k}), \quad \text{if } \forall \mathrm{x} \in \mathrm{Dom}(\rho).\ \exists \hat{v} \in \hat{\rho}(l, \hat{k}).\ \rho(\mathrm{x}) \nearrow \hat{v}$$

Now, with the help of the $\nearrow$ relation defined on values and lexical environments, we can formally explain what it means for analysis results to mimic conservatively the concrete evaluation of a program. The relation also relates caches and analysis results conditional to the provision of a *contour abstraction function.*

$$\begin{array}{l}
\Xi \nearrow_a \mathcal{R} \text{ if} \\
\quad a : \mathrm{Cont} \rightarrow \mathcal{C}ont\ \bigwedge \\
\quad \left[\forall \mathrm{pre}(l,\, k,\, \rho) \in \Xi.\ \delta_{l,a(k)} \neq \emptyset\ \wedge\ \rho \nearrow \hat{\rho}(l, a(k))\right]\ \bigwedge \\
\quad \left[\forall \mathrm{post}(l,\, k,\, v) \in \Xi.\ (\exists \hat{v} \in \alpha_{l,a(k)}.\ v \nearrow \hat{v})\ \vee v \in \mathrm{Err}\right]
\end{array}$$

### 3.5.2 Conservative Mimicking of the Evaluation

Before we proceed with the main theorem, we introduce this little lemma. We do not prove it as quick examination of the semantics of the collecting machine is sufficient to convince oneself that it is true.

**Lemma 3.4** *Let* $e_l \in \mathrm{Exp}$*,* $\rho \in \mathrm{Env}$*, and* $k \in \mathrm{Cont}$*:*

$$\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (v,\,\Xi) \Rightarrow \mathrm{post}(l,\,k,\,v) \in \Xi$$

The following theorem constitutes the main part of the demonstration that any analysis instance coming from the framework is conservative. The proof follows.

**Theorem 3.5** *Let* $e_{l_0} \in \mathrm{Exp}$ *be a program and let* $e_l \in \triangle(e_{l_0})$*. Let the model* $\mathcal{M}$ *be* $(\mathcal{V}al\mathcal{B}$*,* $\mathcal{V}al\mathcal{C}$*,* $\mathcal{V}al\mathcal{P}$*,* $\mathcal{C}ont$*,* $\hat{k}_0$*,* $\mathsf{cc}$*,* $\mathsf{pc}$*,* $\mathsf{call})$*. Let* $\mathcal{R} = \mathrm{FW}(e_{l_0}, \mathcal{M})$ *be the analysis results for* $e_{l_0}$*.*

$$\begin{array}{l} \mathsf{E}\ [\![e_l]\!]\ \rho\ k = (v,\,\Xi)\ \wedge\ \delta_{l,\hat{k}} \neq \emptyset\ \wedge\ \rho \nearrow \hat{\rho}(l,\hat{k}) \\ \Rightarrow\quad \exists a : \mathrm{Cont} \to \mathcal{C}ont.\ \left(\Xi \nearrow_a \mathcal{R}\ \wedge\ a(k) = \hat{k}\right) \end{array}$$

The theorem says that the concrete evaluation of an expression in some evaluation context has an abstract counterpart as long as the expression is evaluated in an appropriate abstract evaluation context. The $a$ function provided by the theorem is the contour abstraction function and it indicates to which abstract contour each concrete contour should be mapped to. The theorem applies only if an appropriate abstract evaluation context is found. That is, it applies only if there is an abstract contour in which $e_l$ gets evaluated and in which the lexical environment abstracts $\rho$. This may seem to weaken the theorem, but note that we do not require $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ to be an actual part of the concrete evaluation of the whole program. It will quickly become apparent in the proof that, if $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is an actual part of the whole evaluation, then there will exist an abstract contour $\hat{k}$ in which $e_l$ is evaluated within an appropriate abstract lexical environment.

**Proof 3.5** We prove the theorem by induction on the number of uses of $\mathsf{E}$ in the evaluation $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$. To have a more precise argumentation, we define the following property $P$:

$$\begin{array}{rl} P(n): & \mathsf{E}\ [\![e_l]\!]\ \rho\ k \text{ is computed in at most } n \text{ uses of } \mathsf{E}\ \wedge \\ & \mathsf{E}\ [\![e_l]\!]\ \rho\ k = (v,\,\Xi)\ \wedge\ \delta_{l,\hat{k}} \neq \emptyset\ \wedge\ \rho \nearrow \hat{\rho}(l,\hat{k}) \\ & \Rightarrow\quad \exists a : \mathrm{Cont} \to \mathcal{C}ont.\ \left(\Xi \nearrow_a \mathcal{R}\ \wedge\ a(k) = \hat{k}\right) \end{array}$$

*Basis.* We must show that $P(1)$ is satisfied. So we only need to consider cases where $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed in exactly one use of $\mathsf{E}$. The only expressions that can get evaluated in one use of $\mathsf{E}$ are the false constant, the variable reference, and the $\lambda$-expression. Let us examine each case in turn.

First case: $e_l = \#\mathtt{f}_l$. We have that:

1. $\mathsf{E}\,[\![e_l]\!]\,\rho\,k = (\#\mathrm{f},\,\Xi)$ where $\Xi = \{\mathrm{pre}(l,\,k,\,\rho), \mathrm{post}(l,\,k,\,\#\mathrm{f})\}$, by the collecting machine semantics;
2. let us define $a : \mathrm{Cont} \to \mathcal{C}ont$ as $[k \mapsto \hat{k}]$; that is, it is only defined in $k$ and $a(k) = \hat{k}$;
3. $\delta_{l,a(k)} \neq \emptyset$, because $\delta_{l,\hat{k}} \neq \emptyset$ and the definition of $a$;
4. $\rho \nearrow \hat{\rho}(l, a(k))$, because $\rho \nearrow \hat{\rho}(l, \hat{k})$ and by def. of $a$;
5. $\forall \mathrm{pre}(l',\,k',\,\rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$, by 3 and 4;
6. $\exists \hat{v}' \in \alpha_{l,a(k)}.\ \#\mathrm{f} \nearrow \hat{v}$, because $\delta_{l,\hat{k}} \neq \emptyset$ implies $\alpha_{l,\hat{k}} \supseteq \mathcal{V}al\mathcal{B}$, by the evaluation constraints of the analysis;
7. $\forall \mathrm{post}(l',\,k',\,v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$, by 6;
8. $\Xi \nearrow_a \mathcal{R}$, by 5 and 7.

So, $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

Second case: $e_l = \mathrm{x}_l$. We have that:

1. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (\rho\ \mathrm{x},\,\Xi)$ where $\Xi = \{\mathrm{pre}(l,\,k,\,\rho), \mathrm{post}(l,\,k,\,\rho\ \mathrm{x})\}$, by the collecting machine semantics;
2. let $a = [k \mapsto \hat{k}]$;
3. $\delta_{l,a(k)} \neq \emptyset$;
4. $\rho \nearrow \hat{\rho}(l, a(k))$;
5. $\forall \mathrm{pre}(l',\,k',\,\rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$;
6. $\exists \hat{v}' \in \hat{\rho}(l, \hat{k})(\mathrm{x}).\ \rho\ \mathrm{x} \nearrow \hat{v}'$, by 4;
7. $\exists \hat{v}' \in \alpha_{l,a(k)}.\ \rho\ \mathrm{x} \nearrow \hat{v}'$, because $\delta_{l,\hat{k}} \neq \emptyset$ implies $\alpha_{l,\hat{k}} \supseteq \mathrm{ref}(\mathrm{x},\,l,\,\hat{k}) = \hat{\rho}(l, \hat{k})(\mathrm{x})$, by the evaluation constraints, the definition of $\hat{\rho}$, and 6;

8. $\forall \text{post}(l',\, k',\, v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \text{Err}$, by 7;

9. $\Xi \nearrow_a \mathcal{R}$.

So, $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

Third case: $e_l = (\lambda_l \text{x}.\ e_{l_1})$. We have that:

1. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (c,\, \Xi)$ where $c = \text{clos}((\lambda_l \text{x}.\ e_{l_1}),\, \rho)$ and $\Xi = \{\text{pre}(l,\, k,\, \rho), \text{post}(l,\, k,\, c)\}$;

2. let $a = [k \mapsto \hat{k}]$;

3. $\delta_{l,a(k)} \neq \emptyset$;

4. $\rho \nearrow \hat{\rho}(l, a(k))$;

5. $\forall \text{pre}(l',\, k',\, \rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$;

6. $\alpha_{l,\hat{k}} \ni \mathsf{cc}(l, \hat{k})$ and $\chi_{\mathsf{cc}(l,\hat{k})} \ni (l, \hat{k})$, because $\delta_{l,\hat{k}} \neq \emptyset$, and by the evaluation constraints;

7. $c \nearrow \mathsf{cc}(l, \hat{k})$, because $\mathsf{cc}(l, \hat{k}) \in \mathcal{V}al\mathcal{C}$ and by 6;

8. $\exists \hat{v}' \in \alpha_{l,a(k)}.\ c \nearrow \hat{v}'$, by 6 and 7;

9. $\forall \text{post}(l',\, k',\, v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \text{Err}$, by 8;

10. $\Xi \nearrow_a \mathcal{R}$.

So, $\Xi \nearrow_a \mathcal{R}$ and $a(k) = \hat{k}$.

For all three possible kinds of expressions, we obtained that $\Xi \nearrow_a \mathcal{R}$ for a function $a : \text{Cont} \to \mathcal{C}ont$ such that $a(k) = \hat{k}$. So $P(1)$ is satisfied.

*Induction hypothesis.* Let us suppose that $P(n-1)$ is satisfied for some $n \geq 2$.

*Induction step.* Now, we must show that $P(n)$ is also satisfied. Note that we have to provide a demonstration only for the cases where $\mathsf{E}\ [\![e_l]\!]\ \rho\ k$ is computed in exactly $n$ uses of $\mathsf{E}$ as the cases for less than $n$ uses are already covered by the induction hypothesis.

Since $n \geq 2$, the only kinds of expressions that are possible for $e_l$ are precisely those that were impossible in the induction basis. In order to avoid starting with the difficult call-expression case, we go through the kinds of expressions from the last to the first.

First case: $e_l = (\texttt{pair?}_l\ e_{l_1})$. The evaluation starts by computing $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1,\ \Xi_1)$. Three sub-cases may occur: $v_1$ is an error, a pair, or a non-pair value.

Let us first consider the sub-case where $v_1 \in \mathrm{Err}$. We have that:

1. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (v_1,\ \Xi)$ where $\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v_1)\} \cup \Xi_1$;
2. the computation of $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ is done in less than $n$ uses of $\mathsf{E}$;
3. $\delta_{l_1,\hat{k}} \neq \emptyset$ by the fact that $\delta_{l,\hat{k}} \neq \emptyset$ and the evaluation constraints;
4. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ because: $e_l$ is not a $\lambda$-expression, so $\mathrm{ref}(\mathrm{x},\ l_1,\ \hat{k}) = \mathrm{ref}(\mathrm{x},\ l,\ \hat{k})$ (for any $\mathrm{x} \in \mathrm{Var}$ in the lexical environment), and so, $\hat{\rho}(l_1, \hat{k}) = \hat{\rho}(l, \hat{k})$;
5. $\Xi_1 \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$, because of 2, 3, 4, and the induction hypothesis;
6. $\delta_{l,a(k)} \neq \emptyset$;
7. $\rho \nearrow \hat{\rho}(l, a(k))$;
8. $\forall \mathrm{pre}(l',\ k',\ \rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$, by 1, 5, 6, and 7;
9. — 11. (non-existent)
12. $\forall \mathrm{post}(l',\ k',\ v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$, by 1, 5, and the fact that $v_1 \in \mathrm{Err}$;
13. $\Xi \nearrow_a \mathcal{R}$.

So, $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

The second sub-case occurs when $v_1 \in \mathrm{ValP}$. Here, we give only the reasoning steps that must be changed from the proof of the first sub-case:

9. $\mathrm{post}(l_1,\ k,\ v_1) \in \Xi_1$, because of the fact that $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1,\ \Xi_1)$ and Lemma 3.4;
10. $\exists \hat{v}' \in \alpha_{l_1,\hat{k}}.\ v_1 \nearrow \hat{v}'$, by 5 and the fact that $v_1 \notin \mathrm{Err}$;
11. $\exists \hat{v}' \in \alpha_{l,\hat{k}}.\ v_1 \nearrow \hat{v}'$, by 10, the fact that $v_1 \in \mathrm{ValP}$ (so $\hat{v}' \in \mathcal{V}al\mathcal{P}$), and the evaluation constraints;

The third sub-case occurs when $v_1 \in \mathrm{ValB} \cup \mathrm{ValC}$. The changes in the reasoning are:

1. $\mathsf{E}\ \llbracket e_l \rrbracket\ \rho\ k = (\#\mathrm{f},\ \Xi)$ where $\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ \#\mathrm{f})\} \cup \Xi_1$;

11. $\exists \hat{v}'' \in \alpha_{l,\hat{k}}.\ \#\mathrm{f} \nearrow \hat{v}''$, by 10, the fact that $v_1 \in \mathrm{ValB} \cup \mathrm{ValC}$ (so $\hat{v}' \in \mathcal{V}al\mathcal{B} \cup \mathcal{V}al\mathcal{C}$), and the evaluation constraints;

Since all three sub-cases are verified, the cache resulting from the evaluation of a pair-membership test expression is abstracted by the analysis results.

Second case: $e_l = (\texttt{cdr}_l\ e_{l_1})$. Again, the evaluation of $e_l$ starts by computing $\mathsf{E}\ \llbracket e_{l_1} \rrbracket\ \rho\ k = (v_1,\ \Xi_1)$. The same three sub-cases as those seen with the `pair`?-expression must be considered. We skip the $v_1 \in \mathrm{Err}$ sub-case since its treatment is almost identical as that of the `pair`?-expression.

So we start by considering the sub-case where $v_1 = (v_1',\ v_1'') \in \mathrm{ValP}$. We have that:

1. $\mathsf{E}\ \llbracket e_l \rrbracket\ \rho\ k = (v_1'',\ \Xi)$ where $\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ v_1'')\} \cup \Xi_1$;

2. the computation of $\mathsf{E}\ \llbracket e_{l_1} \rrbracket\ \rho\ k$ is done in less than $n$ uses of $\mathsf{E}$;

3. $\delta_{l_1,\hat{k}} \neq \emptyset$ because of the fact that $\delta_{l,\hat{k}} \neq \emptyset$ and the evaluation constraints;

4. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ because: $e_l$ is not a $\lambda$-expression, so $\mathrm{ref}(\mathrm{x},\ l_1,\ \hat{k}) = \mathrm{ref}(\mathrm{x},\ l,\ \hat{k})$ (for any $\mathrm{x} \in \mathrm{Var}$ in the lexical environment), and so, $\hat{\rho}(l_1, \hat{k}) = \hat{\rho}(l, \hat{k})$;

5. $\Xi_1 \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$, because of 2, 3, 4, and the induction hypothesis;

6. $\delta_{l,a(k)} \neq \emptyset$;

7. $\rho \nearrow \hat{\rho}(l, a(k))$;

8. $\forall \mathrm{pre}(l',\ k',\ \rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$, by 1, 5, 6, and 7;

9. $\mathrm{post}(l_1,\ k,\ v_1) \in \Xi_1$, because of the fact that $\mathsf{E}\ \llbracket e_{l_1} \rrbracket\ \rho\ k = (v_1,\ \Xi_1)$ and Lemma 3.4;

10. $\exists \hat{v}' \in \alpha_{l_1,\hat{k}}.\ v_1 \nearrow \hat{v}'$, by 5 and the fact that $v_1 \notin \mathrm{Err}$;

11. let $\hat{p} \in \alpha_{l_1,\hat{k}}$ such that $v_1 \nearrow \hat{p}$;

12. $\hat{p} \in \mathcal{V}al\mathcal{P}$ and $\exists (l',\ \hat{p}',\ \hat{p}'',\ \hat{k}') \in \pi_{\hat{p}}.\ v_1' \nearrow \hat{p}'\ \wedge\ v_1'' \nearrow \hat{p}''$, by 11;

13. $\hat{p}'' \in \alpha_{l,\hat{k}}$ by 12 and the evaluation constraints;

14. $\forall \mathrm{post}(l',\ k',\ v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$, by 1, 5, and 12;

15. $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

Note how the $\nearrow$ relation is helpful in the reasoning. It determines that $\hat{p}$ is an abstraction of $v_1$ based on the *observable behaviour* of both values. Let us explain ourselves. The essence of concrete value $v_1$ is that it is a pair, and it contains two values $v_1'$ and $v_1''$ in its fields. The essence of abstract value $\hat{p}$ is that it is a pair and, according to log variable $\pi_{\hat{p}}$, it has, among other things, been formed by consing together $\hat{p}'$ and $\hat{p}''$, that is, abstractions of $v_1'$ and $v_1''$, respectively.

The third sub-case occurs when $v_1 \in \mathrm{ValB} \cup \mathrm{ValC}$. We give only the modified steps:

1. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (\textsc{error},\ \Xi)$ where $\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ \textsc{error})\} \cup \Xi_1$;

9. — 13. (removed)

14. $\forall \mathrm{post}(l',\ k',\ v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$, by 1, 5, and the fact that the result is an error;

This ends the demonstration for case $e_l = (\texttt{cdr}_l\ e_{l_1})$.

Third case: $e_l = (\texttt{car}_l\ e_{l_1})$. Since the reasoning is analogous to that for the `cdr`-expression, we skip it entirely.

Fourth case: $e_l = (\texttt{cons}_l\ e_{l_1}\ e_{l_2})$. The evaluation of one of the sub-expressions may lead to an error. Moreover, the related sub-cases are not really interesting and their demonstration could easily be done by adapting the one for the error sub-case in the `pair?`-expression demonstration. So we concentrate immediately on the interesting sub-case where both sub-expressions evaluate to normal values. Note that we will be a little more concise in the demonstration:

1. let $(v_1,\ \Xi_1) = \mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ where $v_1 \in \mathrm{Val}$;

2. let $(v_2,\ \Xi_2) = \mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k$ where $v_2 \in \mathrm{Val}$;

3. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (p,\ \Xi)$ where $p = \mathrm{pair}(v_1,\ v_2)$ and $\Xi = \{\mathrm{pre}(l,\ k,\ \rho), \mathrm{post}(l,\ k,\ p)\} \cup \Xi_1 \cup \Xi_2$

4. the computation of each of $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ and $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k$ uses $\mathsf{E}$ less than $n$ times;

5. $\delta_{l_1,\hat{k}} \neq \emptyset$ and $\delta_{l_2,\hat{k}} \neq \emptyset$;

6. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ and $\rho \nearrow \hat{\rho}(l_2, \hat{k})$;

7. $\Xi_1 \nearrow_{a_1} \mathcal{R}$ and $\Xi_2 \nearrow_{a_2} \mathcal{R}$, where $a_1(k) = \hat{k}$ and $a_2(k) = \hat{k}$, by 4, 5, 6, and the induction hypothesis;

8. let $a = a_1 a_2$; that is, $a$ contains all the bindings that form both $a_1$ and $a_2$; note that there is conflict in doing so; this is because Theorem 3.2 guarantees us that $\mathrm{Dom}(a_1) \cap \mathrm{Dom}(a_2) = \{k\}$ and we know that $a_1(k) = a_2(k) = \hat{k}$;

9. $\Xi_1 \nearrow_a \mathcal{R}$ and $\Xi_2 \nearrow_a \mathcal{R}$, by 7 and 8;

10. $\forall \mathrm{pre}(l', k', \rho') \in \Xi.\ \delta_{l', a(k')} \neq \emptyset \ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$, by 3, 9, and the theorem preconditions;

11. $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1, \Xi_1) \Rightarrow \mathrm{post}(l_1, k, v_1) \in \Xi_1 \Rightarrow \exists \hat{v}'_1 \in \alpha_{l_1, \hat{k}}.\ v_1 \nearrow \hat{v}'_1$; let $\hat{v}_1$ be that value;

12. $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k = (v_2, \Xi_2) \Rightarrow \mathrm{post}(l_2, k, v_2) \in \Xi_2 \Rightarrow \exists \hat{v}'_2 \in \alpha_{l_2, \hat{k}}.\ v_2 \nearrow \hat{v}'_2$; let $\hat{v}_2$ be that value;

13. let $\hat{p} = \mathsf{pc}(l, \hat{v}_1, \hat{v}_2, \hat{k})$;

14. $\hat{p} \in \alpha_{l, \hat{k}}$ and $(l, \hat{v}_1, \hat{v}_2, \hat{k}) \in \pi_{\hat{p}}$, by the evaluation constraints;

15. $p \nearrow \hat{p}$, by 3, 11, 12, and 14;

16. $\forall \mathrm{post}(l', k', v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l', a(k')}.\ v' \nearrow \hat{v}') \ \vee\ v' \in \mathrm{Err}$, by 3, 9, 14, and 15;

17. $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

Fifth case: $e_l = (\mathtt{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$. There are many sub-cases: the evaluation of the test leads to an error, to a true value, or to a false value. The last two sub-cases can be further subdivided depending on whether the evaluation of the branch that is taken leads to an error on not. As in previous cases, we skip the sub-case where the test evaluates to an error.

We consider the sub-cases where the test evaluates to a true value. In the reasoning, we take care of the situations where the *then*-branch $e_{l_2}$ evaluates or not to an error. We have that:

1. let $(v_1, \Xi_1) = \mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ where $v_1 \in \mathrm{ValC} \cup \mathrm{ValP}$;

2. let $(v_2,\, \Xi_2) = \mathsf{E}\, \llbracket e_{l_2} \rrbracket\, \rho\, k$;

3. $\mathsf{E}\, \llbracket e_l \rrbracket\, \rho\, k = (v_2,\, \Xi)$ where $\Xi = \{\mathrm{pre}(l,\, k,\, \rho), \mathrm{post}(l,\, k,\, v_2)\} \cup \Xi_1 \cup \Xi_2$;

4. the computation of both $\mathsf{E}\, \llbracket e_{l_1} \rrbracket\, \rho\, k$ and $\mathsf{E}\, \llbracket e_{l_2} \rrbracket\, \rho\, k$ is done in less than $n$ uses of $\mathsf{E}$;

5. $\delta_{l_1,\hat{k}} \neq \emptyset$ (note that we cannot say the same thing about $\delta_{l_2,\hat{k}}$ yet);

6. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ and $\rho \nearrow \hat{\rho}(l_2, \hat{k})$;

7. $\Xi_1 \nearrow_{a_1} \mathcal{R}$ where $a_1(k) = \hat{k}$, by 4, 5, 6, and the induction hypothesis;

8. $\mathsf{E}\, \llbracket e_{l_1} \rrbracket\, \rho\, k = (v_1,\, \Xi_1) \Rightarrow \mathrm{post}(l_1,\, k,\, v_1) \in \Xi_1 \Rightarrow \exists \hat{v}_1' \in \alpha_{l_1,\hat{k}}.\ v_1 \nearrow \hat{v}_1'$, since $v_1 \notin \mathrm{Err}$; let $\hat{v}_1$ be this value;

9. $\hat{v}_1 \in \delta_{l_2,\hat{k}}$, because of 8 which implies that $\hat{v}_1 \in \alpha_{l_1,\hat{k}} \cap (\mathcal{V}al\mathcal{C} \cup \mathcal{V}al\mathcal{P})$;

10. $\Xi_2 \nearrow_{a_2} \mathcal{R}$ where $a_2(k) = \hat{k}$, by 4, 6, 9, and the induction hypothesis;

11. let $a = a_1 a_2$; note that $a(k) = \hat{k}$;

12. $\Xi_1 \nearrow_a \mathcal{R}$ and $\Xi_2 \nearrow_a \mathcal{R}$;

13. $\forall \mathrm{pre}(l',\, k',\, \rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$;

14. $\mathsf{E}\, \llbracket e_{l_2} \rrbracket\, \rho\, k = (v_2,\, \Xi_2) \Rightarrow \mathrm{post}(l_2,\, k,\, v_2) \in \Xi_2 \Rightarrow (\exists \hat{v}_2' \in \alpha_{l_2,a(k)}.\ v_2 \nearrow \hat{v}_2')\ \vee\ v_2 \in \mathrm{Err}$ $\Rightarrow (\exists \hat{v}' \in \alpha_{l,a(k)}.\ v_2 \nearrow \hat{v}')\ \vee\ v_2 \in \mathrm{Err}$, because of 12 and evaluation constraints;

15. $\forall \mathrm{post}(l',\, k',\, v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$;

16. $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

These are the modified steps in the reasoning for the sub-cases where the test evaluates to false:

1. let $(v_1,\, \Xi_1) = \mathsf{E}\, \llbracket e_{l_1} \rrbracket\, \rho\, k$ where $v_1 \in \mathrm{ValB}$;

2. let $(v_3,\, \Xi_3) = \mathsf{E}\, \llbracket e_{l_3} \rrbracket\, \rho\, k$;

3. $\mathsf{E}\, \llbracket e_l \rrbracket\, \rho\, k = (v_3,\, \Xi)$ where $\Xi = \{\mathrm{pre}(l,\, k,\, \rho), \mathrm{post}(l,\, k,\, v_3)\} \cup \Xi_1 \cup \Xi_3$;

4. the computation of both $\mathsf{E}\, \llbracket e_{l_1} \rrbracket\, \rho\, k$ and $\mathsf{E}\, \llbracket e_{l_3} \rrbracket\, \rho\, k$ is done in less than $n$ uses of $\mathsf{E}$;

6. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ and $\rho \nearrow \hat{\rho}(l_3, \hat{k})$;

9. $\hat{v}_1 \in \delta_{l_3,\hat{k}}$, because of 8 which implies that $\hat{v}_1 \in \alpha_{l_1,\hat{k}} \cap \mathcal{V}al\mathcal{B}$;

10. $\Xi_3 \nearrow_{a_3} \mathcal{R}$ where $a_3(k) = \hat{k}$, by 4, 6, 9, and the induction hypothesis;

11. let $a = a_1 a_3$; note that $a(k) = \hat{k}$;

12. $\Xi_1 \nearrow_a \mathcal{R}$ and $\Xi_3 \nearrow_a \mathcal{R}$;

14. $\mathsf{E}\ [\![e_{l_3}]\!]\ \rho\ k = (v_3,\, \Xi_3) \Rightarrow \text{post}(l_3,\, k,\, v_3) \in \Xi_3 \Rightarrow (\exists \hat{v}'_3 \in \alpha_{l_3,a(k)}.\ v_3 \nearrow \hat{v}'_3)\ \vee\ v_3 \in \text{Err}$ $\Rightarrow (\exists \hat{v}' \in \alpha_{l,a(k)}.\ v_3 \nearrow \hat{v}')\ \vee\ v_3 \in \text{Err}$, because of 12 and evaluation constraints;

Last case: $e_l = ({}_l e_{l_1}\ e_{l_2})$. There are numerous sub-cases: one of the sub-expressions evaluates to an error; the first one evaluates to a non-closure; a closure is invoked and its body is evaluated, leading or not to an error. The first kind of sub-cases is similar to sub-cases present in all previous cases. We skip them. The second kind of sub-case is similar to the one involving a `cdr`-expression and a non-pair. We skip it too. We only consider the last kind of sub-cases. Here is the reasoning:

1. let $(v_1,\, \Xi_1) = \mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$ where $v_1 = \text{clos}((\lambda_{l_4}\text{x}.\ e_{l_3}),\, \rho')$;

2. let $(v_2,\, \Xi_2) = \mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k$ where $v_2 \in \text{Val}$;

3. let $(v_3,\, \Xi_3) = \mathsf{E}\ [\![e_{l_3}]\!]\ \rho'[\text{x} \mapsto v_2]\ kl$;

4. $\mathsf{E}\ [\![e_l]\!]\ \rho\ k = (v_3,\, \Xi)$ where $\Xi = \{\text{pre}(l,\, k,\, \rho), \text{post}(l,\, k,\, v_3)\} \cup \Xi_1 \cup \Xi_2 \cup \Xi_3$;

5. the computation of each of $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k$, $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k$, and $\mathsf{E}\ [\![e_{l_3}]\!]\ \rho'[\text{x} \mapsto v_2]\ kl$ requires less than $n$ uses of $\mathsf{E}$;

6. $\delta_{l_1,\hat{k}} \neq \emptyset$ and $\delta_{l_2,\hat{k}} \neq \emptyset$;

7. $\rho \nearrow \hat{\rho}(l_1, \hat{k})$ and $\rho \nearrow \hat{\rho}(l_2, \hat{k})$;

8. $\Xi_1 \nearrow_{a_1} \mathcal{R}$ and $\Xi_2 \nearrow_{a_2} \mathcal{R}$, where $a_1(k) = \hat{k}$ and $a_2(k) = \hat{k}$, by 5, 6, 7, and the induction hypothesis;

9. $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ k = (v_1,\, \Xi_1) \Rightarrow \text{post}(l_1,\, k,\, v_1) \in \Xi_1 \Rightarrow \exists \hat{v}'_1 \in \alpha_{l_1,\hat{k}}.\ v_1 \nearrow \hat{v}'_1$ as $v_1 \notin \text{Err}$; let $\hat{c} \in \alpha_{l_1,\hat{k}} \cap \mathcal{V}al\mathcal{C}$ be that closure;

10. $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ k = (v_2,\, \Xi_2) \Rightarrow \text{post}(l_2,\, k,\, v_2) \in \Xi_2 \Rightarrow \exists \hat{v}'_2 \in \alpha_{l_2,\hat{k}}.\ v_2 \nearrow \hat{v}'_2$ as $v_2 \notin \text{Err}$; let $\hat{v}_2 \in \alpha_{l_2,\hat{k}}$ be that value;

11. let $\hat{k}' = \mathsf{call}(l,\ \hat{c},\ \hat{v}_2,\ \hat{k})$;

12. $\exists \hat{k}'' \in \mathcal{C}ont.\ (l_4,\ \hat{k}'') \in \chi_{\hat{c}}$ and $\rho' \nearrow \hat{\rho}(l_4, \hat{k}'')$, because $v_1 \nearrow \hat{c}$;

13. $\hat{v}_2 \in \beta_{\mathrm{x},\hat{k}'}$, by 9, 10, 11, 12, and the evaluation constraints;

14. $\delta_{l_3,\hat{k}'} \neq \emptyset$, because of 13 and the evaluation constraints;

15. $v_2 \nearrow \hat{v}_2 \in \beta_{\mathrm{x},\hat{k}'} = \mathrm{ref}(\mathrm{x}, l_3, \hat{k}') = \hat{\rho}(l_3, \hat{k}')(\mathrm{x}) \Rightarrow \exists \hat{v}' \in \hat{\rho}(l_3, \hat{k}').\ v_2 \nearrow \hat{v}' \Rightarrow \exists \hat{v}' \in \hat{\rho}(l_3, \hat{k}').\ (\rho'[\mathrm{x} \mapsto v_2])\ \mathrm{x} \nearrow \hat{v}'$;

16. for any $\mathrm{y} \in \mathrm{Dom}(\rho')$, $\exists \hat{v}' \in \hat{\rho}(l_4, \hat{k}'').\ \rho'\ \mathrm{y} \nearrow \hat{v}'$, by 12;

17. for any $\mathrm{y} \in \mathrm{Dom}(\rho')$, $\exists \hat{v}' \in \hat{\rho}(l_3, \hat{k}').\ \rho'\ \mathrm{y} \nearrow \hat{v}'$, by 16 and the fact that $\hat{\rho}(l_3, \hat{k}')(\mathrm{y}) \subseteq \hat{\rho}(l_4, \hat{k}'')(\mathrm{y})$ (see the evaluation constraints);

18. for any $\mathrm{y} \in \mathrm{Dom}(\rho')$, $\exists \hat{v}' \in \hat{\rho}(l_3, \hat{k}').\ (\rho'[\mathrm{x} \mapsto v_2])\ \mathrm{y} \nearrow \hat{v}'$, by 17;

19. $\rho'[\mathrm{x} \mapsto v_2] \nearrow \hat{\rho}(l_3, \hat{k}')$, by 15 and 18;

20. $\Xi_3 \nearrow_{a_3} \mathcal{R}$ where $a_3(kl) = \hat{k}'$, by 5, 14, 18, the induction hypothesis;

21. let $a = a_1 a_2 a_3$; there is no conflict as $\mathrm{Dom}(a_1) \cap \mathrm{Dom}(a_2) = \{k\}$, $a_1(k) = a_2(k) = \hat{k}$, and $\mathrm{Dom}(a_3) \cap (\mathrm{Dom}(a_1) \cup \mathrm{Dom}(a_2)) = \emptyset$;

22. $\Xi_1 \nearrow_a \mathcal{R}$, $\Xi_2 \nearrow_a \mathcal{R}$, and $\Xi_3 \nearrow_a \mathcal{R}$;

23. since $\mathrm{post}(l_3,\ kl,\ v_3) \in \Xi_3$, we have that if $v_3 \notin \mathrm{Err}$, then $\hat{v}_3 \in \alpha_{l_3,\hat{k}'}$ such that $v_3 \nearrow \hat{v}_3$, then $\hat{v}_3 \in \gamma_{\hat{c},\hat{k}'} \subseteq \alpha_{l,\hat{k}}$;

24. $\forall \mathrm{pre}(l',\ k',\ \rho') \in \Xi.\ \delta_{l',a(k')} \neq \emptyset\ \wedge\ \rho' \nearrow \hat{\rho}(l', a(k'))$, by 4 and 22;

25. $\forall \mathrm{post}(l',\ k',\ v') \in \Xi.\ (\exists \hat{v}' \in \alpha_{l',a(k')}.\ v' \nearrow \hat{v}')\ \vee\ v' \in \mathrm{Err}$, by 4, 22, and 23;

26. $\Xi \nearrow_a \mathcal{R}$ where $a(k) = \hat{k}$.

This completes the case $e_l = ({}_l e_{l_1}\ e_{l_2})$, the proof that $P(n)$ is satisfied, and the whole proof of Theorem 3.5. □

### 3.5.3 Conservativeness Regarding Dynamic Type Tests

The central property, Theorem 3.5, shows that an analyser instance produced by the framework mimics conservatively parts of the concrete evaluation, provided that certain conditions are met. The following theorem uses this property to show that an optimiser can rely on the analysis results produced by the analyser.

**Theorem 3.6** *Let* $e_{l_0} \in \mathrm{Exp}$ *be a program. Let* $(v_0, \Xi_0) = \mathsf{E}\ [\![e_{l_0}]\!]\ \cdot\ \epsilon$ *be the concrete evaluation result. Let* $\mathcal{M} = (\mathcal{V}al\mathcal{B}, \mathcal{V}al\mathcal{C}, \mathcal{V}al\mathcal{P}, \mathcal{C}ont, \hat{k}_0, \mathsf{cc}, \mathsf{pc}, \mathsf{call})$ *be the abstract model. Let* $\mathcal{R} = \mathrm{FW}(e_{l_0}, \mathcal{M})$ *be the analysis results. Then we have that:*

$$\begin{array}{l}
\exists \mathrm{post}(l,\, k,\, v) \in \Xi_0 \,\wedge\, (\mathtt{car}_{l'}\ e_l) \in \triangle(e_{l_0}) \,\wedge\, v \in \mathrm{ValB} \cup \mathrm{ValC} \Rightarrow \exists \hat{k} \in \mathcal{C}ont.\ \alpha_{l,\hat{k}} \not\subseteq \mathcal{V}al\mathcal{P} \\
\exists \mathrm{post}(l,\, k,\, v) \in \Xi_0 \,\wedge\, (\mathtt{cdr}_{l'}\ e_l) \in \triangle(e_{l_0}) \,\wedge\, v \in \mathrm{ValB} \cup \mathrm{ValC} \Rightarrow \exists \hat{k} \in \mathcal{C}ont.\ \alpha_{l,\hat{k}} \not\subseteq \mathcal{V}al\mathcal{P} \\
\exists \mathrm{post}(l,\, k,\, v) \in \Xi_0 \,\wedge\, ({}_{l'}e_l\ e_{l''}) \in \triangle(e_{l_0}) \,\wedge\, v \in \mathrm{ValB} \cup \mathrm{ValP} \Rightarrow \exists \hat{k} \in \mathcal{C}ont.\ \alpha_{l,\hat{k}} \not\subseteq \mathcal{V}al\mathcal{C}
\end{array}$$

Essentially, it means that if $v \in \mathrm{Err}$ and we confront $\mathcal{R}$ to the safety constraints, then at least one of the safety constraints has to be violated. More accurately, if it is expression $e_l$ that evaluates to an illegal value, then there is a safety constraint concerning $e_l$ that gets violated.

**Proof 3.6** First, observe that in order to make the concrete evaluation to produce an error, one the following three situations must occur: the sub-expression of a `car`- or a `cdr`-expression returns a non-pair, the first sub-expression of a call expression returns a non-closure. Formally, we have that:

$$\begin{array}{rlll}
\exists \mathrm{post}(l,\, k,\, v) \in \Xi_0. & (\mathtt{car}_{l'}\ e_l) \in \triangle(e_{l_0}) & \wedge & v \in \mathrm{ValB} \cup \mathrm{ValC} \quad \vee \\
& (\mathtt{cdr}_{l'}\ e_l) \in \triangle(e_{l_0}) & \wedge & v \in \mathrm{ValB} \cup \mathrm{ValC} \quad \vee \\
& ({}_{l'}e_l\ e_{l''}) \in \triangle(e_{l_0}) & \wedge & v \in \mathrm{ValB} \cup \mathrm{ValP}
\end{array}$$

Second, using Theorem 3.5 it is easy to show that $\mathcal{R}$ abstracts the whole concrete evaluation:

1. $\mathsf{E}\ [\![e_{l_0}]\!]\ \cdot\ \epsilon = (v_0, \Xi_0)$;

2. $\delta_{l_0,\hat{k}_0} \neq \emptyset$, by the evaluation constraints;

3. $\cdot \nearrow \hat{\rho}(l_0, \hat{k}_0)$, that is, the empty environment is abstracted by $\hat{\rho}(l_0, \hat{k}_0)$; this is immediate since $\cdot$ is not defined on any variable;

4. then $\Xi_0 \nearrow_a \mathcal{R}$ where $a(\epsilon) = \hat{k}_0$, by 1, 2, 3, and Theorem 3.5.

Finally, we use this last result to obtain the desired property. In the case where $(\mathtt{car}_{l'}\ e_l) \in \triangle(e_{l_0})$, we have that:

$$\begin{array}{l} \mathrm{post}(l,\, k,\, v) \in \Xi_0 \wedge v \in \mathrm{ValB} \cup \mathrm{ValC} \\ \Rightarrow \exists \hat{v} \in \alpha_{l,a(k)}.\ v \nearrow \hat{v} \quad (\text{since } v \notin \mathrm{Err}) \\ \Rightarrow \exists \hat{v} \in \alpha_{l,a(k)} \cap (\mathcal{V}al\mathcal{B} \cup \mathcal{V}al\mathcal{C}) \quad (\text{by def. of } \nearrow) \\ \Rightarrow \alpha_{l,a(k)} \not\subseteq \mathcal{V}al\mathcal{P} \end{array}$$

Similarly in the other two cases. □

## 3.6 Theoretical Power and Limitations of the Analysis Framework

Because of its great flexibility, our analysis framework is a very powerful tool. In this section, we show that any program that terminates without error can be analysed perfectly well using the framework. What this means is that there exists an abstract model that, when it is used to instantiate an analysis for the program, provides the demonstration that all dynamic type tests can be removed. In the preceding section, we already demonstrated that any program that terminates with an error cannot be analysed perfectly well. That is, for any abstract model, the analysis results that we obtain using it show that at least one type test has to be left in the compiled program. As for the non-terminating programs, there is no general result. Some can be analysed perfectly well and some cannot. This is particularly interesting since non-terminating programs *do not* run into an error (otherwise they would terminate).

Additionally, we give the answer to another question. Since any error-free terminating program can be analysed perfectly well and some non-terminating ones can, too, it would be interesting to be able to find an appropriate model each time it exists. So a natural question is: Is it possible to systematically decide whether there exists an abstract model $\mathcal{M}$ that, when used to analyse a program, provides analysis results that respect all safety constraints? Section 3.6.2 presents a demonstration that the problem is (unfortunately) undecidable. A by-product of this demonstration is the provision of evidence that some non-terminating programs cannot be analysed perfectly well.

### 3.6.1 Programs Terminating Without Error

Programs that terminate without error can be analysed perfectly well. This result is pretty easy to obtain since: a terminating program evaluates completely in a finite number of steps; so it manipulates a finite number of values and evaluation occurs in a finite number of contours; so we simply have to create an abstract model that contains precisely these values and contours and in which $\mathsf{cc}$, $\mathsf{pc}$, and $\mathsf{call}$ behave like in the concrete evaluation.

**Theorem 3.7** *Let* $e_{l_0} \in \mathrm{Exp}$ *be a program. Let* $(v_0, \Xi_0) = \mathsf{E}\ [\![e_{l_0}]\!] \cdot \epsilon$*. Let us suppose that* $v_0 \notin \mathrm{Err}$*. Then there exists an abstract model* $\mathcal{M}$ *such that the analysis results* $\mathcal{R} = \mathrm{FW}(e_{l_0}, \mathcal{M})$ *satisfy all the safety constraints.*

**Proof 3.7** We build the abstract model this way:

$$\begin{array}{lcl}
\mathcal{M} = (\mathcal{V}al\mathcal{B},\ \mathcal{V}al\mathcal{C},\ \mathcal{V}al\mathcal{P},\ \mathcal{C}ont,\ \hat{k}_0,\ \mathsf{cc},\ \mathsf{pc},\ \mathsf{call}) \quad \text{where} & & \\
\mathcal{V}al\mathcal{B} & = & \mathrm{ValB} \\
\mathcal{V}al\mathcal{C} & = & \{\bot_{\mathcal{C}}\} \cup \{v \in \mathrm{ValC} \mid \exists l \in \mathrm{Lab}.\ \exists k \in \mathrm{Cont}.\ \mathrm{post}(l,\ k,\ v) \in \Xi_0\} \\
\mathcal{V}al\mathcal{P} & = & \{\bot_{\mathcal{P}}\} \cup \{v \in \mathrm{ValP} \mid \exists l \in \mathrm{Lab}.\ \exists k \in \mathrm{Cont}.\ \mathrm{post}(l,\ k,\ v) \in \Xi_0\} \\
\mathcal{C}ont & = & \{\bot\} \cup \{k \in \mathrm{Cont} \mid \exists l \in \mathrm{Lab}.\ \exists \rho \in \mathrm{Env}.\ \mathrm{pre}(l,\ k,\ \rho) \in \Xi_0\} \\
\hat{k}_0 & = & \epsilon \\
\mathsf{cc}(l, \hat{k}) & = & \begin{cases} c, & \text{if } e_l = (\lambda_l \mathrm{x}.\ e_{l_1}) \wedge \hat{k} \neq \bot \wedge \mathrm{post}(l,\ \hat{k},\ c) \in \Xi_0 \\ \bot_{\mathcal{C}}, & \text{otherwise} \end{cases} \\
\mathsf{pc}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & \begin{cases} p, & \text{if } \ e_l = (\mathtt{cons}_l\ e_{l_1}\ e_{l_2}) \wedge \hat{v}_1 \in \mathrm{Val} \wedge \hat{v}_2 \in \mathrm{Val} \wedge \hat{k} \neq \bot \\ & \qquad \wedge\, \mathrm{post}(l_1,\ \hat{k},\ \hat{v}_1) \in \Xi_0 \wedge \mathrm{post}(l_2,\ \hat{k},\ \hat{v}_2) \in \Xi_0 \\ & \qquad \wedge\, \mathrm{post}(l,\ \hat{k},\ p) \in \Xi_0 \\ \bot_{\mathcal{P}}, & \text{otherwise} \end{cases} \\
\mathsf{call}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & \begin{cases} \hat{k}l, & \text{if } \ e_l = ({}_l e_{l_1}\ e_{l_2}) \wedge \hat{v}_1 \in \mathrm{Val} \wedge \hat{v}_2 \in \mathrm{Val} \wedge \hat{k} \neq \bot \\ & \qquad \wedge\, \mathrm{post}(l_1,\ \hat{k},\ \hat{v}_1) \in \Xi_0 \wedge \mathrm{post}(l_2,\ \hat{k},\ \hat{v}_2) \in \Xi_0 \\ \bot & \text{otherwise} \end{cases}
\end{array}$$

There is a dummy closure $\perp_{\mathcal{C}}$ and a dummy pair $\perp_{\mathcal{P}}$ added to the values manipulated by the program. And there is a dummy contour $\perp$ added to the set of contours manipulated by the program. They are introduced to allow the abstract creation functions $\mathsf{cc}$, $\mathsf{pc}$, and $\mathsf{call}$ to be defined on their entire domain. Basically, $\mathsf{cc}$, $\mathsf{pc}$, and $\mathsf{call}$ do exactly the same computations as those done in the concrete evaluation. However, for any tuple of arguments that do not correspond to a situation found during the concrete evaluation, they return a dummy answer. We will see later that the dummies do not diminish the power of the model as a smallest fixed-point solution to the evaluation constraints does not include dummies. Note that $\mathsf{cc}$ and $\mathsf{pc}$ are well-defined despite the fact that $c$ and $p$, respectively, are extracted from the cache. This is a consequence of Theorem 3.3, which guarantees that the post-entry from which $c$ or $p$ is extracted is unique.

Some justification has to be given in order to ensure that the model is legal. First, it is easy to verify that $\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, $\mathcal{V}al\mathcal{P}$, and $\mathcal{C}ont$ are finite sets and that $\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, and $\mathcal{V}al\mathcal{P}$ are disjoint. Next, $\hat{k}_0$ is clearly a member of $\mathcal{C}ont$ as the program has been evaluated in contour $\epsilon$. $\mathsf{cc}$, $\mathsf{pc}$, and $\mathsf{call}$ are defined on their entire domain. $\mathsf{cc}$ either returns $\perp_{\mathcal{C}}$ or the value extracted from a post-entry. Since the post-entry contains a value resulting from the evaluation of a $\lambda$-expression, it is clear that the value is an element of ValC. So we can conclude that the return value of $\mathsf{cc}$ is always in $\mathcal{V}al\mathcal{C}$. A similar reasoning applies to $\mathsf{pc}$. There remains to verify that $\mathsf{call}$'s return value always lie in $\mathcal{C}ont$.

In the case where $\mathsf{call}$ returns $\perp$, the verification is immediate. In the case where $\mathsf{call}$ returns a contour of the form $\hat{k}l$, we have to show that the conditions checked by $\mathsf{call}$ are sufficient to imply that $\hat{k}l \in \mathcal{C}ont$. The reasoning is the following:

1. $\hat{k} \neq \perp$
   $\Rightarrow \exists l' \in \text{Lab.}\ \exists \rho \in \text{Env.}\ \text{pre}(l', \hat{k}, \rho) \in \Xi_0$;

2. $e_l = ({}_l e_{l_1}\ e_{l_2})$, $\text{post}(l_1, \hat{k}, \hat{v}_1) \in \Xi_0$, $\text{post}(l_2, \hat{k}, \hat{v}_2) \in \Xi_0$
   $\Rightarrow$ the computation of $\mathsf{E}\ [\![e_l]\!]\ \rho\ \hat{k}$ is required in the global computation $\mathsf{E}\ [\![e_{l_0}]\!]\ \cdot\ \epsilon$;

3. computation of $\mathsf{E}\ [\![e_l]\!]\ \rho\ \hat{k}$ is required
   $\Rightarrow$ computation of $\mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ \hat{k}$ is required
   $\Rightarrow \mathsf{E}\ [\![e_{l_1}]\!]\ \rho\ \hat{k} = (\hat{v}_1, \Xi_1)$ for some cache $\Xi_1 \subseteq \Xi_0$;

4. computation of $\mathsf{E}\ [\![e_l]\!]\ \rho\ \hat{k}$ is required and $\hat{v}_1 \in \text{Val}$
   $\Rightarrow$ computation of $\mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ \hat{k}$ is required

$\Rightarrow \mathsf{E}\ [\![e_{l_2}]\!]\ \rho\ \hat{k} = (\hat{v}_2,\ \Xi_2)$ for some cache $\Xi_2 \subseteq \Xi_0$;

5. $v_0 \in \text{Val} \Rightarrow \hat{v}_1 \in \text{ValC}$ (otherwise there would have been an error and it would contradict the theorem hypothesis);

6. $\hat{v}_1 \in \text{ValC}$ and $\hat{v}_2 \in \text{Val}$
   $\Rightarrow$ computation of $\mathsf{E}\ [\![e_{l_3}]\!]\ \rho'[\text{y} \mapsto \hat{v}_2]\ \hat{k}l$ is required where $\hat{v}_1 = \text{clos}((\lambda\text{y}.\ e_{l_3}),\ \rho')$
   $\Rightarrow \text{pre}(l_3,\ \hat{k}l,\ \rho'[\text{y} \mapsto \hat{v}_2]) \in \Xi_0$
   $\Rightarrow \hat{k}l \in \mathcal{C}ont$

Valid model $\mathcal{M} = (\mathcal{V}al\mathcal{B},\ \mathcal{V}al\mathcal{C},\ \mathcal{V}al\mathcal{P},\ \mathcal{C}ont,\ \hat{k}_0,\ \mathsf{cc},\ \mathsf{pc},\ \mathsf{call})$ allows program $e_{l_0}$ to be analysed perfectly well. To justify this claim, we present an assignment to the abstract variables that is a solution to the evaluation constraints and that also respects the safety constraints. Here is the assignment:

$$\alpha_{l,\hat{k}} = \begin{cases} \{v\}, & \text{if } \hat{k} \neq \bot \wedge \text{post}(l,\ \hat{k},\ v) \in \Xi_0 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\beta_{\text{x},\hat{k}} = \begin{cases} \{v\}, & \text{if } \ \hat{k} \neq \bot \wedge \hat{k} = \hat{k}'l \wedge e_l = ({}_l e_{l_1}\ e_{l_2}) \wedge \\ & \qquad \text{post}(l_1,\ \hat{k}',\ \text{clos}((\lambda_{l_3}\text{x}.\ e_{l_4}),\ \rho)) \in \Xi_0 \wedge \text{post}(l_2,\ \hat{k}',\ v) \in \Xi_0 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\gamma_{\hat{c},\hat{k}} = \begin{cases} \{v\}, & \text{if } \ \hat{c} \neq \bot_{\mathcal{C}} \wedge \hat{k} \neq \bot \wedge \hat{c} = \text{clos}((\lambda_l\text{x}.\ e_{l_1}),\ \rho) \wedge \\ & \qquad \text{post}(l_1,\ \hat{k},\ v) \in \Xi_0 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\delta_{l,\hat{k}} = \left[ \begin{array}{l} \text{if } \hat{k} = \bot \text{ then} \\ \qquad \emptyset \\ \text{else if pre}(l,\ \hat{k},\ \rho) \in \Xi_0 \text{ then} \\ \qquad \left[ \begin{array}{l} \text{if } e_l = e_{l_0} \text{ then} \\ \qquad \{\#\text{f}\} \\ \text{else let } l_1 = \text{parent}(l);\ \text{if } e_{l_1} = (\lambda_{l_1}\text{x}.\ e_l) \text{ then} \\ \qquad \beta_{\text{x},\hat{k}} \\ \text{else if } e_{l_1} = (\texttt{if}_{l_1}\ e_{l_2}\ e_l\ e_{l_3}) \text{ then} \\ \qquad \alpha_{l_2,\hat{k}} \\ \text{else if } e_{l_1} = (\texttt{if}_{l_1}\ e_{l_2}\ e_{l_3}\ e_l) \text{ then} \\ \qquad \alpha_{l_2,\hat{k}} \\ \text{else} \\ \qquad \delta_{l_1,\hat{k}} \end{array} \right. \\ \text{else} \\ \qquad \emptyset \end{array} \right.$$

$$\chi_{\hat{c}} \quad = \quad \begin{cases} \{(l,\, k) \mid e_l = (\lambda_l \mathrm{x}.\ e_{l_1}) \wedge k \in \mathrm{Cont} \wedge \mathrm{post}(l,\, k,\, \hat{c}) \in \Xi_0\}, & \text{if } \hat{c} \neq \bot_{\mathcal{C}} \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\pi_{\hat{p}} \quad = \quad \begin{cases} \left\{ (l,\, v_1,\, v_2,\, k) \;\middle|\; \begin{array}{l} e_l = (\mathtt{cons}_l\ e_{l_1}\ e_{l_2}) \wedge k \in \mathrm{Cont} \wedge \\ \mathrm{post}(l,\, k,\, \hat{p}) \in \Xi_0 \wedge \hat{p} = \mathrm{pair}(v_1,\, v_2) \end{array} \right\}, & \text{if } \hat{p} \neq \bot_{\mathcal{P}} \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\kappa_{\hat{k}} \quad = \quad \begin{cases} \left\{ (l,\, v_1,\, v_2,\, k) \;\middle|\; \begin{array}{l} e_l = ({}_l e_{l_1}\ e_{l_2}) \wedge \mathrm{post}(l_1,\, k,\, v_1) \in \Xi_0 \wedge \\ \mathrm{post}(l_2,\, k,\, v_2) \in \Xi_0 \wedge \hat{k} = kl \end{array} \right\}, & \text{if } \hat{k} \neq \bot \\ \emptyset, & \text{otherwise} \end{cases}$$

Clearly, matrices $\alpha$, $\beta$, $\gamma$, $\chi$, $\pi$, and $\kappa$ are well-defined in terms of $\Xi_0$ and because of the fact that post-entries $\mathrm{post}(l,\, k,\, \_)$ are unique (Theorem 3.3). The $\delta$ matrix is well-defined, too, because it is mostly defined in terms of $\Xi_0$ and the other matrices. The only recursive references to $\delta$ itself are non-cyclic, since we can see the definition of an entry $\delta_{l,\hat{k}}$ as being computed as $f(\Xi_0, \alpha, \beta, \delta_{\mathrm{parent}(l),\hat{k}})$ for some function $f$. Note that $\forall \hat{k} \in \mathit{Cont}.|\kappa_{\hat{k}}| \leq 1$. Note also that no abstract variable contains any of the dummies $\bot_{\mathcal{C}}$, $\bot_{\mathcal{P}}$, and $\bot$.

Now, we have to verify that this assignment to the abstract variables respects all the evaluation constraints. We omit a complete verification as it would be too lengthy and it would be almost completely mechanical. The only point that is more difficult consists in verifying that the constraints related to a variable reference are respected. That is, we verify that, for any variable reference $e_l = \mathrm{x}_l$, $\delta_{l,\hat{k}} \neq \emptyset \Rightarrow \alpha_{l,\hat{k}} \supseteq \mathrm{ref}(\mathrm{x}, l, k)$.

We show by induction on the depth of label $l$ in the syntax tree that if $\mathrm{pre}(l,\, k,\, \rho) \in \Xi_0$ and $\rho\ \mathrm{x}$ is defined, then $\mathrm{ref}(\mathrm{x}, l, k) = \{\rho\ \mathrm{x}\}$.

*Basis.* Label $l$ is at depth $0 \Rightarrow l = l_0 \Rightarrow \mathrm{Dom}(\rho) = \emptyset$.

*Induction hypothesis.* Let us suppose that the desired property is respected for any label $l$ of depth at most $d$.

*Induction step.* Let $\mathrm{pre}(l,\, k,\, \rho) \in \Xi_0$ where $l$ is at depth $d + 1$. Let $l_1 = \mathrm{parent}(l)$. There are two cases. First case:

1. $e_{l_1}$ is not a $\lambda$-expression
2. $\Rightarrow \mathrm{pre}(l_1,\, k,\, \rho) \in \Xi_0$.
3. Suppose that $\rho\ \mathrm{x}$ is defined.

4. $\Rightarrow \mathrm{ref}(\mathrm{x}, l_1, k) = \{\rho\ \mathrm{x}\}$ by induction hypothesis

5. $\Rightarrow \mathrm{ref}(\mathrm{x}, l, k) = \{\rho\ \mathrm{x}\}$ since $\mathrm{ref}(\mathrm{x}, l, k) = \mathrm{ref}(\mathrm{x}, l_1, k)$.

Second case:

1. $e_{l_1} = (\lambda_{l_1}\mathrm{x}.\ e_l)$
2. $\Rightarrow \exists l_2 \in \mathrm{Lab}.$
   $e_{l_2} = ({}_{l_2}e_{l_3}\ e_{l_4}) \wedge k = k' l_2 \wedge \mathrm{post}(l_3,\, k',\, \mathrm{clos}((\lambda_{l_1}\mathrm{x}.\ e_l),\, \rho')) \in \Xi_0 \wedge \mathrm{post}(l_4,\, k',\, v) \in \Xi_0$
   $\wedge\ \rho = \rho'[\mathrm{x} \mapsto v]$
3. There are two sub-cases:
4. first sub-case:
   (a) $\rho$ x is defined
   (b) $\Rightarrow \mathrm{ref}(\mathrm{x}, l, k) = \beta_{\mathrm{x},k} \overset{(*)}{=} \{v\} = \{\rho\ \mathrm{x}\}$ $(*)$ because of the assignment to $\beta$ variables
5. second sub-case:
   (a) Suppose $\rho$ y is defined
   (b) $\Rightarrow \mathrm{ref}(\mathrm{y}, l, k) = \bigcup_{k'' \in K} \mathrm{ref}(\mathrm{y}, l_1, k)$ where
   $K = \{k'' \in \mathrm{Cont} \mid (l_2,\, c,\, v,\, k') \in \kappa_k \wedge (l_1,\, k'') \in \chi_c\}$
   $= \{k'' \in \mathrm{Cont} \mid (l_1,\, k'') \in \chi_{\mathrm{clos}((\lambda_{l_1}\mathrm{x}.\ e_l),\, \rho')}\}$
   (since $\kappa_k = \{(l_2,\, \mathrm{clos}((\lambda_{l_1}\mathrm{x}.\ e_l),\, \rho'),\, v,\, k')\}$)
   $= \{k'' \in \mathrm{Cont} \mid \mathrm{post}(l_1,\, k'',\, \mathrm{clos}((\lambda_{l_1}\mathrm{x}.\ e_l),\, \rho')) \in \Xi_0\}$
   $= \{k'' \in \mathrm{Cont} \mid \mathrm{pre}(l_1,\, k'',\, \rho') \in \Xi_0\}$
   (c) $\Rightarrow \forall k'' \in K.\ \mathrm{pre}(l_1,\, k'',\, \rho') \in \Xi_0$
   (d) $\Rightarrow \forall k'' \in K.\ \mathrm{ref}(\mathrm{y},\, l_1, k'') = \{\rho'\ \mathrm{y}\}$
   (e) $\Rightarrow \mathrm{ref}(\mathrm{y}, l, k) = \{\rho'\ \mathrm{y}\} = \{\rho\ \mathrm{y}\}$

Now that we know that all the evaluation constraints are respected, there remains to do the same with the safety constraints. That would be easy to verify since, by construction of the assignment, the violation of a safety constraint directly imply that the concrete evaluation should have led to an error.

So this concludes the proof, as we gave a model, with which analysis results for the program were obtained, and these results respect both the evaluation and the safety constraints. So the program could be analysed perfectly well.

□

### 3.6.2 Undecidability of the "Perfectly Analysable" Property

We demonstrate here that it is undecidable to determine whether there exists an abstract model that allows a program to be analysed perfectly well. In order to do so, we make a reduction from the Universal Language for the Turing machines to our problem. So, before we state the theorem and give the proof, we introduce the definition of a Turing machine, its behaviour on an input, and the Universal Language.

Our model of Turing machine has a tape that is infinite in both directions. It has a *success* state and a *failure* state. Execution can only stop because the machine has entered one of these special states. It cannot stop because of any kind of illegal operation like, for example, letting the read/write head fall past the end of the tape (in the case of a machine with a semi-infinite tape). The computation may last forever and the execution may not stop.

Formally, a Turing machine $M$ is a tuple $(Q, \Gamma, \Sigma, \delta, \#, q_0, q_{\mathrm{s}}, q_{\mathrm{f}})$ where:

- $Q$ is a (finite) set of states;
- $\Gamma$ is the alphabet of the tape;
- $\Sigma \subset \Gamma$ is the input alphabet;
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the function of transition, where L and R are directions; it is defined for every pair of arguments; given a current state $q \in Q$ and the symbol $c \in \Gamma$ that is currently under the read/write head, $(q', c', d) = \delta(q, c)$ is a tuple giving the new state, the symbol to be written at the current position and the direction in which the head must move;
- $\# \in \Gamma$ is the blank symbol;
- $q_0 \in Q$ is the start state;

- $q_\mathrm{s} \in Q$ is the "success" state;
- $q_\mathrm{f} \in Q$ is the "failure" state; $q_\mathrm{f} \neq q_\mathrm{s}$.

The execution of $M$ on a word $w \in \Sigma^*$ proceeds like this. At the beginning, the tape contains $w$ surrounded by an infinity of # in both directions. The read/write head is positioned on the first symbol of $w$. The state is set to $q_0$. Then, computation is done according to $\delta$. Execution stops if the machine enters the state $q_\mathrm{s}$ or $q_\mathrm{f}$. We say that $M$ *accepts* $w$ if execution ends by having $M$ to enter $q_\mathrm{s}$. We say that $M$ *refuses* $w$ if execution ends by having $M$ to enter $q_\mathrm{f}$. Finally, we say that $M$ *loops on* $w$ if execution never stops. The Universal Language is defined as:

$$\mathrm{UL} = \{(M,\, w) \in \{\text{Turing machines}\} \times \Sigma^* \mid M \text{ accepts } w\}$$

It is well-known that UL is undecidable. For example, see [34].

We can now present the theorem.

**Theorem 3.8** *The following problem is undecidable:*

$$\{e_l \in \mathrm{Exp} \mid \exists \mathcal{M}.\ e_l \textit{ is analysed perfectly well using model } \mathcal{M}\}$$

**Proof 3.8** We prove the theorem by making a reduction of UL to our problem. That is, if our problem were decidable, then UL would be, too, leading to a contradiction. The reduction is a transformation from a machine-word pair $(M,\, w)$ to a program $e_{l_0}$ such that $M$ accepts $w$ if and only if $e_{l_0}$ is analysable perfectly well.

The generated program simulates the execution of $M$ on $w$. If the execution of the machine ends by entering $q_\mathrm{s}$, the program ends by evaluating the expression `#f`. If the execution of the machine ends by entering $q_\mathrm{f}$, the program ends by evaluating the expression (`car #f`), causing an error. If the execution of the machine never ends, the program's evaluation lasts forever.

The tape is represented using two lists: one containing the part of the tape on the right of the head and another contains the reverse of the part of the tape on the left of the head. Of course, the lists cannot contain *all* the symbols appearing on their part of the tape. The end of list represents an infinity of blank symbols. The explicitly represented parts of the tape are lazily extended during the execution. The current state and the symbol under the read/write head are passed around as parameters.

Moreover, two counters are maintained throughout the program evaluation. The value of the first one is always 1 less than the value of the second one. The program tests whether this invariant is still true before each step of the simulation. Of course, the invariant is always true. In the other (necessarily impossible) case, an error is generated by evaluating (`car #f`). These two counters are used later in the proof.

We describe the transformation from $(M, w)$ to $e_{l_0}$ as a sequence of steps.

1. *From a Turing machine to a functional program.* We describe this first step of transformation using a number of compilation functions denoted by '$\mathsf{T}$.'.

```
$\mathsf{T}[\![(M, v)]\!]$ = let d = $\mathsf{T}_\delta[\![\delta]\!]$                /* $\mathrm{d} : Q \to \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ */
              let l = $\lambda$k lt cs rt. lt = #f  ? k #f '#' (cs:rt)
                                          : k (cdr lt) (car lt) (cs:rt)
              let r = $\lambda$k lt cs rt. rt = #f  ? k (cs:lt) '#' #f
                                          : k (cs:lt) (car rt) (cdr rt)
                          /* $l, r : (\Gamma^* \to \Gamma \to \Gamma^* \to \mathrm{Val}^\uparrow) \to \Gamma^* \to \Gamma \to \Gamma^* \to \mathrm{Val}^\uparrow$ */
              letrec s = $\lambda$c1 c2 q lt cs rt.
                         c1 + 1 $\neq$ c2 ? (car #f) :
                         q = $q_\mathrm{s}$ ? #f :
                         q = $q_\mathrm{f}$ ? (car #f) :
                         let (q$'$, cs$'$, dir) = d q cs
                           (dir = L ? l : r) (s (c1 + 1) (c2 + 1) q$'$) lt cs$'$ rt
              s 0 1 $q_0$ #f $\mathsf{T}_\mathrm{cs}[\![w]\!]$ $\mathsf{T}_\mathrm{rt}[\![w]\!]$
$\mathsf{T}_\delta[\![\delta]\!]$ = $\lambda$q cs. q = $q_0$ ? $\mathsf{T}'_\delta[\![\delta\ q_0]\!]$ :
                q = $q_1$ ? $\mathsf{T}'_\delta[\![\delta\ q_1]\!]$ :
                ...
                q = $q_{|Q|-1}$ ? $\mathsf{T}'_\delta[\![\delta\ q_{|Q|-1}]\!]$ :
                #f                                        /* $\leftarrow$ inaccessible case */
$\mathsf{T}'_\delta[\![\delta\ q]\!]$ = cs = $c_0$ ? $\mathsf{T}''_\delta[\![\delta\ q\ c_0]\!]$:
            cs = $c_1$ ? $\mathsf{T}''_\delta[\![\delta\ q\ c_1]\!]$:
```

$$
\begin{array}{ll}
\quad \ldots & \\
\quad \mathrm{cs} = c_{|\Gamma|-1} \ ?\ \mathsf{T}''_{\delta}[\![\delta\ q\ c_{|\Gamma|-1}]\!]\!: & \\
\quad \#\mathtt{f} & /\!*\ \leftarrow \text{inaccessible case } *\!/ \\
\mathsf{T}''_{\delta}[\![\delta\ q\ c]\!] = (q',\, c',\, \mathit{dir}') & /\!*\ \text{where } (q',\, c',\, \mathit{dir}') = \delta(q,\, c)\ *\!/
\end{array}
$$

$$
\mathsf{T}_{\mathrm{cs}}[\![w]\!] \ = \ \begin{cases} \text{`}\#\text{'}, & \text{if } w = \epsilon \\ \text{`}c\text{'}, & \text{if } w = cw' \end{cases}
$$

$$
\mathsf{T}_{\mathrm{rt}}[\![w]\!] \ = \ \begin{cases} \#\mathtt{f}, & \text{if } w = \epsilon \\ \mathsf{T}'_{\mathrm{rt}}[\![w']\!], & \text{if } w = aw' \end{cases}
$$

$$
\mathsf{T}'_{\mathrm{rt}}[\![w]\!] \ = \ \begin{cases} \#\mathtt{f}, & \text{if } w = \epsilon \\ \text{`}a\text{'}{:}\mathsf{T}'_{\mathrm{rt}}[\![w']\!], & \text{if } w = aw' \end{cases}
$$

In the generated program: function ‘d’ is the implementation of the transition function $\underline{\delta}$; functions ‘l’ and ‘r’ update the tape when doing a transition to the left or to the right, respectively; function ‘s’ does a step in the simulation of the machine; note that it verifies counters ‘c1’ and ‘c2’ before doing the step proper; variable ‘q’ holds the current state; variables ‘lt’, ‘cs’, and ‘rt’ hold the left part of the tape, the current symbol, and the right part of the tape, respectively; variable ‘k’ contains the continuation of the execution after an update of the tape.

2. *Removal of syntactic sugar.* We remove tuple manipulation in the generated program using these rules:

$$
\begin{array}{lcl}
\mathtt{let}\ (x,\, y,\, z) = e_1 & \mapsto & \mathtt{let}\ \mathrm{r} = e_1 \\
\quad e_2 & & \mathtt{let}\ x = (\mathtt{car}\ \mathrm{r}) \\
 & & \mathtt{let}\ y = (\mathtt{car}\ (\mathtt{cdr}\ \mathrm{r})) \\
 & & \mathtt{let}\ z = (\mathtt{car}\ (\mathtt{cdr}\ (\mathtt{cdr}\ \mathrm{r}))) \\
 & & \quad e_2 \\
(x,\, y,\, z)\ /\!*\ \text{as a tuple creation } *\!/ & \mapsto & x{:}y{:}z{:}\#\mathtt{f}
\end{array}
$$

We also remove multi-argument functions and calls:

$$
\begin{array}{lcl}
\lambda x_1\ x_2\ \ldots\ .\ e & \mapsto & \lambda x_1 .\ \lambda x_2\ \ldots\ .\ e \\
e_1\ e_2\ e_3\ \ldots & \mapsto & (e_1\ e_2)\ e_3\ \ldots
\end{array}
$$

Recursive use of these last rules may be required.

3. *Elimination of types specific to the simulation of the Turing machine.* We replace state, symbol, and direction constants by numerical counterparts. Let us define the

following coding functions:

$$\begin{aligned}
&\mathsf{T}_Q : Q \to \{0, \ldots, |Q| - 1\} \text{ bijective}\\
&\mathsf{T}_\Gamma : \Gamma \to \{0, \ldots, |\Gamma| - 1\} \text{ bijective}\\
&\mathsf{T}_{\{\mathrm{L},\mathrm{R}\}} : \{\mathrm{L}, \mathrm{R}\} \to \{0, 1\} \text{ bijective}
\end{aligned}$$

We replace each special constant by its code:

$$\begin{array}{lllll}
q & /*\ \in Q\ */ & \mapsto & \mathsf{T}_Q\llbracket q\rrbracket & /*\ \in \mathbb{N}\ */\\
c & /*\ \in \Gamma\ */ & \mapsto & \mathsf{T}_\Gamma\llbracket c\rrbracket & /*\ \in \mathbb{N}\ */\\
\mathit{dir} & /*\ \in \{\mathrm{L}, \mathrm{R}\}\ */ & \mapsto & \mathsf{T}_{\{\mathrm{L},\mathrm{R}\}}\llbracket \mathit{dir}\rrbracket & /*\ \in \mathbb{N}\ */
\end{array}$$

4. *Elimination of numbers.* In turn, we transform arithmetical expressions and constants. We transform the naturals into a unary representation based on lists. Here are the rules:

$$\begin{array}{lll}
e_1 \neq e_2\ ?\ e_3 : e_4 & \mapsto & e_1 = e_2\ ?\ e_4 : e_3\\
e_1 = e_2 \ /*\ \text{not a binding!}\ */ & \mapsto & ((\mathrm{eq}\ e_1)\ e_2)\\
e + 1 & \mapsto & (\mathrm{inc}\ e)\\
n \ /*\ \in \mathbb{N}\ */ & \mapsto & \mathsf{T}_\mathbb{N}\llbracket n\rrbracket
\end{array}$$

where:

$$\mathsf{T}_\mathbb{N}\llbracket n\rrbracket \;=\; \begin{cases} \texttt{\#f}, & \text{if } n = 0\\ \texttt{\#f}\text{:}\mathsf{T}_\mathbb{N}\llbracket n-1\rrbracket, & \text{if } n > 0 \end{cases}$$

After the numeric operations and constants are removed, we apply this last rule once to the whole program:

```
e  ↦   let inc = λn. #f:n
         letrec eq = λn1. λn2. n1 ? (n2 ? ((eq (cdr n1)) (cdr n2))
                                          : #f)
                                     : (n2 ? #f : (#f : #f))
           e
```

5. *Removal of syntactic sugar* (again). We transform many syntactic constructs into base language constructs. Each construct should be completely eliminated before continuing with the next.

**letrec** We remove `letrec`-expressions using the following rule:

$$\begin{array}{lcl} \texttt{letrec}\ \mathrm{v} = (\lambda \mathrm{x}.\ e_1) & \mapsto & \texttt{let}\ \mathrm{v} = (\mathrm{Y}\ (\lambda \mathrm{v}.\ (\lambda \mathrm{x}.\ e_1))) \\ \quad e_2 & & \quad e_2 \end{array}$$

and add the definition of the Y combinator once to the whole program using this rule:

$$\begin{array}{lcl} e & \mapsto & \texttt{let}\ \mathrm{Y} = \lambda \mathrm{f}.\ \texttt{let}\ \mathrm{g} = (\lambda \mathrm{h}.\ (\lambda \mathrm{z}.\ ((\mathrm{f}\ (\mathrm{h}\ \mathrm{h}))\ \mathrm{z}))) \\ & & \qquad\qquad\qquad (\mathrm{g}\ \mathrm{g}) \\ & & \quad e \end{array}$$

**let** We use this rule to remove `let`-expressions:

$$\begin{array}{lcl} \texttt{let}\ \mathrm{v} = e_1 & \mapsto & ((\lambda \mathrm{v}.\ e_2)\ e_1) \\ \quad e_2 & & \end{array}$$

**conditional** We replace the '· ? · : ·' construct by a conditional from the mini-language:

$$e_1\ ?\ e_2 : e_3 \quad \mapsto \quad (\texttt{if}\ e_1\ e_2\ e_3)$$

**cons** We apply the following rule while taking care of respecting the fact that the ':' operator is right-associative:

$$e_1{:}e_2 \quad \mapsto \quad (\texttt{cons}\ e_1\ e_2)$$

6. $\alpha$-conversion and proper labelling. We make sure each variable has a distinct name and add unique labels to all the expressions of the program.

We can make the following observations about the generated program $e_{l_0}$. First, the only expressions that may cause an error are the two (`car #f`) expressions. By construction of the program, we know that the evaluation of the other expressions cannot go wrong.

Second, the first (`car #f`) expression, although it would necessarily cause an error if it were evaluated, does not get evaluated in the first place. It is obvious that counters 'c1' and 'c2', after beginning with values 0 and 1, respectively, are each incremented by 1 after each simulation step. So the invariant c1 + 1 = c2 is true during the whole evaluation of $e_{l_0}$.

Third, if $M$ accepts $w$, the evaluation of $e_{l_0}$ ends by returning #f as a result. By

Theorem 3.7, it follows that there is a model $\mathcal{M}$ allowing $e_{l_0}$ to be analysed perfectly well.

Fourth, in the opposite case, $M$ refuses $w$ by entering state $q_{\mathrm{f}}$. In this case, the second (`car` #`f`) expression gets evaluated and an error occurs. By Theorem 3.6, there cannot be a model allowing $e_{l_0}$ to be analysed perfectly well.

Fifth and last observation, if $M$ loops on $w$, the evaluation of $e_{l_0}$ never ends and no error ever occurs but, nevertheless, $e_{l_0}$ is not perfectly analysable. This fact is not necessarily trivial to verify. We do not provide a complete and formal proof, we only give the following reasoning:

1. Let us suppose that $e_{l_0}$ can be analysed perfectly well using model $\mathcal{M}$. Note that we must have that $|\mathcal{V}al\mathcal{P}| < \infty$ for $\mathcal{M}$ to be a legal model.

2. Note also that counters 'c1' and 'c2' go through all values in $\mathbb{N}$ and $\mathbb{N} - \{0\}$, respectively, during the infinite evaluation.

3. The following point is not directly established by Theorems 3.5 and 3.6, but we will stretch the scope of these a little bit.

   In our present case, the evaluation is infinite, so our collecting machine would not stop computing and there would be no cache returned by it. However, we could define a variant of the collecting machine to which we pass an argument indicating the maximum number of steps that the machine should make. In the case of an infinite evaluation, we could obtain a cache describing the beginning of the evaluation. On top of it, we could adapt both theorems to make them able to handle partial caches. So, we suppose that we have results similar to those given by the theorems despite the fact that the evaluation is infinite.

   Now, this is where the counters 'c1' and 'c2' come into play. Each time the generated program $e_{l_0}$ tests whether the invariant about counters 'c1' and 'c2' is still true, the expression $e_{l_{\mathrm{test}}} = ({}_{l_{\mathrm{test}}}(\mathrm{eq}\ (\mathrm{inc}\ \mathrm{c1}_{l_1}))\ \mathrm{c2}_{l_2})$ is evaluated. So there are an infinity of contours $k \in \mathrm{Cont}$ and $n \in \mathbb{N}$ such that $\mathrm{post}(l_1,\, k,\, \mathsf{T}_{\mathbb{N}}[\![n]\!])$ and $\mathrm{post}(l_2,\, k,\, \mathsf{T}_{\mathbb{N}}[\![n+1]\!])$ are in the cache.[4]

[4] We make a slightly abusive use of $\mathsf{T}_{\mathbb{N}}$ as it is supposed to produce *code*, not *values*. However, each instance of code generated by $\mathsf{T}_{\mathbb{N}}$ can only evaluate to a single value, no matter in which environment or contour it is evaluated.

4. Among the abstract pairs in $\mathcal{V}al\mathcal{P}$, there is necessarily one that is the abstraction of more than one number (of more than one list of Booleans). Let $\hat{p} \in \mathcal{V}al\mathcal{P}$ be that pair. Moreover, let $m \neq n \in \mathbb{N}$ such that $\mathsf{T}_{\mathbb{N}}[\![m]\!] \nearrow \hat{p}$ and $\mathsf{T}_{\mathbb{N}}[\![n]\!] \nearrow \hat{p}$.

5. Let $k \in$ Cont such that post($l_1$, $k$, $\mathsf{T}_{\mathbb{N}}[\![m]\!]$) and post($l_2$, $k$, $\mathsf{T}_{\mathbb{N}}[\![m+1]\!]$) are in the cache. Let $\hat{k} \in \mathcal{C}ont$ and $\hat{p}' \in \mathcal{V}al\mathcal{P}$ abstract $k$ and $\mathsf{T}_{\mathbb{N}}[\![m+1]\!]$, respectively. By our "extended" Theorem 3.5, we know that $\hat{p} \in \alpha_{l_1,\hat{k}}$ and $\hat{p}' \in \alpha_{l_2,\hat{k}}$. By the ambiguity of what is abstracted by $\hat{p}$, we conclude that the abstract evaluation of the test has to include the possibility that the test is negative, leading to the evaluation of (`car #f`).

   More precisely, the abstract evaluation of $e_{l_{\text{test}}}$ in contour $\hat{k}$ represents the test inc($m$) = $m+1$ (which is true and which is expected by conservativeness) *and* the test inc($n$) = $m+1$ (which is false). So $\alpha_{l_{\text{test}},\hat{k}}$ contains an expected abstract true value (i.e. $\in \mathcal{V}al\mathcal{C} \cup \mathcal{V}al\mathcal{P}$) and an abstract false value (i.e. $\in \mathcal{V}al\mathcal{B}$).

6. Because the test may apparently be false, the expression (`car #f`) is abstractly evaluated in contour $\hat{k}$, leading to the violation of a safety constraint. Since this reasoning holds for an arbitrary model, we conclude that $e_{l_0}$ cannot be analysed perfectly well.

This concludes the proof that the generated program $e_{l_0}$ is analysable perfectly well if and only if $M$ accepts $w$. Since UL is undecidable, it is impossible to always be able to decide if there exists a model that allows an arbitrary $e_{l_0}$ to be analysed perfectly well. □

## 3.7 Flexibility in Practice

The flexibility of the analysis framework can be illustrated in another way. The framework is able to imitate many conventional analyses.

For example, we can define models that produce analysis instances similar to polynomial variants of Shivers' $K$-cfa [55, 61, 37]. The proposed models are intended for the analysis of program $e_{l_0}$.

$$\begin{aligned}
\mathcal{V}al\mathcal{B} &= \{\#\text{f}\} \\
\mathcal{V}al\mathcal{C} &= \{\bot_{\mathcal{C}}\} \cup \{\lambda_l\ \hat{k} \mid l \in \triangle(l_0)\ \wedge\ e_l \text{ is a } \lambda\text{-expression}\ \wedge\ \hat{k} \in \mathcal{C}ont\} \\
\mathcal{V}al\mathcal{P} &= \{P\} \\
\mathcal{C}ont &= \{\hat{k} \in \text{Lab}^* \mid |\hat{k}| \leq K\}
\end{aligned}$$

$$\begin{array}{lcl}
k_0 & = & \epsilon \\
\mathsf{cc}(l, \hat{k}) & = & \begin{cases} \lambda_l\ \hat{k}, & \text{if } e_l \text{ is a } \lambda\text{-expression} \\ \perp_{\mathcal{C}}, & \text{otherwise} \end{cases} \\
\mathsf{pc}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & P \\
\mathsf{call}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & \text{the longest suffix of } \hat{k}l \text{ in } \mathcal{C}ont
\end{array}$$

A contour is a chain of the labels of the enclosing $K$ sites where calls occurred that lead to the current evaluation. It is usually referred to as a *call chain*. By the definition of $\mathcal{C}ont$, there is only a polynomial number of abstract contours (relative to the size of the program). There is also a polynomial number of values. Pairs are represented coarsely by a single abstract pair. Distinct $\lambda$-expressions produce distinct closures. Moreover, the contour in which a closure was created is captured by the closure. It allows closures to behave differently depending on the evaluation context in which they were created. That does not directly correspond to remembering the lexical environment but, in favourable cases, it acts as a good substitute.

Note that in the particular case where $K = 0$, there is only one contour ($\epsilon$) for the whole abstract evaluation and one closure per $\lambda$-expression.

By its equivalence with the 0-cfa, set-based analysis [29, 37] is also imitated by an instantiation of an analysis using our framework.

More elaborate analyses can also be imitated by the framework. The following example is inspired from one in [37]. To obtain a more precise analysis, it is sometimes necessary to distinguish contours by the type of the values that are manipulated by the program. The advantage of contours based on types is that types constitute the information that is *really* used in the concrete evaluation. That is, a program may test whether a particular value is a pair, but never tests whether the function body being evaluated was called from expression $e_l$. Contours directly conveying the really useful information normally improve the analysis accuracy more than contours conveying information that is, in the best of cases, only *correlated* to the useful information. Here is the definition of a model using type-based contours.

$$\begin{array}{lcl}
\mathcal{V}al\mathcal{B} & = & \{\#\mathrm{f}\} \\
\mathcal{V}al\mathcal{C} & = & \{\perp_{\mathcal{C}}\} \cup \{\lambda_l\ \hat{k} \mid l \in \triangle(l_0)\ \wedge\ e_l \text{ is a } \lambda\text{-expression}\ \wedge\ \hat{k} \in \mathcal{C}ont\} \\
\mathcal{V}al\mathcal{P} & = & \{P\}
\end{array}$$

$$
\begin{array}{lcl}
\mathcal{C}ont & = & \{\perp\} \cup \left\{\hat{k} \in \{\textsc{b}, \textsc{c}, \textsc{p}\}^* \mid |\hat{k}| \leq L\right\} \\
& & \text{where } L \text{ is the maximum number of variables} \\
& & \qquad \text{visible from any } e_l \in \triangle(e_{l_0}) \\
k_0 & = & \epsilon \\
\mathsf{cc}(l, \hat{k}) & = & \begin{cases} \lambda_l\ \hat{k}, & \text{if } e_l \text{ is a } \lambda\text{-expression and } \hat{k} \neq \perp \\ \perp_{\mathcal{C}}, & \text{otherwise} \end{cases} \\
\mathsf{pc}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & P \\
\mathsf{call}(l, \hat{v}_1, \hat{v}_2, \hat{k}) & = & \begin{cases} \perp, & \text{if } \hat{k} = \perp \text{ or } \hat{v}_1 \notin \mathcal{V}al\mathcal{C} \text{ or } \hat{v}_2 = \perp_{\mathcal{C}} \text{ else} \\ \textsc{b}\hat{k}', & \text{if } \hat{v}_1 = \lambda_l\ \hat{k}' \text{ and } \hat{v}_2 \in \mathcal{V}al\mathcal{B} \text{ else} \\ \textsc{c}\hat{k}', & \text{if } \hat{v}_1 = \lambda_l\ \hat{k}' \text{ and } \hat{v}_2 \in \mathcal{V}al\mathcal{C} \text{ else} \\ \textsc{p}\hat{k}', & \text{let } \hat{v}_1 = \lambda_l\ \hat{k}' \text{ and } \hat{v}_2 \in \mathcal{V}al\mathcal{P} \end{cases}
\end{array}
$$

The two main differences with this new model are the following. Contours are made of type indicators instead of labels. And it is the contour contained in the invoked closure that is extended instead of the contour that prevails when the call occurs. The contour in which an expression is evaluated indicates the (top-level) type of the value to which each variable in the environment is bound. The analysis instance obtained using this model has exponential complexity in the size of the program. The worst case occurs when the longest lexical environment in the program contains a number of variables that is a significant fraction of the size of the program.

Note that an abstract variable like $\alpha_{l,\hat{k}}$ always exists, even if the number of variables visible from $e_l$ and the number of indicators in $\hat{k}$ do not match. In such a case, a minimal solution to the evaluation constraints always includes the assignment $\alpha_{l,\hat{k}} = \emptyset$ because the expression never gets evaluated in that contour.

Despite its great flexibility, our framework has its limits. As an instance, the analysis based on *polymorphic splitting* presented by Jagannathan and Wright [38] cannot be imitated by the framework. Polymorphic splitting is presented as a method of obtaining, in abstract interpretation, an analogue to the let-polymorphism used in Hindley-Milner polymorphic type inference [43]. Abstract closures that are bound to a variable in a `let`-expression receive a special treatment. First, their associated contour is extended when they are bound to the variable. Next, their contour is *modified* by each reference to the variable. Moreover, two distinct references to the variable produce two different modifications to the closure. This is clearly not feasible within our framework. In our case, a reference to a variable cannot modify the value it is bound to, neither can it modify a part of that value.

# Chapter 4

# Demand-Driven Analysis

## 4.1 A Cyclic Process

Now that we have a precise objective and a powerful analysis framework, we propose a coarse sketch of the demand-driven analysis. Demand-driven analysis should start by performing a *preliminary analysis* for the program. The preliminary analysis is an inexpensive analysis that provides relatively coarse initial analysis results. Typically, the preliminary analysis results do not bring sufficient evidence to let the optimiser to remove all dynamic safety tests. Demand-driven analysis then continues with a *model-update, re-analysis cycle.* A model-update phase proposes and performs changes on the abstract model, based on the most recent analysis results and on the dynamic tests that are remaining. Instead of "updated", we might as well say that the model has been *refined.* The re-analysis phase computes new analysis results for the program using the new abstract model. This cycle continues until there are no resources left for the analysis or all safety tests could be removed.

Of course, this sketch is very general and leads to many questions. We ask some questions ourselves and bring answers to some of them immediately.

What can one expect from the use of an updated, or refined, abstract model? Normally, the updated model produces a more accurate analysis instance. This more accurate analysis may provide analysis results containing less superfluous values. And, with chance, these allow the optimiser to remove some additional safety tests. We use the term *more informative* to describe analysis results that contain less superfluous values.

While it is clear that analysis results containing less superfluous values do not automatically imply that some additional safety tests can be removed, it may not be obvious why a more accurate analysis does not necessarily lead to more informative results. We give an example scenario. Let a program $e_{l_0}$ be analysed using model $\mathcal{M}$. Also, let $\mathcal{V}al\mathcal{P}$ contain only one pair. Let $e_l$ be some expression evaluating only to this pair. Now, for some reason, a more precise description of the values to which $e_l$ evaluates is required, and, consequently, $\mathcal{M}$ is refined into $\mathcal{M}'$ such that $\mathcal{V}al\mathcal{P}'$ contains nine pairs. The nine pairs indicate the types of the two values that are stored in the CAR- and CDR-fields (three different types for the CAR-field and three for the CDR-field). Suppose that $\mathsf{pc}$ is changed accordingly. A re-analysis is done and suppose that the results obtained for $e_{l_0}$ using $\mathcal{M}'$ reveal that $e_l$ may evaluate to any of the nine pairs in $\mathcal{V}al\mathcal{P}'$. Then, in the precise case of $e_l$, the analysis results are finer but not more informative.

How can the model be refined? In principle, there is no problem at all if one wants to refine a model since a model is a simple collection of framework parameters and new parameters can easily be chosen, as long as the new model is legal. Of course, automatic updates of the model are more involving. It depends a lot on the modelling strategy. But it is clearly feasible. Chapter 5 presents our proposal of a modelling strategy and the means to update models automatically.

How should the modifications to the model be chosen? That is, among changes to $\mathcal{V}al\mathcal{P}$ and $\mathsf{pc}$, changes to $\mathcal{V}al\mathcal{C}$ and $\mathsf{cc}$, changes to $\mathcal{C}ont$ and $\mathsf{call}$, or some combinations of these, which should be the most helpful in removing safety tests? This is the most interesting question. It is not obvious *a priori* as computations in the program to analyse can be very intricate. A change in the representation of pairs may help to obtain better information as to which functions can be invoked at a certain call, which in turn, may cause one of these functions not to be passed the Boolean that caused an error in the evaluation of its body.

Here are desirable characteristics of the method that chooses modifications to the model. Naturally, this method should be systematic. Requiring the intervention of the user would make it unusable. Also, it should tend to select *appropriate*, or *useful* modifications. To expect guarantees that all selected modifications are useful is utopian, as the general task is uncomputable. These reasons are generalities, but a more practical characteristic, and an important one, is that we want the method not to become a large AI program, or an expert system. We speculate that an AI engine driving the model modifications would probably obtain better results than a more simple and mechanistic approach. However, we wish to develop something that has at least some genericity, that could be adapted to other analyses or to other languages. As we mention earlier, the real goal is more a proof of concept than an attempt to get the best possible type analysis. The next section presents a proposal of a method for the selection of modifications to the model.

## 4.2 Generation and Propagation of Demands

We propose a method for the selection of modifications to the model that is based on *demands*. Roughly speaking, a demand is a request for the demonstration of a certain fact or for the execution of a certain action. It is emitted because there are good reasons to believe that its accomplishment would ultimately improve the analysis of the program. Also, it is emitted because there are reasons to believe that it does represent an actual fact (in the case of a request for demonstration) and consequently that it might be achievable.

In a model-update phase, demands are first generated, then processed, usually leading to the emission of new, subordinate demands. We do not want to give in this chapter a complete proposition as to precisely what demands are, how they are generated and how they are processed. A complete proposition is given in Chapter 5. Nevertheless, we present many general ideas here.

The processing of the demands is the process by which the direct needs of the optimiser, expressed as the initial demands, are ultimately translated into other demands that are precise indications on the way to update the model.

The initial demands are generated at the start of the model-update phase and directly mirror the needs of the optimiser. For each expression for which a safety test seems to be still required, according to the current analysis results, a demand is emitted asking for a demonstration to be made to show that, in fact, the values manipulated by the expression are all correct ones and no test is required. For example, if a safety test seems to be required for expression $(\mathtt{car}_l\ e_{l_1})$ or $(\mathtt{cdr}_l\ e_{l_1})$, a demand is generated to ask for a demonstration that, in fact, $e_{l_1}$ may only evaluate to pairs. Clearly, the fact that a demand is emitted implies that the current results suggest that $e_{l_1}$ may evaluate to something else than pairs. But the presence of the expression as it is suggests that the programmer believes that $e_{l_1}$

may only evaluate to pairs. The generation of the initial demands could hardly be a simpler operation.

Demand processing uses rules to determine what actions should be done in the hope of fulfilling the request stated in the demand. The actions to perform depend on the kind of demand to process and on the context. The context includes the current state of the model and the current analysis results. The existence of more than one kind of demands seems inevitable.

The initial demands are all similar: they all ask to show that a certain expression may only evaluate to pairs or to closures. However, other kinds of demands can be generated by the processing of the initial demands, and the processing of their sub-demands, and that of these new demands, etc. Even if different sets of demands may be used for different demand-driven methods, some kinds of demands seem inevitable. For example, a demand may ask for a demonstration that a particular expression does not get evaluated at all. Or, at least, not in certain circumstances. Another example: a demand might ask for a change to the model in such a way that more precise contours be introduced to cause a certain expression to evaluate only to pairs in a particular contour, and only to Booleans in another.

The precise set of demands that is required to implement a model-update phase depends on the way one models the values and contours, on the way one wants the demands to be processed (the processing rules), on the kind of sub-demands the processing rules produce, etc.

Depending on the context, the processing of certain demands may lead to trivial success, or trivial failure, to a modification to the model, or, generally, to a combination of actions on some auxiliary data structures and the emission of new demands. Trivial success occurs when, for example, the demand asks to show that an expression returns only pairs and that the current analysis results indicate that it is already the case. Trivial failure occurs when, for example, the demand asks to show that the main expression of the program does not get evaluated, which is simply false.

Sketches of processing rules for typical demands are presented just after an informal example of demand-driven analysis.

## 4.3 A Demand-Driven Analysis Example

We present an example of demand-driven analysis for a small program $e_1$. It is not a complex program and only one judicious modification to the basic abstract model will be sufficient to analyse it perfectly well. The model-update phase that is presented is not very complicated but it still provides the opportunity to informally introduce some considerations that are fundamental in the development of a complete demand-driven approach.

The program to analyse is the following:

```
($_1$($\lambda_2$f. ($_3$f$_4$ ($_5$f$_6$ (cons$_7$ #f$_8$ #f$_9$))))
  ($\lambda_{10}$x. (if$_{11}$ x$_{12}$
              (car$_{13}$ (pair?$_{14}$ x$_{15}$))
              ($\lambda_{16}$y. y$_{17}$))))
```

Its evaluation does not cause an error but it is designed to cause confusion during a naïve analysis, as we see next. The initial model we use for the analysis of $e_1$ is:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{M} = (\mathcal{V}al\mathcal{B},\ \mathcal{V}al\mathcal{C},\ \mathcal{V}al\mathcal{P},\ \mathcal{C}ont,\ K,\ \mathsf{cc},\ \mathsf{pc},\ \mathsf{call})} \\
\mathcal{V}al\mathcal{B} & = & \{\#\mathrm{f}\} \\
\mathcal{V}al\mathcal{C} & = & \{\lambda_2,\ \lambda_{10},\ \lambda_{16}\} \\
\mathcal{V}al\mathcal{P} & = & \{P\} \\
\mathcal{C}ont & = & \{K\} \\
\mathsf{cc}(l,k) & = & \begin{cases} \lambda_l, & \text{if } l \in \{2,10,16\} \\ \lambda_2, & \text{otherwise} \end{cases} \\
\mathsf{pc}(l,v_1,v_2,k) & = & P \\
\mathsf{call}(l,f,v,k) & = & K
\end{array}
$$

The results that we obtain by analysing $e_1$ using $\mathcal{M}$ are the following. We limit the presentation of the results to that of the $\alpha$ matrix.

$$
\begin{array}{llll}
\alpha_{1,K} = \{\#\mathrm{f}, \lambda_{16}\} & \alpha_{2,K} = \{\lambda_2\} & \alpha_{3,K} = \{\#\mathrm{f}, \lambda_{16}\} & \alpha_{4,K} = \{\lambda_{10}\} \\
\alpha_{5,K} = \{\#\mathrm{f}, \lambda_{16}\} & \alpha_{6,K} = \{\lambda_{10}\} & \alpha_{7,K} = \{P\} & \alpha_{8,K} = \{\#\mathrm{f}\} \\
\alpha_{9,K} = \{\#\mathrm{f}\} & \alpha_{10,K} = \{\lambda_{10}\} & & \\
\hline
\alpha_{11,K} = \{\#\mathrm{f}, \lambda_{16}\} & \alpha_{12,K} = \{\#\mathrm{f}, \lambda_{16}, P\} & \alpha_{13,K} = \{\#\mathrm{f}\} & \alpha_{14,K} = \{\#\mathrm{f}, P\} \\
\alpha_{15,K} = \{\#\mathrm{f}, \lambda_{16}, P\} & \alpha_{16,K} = \{\lambda_{16}\} & \alpha_{17,K} = \emptyset &
\end{array}
$$

When looking at the results, it is immediately apparent that only one dynamic safety test is still needed. All calls are safe since $\alpha_{2,K}$, $\alpha_{4,K}$, and $\alpha_{6,K}$ contain only closures. However, the only other potentially erroneous expression, $e_{13}$, still needs its safety test because its sub-expression, $e_{14}$, may evaluate to something else than pairs. Closer inspection of the results shows that the two invocations of $\lambda_{10}$ are merged together. For example, the values it returns (i.e. $\alpha_{11,K}$) include the abstractions for both values that are returned during the concrete evaluation of the program. The parameter 'x' (i.e. $\alpha_{12,K}$ and $\alpha_{15,K}$) contains abstractions for both arguments passed during the concrete evaluation, but it contains also $\lambda_{16}$ which is "prematurely" returned by the first invocation and passed as an argument in the second invocation.

This first analysis is considered to be the preliminary analysis of the whole demand-driven approach. The model used in the preliminary analysis is generally very simple, like in this example. The next step is a model-update phase, since a safety test is still required for the program. During the course of the model-update phase, we first generate initial demands and then process them.

There is only one safety test left so we generate only one initial demand. In fact, we generate only one initial demand for the safety test because there is only one contour, also. The demand directly mirrors the needs of the optimiser and we will denote it like this:

$$D_1 \equiv \textbf{show } \alpha_{14,K} \subseteq \mathcal{V}al\mathcal{P}$$

A literal reading of the demand does not make sense. Clearly, with the current model, the contents of abstract variable $\alpha_{14,K}$ are not restricted to pairs. But the intent is that *something* should be done with $\mathcal{M}$ in order to eventually have that $\alpha_{14,K}$ or, more likely, *specialisations* of $\alpha_{14,K}$ to all lie inside of the given bound.

What could *specialisations* of $\alpha_{14,K}$ be? Variable $\alpha_{14,K}$ represents the value of $e_{14}$ in any possible evaluation contexts. This is because contour $K$ is unique and, as such, represents all evaluation contexts. But a change to the model could introduce different contours (e.g. $K_1$, $K_2$, …). Each of them would represent a distinct subset the evaluation contexts. So $\alpha_{14,K_1}$, $\alpha_{14,K_2}$, … would represent the value of $e_{14}$ in each set of evaluation contexts. Having said that, we can interpret the demand as "do any necessary modifications to the model to have, for any contour $K'$ that is a specialisation of $K$, the constraint $\alpha_{14,K'} \subseteq \mathcal{V}al\mathcal{P}$ to be satisfied". Note that the modifications to the model need not necessarily introduce specialisations of $K$ but could modify the representation of closures or that of pairs to obtain the desired effect.

Now, let us turn to the processing of $D_1$. Ultimately, we want $e_{14}$ to return nothing else than pairs. To control the value of an expression, one normally has to control the source of its value. $e_{14}$ is a `pair`?-expression, and the value of that kind of expression depends solely on the value of its sub-expression. By the semantics of a `pair`?-expression, it would be sufficient to have $e_{15}$ to return only pairs. So we would generate this new demand:

$$D_2 \equiv \textbf{show } \alpha_{15,K} \subseteq \mathcal{V}al\mathcal{P}$$

As will be made apparent when we will be involved in the design of processing rules for the demands, more than one strategy is usually available. For example, another sufficient achievement consists in proving that $e_{14}$ does not get evaluated at all, namely:

$$D_2' \equiv \textbf{show } \delta_{14,K} = \emptyset$$

Consequently, it would not evaluate to any value, and the CAR-field extraction would certainly not operate on non-pairs. Is one of these two processing methods better than the other? Are there other ways to process $D_1$?

The answer to the second question is: yes. But we will explore other possibilities when we present a complete approach in Chapter 5. To the first question, we answer that the first processing method is better. Here is the reason. Although the fulfilling of any of $D_2$ and $D_2'$ is sufficient to fulfil $D_1$, only $D_2$ is *necessary*. That is, $\alpha_{14,K} \subseteq \mathcal{V}al\mathcal{P}$ implies $\alpha_{15,K} \subseteq \mathcal{V}al\mathcal{P}$. But it is not the case that $\alpha_{14,K} \subseteq \mathcal{V}al\mathcal{P}$ implies $\alpha_{15,K} = \emptyset$.

Now, why is it preferable to use sufficient and necessary sub-demands? Because of the following reasoning. Since (`car`$_{13}$ $e_{14}$) is a part of the program, it is reasonable to expect $e_{14}$ to return only pairs. It is not an absolute truth at all, but simply a reasonable assumption. Since the demonstration that $e_{13}$ returns only pairs is necessary to satisfy $D_1$, $D_2$ seems to be a reasonable demand. The fact that $D_2$ is also sufficient makes it even more attractive. On the other hand, the property expressed in $D_2'$ is not necessary, so the program could possibly behave in such a way that the property expressed by $D_2'$ is violated while the one in $D_1$ is satisfied nevertheless. It follows that $D_2'$ could be false and, consequently, impossible to satisfy. In the case considered in this example, the property in $D_2'$ is effectively false as $e_{14}$ *is* evaluated.

Having chosen sub-demand $D_2$, we then have to process it. Although it is tempting to interpret $D_2$ as saying "show that 'x' can only be bound to pairs", the right interpretation is more like "show that 'x' can only be bound to a pair *when $e_{15}$ is evaluated*". Showing that $e_{15}$ is not evaluated at all would solve our problems but this property is not a necessary one, again. So we reject it. Let us study the situation carefully. Currently, 'x' seems to possibly be bound to objects of any type. However, in the case where 'x' is bound to a pair, the property in $D_2$ is satisfied, so it is fine. And in the cases where 'x' is bound to #f or to a closure, it appears that the property is violated. However, in the #f case, the conditional causes $e_{15}$ not to be evaluated. Consequently, there is no problem in this case, too. But let us suppose that the processing rules cannot make such a reasoning. So a sensible approach consists in first *separating* the cases associated to each type. In simple words, evaluation of body $e_{11}$ should occur in different contours depending on the type of 'x'. We express this new demand by:

$$D_3 \equiv \textbf{split}\ \alpha_{15,K}\ \star$$

The '$\star$' is the *split point* symbol. It indicates where additional precision in the abstract values is desired. It means "do the appropriate modifications to $\mathcal{M}$ so that, in $K$ or in each of its eventual substitutes $K_1, \ldots, K_n$, $e_{15}$ evaluates to values of only one type". If the request in this demand could be achieved, then we would have made progress in the resolution of our problem since it would be decomposed into three sub-cases. The sub-case in which 'x' is bound to a pair would not be a problem. Neither would the sub-case in which 'x' is bound to #f. There would remain the case where 'x' is bound to a closure. The THEN-branch of the conditional would be evaluated and CAR-field extraction would be attempted on #f. But, at least, the situation would be clearer because evaluation in this case would necessarily lead to an error, so it would be legitimate to emit this demand:

$$D_4 \equiv \textbf{show}\ \delta_{15,K_{\mathcal{C}}} = \emptyset$$

where $K_{\mathcal{C}}$ would be the contour in which 'x' is bound to a closure.

But let us not skip important steps. We first have to take care of $D_3$. Separating evaluation contexts to distinguish the type of the values bound to a variable is easy since contours are selected by the call function. And we have total control over call. Let us process

$D_3$ by modifying $\mathcal{M}$. We include only the modifications to $\mathcal{M}$:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{M}' = (\mathcal{V}al\mathcal{B},\, \mathcal{V}al\mathcal{C},\, \mathcal{V}al\mathcal{P},\, \mathcal{C}ont',\, K,\, \mathsf{cc},\, \mathsf{pc},\, \mathsf{call}')} \\
\mathcal{C}ont & = & \{K, K_{\mathcal{B}}, K_{\mathcal{C}}, K_{\mathcal{P}}\} \\
\mathsf{call}(l, f, v, k) & = & \begin{cases} K_{\mathcal{B}}, & \text{if } f = \lambda_{10} \ \wedge \ v \in \mathcal{V}al\mathcal{B} \\ K_{\mathcal{C}}, & \text{if } f = \lambda_{10} \ \wedge \ v \in \mathcal{V}al\mathcal{C} \\ K_{\mathcal{P}}, & \text{if } f = \lambda_{10} \ \wedge \ v \in \mathcal{V}al\mathcal{P} \\ K, & \text{otherwise} \end{cases}
\end{array}$$

Note how the evaluation of the body of $\lambda_{10}$ will occur in different contours depending on the type of the argument. The rest of the program is evaluated in contour $K$. Here are the analysis results that we obtain for $e_1$ using $\mathcal{M}'$ (only the non-empty entries are listed):

$$\begin{array}{llll}
\alpha_{1,K} = \{\lambda_{16}\} & \alpha_{2,K} = \{\lambda_{2}\} & \alpha_{3,K} = \{\lambda_{16}\} & \alpha_{4,K} = \{\lambda_{10}\} \\
\alpha_{5,K} = \{\#\mathrm{f}\} & \alpha_{6,K} = \{\lambda_{10}\} & \alpha_{7,K} = \{P\} & \alpha_{8,K} = \{\#\mathrm{f}\} \\
\alpha_{9,K} = \{\#\mathrm{f}\} & \alpha_{10,K} = \{\lambda_{10}\} & & \\
\hline
\alpha_{11,K_{\mathcal{B}}} = \{\lambda_{16}\} & \alpha_{12,K_{\mathcal{B}}} = \{\#\mathrm{f}\} & & \\
 & \alpha_{16,K_{\mathcal{B}}} = \{\lambda_{16}\} & & \\
\hline
\alpha_{11,K_{\mathcal{P}}} = \{\#\mathrm{f}\} & \alpha_{12,K_{\mathcal{P}}} = \{P\} & \alpha_{13,K_{\mathcal{P}}} = \{\#\mathrm{f}\} & \alpha_{14,K_{\mathcal{P}}} = \{P\} \\
\alpha_{15,K_{\mathcal{P}}} = \{P\} & & &
\end{array}$$

These results are much more accurate. We see fewer superfluous values in the $\alpha$ matrix. Obviously, $D_3$ has been processed with success since 'x' contains only values of the type indicated by the contour, if at all, i.e. $\alpha_{12,K_{\mathcal{B}}} \subseteq \mathcal{V}al\mathcal{B}$, $\alpha_{12,K_{\mathcal{C}}} \subseteq \mathcal{V}al\mathcal{C}$, and $\alpha_{12,K_{\mathcal{P}}} \subseteq \mathcal{V}al\mathcal{P}$. As expected, there is no problem in contours $K_{\mathcal{B}}$ and $K_{\mathcal{P}}$. But the good news is that there is no problem in contour $K_{\mathcal{C}}$ either because the first invocation of $\lambda_{10}$ no longer returns $\lambda_{16}$ "prematurely" and so the second invocation does not receive $\lambda_{16}$ as an argument.

The last safety test can now be removed without risk for the safety of the program. Indeed, $\forall k \in \mathcal{C}ont.\ \alpha_{14,k} \subseteq \mathcal{V}al\mathcal{P}$. On the other hand, if it would not have been the case that $\alpha_{14,K_{\mathcal{C}}} \subseteq \mathcal{V}al\mathcal{P}$, then it would have been necessary to continue with the processing of $D_4$.

## 4.4 Preliminary Analysis

The choice of a good initial model to be used in the preliminary analysis is important. We do not give one here explicitly as it depends on the abstract value representation strategy. But there are some principles that must be considered during the choice of the initial model.

The initial model has to be a compromise between contradictory tendencies: an ideal preliminary analysis should be relatively fast *and* accurate. The problem with a preliminary analysis that is too slow is that it may consume all the work units available to the analyser. And, in case of exhaustion of the work units during the preliminary analysis, one has to choose between two bad solutions. First bad solution: let the preliminary analysis finish. In this case, the time limit prescribed by the user is not respected. Second bad solution: interrupt the preliminary analysis. In this case, the analysis results have to be completely discarded as the minimal *valid* solution has not been reached yet and, consequently, there is no guarantee that the results are conservative.

On the other hand, the problem with a preliminary analysis that is not accurate enough is that the results may be almost unusable. It means that the results could contain so many superfluous values that almost all safety tests would seem to be required. It follows that almost all the work would be left to the model-update, re-analysis cycle. The cycle is powerful but the cost of removing one safety test with it is much greater than the cost of removing one safety test with the preliminary analysis.

## 4.5 Model-Update, Re-Analysis Cycle

The proposition of a complete approach for the cycle is presented in Chapter 5. Here, we only present considerations related to the model-update, re-analysis cycle and especially to demand processing. Many of the considerations have been introduced informally in the example.

The purpose of the model-update, re-analysis cycle is to modify the model in such a way that an increasing number of dynamic safety tests can be removed from the executable code generated for the program to compile. As proposed, the model-update phase consists in the generation and processing of demands in order to translate the needs of the optimiser into prescriptions of model updates.

When processing a demand, the corresponding processing rule should always translate it into *necessary* and, when possible, into *sufficient* sub-demands. Sub-demands are sufficient when achievement of the requests in the sub-demands implies achievement of the request in the processed demand. Sub-demands are necessary when the achievement of the processed demand necessarily implies the achievement of the sub-demands. It is not always possible to find sufficient *and* necessary sub-demands, depending on the demand to process and the current analysis results. We give examples of the four possible cases.

**Necessary and sufficient** This case occurred in the demand-driven analysis example. For demand $D_1$:

$$D_1 \equiv \textbf{show } \alpha_{14,K} \subseteq \mathcal{V}al\mathcal{P} \quad \text{where } e_{14} = (\texttt{pair?}_{14}\ e_{15})$$

we can emit one sub-demand $D_2$:

$$D_2 \equiv \textbf{show } \alpha_{15,K} \subseteq \mathcal{V}al\mathcal{P}$$

$D_2$ is sufficient because its achievement would automatically imply the achievement of $D_1$, as a `pair?`-expression evaluates to a pair when its sub-expression evaluates to that precise pair. $D_2$ is also necessary because the only way we can have that $e_{14}$ returns only pairs (or nothing) is to have $e_{15}$ to return only pairs (or nothing). This is the ideal case.

**Necessary but insufficient** Let us consider a demand $D_3$:

$$D_3 \equiv \textbf{split } \alpha_{23,k}\ \star \quad \text{where } e_{23} = (\texttt{if}_{23}\ e_{24}\ e_{25}\ e_{26})$$

Suppose that both $e_{25}$ and $e_{26}$ evaluate to values of more than one types in contour $k$. Since the value of $e_{23}$ is the union of the values of $e_{25}$ and $e_{26}$, then it is necessary to split the values coming from $e_{25}$ and $e_{26}$. That is, if the model were magically modified in such a way that $D_3$ is achieved, we would necessarily observe that, in each sub-contour $k_i$ specialising $k$, $e_{25}$ would evaluate to values of a single type. Similarly for $e_{26}$. So let us emit the following sub-demands:

$$D_4 \equiv \textbf{split}\ \alpha_{25,k}\ \star \qquad D_5 \equiv \textbf{split}\ \alpha_{26,k}\ \star$$

If both sub-demands are eventually satisfied, then both branches of the conditional will be well-split according to the type of the values to which they evaluate. That is, $k$ will have been replaced by sub-contours $k_1$, $k_2$, ... such that in each $k_i$, each branch, taken individually, evaluates to values of only one type, if at all. But it does not automatically imply that $D_3$ is achieved. In a contour, say $k_7$, $e_{25}$ could evaluate only to pairs while $e_{26}$ could evaluate only to Booleans, meaning that, in $k_7$, $e_{23}$ evaluates to values of more than one type. So the processing of $D_3$ produced necessary but insufficient sub-demands.

**Sufficient but unnecessary** Let us consider a demand $D_6$:

$$D_6 \equiv \textbf{show}\ \alpha_{18,K} \subseteq \mathcal{V}al\mathcal{P} \quad \text{where } e_{18} = (\texttt{if}_{18}\ e_{19}\ e_{20}\ e_{21})$$

Suppose that $e_{19}$ and $e_{20}$ evaluate to values of all types and that $e_{21}$ evaluates only to pairs. We could emit the following sub-demand:

$$D_7 \equiv \textbf{show}\ \alpha_{19,K} \subseteq \mathcal{V}al\mathcal{B}$$

The advantage of using $D_7$ is that its achievement is sufficient to cause the achievement of $D_6$. However, it does not express a necessary property of the computations made by the program. To see why, imagine that the model is magically modified in such a way that $D_6$ is achieved. It could be the result of having $e_{20}$ to return only pairs and leaving the results of $e_{19}$ unchanged. In this case, the property in $D_7$ would not be satisfied and it could even be impossible to satisfy $D_7$. So, processing $D_6$ as we suggested here is risky.

**Insufficient and unnecessary** Let us consider a demand $D_8$:

$$D_8 \equiv \textbf{show}\ \alpha_{31,X} \subseteq \mathcal{V}al\mathcal{P} \quad \text{where } e_{31} = ({}_{31}e_{32}\ e_{33})$$

Suppose that $\alpha_{32,X} = \{c_1, c_2\}$, $\alpha_{33,X} = \{b, p\}$ (for $\underline{\text{B}}$oolean and $\underline{\text{p}}$air), and that the results obtained by performing each possible invocation is summarised in this table:

| on | $b$ | $p$ |
|---|---|---|
| $c_1$ | $\{b, p\}$ | $\{b, p\}$ |
| $c_2$ | $\{b\}$ | $\{p\}$ |

Here is an insufficient and unnecessary sub-demand:

$$D_9 \equiv \textbf{show } c_1 \notin \alpha_{32,X}$$

$D_9$ would be insufficient because the local information currently available indicates that it is possible that $c_2$ would be still called on $b$ and still returned $\{b\}$. Also, $D_9$ would be unnecessary because the real computations happening at $e_{31}$ could be that a concrete closure abstracted by $c_1$ does get called on some argument but that the return value is a pair.

The last case was included in the list for completeness. It is not clear how the generation of unnecessary and insufficient sub-demands could help in the model-update phase.

In general, unnecessary sub-demands should be avoided since the property they contain may possibly be false. Since there is no hope of ever finding a demonstration for such properties, a considerable amount of time could be lost in the processing of the unnecessary demands. Note however that it does no harm as far as the safety of the generated executable code is concerned. Dynamic safety tests are removed only when there is indisputable evidence in the analysis results that they are redundant.

The whole demand-driven approach is based on the following reasoning. We use the arrow '$\leadsto$' to indicate that the steps in the reasoning are not logical implications but reasonably reliable conclusions instead.

When a programmer uses a possibly erroneous operation such as `car`, he expects
the safety test to always succeed
$\leadsto$ the safety test truly succeeds all the time
$\leadsto$ there exists a mathematical proof that the safety test always succeeds
$\leadsto$ there exists an abstract model that forms a demonstration that the safety test always succeeds

$\leadsto$ the demand-driven cycle can find such an abstract model through demand generation and processing

This reasoning clearly shows the difference between "trusting the programmer on how to invest analysis efforts" and "trusting the programmer on which safety tests should be omitted". Only the first kind of trust is granted as it does not compromise the safety of the executable code. The reasoning also illustrates why it is so important to use processing rules that produces only necessary sub-demands. By the fact that the initial demand, generated from the possibly erroneous expression, is most probably necessary, then all its sub-demands and sub-sub-demands, recursively, are necessary, too. It is then reasonable to expect these sub-demands to be achievable. On the other hand, there is no "reasonably reliable" chain of deductions to support the belief that an unnecessary sub-demand has a good potential of being satisfiable.

The complete set of demands naturally depends on the whole demand-driven approach. However, three kinds of demands that we already mentioned previously seem to be unavoidable. Namely, *bound demands*, such as **show** $\alpha_{12,k} \subseteq \mathcal{V}al\mathcal{P}$, *split demands*, such as **split** $\alpha_{12,k}\ \star$, and *never demands*, such as **show** $\delta_{12,k} = \emptyset$. Normally, we expect the processing rules to be relatively simple for most of the demand kinds and in most situations. As an instance, the reasonings involved in the example of Section 4.3 were all reasonable and intuitive. However, the biggest problems are to be expected from the processing of the demands related to conditionals and call expressions. Especially from the calls as the undecidability of the optimisation task would disappear if calls were removed from the source language.

The importance of having necessary demands leads to an important principle in the design of the processing rules. This principle says that the *good cases* should always be separated from the *bad cases* before an attempt is made to show that some cases do not occur. The wording of the principle is deliberately vague as it applies to many situations. The meaning of the principle is better illustrated by examples.

A first example relates to the bound demands. Usually, some values lie inside the bound and the others, outside. Let us consider demand $D \equiv$ **show** $\alpha_{3,K} \subseteq \mathcal{V}al\mathcal{C}$ and let us suppose that $\alpha_{3,K}$ contains abstract closures and pairs. The closures are the good cases since their presence in $\alpha_{3,K}$ does not give rise to problems. On the other hand, the pairs are the bad cases as $D$ precisely asks for a demonstration that they should not appear in $\alpha_{3,K}$. One cannot take appropriate measures to achieve $D$ by letting $\alpha_{3,K}$ contain both types of values. Neither can one do so by trying to eliminate all values from $\alpha_{3,K}$ as the resulting sub-demands would not express necessary properties. Because, as far as we know, there is no indications that closures should not appear in $\alpha_{3,K}$. So, appropriate measures must first include a sub-demand asking to make a separation between pairs and closures in $\alpha_{3,K}$. Using demand $D' \equiv \textbf{split}\ \alpha_{3,K}\ \star$, actually. Only when $D'$ has been achieved can one continue with the normal processing of $D$. In the general situation, after the successful processing of $D'$, contour $K$ has been replaced by specialised versions of $K$: $\{K_1, \ldots, K_n\} = G \dot{\cup} B$, and in contours $K_i \in G$, $\alpha_{3,K_i}$ contains only closures and in contours $K_j \in B$, $\alpha_{3,K_j}$ does not contain any closure. $D$ is trivially satisfied in contours $K_i \in G$ (the Good cases). In contours $K_j \in B$ such that $\alpha_{3,K_j} \neq \emptyset$ (the Bad cases), it is now legitimate to emit a demand like $D''_j \equiv \textbf{show}\ \delta_{3,K_j} = \emptyset$. Now, $D''_j$ is as necessary as $D$. That is, violation of $D$ causes a safety test to stay required and violation of $D''_j$ does the same.

Here is another example relates to calls. In a single call, some invocations may be considered to be hazardous and some, not. Let us consider call $({}_{40}e_{41}\ e_{42})$ in contour $K$, where $\alpha_{41,K} = \{c_1, c_2\}$, $\alpha_{42,K} = \{v\}$, and a demand $D$ asking for a demonstration that $c_1$ is not invoked on $v$ at $e_{40}$ in contour $K$. Doing nothing is not an appropriate method to achieve $D$. On the contrary, emitting demands like $\textbf{show}\ \delta_{40,K} = \emptyset$ or $\textbf{show}\ \delta_{41,K} = \emptyset$ is not appropriate either as they do not express necessary properties. That is, it may be the case that a concrete closure, represented by $c_2$ is truly invoked on a value, represented by $v$, at $e_{40}$ in some context, represented by $K$. Consequently, closures $c_1$ (the bad case) and $c_2$ (the good case) must be separated before any attempt to demonstrate that some expression does not get evaluated in some contour is made.

Apart from the fundamental mechanism of generation and processing of demands, many considerations are related to the infrastructure required by the demand-driven analysis. A first consideration is that there has to be some kind of concurrency in the model-update, re-analysis phase. The cycle cannot proceed by working on the removal of one safety test, then on another, etc. Any safety test may be arbitrarily difficult to remove, if possible at all. So a sequential approach for the removal of tests may block at one of the first tests, leading to the consumption of all the time units available. This is a bad use of the resources as many more safety tests might have been removed by working on all tests concurrently. This way, all the easily removable tests disappear after little effort has been invested on them. Only the tests that are the most difficult to remove, or impossible, remain.

Deriving from the concurrency, there is the problem of the obsolescence of demands. The effort that is invested on some tests frequently results in an update of the model. This update causes some demands to become either trivially satisfied, or expressed in out-fashioned terms. Demands that are trivially satisfied not even have to be those that are responsible for the model update. They become a simple nuisance as processing them is a waste of time. Proper testing may be done before processing each demand in order to avoid wasting time on already satisfied ones. On the other hand, demands that are expressed in out-fashioned terms are a more serious problem as their meaning is not related to the abstract model anymore. For example, consider the out-fashioned demand $D \equiv \textbf{split}\ \alpha_{7,k} \star$ where contour $k$ has been replaced by the more precise contours $k_1$, $k_2$, and $k_3$. As it is, $D$ is no longer a valid demand. It should be replaced by specialised demands $D_1$, $D_2$, and $D_3$ where $D_i \equiv \textbf{split}\ \alpha_{7,k_i} \star$. Continuing to manipulate $D$ is problematic as the following situation could occur. Each $D_i$ may be trivially satisfied. That is, each abstract variable $\alpha_{7,k_i}$ may contain values of only one type. That would mean that $D$ would be satisfied. However, if we interpreted $\alpha_{7,k}$ as $\cup_i\ \alpha_{7,k_i}$, then we could be brought to believe that $D$ is not satisfied, as $\cup_i\ \alpha_{7,k_i}$ could contain values of different types.

A last consideration concerns the sharing of the abstract model between *threads* of demand processing. Note that the computation effort that is put into proving the redundancy of a particular safety test can be viewed as a thread in the global, concurrent model-update, re-analysis phase. Sharing the abstract model between threads means that, each time one of the threads selects an update to the model, it is applied to a single global model. On the contrary, not sharing the model means that each thread has its own private model. The advantage of sharing is that useful information can flow quickly between threads. And updating a model means that the new analysis results will mimic the concrete evaluation more accurately. However, the inconvenience of sharing is that the frequent model updates coming from all threads cause demands to be frequently rewritten in new terms. These frequent rewritings tend to cause a proliferation of demands. Hybrid approaches can be chosen that try to obtain the best of both worlds and keep the inconvenience to a minimum. For example, the model held by a thread is communicated to the other threads only if its corresponding safety test has been proven to be redundant. So only “clearly useful” model updates propagate to the model of the other threads.

## 4.6 Discussion

To summarise the contents of this chapter, we would say that it is a proposal of a demand-driven analysis being composed of a preliminary analysis followed by a model-update, re-analysis cycle. Instead of the description of a complete approach, the most important considerations to take care of in the design of a complete approach are highlighted. The major considerations are: the balance between accuracy and cost in the preliminary analysis; the concepts of necessity and sufficiency in the processing of demands; the necessary concurrency in the cycle and its consequences; and the eventual sharing of model updates. Chapter 5 proposes a complete approach that tries to stay as simple as possible while taking care of these considerations.

Of course, without the proposition of a complete approach and the execution of experiments, it is hard to evaluate the potential of a demand-driven type analysis. However, the eventuality that the demand-driven analysis could be less powerful than an oracle in choosing an abstract model could well be real. That is, an oracle would choose an abstract model allowing the program to be analysed perfectly well each time such a model exists. Of course, this task is uncomputable and we cannot expect the demand-driven approach to do the same in finite time for each program. But we could have hoped that, given an unbounded amount of time, it has the ability to eventually find an appropriate model each time such a model exists while having the freedom to possibly loop each time the model does not exist. However, even this reduced requirement may not be achievable. That is, the demand-driven analysis does not try every possible abstract model by brute force. Each modification has to be *needed* according to the current state of the model and the current analysis results. So there exists the possibility that a program could be so intricate that no useful suggestion for updating the model is proposed after a certain point. I.e. that all useful modifications to the model seem to be unnecessary.

We expect that a model-update phase based on demand manipulation ought to propose interesting modifications to the model. The expectations come from the necessity of the property in each demand. Necessity that ultimately comes from the supposition of the programmer being *probably* right when he believes that some values have to be pairs or closures. Consequently, we say that calls and `car`- and `cdr`-expressions are *reliable hints* to have guidance of the demand-driven analysis. Of course, these expressions are precisely those that normally include dynamic safety tests that the optimiser wants to remove. But the legitimacy behind the expectations comes from the reliability of the hints and not from the importance of having the optimiser to perform its task. If the optimisation to perform were the detection of calls where inlining of functions can occur,[1] there would not be the same legitimacy. To see why, when the programmer writes a call expression in his program, that does not mean that he believes that only one function can be invoked from this call. At least, there is no syntactic evidence to support the existence of such a belief. So there is no reasonable chain of conclusions that we can draw from the call. However, other reliable sources of properties exist. As an instance, profiling[2] can provide statistics about the execution of a program and these statistics may reveal the existence of properties with possibly high degree of reliability. For example, if all closures observed at a certain call $e_l$ came from the same $\lambda$-expression $e_{l'}$ during each of the several million invocations having occurred there, then it is an opportunity for inlining. A demand could be emitted that requests a demonstration of the statement that all the closures invoked at $e_l$ come from $e_{l'}$.

[1] The *inlining* is an optimisation technique in which a call is replaced by the body of a function, when it is known that only that function could be invoked at that call.

[2] *Profiling* a program consists in gathering different statistics on the details of the execution of a program. The nature of the statistics may vary wildly as they go from execution frequency for expressions to the type of the objects seen at a particular point in the program.

# Chapter 5

# Pattern-Based Demand-Driven Analysis

We now present a complete approach for performing a demand-driven analysis. That is, we present particular choices for the representation of the abstract values and abstract contours, the implementation of models, and the global algorithm. The choices are intended to form the simplest and most intuitive representation for the abstract values and contours. Values and contours are based on *patterns* or, in informal terms, data structures with holes. A pattern presents a shallow description of a concrete value or contour. It is similar to patterns found in high-level languages that feature pattern-matching for the definition of functions. Models are represented using pattern-matchers.

The chapter starts by giving a complete presentation of the abstract models and demands. Then the processing rules for the demands are presented with a discussion on our particular choices. Next, the whole approach is presented. It is a description of the *current* prototype. A history of the different attempts to create a working prototype follows. Finally, we discuss the pros and cons of the current pattern-based approach and mention extensions to it.

## 5.1 Pattern-Based Modelling

We use patterns to represent values and contours. The syntax of these patterns and their meaning is first presented. The reasons behind the choice of the patterns follow.

Next, the abstract models are described. It includes the definition of the pattern-matchers, their use and the properties that they must obey. Algorithms used to update the pattern-matchers, that is, the abstract models, are also presented.

Finally, the syntax and meaning of the demands are presented. These are presented in that section because the definition of demands is closely related to the patterns and the representation of values and contours.

### 5.1.1 Representation of the Abstract Values and Contour

The abstract values are represented using patterns which are shallow versions of the concrete values. That is, the type and contents of the sub-values are known up to a certain depth. The depth where the details are still available need not be the same in every part of a value. At the point where no more details are available, a special pattern is used to indicate that anything could go there. For example, an abstract pair could contain the Boolean #**f** in its CAR-field and the special any-value pattern in its CDR-field.

There are two reasons why we have adopted such a representation. We believe that it is the simplest and most intuitive representation that still features an arbitrary level of accuracy. Also, following the explanations of Section 3.7, we think that data abstractions of what is directly used in the concrete evaluation should perform better than abstractions that are indirectly linked to the concrete evaluation. For example, we expect to obtain better results by manipulating abstract pairs containing (incomplete) description of the two values they contain than by manipulating ones memorising at which label and in which contour they were created.

#### Overview

Here is an overview of the abstract representation for the values of each type. There is only one abstract Boolean since there is only one concrete Boolean to keep track of. Pairs are more or less shallow representations of concrete pairs. They do not memorise how they were created but rather what they contain. On the other hand, closures remember which $\lambda$-expression they come from and what contour was prevailing during their creation. Two distinct $\lambda$-expressions cannot produce the same abstract closure. However, the memorised contour can be an approximation of the one that prevailed at creation time. This may seem to be in sharp contradiction with the spirit in which we want to represent abstract values. But this apparent contradiction disappears when we see what abstract contours are.

The choice of the representation for the contours is a direct application of the principle that the best abstract representation should be a partial description of the concrete entity. Abstract contours are essentially shallow versions of lexical environments, but without the variables. An abstract contour is a list of abstract values where each abstract value represents the value to which a visible variable is bound to. As abstract values, abstract contours may feature various degrees of accuracy in the representation of the value of each variable. The first value is a bound on the contents of the variable introduced by the innermost enclosing $\lambda$-expression. The last corresponds to the value of the outermost visible variable. By construction of the abstract models, expressions get abstractly evaluated in contours that have a length corresponding to that of the lexical environment.

### Syntax

Abstract values and contours are denoted using the syntax of the *modelling [contour] patterns*. We call these *modelling* patterns to distinguish them from the *split patterns* that are introduced later. Figure 5.1 presents the syntax of the modelling patterns. There is a different modelling pattern for each type of abstract value. Also, there is a special pattern that represents all values: $\forall$. There is another special pattern that represents all closures: $\lambda_\forall$. These special patterns mark the limits of the description of the abstract values. For example, pattern $(\#\mathbf{f}, \forall)$ is the notation for the pair mentioned above. The abstract pair contains a Boolean in its CAR-field and contains anything in its CDR-field.

Without the special patterns, the syntax of modelling patterns could only denote concrete values. In order to be able to identify the type of the abstract values, model parameters $\mathsf{cc}$ and $\mathsf{pc}$ are not allowed to return $\forall$ as abstract closure or as abstract pair, respectively. Also, to avoid blending all closures together, parameter $\mathsf{cc}$ is not allowed to return $\lambda_\forall$ as abstract closure.

$$\begin{array}{lcll}
\mathrm{MPat} & := & \forall \mid & \\
 & & \#\mathbf{f} \mid & \\
 & & \lambda_{\forall} \mid & \\
 & & \lambda_l\ k \mid & \text{where } l \in \mathrm{Lab} \text{ and } k \in \mathrm{MCtPat} \\
 & & (P_1,\, P_2) & \text{where } P_1, P_2 \in \mathrm{MPat} \\
\mathrm{MCtPat} & := & (P_1\ \ldots\ P_n) & \text{where } n \geq 0 \text{ and } P_1, \ldots, P_n \in \mathrm{MPat}
\end{array}$$

Figure 5.1: Syntax of the modelling patterns

Modelling *contour* patterns are represented as lists of patterns. They have as many entries as there are variables in the environment of the expressions for which the contours are intended. In particular, the main expression and its immediate sub-expressions get evaluated in the empty contour: $()$. This is normal as there is no variable visible from these expressions. To illustrate the contours let us we consider this partial program:

$$\begin{array}{l}
(_1(\lambda_2\mathrm{x}.\ (_3(\lambda_4\mathrm{y}.\ e_5) \\
\qquad\qquad\quad \ldots)) \\
\quad \ldots)
\end{array}$$

The contours in which $e_5$ is to be evaluated have two entries: the first for the value of 'y', the second for the value of 'x'. For example, a contour indicating that 'x' is a closure and 'y' is a pair looks like:

$$k = ((\forall,\, \forall)\ \lambda_{\forall})$$

By construction of our abstract models, it is guaranteed that a reference to 'x' made in $k$ (i.e. from an expression in $\underline{\triangle}(e_5)$) can only yield closures and a reference to 'y' can only yield pairs.

### Conformance

Modelling patterns denote abstractions of concrete values. Most of the abstract values happen to represent more than one concrete values. When a concrete value is represented by an abstract value, we say that the concrete value *conforms to*, or *is abstracted by*, the abstract value. Here, we give a formal definition of the conformance relation. We use the notation $\nearrow$ (already used in Section 3.5.1) to denote the "is abstracted by" relation. However, we give here a new definition that gives a direct correspondence between concrete values and modelling patterns, without any kind of reference to some analysis results. Figure 5.2

$$
\begin{array}{ll}
\nearrow \;\subseteq \mathrm{Val} \times \mathrm{MPat} & \\
v \nearrow \forall & \\
\#\mathrm{f} \nearrow \#\mathbf{f} & \\
c \nearrow \lambda_{\forall}, & \text{if } c \in \mathrm{ValC} \\
\mathrm{clos}((\lambda_l \mathrm{x}.\ e), \rho) \nearrow \lambda_l\ k, & \text{if } \rho \nearrow_l k \\
\mathrm{pair}(v_1, v_2) \nearrow (P_1, P_2), & \text{if } v_1 \nearrow P_1 \text{ and } v_2 \nearrow P_2 \\
 & \\
\nearrow_l \;\subseteq \mathrm{Env} \times \mathrm{MCtPat}, & l \in \mathrm{Lab} \\
\rho \nearrow_l (), & \text{if } \rho \text{ is valid at label } l \text{ and } \mathrm{Dom}(\rho) = \emptyset \\
\rho \nearrow_l (P_1\ P_2\ \ldots\ P_n), & \text{if } \rho \text{ is valid at label } l, \\
 & \quad \mathrm{x} \text{ is the innermost variable among those in } \mathrm{Dom}(\rho), \\
 & \quad \rho\ \mathrm{x} \nearrow P_1, \\
 & \quad e_{l_1} = (\lambda_{l_1} \mathrm{x}.\ e), \text{ and} \\
 & \quad \rho[\mathrm{x} \mapsto \bot] \nearrow_{l_1} (P_2\ \ldots\ P_n)
\end{array}
$$

Figure 5.2: Definition of the conformance relation

presents the definition of relation $\nearrow$. The correspondence between lexical environments and modelling contour patterns is also presented. In this case, a label must be provided to the relation as an index. We say that an environment is abstracted, at label $l$, by a contour pattern when the values to which variables are bound conform to the corresponding abstract values in the contour. The label is necessary because otherwise the same contour could abstract lexical environments containing bindings for different sets of variables. In Figure 5.2, we use the notation $\rho[\mathrm{x} \mapsto \bot]$ to denote an environment identical to $\rho$ except that the new one is not defined on 'x'. The symbol $\bot$ can be seen as an undefined value.

An extension to the definition of the relation $\nearrow$ that we use later is that of the conformance *between* modelling patterns. We say that $P_1$ *conforms to* $P_2$ when all concrete values that conform to $P_1$ also conform to $P_2$ and we denote it by $P_1 \nearrow P_2$. Verifying the following property about two modelling patterns $P_1$ and $P_2$:

$$
\begin{array}{l}
P_1 \nearrow P_2 \text{ if} \\
\forall v \in \mathrm{Val}.\ v \nearrow P_1 \Rightarrow v \nearrow P_2
\end{array}
$$

is mathematically sound but does not form an algorithm. However, it is easy to present one. The conformance relation between modelling contour patterns is also presented. Technically, the label index is not necessary to compare contour patterns directly anymore because the only requirement on them is to be of the same length. But we keep it to let the notation

$$
\begin{array}{ll}
\nearrow \;\subseteq \text{MPat} \times \text{MPat} & \\
P_1 \nearrow \forall & \\
\#\mathbf{f} \nearrow \#\mathbf{f} & \\
\lambda_\forall \nearrow \lambda_\forall & \\
\lambda_l\ k \nearrow \lambda_\forall & \\
\lambda_l\ k_1 \nearrow \lambda_l\ k_2, & \text{if } k_1 \nearrow_l k_2 \\
(P_1, P_2) \nearrow (P_1', P_2'), & \text{if } P_1 \nearrow P_1' \text{ and } P_2 \nearrow P_2'
\end{array}
$$

$$
\begin{array}{ll}
\nearrow_l \;\subseteq \text{MCtPat} \times \text{MCtPat} \quad l \in \text{Lab} & \\
(P_1\ \ldots\ P_n) \nearrow_l (P_1'\ \ldots\ P_n'), & \text{if there are } n \text{ visible variables at label } l \text{ and} \\
& \quad P_1 \nearrow P_1',\ \ldots,\ P_n \nearrow P_n'
\end{array}
$$

Figure 5.3: Algorithm for the conformance relation between modelling patterns

be consistent with the concrete-abstract case and to keep a connection with the following mathematical definition of conformance between contour patterns:

$$
\begin{array}{l}
(P_1\ \ldots\ P_n) \nearrow_l (P_1'\ \ldots\ P_n') \text{ if} \\
\forall \rho \in \text{Env}.\ \rho \text{ is valid at label } l \Rightarrow \Big(\rho \nearrow_l (P_1\ \ldots\ P_n) \Rightarrow \rho \nearrow_l (P_1'\ \ldots\ P_n')\Big)
\end{array}
$$

Note that we will never have to test conformance between two contours that abstract two incompatible lexical environments, i.e. lexical environments that have different domains. Figure 5.3 presents the formal definition of an algorithm testing the conformance between two modelling patterns. Proving that this definition of conformance is identical to the mathematical definitions is very simple and so we do not make the proof.

### 5.1.2 Models

We build upon the definition of the abstract values and define the abstract models. Abstract models are made of a certain number of *pattern-matchers*. These pattern-matchers regulate the accuracy of the modelling patterns that act as abstract values. As is soon presented, abstract operations on values are performed similarly to concrete operations except that pattern-matchers are used to determine the appropriate level of details in the resulting values. For example, while a concrete pair holding values $v_1$ and $v_2$ is $\text{pair}(v_1, v_2)$, an abstract pair holding values $\hat{v}_1$ and $\hat{v}_2$ is obtained by passing $(\hat{v}_1, \hat{v}_2)$ through a pattern-matcher. The latter may choose to reduce the accuracy in certain points of the new pair.

A theoretical definition of pattern-matchers is first presented. There are many properties to which they must obey. Models are defined using these pattern-matchers. Then, the implementation of the pattern-matchers is described. Finally, the algorithms allowing pattern-matchers (and, consequently, the abstract model) to be updated are presented.

### Theoretical Definition of Pattern-Matchers

The task of a pattern-matcher consists in choosing, for each concrete value $v$, a modelling pattern $P$ that is going to be its corresponding abstract value. Naturally, $P$ must be chosen so that $v$ conforms to it. That is, is has to choose $P$ such that $v \nearrow P$. We define a pattern-matcher to be a set of modelling patterns. For a concrete value $v$ and a pattern-matcher $M$, the abstract value $P$ returned as a representative for $v$ is the element $P \in M$ such that $v \nearrow P$.

Note that we just used the words "*the* abstract value". That means that such an abstract value must be present in $M$. This leads to the following property of pattern-matchers. A correct pattern-matcher has to be *exhaustive*. That is, $M$ is exhaustive if:

$$\forall v \in \text{Val}.\ \exists P \in M.\ v \nearrow P$$

But it is not yet sufficient to allow us to use the words "the abstract value". For most of the values in Val, there is more than one pattern to which it conforms. So there may be more than one $P' \in M$ such that $v \nearrow P'$. So a particular pattern-matcher has to commit to certain patterns so that its results are always unique. That is, it has to be *non-redundant*. Formally, $M$ is exhaustive and non-redundant if:

$$\forall v \in \text{Val}.\ \exists_1 P \in M.\ v \nearrow P$$

The modelling pattern $P$ chosen by the existential quantifier is the abstract value selected by the pattern-matcher to be the abstract representative for $v$.

The preceding example—the construction of an abstract pair—also used a pattern-matcher. However, the pattern-matcher was used on a modelling pattern, not on a concrete value. Normally, it does not make a difference. For a modelling pattern $P$ and pattern-matcher $M$, we simply search for $P' \in M$ such that $P \nearrow P'$. However, if $P$ is not accurate enough, there may be no appropriate $P'$ in $M$. But, as long as $P$ is accurate enough, we can use the pattern-matcher to find an abstract value that abstracts it.

We present a simple example of modelling pattern that is not accurate enough. Let $M$ be:

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

If $P = \forall$, then we cannot select $P' \in M$ such that $P \nearrow P'$. $P$ is too inaccurate. In fact, $\forall$ is the only modelling pattern that is too inaccurate to have an abstract representative in $M$.

Mathematically, selecting an abstract value in a pattern-matcher is equivalent to a *projection*. For $v \in \text{Val} \cup \text{MPat}$, the abstract value $v' \in \text{MPat}$ selected by the pattern-matcher $M$ is usually different from $v$. But if we want to select the abstraction for $v'$, we get $v'$ again. Intuitively, it makes sense as the task of the pattern-matcher is to "erase" unwanted details in values. Once their unwanted details are erased, values do not change anymore if they go through the pattern-matcher again.

Now that we have a precise definition of a pattern-matcher, we can introduce the model. A pattern-based abstract model is built on a group of pattern-matchers: one for the abstract values and the others for the abstract contours. All of them have to be exhaustive and non-redundant. The pattern-matcher projecting the abstract values is used for all three types. It has to be able to project all values in Val. We will usually denote it as $M_\mathcal{V}$. As for the contour pattern-matchers, there is one for each $\lambda$-expression of the program. The contour pattern-matcher $M_l$ selects the contour in which the body of a $\lambda$-expression ($\lambda_l$x. $e_{l_1}$) is to be evaluated when a closure originating from $e_l$ is invoked.

In order to obtain a legal model, $M_\mathcal{V}$ must contain distinct patterns for values of the three types. That is, it cannot be $\{\forall\}$. Also, $M_\mathcal{V}$ must provide distinct abstract closures for each $\lambda$-expression. So this model is not accurate enough:

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

Indeed, blending all closure together would keep our model from being able to feature sets of contours customised for each closure body.

The contour pattern-matchers do not project (simple) modelling patterns, but modelling contour patterns. Pattern-matcher $M_l$ contain contour patterns of length $n$, where $n$ is the number of variables in the environment of the body of $(\lambda_l \mathrm{x}.\ e_{l_1})$. However, what we presented above about a value pattern-matcher applies almost immediately to contour pattern-matchers. The only difference lies in the meaning of the exhaustiveness property. Instead, of being able to project all values in Val, $M_l$ has to be able to project all lists of values of length $n$. There is no minimal accuracy required from the contour pattern-matchers.

An abstract model $\mathcal{M}$ for program $e_{l_0}$ can be built using pattern-matchers provided that there is a value pattern-matcher $M_{\mathcal{V}}$ and one contour pattern-matcher $M_l$ per $\lambda$-expression $e_l$. Each pattern-matcher must be exhaustive and non-redundant. We define each framework parameter in $\mathcal{M}$ as:

- $\mathcal{V}al\mathcal{B}$ is $\{\#\mathbf{f}\}$
- $\mathcal{V}al\mathcal{C}$ is $\{\lambda_l\ k \in M_{\mathcal{V}}\}$
- $\mathcal{V}al\mathcal{P}$ is $\{(P_1,\, P_2) \in M_{\mathcal{V}}\}$
- $\mathcal{C}ont$ is $\bigcup_{l \in L} M_l$ where $L = \{l \in \triangle(e_{l_0}) \mid e_l$ is a $\lambda$-expression$\}$
- $k_0$ is $(\,)$
- $\mathsf{cc}(l, k)$ is the projection by $M_{\mathcal{V}}$ of $\lambda_l\ k$
- $\mathsf{pc}(l, P_1, P_2, k)$ is the projection by $M_{\mathcal{V}}$ of $(P_1,\, P_2)$
- $\mathsf{call}(l_1, \lambda_l\ (P_1\ \ldots\ P_n), P, k)$ is the projection by $M_l$ of $(P\ P_1\ \ldots\ P_n)$

The definition of the first five parameters is straightforward. On the other hand, the definition of the three creation functions deserves some explanation. The cc and pc functions consist in performing a projection on a pattern built in a natural way. Patterns $\lambda_l\ k$ and $(P_1,\, P_2)$, respectively, are both projected using $M_{\mathcal{V}}$. The raw closure $\lambda_l\ k$ contains full contour information. Some of it is forgotten by the projection. Similarly, for the raw pair $(P_1,\, P_2)$. The definition of call summarises well the mechanisms implementing our contour selection strategy. Since contours are abstract representatives for lexical environments, the contour selected for the evaluation of the body of a closure reflects the lexical environment by combining the closure's contour (the abstract closure's definition environment) and the argument in the invocation. The contour that prevails at the site where the invocation occurs is not considered. Neither is the label of the call. The raw contour $(P\ P_1\ \ldots\ P_n)$ is first created and is then projected using the pattern-matcher specialised for invocations of closures of the form '$\lambda_l$ _', that is, $M_l$. Note how the value of the innermost variable, $P$, is inserted at the beginning of the new contour, maintaining the invariant that the value of the variables are listed from the innermost to the outermost.

Before we are done with the theoretical presentation of the pattern-based models, we need to take care of a last problem: that of *consistency* between pattern-matchers. The projection of a concrete value using a pattern-matcher is always possible, as long as the pattern-matcher is exhaustive and non-redundant. However, not all abstract values can be projected using a pattern-matcher, even if the pattern-matcher is exhaustive and non-redundant. Certain abstract values are too inaccurate. That is, too inaccurate in at least some of their sub-components. The problem with abstract values (or contours) that are too inaccurate is that, when they are used to form a raw pattern $P$ and a projection of $P$ is attempted using a pattern-matcher $M$, there may not be any $P' \in M$ such that $P \nearrow P'$. Consequently, the pattern-matchers on which an abstract model is built must represent projections that return values accurate enough to be included into raw patterns and projected again by the same or other pattern-matchers.

Let us give an example of a value pattern-matcher that produces values that are too inaccurate for its own needs. Let $M_{\mathcal{V}}$ be:

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

Normally, it would not be considered as a valid value pattern-matcher because it blends all closures together. But we prefer to keep the example simple as we are not interested by closures here. $M_{\mathcal{V}}$ projects Booleans and closures to their simplest formulation. But it lets pairs have more details. The type of the value in the CDR-field of pairs is explicit and in the case where this value is a pair too, the CAR-field of this internal pair contains an extra level of details. Note also that all abstract pairs have no information about the contents of their CAR-field. However, type information on the value in the CAR-field of pairs is sometimes required during the construction of new pairs. To clearly illustrate when the problem occurs, we consider the following expression:

$$(\texttt{cons}_{22}\ \texttt{\#f}_{23}\ (\texttt{cons}_{24}\ (\lambda_{25}\texttt{x}.\ \texttt{x}_{26})\ \texttt{\#f}_{27}))$$

Concretely evaluating $e_{22}$ in, say, the empty environment '$\cdot$', gives:

$$\mathrm{pair}(\#\mathrm{f},\ \mathrm{pair}(\mathrm{clos}((\lambda_{25}\mathrm{x}.\ \mathrm{x}_{26}),\ \cdot),\ \#\mathrm{f}))$$

Just out of curiosity, we may project this value using pattern-matcher $M_{\mathcal{V}}$ to obtain the corresponding abstract value:

$$(\forall,\ (\lambda_{\forall},\ \forall))$$

However, during an abstract evaluation of $e_{22}$, the abstract value must be built step by step. So the creation of the inner pair (evaluation of $e_{24}$) produces the following raw and projected pattern:

$$(\lambda_{25}\ (),\ \#\mathbf{f})\ \overset{M_{\mathcal{V}}}{\mapsto}\ (\forall,\ \#\mathbf{f})$$

The creation of the outer pair (evaluation of $e_{22}$) produces:

$$(\#\mathbf{f},\ (\forall,\ \#\mathbf{f}))\ \overset{M_{\mathcal{V}}}{\mapsto}\ ?$$

The projection cannot be done because the internal abstract pair is not accurate enough to be handled by $M_{\mathcal{V}}$.

This was an example of the value pattern-matcher not being consistent with itself. But to obtain a valid pattern-based model, it is not only necessary for $M_{\mathcal{V}}$ to produce abstract values accurate enough for its own needs, but it must do the same for the needs of each $M_l$, and each $M_l$ must produce contours accurate enough for the needs of $M_{\mathcal{V}}$. Indeed, by the definition of the creation functions $\mathsf{cc}$ and $\mathsf{pc}$, $M_{\mathcal{V}}$ is used to project raw patterns containing abstract values and contours coming from itself and from the different $M_l$, respectively. And by the definition of the selection function $\mathsf{call}$, all the $M_l$ are used to project raw contour patterns containing abstract values coming from $M_{\mathcal{V}}$ (and from closure contours coming themselves from $M_{\mathcal{V}}$). So the implementation of abstract models has to ensure that consistency is maintained between the pattern-matchers after each model update.

### Implementation of Pattern-Matchers

The mathematical definition of pattern-matchers is simple and precise but if pattern-matchers were directly implemented this way, projections would be rather inefficient. So, instead, pattern-matchers are implemented as decision trees. A fast traversal of a pattern allows the pattern-matcher to determine what the result of the projection is. The traversal takes linear time in the size of the pattern. More precisely, it takes time linear in the size of the *inspected* part of the pattern. Some sub-parts of a pattern need not be inspected as they are matched to the sub-pattern $\forall$ in the pattern-matcher.

We have adopted a *breadth-first* traversal of the patterns. Note that a depth-first traversal would work too. In fact, any valid order would work; as long as any part of the pattern is inspected before its sub-parts are. But we have a reason to prefer the breadth-first traversal.

During a demand-driven analysis, there typically are considerable differences in the level of details needed in some values (or contours) compared to that in other values. For example, the pairs having a non-pair in the CDR-field may be uninteresting for the analysis while those having a pair in the CDR-field may become very detailed in both fields. When coarse and detailed values coexist, the point at which there is a distinction between the two kinds of values occurs at a low depth in the pattern (because a coarse value is not very deep, to start with). So, in order to avoid considering unnecessary details in the uninteresting values, the inspection of the distinguishing point should appear as high as possible in the decision tree. When a breadth-first traversal is used, this point cannot appear below a certain depth because traversing all levels above the point can only introduce a bounded number of stages in the decision tree. On the other hand, if a depth-first traversal is used, an arbitrary number of points may have to be inspected before the distinguishing point is reached. This is because full-detail inspection is necessary as long as the point distinguishing uninteresting and interesting values is not met. To come back to the example, a decision tree performing a depth-first traversal of the pairs would have to inspect the value in the CAR-field with full precision in the eventuality that the pairs are interesting, i.e. in the eventuality that the CDR-field contains a pair. Complete traversal of the value in the CAR-field may be arbitrarily long. So a depth-first traversal may lead to a decision tree that is exaggerately big if a bad case occurs.

Now, let us describe the data structures used to implement the pattern-matchers. First,

$$
\begin{array}{lll}
\mathrm{PM} & := \mathrm{PM}_O \mid \mathrm{PM}_C \mid \mathrm{PM}_L & \\
\mathrm{PM}_O & := \texttt{Onode}\ [\mathcal{V}al \Rightarrow M] \mid & \text{where } M \in \mathrm{PM} \\
 & \phantom{:=}\ \texttt{Onode}\ [\mathcal{V}al\mathcal{B} \Rightarrow M_1,\ \mathcal{V}al\mathcal{C} \Rightarrow M_2,\ \mathcal{V}al\mathcal{P} \Rightarrow M_3] & \\
 & & \text{where } M_1, M_2, M_3 \in \mathrm{PM} \\
\mathrm{PM}_C & := \texttt{Cnode}\ [\mathrm{Lab} \Rightarrow M] \mid & \text{where } M \in \mathrm{PM} \\
 & \phantom{:=}\ \texttt{Cnode}\ [l_1 \Rightarrow M_1,\ \ldots,\ l_n \Rightarrow M_n] & \text{where } M_1, \ldots, M_n \in \mathrm{PM} \\
 & & \text{and } \{l_1,\ \ldots,\ l_n\} = \\
 & & \quad \{l \in \mathrm{Lab} \mid e_l \text{ is a } \lambda\text{-expr.}\} \\
\mathrm{PM}_L & := \texttt{Leaf}\ P & \text{where } P \in \mathrm{MPat} \cup \mathrm{MCtPat}
\end{array}
$$

Figure 5.4: Implementation of the pattern-matchers

the decision tree is made of inspection nodes—the internal nodes—and of result nodes—the leafs. The leafs contain the results of the projection of raw patterns. There are two kinds of inspection nodes: the object nodes and the closures nodes. The two kinds of inspection nodes both come in two variants: the blind variant and the discriminating variant. An object node expects a value and (possibly) discriminates on the type of the value. A closure node expects a closure and (possibly) discriminates on the label attached to the closure. A blind variant does not inspect its corresponding sub-pattern and has a single sub-tree. A discriminating variant inspects its corresponding sub-pattern and dispatches the remainder of the traversal to one of its sub-trees depending on the type or the label.

Figure 5.4 presents the data structures used to represent decision trees. The inspection nodes are built with a constructor that indicates if they are object or closure nodes. Then a list of alternatives follows. We believe that the notation for the alternatives speaks for itself. The leaf nodes contain modelling patterns or modelling contour patterns, depending on whether they are part of a value or contour pattern-matcher, respectively.

The breadth-first traversal of data structures typically requires a queue to temporarily hold the sub-structures until they are traversed. It is the case for the traversal of patterns. The general treatment for a pattern depends on the inspection node variant that is inspecting it. When the inspection node is blind, the pattern is simply extracted from the queue. When the inspection node is discriminating, the pattern is extracted from the queue *and* then its sub-patterns, when they exist, are inserted in the queue for future inspection. One may have noted that the data structures used to implement the pattern-matchers do not include nodes to inspect contour patterns explicitly. The inspection of contours always starts by breaking them into the individual values they contain and inserting each value one after the

$$
\begin{array}{lrl}
\multicolumn{3}{l}{\mathrm{pm} : \mathrm{PM} \times \langle \text{queue of MPat} \rangle \to \mathrm{MPat} \cup \mathrm{MCtPat}} \\
\mathrm{pm}(\texttt{Onode}\ [\mathcal{V}al \Rightarrow M], & P \triangleleft q) & = \mathrm{pm}(M,\ \ q) \\
\mathrm{pm}(\texttt{Onode}\ [\mathcal{V}al\mathcal{B} \Rightarrow M_1,\ \ldots], & \#\mathbf{f} \triangleleft q) & = \mathrm{pm}(M_1,\ q) \\
\mathrm{pm}(\texttt{Onode}\ [\ldots,\ \mathcal{V}al\mathcal{C} \Rightarrow M_2,\ \ldots], & P \triangleleft q) & = \mathrm{pm}(M_2,\ q \triangleleft P), \\
 & & \quad \text{if } P = \lambda_\forall \text{ or } P = \lambda_l\ (P_1\ \ldots\ P_n) \\
\mathrm{pm}(\texttt{Onode}\ [\ldots,\ \mathcal{V}al\mathcal{P} \Rightarrow M_3], & (P_1,\ P_2) \triangleleft q) & = \mathrm{pm}(M_3,\ q \triangleleft P_1 \triangleleft P_2) \\
\mathrm{pm}(\texttt{Cnode}\ [\mathrm{Lab} \Rightarrow M], & P \triangleleft q) & = \mathrm{pm}(M,\ \ q) \\
\mathrm{pm}(\texttt{Cnode}\ [\ldots,\ l_i \Rightarrow M_i,\ \ldots],\ \lambda_{l_i}\ (P_1\ \ldots\ P_n) & \triangleleft q) & = \mathrm{pm}(M_i,\ q \triangleleft P_1 \triangleleft \ldots \triangleleft P_n) \\
\mathrm{pm}(\texttt{Leaf}\ P, & [\,]) & = P
\end{array}
$$

Figure 5.5: Algorithm for pattern-matching

other in the queue.

Queues are denoted using square brackets and queue elements are separated by commas. Insertion is performed to the right and extraction, to the left of queues. The projection of (simple) modelling pattern $P$ is done by using $[P]$ as an initial queue. The projection is done by computing:

$$\mathrm{pm}(M_{\mathcal{V}},\ [P])$$

where 'pm' is the pattern-matching function. The projection of modelling contour pattern $(P_1\ \ldots\ P_n)$ using pattern-matcher $M_l$ is done by computing:

$$\mathrm{pm}(M_l,\ [P_1,\ \ldots,\ P_n])$$

Figure 5.5 presents the algorithm that performs projections using the pattern-matchers. It takes a pattern-matcher node and a queue of values as arguments. To make the algorithm easier to read, we use a *view*[1] on queues, denoted by '$\triangleleft$', to indicate both insertions and extractions. As usual, insertions are done to the right of queues and extractions, to the left.

The object nodes accept all kinds of modelling patterns. But the closure nodes expect only modelling patterns of closures. By construction of the pattern-matchers, a closure is

[1] A view is an implicit transformation that is performed on data structures to present them under a different aspect, or point of view, that is more helpful. Views are used both in pattern-matching and in the construction of values. Here is an example using a Haskell-like syntax. We can extract the first two elements of a list along with the rest of the list using the pattern `a:b:xs`. However, if the real intent was to obtain a list of the first two elements and the rest of the list, the use of the concatenation view, `++`, in pattern `[a,b]++xs`, would be more natural. In the first pattern, it is the real constructor that is used to perform the pattern-matching. But in the second, a fictitious but more convenient representation of the values is obtained by the use of the `++` view.

always the next pattern to extract from the queue each time pattern-matching goes through a closure node. Also, a leaf always coincide with an emptied queue. This is guaranteed by construction of the pattern-matchers.

Let us have a closer look at the algorithm. When an object node is reached, four cases are possible: the node is blind or the node is discriminating and the extracted pattern is that of a Boolean, that of a closure, or that of a pair. A blind node simply discards the pattern. In the other cases, sub-patterns, if they exist, are inserted back into the queue. In the Boolean case, there is no sub-pattern to insert back. In the closure case, the whole closure is inserted back for future examination by a closure node. The inspection of the label of the closure, if it occurs at all, is considered to be an operation done deeper by one level in the pattern than the inspection of its type. In the pair case, both sub-patterns are inserted back into the queue.

When a closure node is reached, there are only two cases: the node is blind or it is discriminating on the label of the $\lambda$-expression from which the extracted closure originates. When the node is blind, the label and the whole closure are not considered and the closure is discarded. When the node is discriminating, there is a case for each $\lambda$-expression label. The contour of the closure is broken and the values it contains are inserted into the queue, from the first to the last.

When a leaf is reached, the result of the projection is simply extracted from the leaf and returned.

The presented data structures and pattern-matching algorithm provide an implementation for the abstract models that is relatively fast. The raw patterns that must be projected because of the use of creation functions of the model can be processed in linear time with the size of the part of the pattern that is inspected by the decision trees.

### Model Updates

An update of the model consists in changing one or a few of the pattern-matchers to make them more accurate. That is, more accurate with respect to the projection results and more accurate with respect to their inspection of the projected patterns. A single model update may require more than one change to the same pattern-matcher. Typically, this is the case for $M_{\mathcal{V}}$. The changes to the pattern-matchers must be done with care. In particular, the

$$
\begin{array}{lll}
\text{SPat} & := \star \mid & \\
 & \lambda_\star \mid & \\
 & \lambda_l \; k \mid & \text{where } l \in \text{Lab and } k \in \text{SCtPat} \\
 & (P_1,\, P_2) \mid & \text{where } P_1 \in \text{SPat and } P_2 \in \text{MPat} \\
 & (P_1,\, P_2) & \text{where } P_1 \in \text{MPat and } P_2 \in \text{SPat} \\
\text{SCtPat} & := (P_1 \; \ldots \; P_{i-1} \; P_i \; P_{i+1} \; \ldots \; P_n) & \text{where } P_i \in \text{SPat} \\
 & & \text{and } \forall j \in \{1, \ldots, n\} - \{i\}.\; P_j \in \text{MPat}
\end{array}
$$

Figure 5.6: Syntax of the split patterns

pattern-matchers must stay consistent with the others. However, a systematic updating procedure that ensures that the updates are done properly is relatively easy to elaborate.

The first tools we need to describe are the *split patterns* and the *split contour patterns*. The split patterns specify a point in the abstract values or abstract contours where an increase in precision is sought. Split patterns are usually generated by the processing of demands. Figure 5.6 presents their syntax. The '$\star$' sign is called the split point. Every split (contour) pattern contains exactly one split point. Normally, the split point causes the values to be more accurate by one extra level. But, in order to stay as general as possible, we do not rely on that supposition.

The syntax of the split patterns allows one to indicate which abstract values should be affected by the update. For example, the following two patterns are not equivalent:

$$(\lambda_\forall,\, \star) \qquad (\forall,\, \star)$$

Both ask for additional accuracy in the representation of the value in the CDR-field of pairs. But the first asks for additional accuracy only for the pairs that have a closure in their CAR-field while the second asks it for all pairs. Naturally, more complex restrictions can be expressed using the "modelling part" of the patterns. However, there are limitations related to the fact that patterns are traversed in a breadth-first manner. For example, the following patterns describe the same split:

$$\lambda_{12}\ ((\lambda_\forall,\, \#\mathbf{f})\ \star) \qquad \lambda_{12}\ ((\forall,\, \forall)\ \star)$$

because the node affected by the split point is higher in the decision tree than those corresponding to the fields of the pair. So, the choice between blindness and discrimination for the higher node cannot depend on the path that is followed in lower nodes. Even if, concretely, updates are performed directly on decision trees, here, we prefer to describe the update process while considering pattern-matchers to be sets.

The model update requests take the form of one or more *pattern-matcher update requests*. Pattern-matcher update requests are denoted using the syntax:

$$\text{update } M \text{ with } P \quad \text{where } M \in \text{PM and } P \in \text{SPat}$$

In fact, the *name* of the pattern matcher is important. The reasons are presented later. As an example, the processing of demands might generate the following pattern-matcher update requests:

$$\begin{array}{l} \text{update } M_2 \text{ with } (\star) \\ \text{update } M_{\mathcal{V}} \text{ with } \lambda_4\ (\star) \\ \text{update } M_4 \text{ with } (\#\mathbf{f}\ \star) \end{array}$$

Each pattern-matcher update request can be processed individually.

The first step in the processing of a pattern-matcher update request like:

$$\text{update } M_{\mathcal{V}} \text{ with } P \quad \text{or} \quad \text{update } M_l \text{ with } k$$

consists in simplifying the pattern $P$—or $k$. Unnecessary details ought to be removed from the pattern since they do not have an influence on the signification of the pattern. We illustrate the simplification by using once again the above example:

$$\lambda_{12}\ ((\lambda_\forall,\ \#\mathbf{f})\ \star) \quad \mapsto \quad \lambda_{12}\ ((\forall,\ \forall)\ \star)$$

The implementation of the simplification is relatively simple. It only requires a breadth-first traversal of the pattern. The elements of the pattern are noted. Since there is no decision tree to guide the traversal, explicit markers are manipulated along with the sub-patterns. The '`O`' and '`C`' markers indicate object and closure inspections, respectively. When the split point is found, the rest of the traversal operates an *erasure* of the remaining sub-patterns. The top of the pattern is then rebuilt on top of these simplified sub-patterns. Figure 5.7 presents the algorithm formally. 'S' is an overloaded function that simplifies both split patterns and split contour patterns. Function '$\text{S}_Q$' deconstructs and reconstructs the higher parts of the pattern. Function '$\text{S}^\star_Q$' is the detail-erasure operation. Note that a queue is

$$
\begin{array}{lcl}
\mathrm{S} : \mathrm{SPat} \rightarrow \mathrm{SPat} & & \\
\mathrm{S}(P) & = & P' \\
 & & \text{where } P' \triangleleft [\,] = \mathrm{S}_Q([\,] \triangleleft \mathtt{O}\ P)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{S} : \mathrm{SCtPat} \rightarrow \mathrm{SCtPat} & & \\
\mathrm{S}((P_1\ \ldots\ P_n)) & = & (P'_1\ \ldots\ P'_n) \\
 & & \text{where } P'_n \triangleleft \ldots \triangleleft P'_1 \triangleleft [\,] = \mathrm{S}_Q([\,] \triangleleft \mathtt{O}\ P_1 \triangleleft \ldots \triangleleft \mathtt{O}\ P_n)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{S}_Q : \langle \text{queue of } \{\mathtt{O}, \mathtt{C}\} \times (\mathrm{SPat} \cup \mathrm{MPat})\rangle \rightarrow \langle \text{queue of } (\mathrm{SPat} \cup \mathrm{MPat})\rangle & & \\
\mathrm{S}_Q(\mathtt{O}\ \forall \triangleleft q) & = & \mathrm{S}_Q(q) \triangleleft \forall \\
\mathrm{S}_Q(\mathtt{O}\ \star \triangleleft q) & = & \mathrm{S}^{\star}_Q(q) \triangleleft \star \\
\mathrm{S}_Q(\mathtt{O}\ \#\mathbf{f} \triangleleft q) & = & \mathrm{S}_Q(q) \triangleleft \#\mathbf{f} \\
\mathrm{S}_Q(\mathtt{O}\ P \triangleleft q) & = & q' \triangleleft P', \quad \text{if } P \text{ is } \lambda_{\forall},\ \lambda_{\star}, \text{ or } \lambda_l\ k \\
 & & \text{where } P' \triangleleft q' = \mathrm{S}_Q(q \triangleleft \mathtt{C}\ P) \\
\mathrm{S}_Q(\mathtt{O}\ (P_1, P_2) \triangleleft q) & = & q' \triangleleft (P'_1, P'_2) \\
 & & \text{where } P'_2 \triangleleft P'_1 \triangleleft q' = \mathrm{S}_Q(q \triangleleft \mathtt{O}\ P_1 \triangleleft \mathtt{O}\ P_2) \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_{\forall} \triangleleft q) & = & \mathrm{S}_Q(q) \triangleleft \lambda_{\forall} \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_{\star} \triangleleft q) & = & \mathrm{S}^{\star}_Q(q) \triangleleft \lambda_{\star} \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_l\ (P_1\ \ldots\ P_n) \triangleleft q) & = & q' \triangleleft \lambda_l\ (P'_1\ \ldots\ P'_n) \\
 & & \text{where } P'_n \triangleleft \ldots \triangleleft P'_1 \triangleleft q' = \mathrm{S}_Q(q \triangleleft P_1 \triangleleft \ldots \triangleleft P_n)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{S}^{\star}_Q : \langle \text{queue of } \{\mathtt{O}, \mathtt{C}\} \times \mathrm{MPat}\rangle \rightarrow \langle \text{queue of } \mathrm{MPat}\rangle & & \\
\mathrm{S}^{\star}_Q(\mathtt{O}\ P \triangleleft q) & = & \mathrm{S}^{\star}_Q(q) \triangleleft \forall \\
\mathrm{S}^{\star}_Q(\mathtt{C}\ P \triangleleft q) & = & \mathrm{S}^{\star}_Q(q) \triangleleft \lambda_{\forall} \\
\mathrm{S}^{\star}_Q([\,]) & = & [\,]
\end{array}
$$

Figure 5.7: Simplification of split patterns

used for the deconstruction of the pattern and another for the reconstruction. The order of the sub-patterns in the reconstruction queue is reversed. Figure 5.8 shows a trace of the simplification of the above example.

The next step in the processing of a pattern-matcher update request consists in ensuring that the pattern-matchers remain consistent. Updating a certain pattern-matcher may lead to a cascade of updates. This is because the updated values may be created by projecting raw patterns obtained from other values and these other values might not be accurate enough yet. Figure 5.9 shows the rules that are used to generate new update requests from complex ones in order to maintain consistency. The new requests have to go through the rules themselves, and so on, until a base case is reached. The set of requests obtained this way can be processed in any order by the third step: the model could be in an inconsistent state during the update, but once all the pattern-matcher update requests are achieved,

$$
\begin{array}{l}
\mathrm{S}(\lambda_{12}\ ((\lambda_{\forall},\ \#\mathbf{f})\ \star))\\
\quad \mathrm{S}_Q([\texttt{0}\ \lambda_{12}\ ((\lambda_{\forall},\ \#\mathbf{f})\ \star)])\\
\quad\quad \mathrm{S}_Q([\texttt{C}\ \lambda_{12}\ ((\lambda_{\forall},\ \#\mathbf{f})\ \star)])\\
\quad\quad\quad \mathrm{S}_Q([\texttt{0}\ (\lambda_{\forall},\ \#\mathbf{f}), \texttt{0}\ \star])\\
\quad\quad\quad\quad \mathrm{S}_Q([\texttt{0}\ \star, \texttt{0}\ \lambda_{\forall}, \texttt{0}\ \#\mathbf{f}])\\
\quad\quad\quad\quad\quad \mathrm{S}^{\star}_Q([\texttt{0}\ \lambda_{\forall}, \texttt{0}\ \#\mathbf{f}])\\
\quad\quad\quad\quad\quad\quad \mathrm{S}^{\star}_Q([\texttt{0}\ \#\mathbf{f}])\\
\quad\quad\quad\quad\quad\quad\quad \mathrm{S}^{\star}_Q([\,])\\
\quad\quad\quad\quad\quad\quad\quad \Rightarrow [\,]\\
\quad\quad\quad\quad\quad\quad \Rightarrow [\forall]\\
\quad\quad\quad\quad\quad \Rightarrow [\forall, \forall]\\
\quad\quad\quad\quad \Rightarrow [\forall, \forall, \star]\\
\quad\quad\quad \Rightarrow [\star, (\forall,\ \forall)]\\
\quad\quad \Rightarrow [\lambda_{12}\ ((\forall,\ \forall)\ \star)]\\
\quad \Rightarrow [\lambda_{12}\ ((\forall,\ \forall)\ \star)]\\
\Rightarrow \lambda_{12}\ ((\forall,\ \forall)\ \star)
\end{array}
$$

Figure 5.8: Example of simplification of a split pattern

$$
\begin{array}{llll}
\text{update } M_{\curlyvee} \text{ with } \star & \rightarrow & \textsc{none} & \\
\text{update } M_{\curlyvee} \text{ with } \lambda_{\star} & \rightarrow & \textsc{none} & \\
\text{update } M_{\curlyvee} \text{ with } \lambda_l\ (P_1\ P_2\ \ldots\ P_n) & \rightarrow & \text{update } M_{l'} \text{ with } (P_1\ P_2\ \ldots\ P_n) & \\
 & & \text{where } e_{l'} = (\lambda_{l'}\text{x. } e_{l''}) \text{ and } l \in \underline{\triangle}(l'') & \\
\text{update } M_{\curlyvee} \text{ with } (P_1,\ P_2) & \rightarrow & \text{update } M_{\curlyvee} \text{ with } P_1, & \text{if } P_1 \in \mathrm{SPat}\\
\text{update } M_{\curlyvee} \text{ with } (P_1,\ P_2) & \rightarrow & \text{update } M_{\curlyvee} \text{ with } P_2, & \text{if } P_2 \in \mathrm{SPat}\\
\text{update } M_l \text{ with } (P_1\ P_2\ \ldots\ P_n) & \rightarrow & \text{update } M_{\curlyvee} \text{ with } P_1, & \text{if } P_1 \in \mathrm{SPat}\\
\text{update } M_l \text{ with } (P_1\ P_2\ \ldots\ P_n) & \rightarrow & \text{update } M_{\curlyvee} \text{ with } \lambda_l\ (P_2\ \ldots\ P_n), & \text{if } P_1 \notin \mathrm{SPat}
\end{array}
$$

Figure 5.9: Generation of pattern-matcher update requests to ensure consistency

it becomes consistent again. Figure 5.10 continues the example of Figure 5.8 and lists the pattern-matcher update requests that ensure a consistent update of the model. The example supposes that $\lambda$-expression $e_{12}$ is an immediate sub-expression of $\lambda$-expression $e_7$ and that $\lambda$-expression $e_7$ is an immediate sub-expression of $\lambda$-expression $e_3$.

The third step is the *slicing* of patterns. The pattern in a pattern-matcher update request may be asking for an increase in accuracy that adds more than one extra level in the concerned abstract values. To avoid manipulating complex update situations, we perform slicing on the pattern. The slicing of a pattern transform it in a sequence of patterns of increasing accuracy. It makes sure that nodes in the decision tree are upgraded from the

$$\begin{array}{rl} & \text{update } M_{\mathcal{V}} \text{ with } \lambda_{12}\ ((\forall,\, \forall)\ \star) \\ \to & \text{update } M_7 \text{ with } ((\forall,\, \forall)\ \star) \\ \to & \text{update } M_{\mathcal{V}} \text{ with } \lambda_7\ (\star) \\ \to & \text{update } M_3 \text{ with } (\star) \\ \to & \text{update } M_{\mathcal{V}} \text{ with } \star \end{array}$$

Figure 5.10: Example of an update request and the sub-requests generated for consistency

highest to the lowest. Visually, the sequence of patterns have a split point that moves further away from the top when we consider them in the order they appear in the sequence. The split point moves further away from the top in a breadth-first order. Figure 5.11 presents the slicing algorithm formally. The sequence that is produced by the algorithm is denoted using the syntax of queues. However, insertions to the front of the sequence are done using the '$\triangleright$' operator. The 'map' function is the usual function. It takes a function and a sequence as arguments and applies the function to each element of the sequence, producing a new sequence. On the contrary of to the preceding step, the patterns obtained with the slicing algorithm are ordered and the order must be respected. Figure 5.12 shows a split pattern and the sequence of patterns obtained by slicing it. Notice how a split point is inserted for each point of the sub-pattern that is not a "universal" one, i.e. $\forall$ or $\lambda_\forall$.

The fourth step consists in applying the simplified, consistent, and sliced patterns to the pattern-matchers. Let us consider a pattern-matcher update request like:

$$\text{update } M_{\mathcal{V}} \text{ with } P \quad \text{or} \quad \text{update } M_l \text{ with } k$$

We suppose that the order among the sliced patterns produced by the slicing algorithm is respected. The upgrade may not even be necessary if the node to upgrade is already discriminating. Even if upgrades are actually performed on decision trees, we prefer to present upgrades on set-based pattern-matchers to keep things simpler. Also, we give the textual explanations only for an update of $M_{\mathcal{V}}$. The operations are almost identical in the case of the update of $M_l$.

When we update $M_{\mathcal{V}}$ with a split pattern $P$, some of the modelling patterns in $M_{\mathcal{V}}$ change while others do not. So we need a tool to decide which modelling patterns are affected by $P$. To fulfil our needs, we extend the $\nearrow$ relation to make it able to test if a modelling pattern conforms to a split pattern. If we decree that '$\star$' is equivalent to $\forall$

$$\begin{array}{lcl}
\mathrm{S} : \mathrm{SPat} \to \langle \text{sequence of SPat} \rangle & & \\
\mathrm{S}(P) & = & \mathrm{map}\ (\lambda\,(P' \triangleleft [\,]).\ P')\ \sigma \\
 & & \text{where } (\_,\ \sigma) = \mathrm{S}_Q([\,] \triangleleft P)
\end{array}$$

$$\begin{array}{lcl}
\mathrm{S} : \mathrm{SCtPat} \to \langle \text{sequence of SCtPat} \rangle & & \\
\mathrm{S}((P_1\ \ldots\ P_n)) & = & \mathrm{map}\ (\lambda\,(P'_n \triangleleft \ldots \triangleleft P'_1 \triangleleft [\,]).\ (P'_1\ \ldots\ P'_n))\ \sigma \\
 & & \text{where } (\_,\ \sigma) = \mathrm{S}_Q([\,] \triangleleft P_1 \triangleleft \ldots \triangleleft P_n)
\end{array}$$

$$\begin{array}{lcl}
\mathrm{S}_Q : \langle \text{queue of } \{\mathtt{O}, \mathtt{C}\} \times (\mathrm{SPat} \cup \mathrm{MPat}) \rangle \to & & \\
\quad \langle \text{queue of MPat} \rangle \times \langle \text{sequence of } \langle \text{queue of } (\mathrm{SPat} \cup \mathrm{MPat}) \rangle \rangle & & \\
\mathrm{S}_Q(\mathtt{O}\ \forall \triangleleft q) & = & (q' \triangleleft \forall,\ \mathrm{map}\ (\lambda\, q.\ q \triangleleft \forall)\ \sigma) \\
 & & \text{where } (q',\ \sigma) = \mathrm{S}_Q(q) \\
\mathrm{S}_Q(\mathtt{O}\ \star \triangleleft q) & = & (q' \triangleleft \forall,\ (q' \triangleleft \star) \triangleright [\,]) \\
 & & \text{where } (q',\ [\,]) = \mathrm{S}_Q(q) \\
\mathrm{S}_Q(\mathtt{O}\ \#\mathbf{f} \triangleleft q) & = & (q' \triangleleft \forall,\ (q' \triangleleft \star) \triangleright (\mathrm{map}\ (\lambda\, q.\ q \triangleleft \#\mathbf{f})\ \sigma)) \\
 & & \text{where } (q',\ \sigma) = \mathrm{S}_Q(q) \\
\mathrm{S}_Q(\mathtt{O}\ P \triangleleft q) & = & (q' \triangleleft \forall,\ (q' \triangleleft \star) \triangleright (\mathrm{map}\ (\lambda\,(P' \triangleleft q).\ q \triangleleft P')\ \sigma)), \\
 & & \text{if } P \text{ is } \lambda_\forall,\ \lambda_\star, \text{ or } \lambda_l\ k \\
 & & \text{where } (\lambda_\forall \triangleleft q',\ \sigma) = \mathrm{S}_Q(q \triangleleft \mathtt{C}\ P) \\
\mathrm{S}_Q(\mathtt{O}\ (P_1,\ P_2) \triangleleft q) & = & (q' \triangleleft \forall,\ (q' \triangleleft \star) \triangleright (\mathrm{map}\ (\lambda\,(P'_2 \triangleleft P'_1 \triangleleft q).\ q \triangleleft (P'_1,\ P'_2))\ \sigma)) \\
 & & \text{where } (\forall \triangleleft \forall \triangleleft q',\ \sigma) = \mathrm{S}_Q(q \triangleleft \mathtt{O}\ P_1 \triangleleft \mathtt{O}\ P_2) \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_\forall \triangleleft q) & = & (q' \triangleleft \lambda_\forall,\ \mathrm{map}\ (\lambda\, q.\ q \triangleleft \lambda_\forall)\ \sigma) \\
 & & \text{where } (q',\ \sigma) = \mathrm{S}_Q(q) \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_\star \triangleleft q) & = & (q' \triangleleft \lambda_\forall,\ (q' \triangleleft \lambda_\star) \triangleright [\,]) \\
 & & \text{where } (q',\ [\,]) = \mathrm{S}_Q(q) \\
\mathrm{S}_Q(\mathtt{C}\ \lambda_l\ (P_1\ \ldots\ P_n) \triangleleft q) & = & (q' \triangleleft \lambda_\forall, \\
 & & \ (q' \triangleleft \lambda_\star) \triangleright \\
 & & \ (\mathrm{map}\ (\lambda\,(P'_n \triangleleft \ldots \triangleleft P'_1 \triangleleft q).\ q \triangleleft \lambda_l\ (P'_1\ \ldots\ P'_n))\ \sigma)) \\
 & & \text{where } (\underbrace{\forall \triangleleft \ldots \triangleleft \forall \triangleleft}_{n \text{ times}} q',\ \sigma) = \mathrm{S}_Q(q \triangleleft P_1 \triangleleft \ldots \triangleleft P_n) \\
\mathrm{S}_Q([\,]) & = & ([\,],\ [\,])
\end{array}$$

Figure 5.11: Slicing of split patterns

$$\mathrm{S}\,(\,(\lambda_{12}\ (\star),\ (\#\mathbf{f},\ \forall))\,) =$$

$$\left[\begin{array}{ccc}
\star\ , & (\star,\ \forall)\ , & (\lambda_\forall,\ \star)\ , \\
(\lambda_\star,\ (\forall,\ \forall))\ , & (\lambda_{12}\ (\forall),\ (\star,\ \forall))\ , & (\lambda_{12}\ (\star),\ (\#\mathbf{f},\ \forall))
\end{array}\right]$$

Figure 5.12: Example of the slicing of a split pattern

$$\begin{array}{ll}
\nearrow\ \subseteq \mathrm{MPat} \times (\mathrm{MPat} \cup \mathrm{SPat}) & \\
\ldots & \text{(rules for the MPat} \times \text{MPat cases)} \\
P \nearrow \star & \\
\lambda_\forall \nearrow \lambda_\star & \\
\lambda_l\ k \nearrow \lambda_\star & \\
\lambda_l\ k_1 \nearrow \lambda_l\ k_2, & \text{if } k_1 \nearrow_l k_2 \\
(P_1, P_2) \nearrow (P_1', P_2'), & \text{if } P_1 \nearrow P_1' \text{ and } P_2 \nearrow P_2'
\end{array}$$

$$\begin{array}{ll}
\nearrow_l\ \subseteq \mathrm{MCtPat} \times (\mathrm{MCtPat} \cup \mathrm{SCtPat}) & l \in \mathrm{Lab} \\
\ldots & \text{(rule for the MCtPat} \times \text{MCtPat cases)} \\
(P_1\ \ldots\ P_n) \nearrow_l (P_1'\ \ldots\ P_n'), & \text{if these are } n \text{ visible variables at label } l \text{ and} \\
 & \quad P_1 \nearrow P_1',\ \ldots,\ P_n \nearrow P_n'
\end{array}$$

Figure 5.13: Extension of the definition of conformance between modelling and split patterns

(and that '$\lambda_\star$' is equivalent to '$\lambda_\forall$'), then, by the slicing, we can always decide whether each pattern in $M_{\mathcal{V}}$ conforms to $P$. Indeed, the only point in which conforming patterns in $M_{\mathcal{V}}$ may not be as precise as $P$ is exactly at the split point. The decree is reasonable as, although '$\star$' asks for higher accuracy, it has not committed to a particular choice among the available options. Figure 5.13 presents the extension to $\nearrow$. The extension makes use of the previous definition without any special indication.

With the help of the conformance relation, it is now easy to express the algorithm that upgrades the inspection points. Figure 5.14 presents the algorithm. A pattern-matcher update request 'update $M_{\mathcal{V}}$ with $P$' is performed by the (overloaded) function '$U$' and the resulting pattern-matcher is $U(M_{\mathcal{V}}, P)$. Basically, each modelling pattern in $M_{\mathcal{V}}$ is first tested for conformance to the split pattern. If it is conforming, it is "exploded" into more accurate modelling patterns, if it is not already accurate enough. When $\forall$ is exploded, it provides the basic patterns of the three types. When '$\lambda_\forall$' is exploded, it provides the basic patterns of all closures of the program. The program is assumed to be $e_{l_0}$. Note that some cases are not treated by $\sqcap$. This is because of the conformance test previously made: the definition of $\sqcap$ contains only the possible cases.

A small example of upgrading is presented in Figure 5.15. The value pattern-matcher for a little program is upgraded using pattern $P = \lambda_{10}\ (\forall\ \star)$. The current state of $M_{\mathcal{V}}$ is shown. Note that $M_{\mathcal{V}}$ is precise enough to be ready to be split using $P$. This is always the case because of the slicing of split patterns. We suppose that the program has two $\lambda$-expressions,

$$
\begin{array}{l}
U : \mathrm{PM} \times \mathrm{SPat} \\
U(M_{\mathcal{V}}, P) \quad = \quad \displaystyle\bigcup_{P' \in M_{\mathcal{V}}} \left( \begin{array}{r} \text{if } P' \nearrow P \text{ then } P' \sqcap P \\ \text{else } \ \{P'\} \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
U : \mathrm{PM} \times \mathrm{SCtPat} \\
U(M_l, k) \quad = \quad \displaystyle\bigcup_{k' \in M_l} \left( \begin{array}{r} \text{if } k' \nearrow_{l'} k \text{ then } k' \sqcap k \\ \text{else } \ \{k'\} \end{array} \right) \\
\qquad\qquad\qquad \text{where } e_l = (\lambda_l \mathrm{x}.\ e_{l'})
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\sqcap : \mathrm{MPat} \times (\mathrm{MPat} \cup \mathrm{SPat}) \to 2^{\mathrm{MPat}}} \\
P \sqcap \forall & = & \{P\} \\
\forall \sqcap \star & = & \{\#\mathbf{f}, \lambda_{\forall}, (\forall,\, \forall)\} \\
P \sqcap \star & = & \{P\}, \quad \text{if } P \neq \forall \\
\#\mathbf{f} \sqcap \#\mathbf{f} & = & \{\#\mathbf{f}\} \\
P \sqcap \lambda_{\forall} & = & \{P\} \\
\lambda_{\forall} \sqcap \lambda_{\star} & = & \left\{ \lambda_l\ (\underbrace{\forall\ \ldots\ \forall}_{n \text{ times}}) \;\middle|\; \begin{array}{l} l \in \triangle(e_{l_0}) \ \wedge \\ e_l \text{ is a } \lambda\text{-expression} \ \wedge \\ \text{there are } n \text{ visible variables at label } l \end{array} \right\} \\
\lambda_l\ k \sqcap \lambda_{\star} & = & \{\lambda_l\ k\} \\
\lambda_l\ k' \sqcap \lambda_l\ k & = & \left\{ \lambda_l\ k'' \;\middle|\; k'' \in k' \sqcap k \right\} \\
(P_1',\, P_2') \sqcap (P_1,\, P_2) & = & \left\{ (P_1'',\, P_2'') \;\middle|\; P_1'' \in P_1' \sqcap P_1,\ P_2'' \in P_2' \sqcap P_2 \right\}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\sqcap : \mathrm{MCtPat} \times (\mathrm{MCtPat} \cup \mathrm{SCtPat}) \to 2^{\mathrm{MCtPat}}} \\
(P_1'\ \ldots\ P_n') \sqcap (P_1\ \ldots\ P_n) & = & \left\{ (P_1''\ \ldots\ P_n'') \;\middle|\; P_1'' \in P_1' \sqcap P_1,\ \ldots,\ P_n'' \in P_n' \sqcap P_n \right\}
\end{array}
$$

Figure 5.14: Algorithm for the upgrade of inspection points in pattern-matchers

$e_5$ and $e_{10}$, having respectively one and two variables in their lexical environment.

If the upgrade of pattern-matchers is done directly on the decision trees, it can be made more efficient. Essentially, the efficient algorithm consists in performing the conformance test and the (eventual) node upgrade together. Branches of the trees that do not conform to the pattern are unchanged. Branches that conform may change, depending on the fact that the inspection node corresponding to the split point is blind or not. Projection results at the leafs must be updated to reflect the increase in accuracy. Essentially, the update of the leafs is done similarly to $\sqcap$. Note that the upgrade of a blind node into a discriminating node changes the use of the queue. New blind nodes have to be introduced on lower levels in the trees to consume the sub-patterns that are to be inserted in the queue by the new discriminating node. As an instance, when a blind object node is turned into a discriminating node, two blind object nodes have to be added in the "pair" branch and a blind closure node has to be added in the "closure" branch. No new node is required in the "Boolean" branch as no sub-pattern gets inserted in the queue when **#f** is encountered.

### 5.1.3 Demands

We present the different kinds of demands that we manipulate during the demand processing phases of the demand-driven analysis. Most of these kinds were introduced informally in the previous chapter as "inevitable" ones. We now give a complete presentation of each kind along with its syntax and meaning.

Figure 5.16 presents the syntax of demands. The first three kinds of demands were informally mentioned in the previous chapter. The bad call demands are added to the list. Here is a precise description of each kind of demands. Each demand more or less directly asks for modifications to the abstract model.

**Bound demands.** Demand '**show** $\alpha_{l,k} \subseteq B$' requests a demonstration that $e_l$, when evaluated in contour $k$, provides only values contained in bound $B$. The possible bounds include each of the three types ($\mathcal{V}al\mathcal{B}$, $\mathcal{V}al\mathcal{C}$, and $\mathcal{V}al\mathcal{P}$) and also the values acting as true Boolean values in conditionals ($\mathcal{V}al\mathcal{T}rues = \mathcal{V}al\mathcal{C} \cup \mathcal{V}al\mathcal{P}$). The demonstration obtained when the demand is achieved usually have contour $k$ split into a certain number of more specialised contours $k_1, \ldots, k_n$ such that $\forall\, 1 \leq i \leq n.\ \alpha_{l,k_i} \subseteq B$. The

$$M_{\mathcal{V}} = \left\{ \begin{array}{l} \#\mathbf{f}, \\ \lambda_5\ (\forall), \\ \lambda_{10}\ (\#\mathbf{f}\ \forall),\ \lambda_{10}\ (\lambda_\forall\ \forall),\ \lambda_{10}\ ((\forall,\ \forall)\ \forall), \\ (\forall,\ \forall) \end{array} \right\}$$

$$U(M_{\mathcal{V}}, \lambda_{10}\ (\forall\ \star)) =$$

$$\left[ \begin{array}{ccc}
\#\mathbf{f} & \overset{\text{non-conforming}}{\Rightarrow} & \{\#\mathbf{f}\} \\
 & & \cup \\
\lambda_5\ (\forall) & \overset{\text{non-conforming}}{\Rightarrow} & \{\lambda_5\ (\forall)\} \\
 & & \cup \\
\lambda_{10}\ (\#\mathbf{f}\ \forall) & \overset{\text{conforming}}{\Rightarrow} & \left\{ \begin{array}{l} \lambda_{10}\ (\#\mathbf{f}\ \#\mathbf{f}), \\ \lambda_{10}\ (\#\mathbf{f}\ \lambda_\forall), \\ \lambda_{10}\ (\#\mathbf{f}\ (\forall,\ \forall)) \end{array} \right\} \\
 & & \cup \\
\lambda_{10}\ (\lambda_\forall\ \forall) & \overset{\text{conforming}}{\Rightarrow} & \left\{ \begin{array}{l} \lambda_{10}\ (\lambda_\forall\ \#\mathbf{f}) \\ \lambda_{10}\ (\lambda_\forall\ \lambda_\forall) \\ \lambda_{10}\ (\lambda_\forall\ (\forall,\ \forall)) \end{array} \right\} \\
 & & \cup \\
\lambda_{10}\ ((\forall,\ \forall)\ \forall) & \overset{\text{conforming}}{\Rightarrow} & \left\{ \begin{array}{l} \lambda_{10}\ ((\forall,\ \forall)\ \#\mathbf{f}) \\ \lambda_{10}\ ((\forall,\ \forall)\ \lambda_\forall) \\ \lambda_{10}\ ((\forall,\ \forall)\ (\forall,\ \forall)) \end{array} \right\} \\
 & & \cup \\
(\forall,\ \forall) & \overset{\text{non-conforming}}{\Rightarrow} & \{(\forall,\ \forall)\}
\end{array} \right]$$

$$\left\{ \begin{array}{lll}
\#\mathbf{f}, & & \\
\lambda_5\ (\forall), & & \\
\lambda_{10}\ (\#\mathbf{f}\ \#\mathbf{f}), & \lambda_{10}\ (\lambda_\forall\ \#\mathbf{f}), & \lambda_{10}\ ((\forall,\ \forall)\ \#\mathbf{f}), \\
\lambda_{10}\ (\#\mathbf{f}\ \lambda_\forall), & \lambda_{10}\ (\lambda_\forall\ \lambda_\forall), & \lambda_{10}\ ((\forall,\ \forall)\ \lambda_\forall), \\
\lambda_{10}\ (\#\mathbf{f}\ (\forall,\ \forall)), & \lambda_{10}\ (\lambda_\forall\ (\forall,\ \forall)), & \lambda_{10}\ ((\forall,\ \forall)\ (\forall,\ \forall)), \\
(\forall,\ \forall) & &
\end{array} \right\}$$

Figure 5.15: Example of the upgrade of a pattern-matcher

$$
\begin{array}{lcll}
\text{Demand} & := & \textbf{show}\ V \subseteq B \mid & \text{where } V \in \alpha\text{-var},\ B \in \text{Bound} \\
 & & \textbf{split}\ V\ P \mid & \text{where } V \in \text{Splittee},\ P \in \text{SPat} \\
 & & \textbf{show}\ V = \emptyset \mid & \text{where } V \in \delta\text{-var} \\
 & & \textbf{bad-call}\ l\ f\ v\ k & \text{where } l \in \text{Lab},\ f, v \in \text{MPat},\ k \in \text{MCtPat} \\
\text{Bound} & := & \mathcal{V}al\mathcal{B} \mid \mathcal{V}al\mathcal{C} \mid \mathcal{V}al\mathcal{P} \mid \mathcal{V}al\mathcal{T}rues & \\
\text{Splittee} & := & \mathcal{V}al\mathcal{C} \mid \mathcal{V}al\mathcal{P} \mid V & \text{where } V \in \alpha\text{-var} \cup \beta\text{-var} \cup \gamma\text{-var} \\
\alpha\text{-var} & := & \alpha_{l,k} & \text{where } l \in \text{Lab},\ k \in \text{MCtPat} \\
\beta\text{-var} & := & \beta_{\text{x},k,l} & \text{where x} \in \text{Var},\ k \in \text{MCtPat},\ l \in \text{Lab} \\
\gamma\text{-var} & := & \gamma_{c,k} & \text{where } c \in \text{MPat},\ k \in \text{MCtPat} \\
\delta\text{-var} & := & \delta_{l,k} & \text{where } l \in \text{Lab},\ k \in \text{MCtPat}
\end{array}
$$

Figure 5.16: Syntax of the demands

$$
\begin{array}{ll}
\boxtimes \subseteq 2^{\text{MPat}} \times \text{SPat} & \\
S \boxtimes P, & \text{if } \forall \in S \\
S \boxtimes \star, & \text{if } (\#\mathbf{f} \in S\ \wedge\ S \setminus \{\#\mathbf{f}\} \neq \emptyset)\ \vee \\
 & \quad (S \cap T \neq \emptyset\ \wedge\ S \setminus T \neq \emptyset) \\
 & \text{where } T = \{(P_1,\ P_2) \mid P_1, P_2 \in \text{MPat}\} \\
S \boxtimes \lambda_\star, & \text{if } \lambda_\forall \in S\ \ \vee \\
 & \quad (\lambda_l\ k, \lambda_{l'}\ k' \in S\ \wedge\ l \neq l') \\
S \boxtimes \lambda_l\ (P_1\ \ldots\ P_n), & \text{if } P_i \in \text{SPat}\ \wedge\ T \boxtimes P_i \\
 & \text{where } T = \left\{\ P'_i\ \middle|\ \begin{array}{l} \lambda_l\ (P'_1\ \ldots\ P'_n) \in S\ \wedge \\ \lambda_l\ (P'_1\ \ldots\ P'_n)\ \overset{\exists}{\cap}\ \lambda_l\ (P_1\ \ldots\ P_n) \end{array} \right\} \\
S \boxtimes (P_1,\ P_2), & \text{if } P_i \in \text{SPat}\ \wedge\ T \boxtimes P_i \\
 & \text{where } T = \left\{\ P'_i\ \middle|\ (P'_1,\ P'_2) \in S\ \wedge\ (P'_1,\ P'_2)\ \overset{\exists}{\cap}\ (P_1,\ P_2)\ \right\}
\end{array}
$$

Figure 5.17: Algorithm for the “is spread on” relation

$$
\begin{array}{ll}
\overset{\exists}{\cap} \subseteq (\mathrm{MPat} \cup \mathrm{SPat}) \times (\mathrm{MPat} \cup \mathrm{SPat}) & \\
\forall \overset{\exists}{\cap} P_2 & \\
P_1 \overset{\exists}{\cap} \forall & \\
\star \overset{\exists}{\cap} P_2 & \\
P_1 \overset{\exists}{\cap} \star & \\
\#\mathbf{f} \overset{\exists}{\cap} \#\mathbf{f} & \\
\lambda_\forall \overset{\exists}{\cap} P_2, & \text{if } P_2 \text{ is } \lambda_\forall,\ \lambda_\star, \text{ or } \lambda_l\ k \\
P_1 \overset{\exists}{\cap} \lambda_\forall, & \text{if } P_1 \text{ is } \lambda_\star \text{ or } \lambda_l\ k \\
\lambda_\star \overset{\exists}{\cap} P_2, & \text{if } P_2 \text{ is } \lambda_\star \text{ or } \lambda_l\ k \\
P_1 \overset{\exists}{\cap} \lambda_\star, & \text{if } P_1 \text{ is } \lambda_l\ k \\
\lambda_l\ (P_1\ \ldots\ P_n) \overset{\exists}{\cap} \lambda_l\ (P'_1\ \ldots\ P'_n), & \text{if } P_i \overset{\exists}{\cap} P'_i,\ \forall 1 \leq i \leq n \\
(P_1, P_2) \overset{\exists}{\cap} (P'_1, P'_2), & \text{if } P_1 \overset{\exists}{\cap} P'_1 \text{ and } P_2 \overset{\exists}{\cap} P'_2
\end{array}
$$

Figure 5.18: Definition of the "have a non-empty intersection" relation

property $\alpha_{l,k} \subseteq B$ may not (and need not) necessarily be satisfied literally.[2]

Most of the time, bound demands are generated as initial demands and directly express the needs of the optimiser.

The set of bounds that can be used in bound demands may seem restricted. One may estimate that more complex bounds are necessary. However, by the choice of the demand processing rules, bound demands are quickly transformed into other demands. The four different bounds that are mentioned are just sufficient for our approach.

**Never demands.** Demand '**show** $\delta_{l,k} = \emptyset$' asks for a demonstration that $e_l$ is not really evaluated in contour $k$. Once again, various modifications to the abstract model are generally needed among which there is typically a split of contour $k$ into more specialised ones, $k_1, \ldots, k_n$, such that $\forall\, 1 \leq i \leq n.\ \delta_{l,k_i} = \emptyset$.

Usually, never demands are generated because there is evidence that, if the expression gets evaluated, then it necessarily leads to an error.

**Split demands.** These demands ask for an increase in the accuracy of the modelling. The *splittee* is the entity for which greater accuracy is required, i.e. that should be split. The desired improvement in accuracy is specified by the split pattern. There are split demands that directly ask for an update of the model. These have $\mathcal{V}al\mathcal{C}$ or

[2] Moreover, after the split of $k$ is done, $k$ no longer exists. So, strictly speaking, talking about abstract variable $\alpha_{l,k}$ is an abuse of notation.

$\mathcal{V}al\mathcal{P}$ as splittee. The others ask for increased accuracy—or separation—in the values contained in an abstract variable. The abstract variable represents the value of an expression ($\alpha_{l,k}$), the value of a reference to a variable ($\beta_{\mathrm{x},k,l}$), or the return value of a closure ($\gamma_{c,k}$).

Split demands having an abstract variable as splittee are mainly generated to separate the so-called good cases from the bad cases. In the previous chapter, we explain the importance of separating good and bad cases before any attempt to remove the bad cases is made.

A splittee of the form $\beta_{\mathrm{x},k,l}$ denotes a *reference* to 'x' in contour $k$ and from label $l$. Recall that this is different from the abstract variable $\beta_{\mathrm{x},k}$. The contours that are valid where the variable is bound and those where the variable is referenced may differ completely. For example, it is the case when the reference occurs in an expression deeply nested inside of the closure that introduced the binding to the variable.

Split demands on abstract variables request that the model be modified in such a way that (in the case of a splittee from '$\alpha$-var') $k$ is split into specialised contours $k_1, \ldots, k_n$ such that, in each $k_i$, the values fall on *only one side of the pattern.* To formally express this concept, we need the help of the "is spread on" relation to indicate when abstract values happen not fall all on the same side of the pattern. Figure 5.17 gives a formal definition of this relation. The spread relation between a set of values $S$ and a split pattern $P$ is denoted by $S \boxtimes P$. In turn, this relation is based on another one: the "have a non-empty intersection". This one indicates if two split or modelling patterns have an intersection, i.e. if there exists a concrete value that is abstracted by both patterns. We write $P_1 \overset{\exists}{\cap} P_2$ when patterns $P_1$ and $P_2$ have a non-empty intersection. Figure 5.18 formally defines the relation. Again, we decree that split pattern '$\star$' abstracts all values and '$\lambda_\star$' abstracts all closures. So, the achievement of '**split** $\alpha_{l,k}$ $P$' consists in modifying the abstract model such that $k$ is specialised into $k_1, \ldots, k_n$ such that $\forall 1 \leq i \leq n.\ \neg(\alpha_{l,k_i} \boxtimes P)$. Similarly for other split demands where the splittee is an abstract variable.

To help to understand the meaning of $\boxtimes$, we use a picture. Imagine that the set of all modelling patterns lie on a (two-dimensional) plan. Since patterns are discrete entities, we will imagine them as sand granules. Now, our set of abstract values $S$ is represented by a subset of these granules. Imagine that our split pattern $P$ is a riddle—a coarse sieve. It has a certain number of holes. It may be as vast as the plan or may occupy only a tiny fraction of the plan. Testing whether $S \boxtimes P$ hold is equivalent to sifting the sand granules using the riddle: let the sand granules levitate above the ground, each at their respective $x$ and $y$ coordinates; let the riddle lie between the granules and the ground; and finally let the granules fall free. Some granules fall into the riddle, some do not. Among the granules that fall into the riddle, some may go through a different hole than others. We say that the sand was *spread over* the riddle if more than one hole was passed through by the sand. The granules that fell outside of the riddle do not matter.

With the test $S \boxtimes P$, a similar thing occurs. Some abstract values do not have an intersection with $P$: they fall outside of $P$. Others fall into $P$ and pass through one of the "holes" of $P$, depending on the type or on the label of a sub-pattern. Moreover, some values may even be too coarse to be able to go through one of the holes; they get stuck on $P$. Let us give some examples:

- pattern $P = (\star, \forall)$ sifts pairs; it has three holes that are $(\#\mathbf{f}, \forall)$, $(\lambda_\forall, \forall)$, and $((\forall, \forall), \forall)$; Booleans and closures fall outside of $P$; $\forall$ cannot go through $P$ but cannot fall outside either;
- pattern $P = \star$ has three holes and no values can fall outside of it;
- pattern $P = (\lambda_{12}\ (\lambda_\forall\ \#\mathbf{f}), (\lambda_\star, \#\mathbf{f}))$ occupies a small fraction of the plan and has as many holes as there are $\lambda$-expressions in the program.

So $S \boxtimes P$ holds if there is a value in $S$ that gets stuck in $P$ or if there are values in $S$ going through different "holes" of $P$. Values having no intersection with $P$ do not matter.

**Bad call demands.** Demand '**bad-call** $l\ f\ v\ k$' asks for a demonstration that the described invocation actually does not occur. The invocation is that of closure $f$ on argument $v$ at call $e_l$ in contour $k$. The achievement of this demand usually requires to first perform changes on the abstract model and then to have all "bad" specialisations of the invocations not to occur.

Bad call demands originate from the processing of never demands. In order to show that a certain expression (that happens to be the body of a closure) does not get evaluated, it is necessary to show that certain calls do not occur.

**Call site monitoring.** Although call site monitoring is not a kind of demand, we mention it here simply to introduce its syntax. When a call expression $e_l$ has to be monitored, we write the pseudo-demand '**monitor-call** $l$ $k$'. It specifies in which contour the monitoring must be done.

We intentionally omit to explain what monitoring is exactly. We simply mention that it consists in taking actions to achieve all bad call demands related to the specified call site.

## 5.2 Demand Processing

We present the processing rules for the demands. The processing rules for each kind of demands are presented in the following sections. They are considered in the following order: the bound demands, the never demands, the bad call demands, and the split demands. The processing of the split demands is clearly the most involving. Then, we continue by describing the monitoring of call sites. Finally, we present an important function that is used to minimally separate couples according to some given property: the Split-Couples function (SC) is useful in the processing of a few demands.

The demand processing rules depend on a certain number of hypothesis. They suppose that the complete demand-driven approach is the one presented in a later section. Changing the global approach would require some adaptation of the processing rules. As described, the processing rules are intended to be used during a model-update phase. Let $e_{l_0}$ be the program to analyse. The current abstract model is

$$\mathcal{M} = (\mathcal{V}al\mathcal{B},\, \mathcal{V}al\mathcal{C},\, \mathcal{V}al\mathcal{P},\, \mathcal{C}ont,\, (\,),\, \mathsf{cc},\, \mathsf{pc},\, \mathsf{call})$$

and is built on the pattern-matchers

$$\{M_{\mathcal{V}}\} \cup \{M_l \mid (\lambda_l \mathrm{x}.\ e) \in \triangle(e_{l_0})\}$$

The analysis results for the program using $\mathcal{M}$ are assumed to be available as

$$\mathcal{R} = (\alpha,\, \beta,\, \gamma,\, \delta,\, \chi,\, \pi,\, \kappa) = \mathrm{FW}(e_{l_0}, \mathcal{M})$$

Despite the hypotheses that we pose, many processing rules take care of more cases than it is strictly necessary. This is because, most of the time, it is simpler to treat all cases, even impossible ones, than to argue why some of them are impossible.

We show the results of processing a demand using a double arrow. It is usually preceded by a condition. It looks like:

If *some condition*:<br>
$\Rightarrow$*results*

Most of the results of demand processing are emissions of new demands. But some results constitute one or many model updates. When the processing of a demand is complete and does not emit sub-demands, a comment is added to indicate whether its processing is successful. Comment (SUCCESS) indicates that the demand is achieved. Comment (FAILURE) indicates that the demand cannot be achieved. Normally, demands that depend on the failed demand cannot be achieved either. As we explain during the description of the global demand-driven approach, the comments are ignored. We insert them to make the presentation clearer. However, a modified approach could make use of the comments.

### 5.2.1 Bound Demands

Let us consider bound demand $D \equiv$ '**show** $\alpha_{l,k} \subseteq B$' where $B$ is one of the four bounds. When $D$ is processed, one of three situations can occur. The first is that the bound is respected, so $D$ is trivially achieved:

If $\alpha_{l,k} \subseteq B$:<br>
$\Rightarrow$(SUCCESS)

The second situation occurs when no value resulting from the evaluation of $e_l$ in $k$ lies inside of $B$. This is a relatively simple situation as only *bad cases* occur. Only bad values can come from $e_l$ so the sufficient and necessary way to achieve $D$ is by showing that $e_l$ does not get evaluated in $k$ at all:

If $\alpha_{l,k} \cap B = \emptyset$:<br>
$\Rightarrow$**show** $\delta_{l,k} = \emptyset$

Note that, if $\alpha_{l,k}$ is empty, we can say that, in fact, it falls into the first two situations. However, the first situation is more favourable and should be used. Each time the conditions attached to a situation are met, this situation should be considered to have priority over the next ones as we list the more favourable situations first.

The last situation occurs when there are both good and bad cases. That is, when there are values that lie inside of the bound and others, outside. The first step in trying to achieve $D$ then consists in separating the good and bad cases:

Otherwise:
⇒**split** $\alpha_{l,k}$ $\star$

Because of the simplicity of the valid bounds, a split demand requesting the values in $\alpha_{l,k}$ to be split according to their (top-level) type is sufficient to separate good cases from bad cases. In the eventuality that this new demand is achieved, then a reiteration of the processing of $D$ (in fact, of its specialisations) will be able to proceed using one of the first two situations.

### 5.2.2 Never Demands

Let us consider never demand $D \equiv$ '**show** $\delta_{l,k} = \emptyset$' to be the demand to process. The first and simplest situation occurs when the property to verify is already true. Then, the demand is trivially achieved:

If $\delta_{l,k} = \emptyset$:
⇒(SUCCESS)

Another simple situation consists in $D$ asking for a demonstration that the program does not get evaluated in the main contour, which is patently false. The abstract interpretation of the program, for analysis purpose, *is* started by the constraint $\delta_{l_0,()} \supseteq \mathcal{V}al\mathcal{B}$. It follows that $D$ fails:

If $l = l_0$ and $k = ()$:
⇒(FAILURE)

The other situations require some active processing. First, note that, most of the time, the fact that an expression is evaluated is controlled by its parent expression, i.e. by $e_{l'}$, where $l' = \text{parent}(l)$.[3] Often, the evaluation of $e_l$ depends only on the fact that $e_{l'}$ is

[3] The attentive reader may notice that we do not mention the case where $l = l_0$ and $k \neq ()$. This is because $e_{l_0}$ is not inside the body of a closure. Its evaluation cannot be triggered because of some invocation. So the only way $e_{l_0}$ gets evaluated is by the starting constraint which uses contour $()$. Consequently, for all $k \neq ()$, we have that $\delta_{l_0,k} = \emptyset$, and this case is caught by the first situation.

evaluated too. But let us start by considering the special cases first.

If $e_{l'} = (\lambda_{l'}\mathrm{x}.\ e_l)$, the events that cause the evaluation of $e_l$ in contour $k$ are that some closure originating from $e_{l'}$ gets invoked and $\mathsf{call}$ selects $k$ as the contour in which $e_l$ ought to be evaluated. So, in order to try to achieve $D$, a demonstration that each such invocation cannot happen is needed. Bad call demands are emitted for each invocation leading to the evaluation of $e_l$ in $k$. If all of these sub-demands are eventually achieved, then $D$ clearly becomes so, too:

If $e_{l'} = (\lambda_{l'}\mathrm{x}.\ e_l)$:
$\Rightarrow\left\{ \ \textbf{bad-call}\ l''\ (\lambda_{l'}\ k'')\ v\ k' \ \middle|\ (l'',\ (\lambda_{l'}\ k''),\ v,\ k') \in \kappa_k \ \right\}$

The situation in which $e_{l'}$ is a conditional and $e_l$ is its THEN-branch is a special case as it is not true that $e_l$ is evaluated if and only if $e_{l'}$ is. In fact, $e_l$ is not evaluated if and only if the test in $e_{l'}$ does not return "true" values (closures and pairs). So $D$ is achieved if and only it can be showed that the test returns nothing else than "false" 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 situation in which $e_l$ is the ELSE-branch of $e_{l'}$ is symmetric to the THEN-branch situation:

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 remaining situations are all those in which $e_l$ is evaluated if and only if $e_{l'}$ is. Those include the case where $e_l$ is the test of the conditional $e_{l'}$ and the cases where $e_{l'}$ is not a $\lambda$-expression nor a conditional. The appropriate processing in these situations is to ask for a demonstration that $e_{l'}$ does not get evaluated either (at least in contour $k$):

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

### 5.2.3 Bad Call Demands

Let us consider demand $D \equiv$ '**bad-call** $l\ f\ v\ k$'. Since the parameters in $D$ describe the circumstances of an invocation, we know that $e_l$ is a call. Let $e_l = ({}_l e_{l'}\ e_{l''})$. In processing

$D$, the first situation that we could face is that of $D$ being trivially achieved. The described invocation does not occur if at least one of $f$ and $v$ does not appear at the call in the specified contour:

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

Otherwise, the invocation really occurs. At least, according to the current analysis results. The natural processing for this bad call would consist in separating the specified invocation from the others, if they exist, and then trying to show that $e_l$ does not get evaluated in the sub-contour that contains the specified invocation. Prior separation of the specified invocation from the others, if they exist, is essential, since the other invocations need not necessarily be bad. Indeed, the other invocations may even represent actual concrete invocations, and, as such, should not be subject to an attempt to demonstrate that they do not occur. Non-occurrence of the other invocations is *not necessary.*

However, we do not process $D$ in the way we just described. The described method lead to too many splits. Imagine that many closures are invoked on many different arguments at $e_l$ in $k$, and that half of the invocations are considered to be bad. The described processing requires every bad invocation to be completely separated from all the others. But the only separation that is really needed is one that separates all bad calls from all (presumably) good calls. This global separation may be much simpler than the combination of all individual separations. So instead of immediately taking measures to achieve $D$, we prefer to put it in a *log of bad calls.* Later, all bad calls related to $e_l$ and contour $k$ are processed together in what we designate as *the monitoring of* $e_l$ *in* $k$. We denote the log of bad calls by $L_{\mathrm{BC}}$ and the invocations that are marked as bad for call $e_l$ in contour $k$ are listed in $L_{\mathrm{BC}}(l, k)$.

```
Otherwise:
$\Rightarrow$Insert $(f, v)$ in $L_{\mathrm{BC}}(l, k)$
  Flag $(l, k)$ as a candidate for monitoring
```

### 5.2.4 Split Demands

The processing of a split demand '**split** $V$ $P$' depends considerably on the splittee $V$. Processing of the demand for each kind of splittee is presented in separate sections.

### Direct Split on the Model

A direct split on the model is requested by a demand like $D \equiv$ '**split** $V$ $P$' where $V$ is $\mathcal{V}al\mathcal{C}$ or $\mathcal{V}al\mathcal{P}$. No matter which of the two splittees is used in $D$, the demand is processed identically. The value pattern-matcher is updated using $P$:

$\Rightarrow$Update $M_{\mathcal{V}}$ with $P$

### Split on $\alpha$-Variables

The splits on $\alpha$-variables are the most involving part of the demand processing. Let us consider split demand $D \equiv$ '**split** $\alpha_{l,k}$ $P$'. The simplest situation is the one in which $D$ is trivially achieved. It occurs when the values in the abstract variable are not spread on the pattern:

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

Otherwise, the complexity of the processing becomes immediately apparent. The values in the abstract variable are spread on the pattern and some measures have to be taken in order to cause this spreading to disappear. We know that the values in an $\alpha$-variable result from the abstract interpretation of expression $e_l$. And the interpretation of $e_l$ depends on the kind of expression it is. So, similarly to the processing of never demands that depended on the kind of the parent expression, the processing of split demands on $\alpha$-variables depends on the kind of the expression itself. We consider each kind of expression in turn.

**Boolean Constant** Let $e_l = \#\mathtt{f}_l$. This situation is actually impossible, as we explain next, but we include it for completeness. Abstract variable $\alpha_{l,k}$ contains either all Booleans ($\mathcal{V}al\mathcal{B}$) or nothing, depending on whether $e_l$ gets evaluated in contour $k$ or not. If $\alpha_{l,k} = \mathcal{V}al\mathcal{B}$, inspection of each possible split pattern $P'$ shows that we cannot have that $\alpha_{l,k} \boxtimes P'$. Intuitively, this is because abstract Booleans in $\mathcal{V}al\mathcal{B}$ represent perfectly accurately the concrete Boolean. A concrete value by itself cannot be spread on a split pattern. And if $\alpha_{l,k} = \emptyset$, then there clearly in no spreading. But we give the processing rule for $D$ nevertheless and we indicate that $D$ is trivially achieved:

If $e_l = \#\mathtt{f}_l$:

$\Rightarrow$(SUCCESS)

**Variable Reference** Let $e_l = \mathrm{x}_l$. Processing $D$ is very simple as it translates directly into a split demand on a $\beta$-variable:

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

**Call** Let $e_l = ({}_l e_{l'}\ e_{l''})$. This situation is *the* difficult one. Here are a couple of reasons. First, the result of the evaluation of $e_l$ comes from the invocation of each closure on each argument and blending the individual results together. So the values in $\alpha_{l,k}$ are not directly function of the values in $\alpha_{l',k}$ and $\alpha_{l'',k}$. Second, the calls are responsible for making our mini-language a Turing-complete one. Without them, analysing a program could simply consist in concretely evaluating it, given the guarantee not to loop.

In order to process $D$, the result of each invocation has to be inspected. For closure $f$ invoked on argument $v$, with return values in $\gamma_{f,k'}$, for some $k'$, there are three cases: $\gamma_{f,k'}$ has no intersection with $P$, $\gamma_{f,k'}$ is not spread on $P$, or $\gamma_{f,k'}$ is spread on $P$. In the first case, the invocation does not contribute to the value of $\alpha_{l,k}$ in a way that is observable by $P$. In the second case, we can determine into which "hole" of $P$ the result of the invocation falls. In the third case, we cannot even determine into which "hole" the invocation falls. The treatment of each invocation is different depending on the case to which it belongs.

Invocations having an empty return value or a return value that has no intersection with $P$ are ignored.

Invocations having a return value that is spread on $P$ lead to a request for having a more precise description of the computations occurring in the invoked closure. Eventually, the more precisely described closure may have return values that cease to be spread on $P$. That is, $\gamma_{f,k'}$ may be replaced by a number of more specialised invocation results, each causing no spreading on $P$. Being able to determine in which "hole" of $P$ each invocation result goes is vital to a successful processing of $D$. Until the invocation of $f$ on $v$ is split into non-spreading evaluation results, it cannot be used to select useful splits on sub-expressions $e_{l'}$ and $e_{l''}$.

Invocations having a return value that is not spread on $P$ are immediately used in selecting splits on the sub-expressions of $e_l$. The return value of such an invocation goes through a single "hole" of $P$ but that does not automatically mean that all non-spreading return values, once united together, do not spread on $P$. In order to make progress in the achievement of $D$, non-spreading return values are collected together along with their corresponding closure-argument couple. The Split-Couples function is then used to select splits on the closure component and/or on the argument component. That is, on the value of $e_{l'}$ and that of $e_{l''}$, respectively. The SC function selects splits such that *incompatible* couples are separated by the splits. We say that two couples $(f_1, v_1)$ and $(f_2, v_2)$ are incompatible if their associated return values $\gamma_{f_1,k_1}$ and $\gamma_{f_2,k_2}$ go through different "holes" of $P$. If all the splits selected by SC are to be achieved, then no incompatible couples will appear in the same contour anymore.

We give the processing rule and then give an example:

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

$$
\begin{array}{l}
\Rightarrow \left\{ \textbf{split}\ \gamma_{f,k'}\ P \;\middle|\; \begin{array}{l} f \in \alpha_{l',k} \cap \mathcal{V}al\mathcal{C} \quad \wedge\ v \in \alpha_{l'',k}\ \wedge \\ k' = \mathsf{call}(l,\, f,\, v,\, k) \wedge \gamma_{f,k'} \boxtimes P \end{array} \right\} \\
\quad \cup\ \left\{ \textbf{split}\ \alpha_{l',k}\ P' \;\middle|\; P' \in B \right\} \\
\quad \cup\ \left\{ \textbf{split}\ \alpha_{l'',k}\ P'' \;\middle|\; P'' \in C \right\} \\
\quad \text{where } A \qquad = \left\{ ((f,\, v),\, \gamma_{f,k'}) \;\middle|\; \begin{array}{l} f \in \alpha_{l',k} \cap \mathcal{V}al\mathcal{C} \quad \wedge\ v \in \alpha_{l'',k}\ \wedge \\ k' = \mathsf{call}(l,\, f,\, v,\, k) \wedge \\ \exists v' \in \gamma_{f,k'}.\ v' \overset{\exists}{\cap} P\ \wedge \neg\, (\gamma_{f,k'} \boxtimes P) \end{array} \right\} \\
\qquad\qquad (B,\, C) = \textsc{sc}(A,\, P)
\end{array}
$$

In our example, we consider demand $D =\equiv$ '**split** $\alpha_{l,k}$ $\star$' where $e_l = ({}_l e_{l'}\ e_{l''})$. So we want to have $k$ (and possibly other abstract entities) split into $k_1, \ldots, k_n$ so that, for each $1 \leq i \leq n$, the contents of $\alpha_{l,k_i}$ is of a single type, if not empty. In order to have an actual situation with which we can work, let us suppose that two different closures may be invoked on two different values. That is, $\alpha_{l',k} = \{f_1, f_2\}$ and $\alpha_{l'',k} = \{v_1, v_2\}$. For convenience, we also suppose that $f_1$ and $f_2$ originate from two different $\lambda$-expressions and that $v_1$ and $v_2$ are values of different types. These last convenient suppositions are used below to keep things simple. During abstract interpretation, each closure is invoked on each argument and each time a contour is selected by call. We denote by $k_{ij}$ the contour selected when $f_i$ is invoked on $v_j$, for $i,\, j \in \{1, 2\}$. That is, $k_{ij} = \mathsf{call}(l,\, f_i,\, v_j,\, k)$. Let the spreading or non-spreading on $\star$ of the result of each invocation be given by this table:

$$\begin{array}{rr} \neg\,(\gamma_{f_1,k_{11}} \boxtimes \star), & \gamma_{f_1,k_{12}} \boxtimes \star, \\ \gamma_{f_2,k_{21}} \boxtimes \star, \text{ and} & \neg\,(\gamma_{f_2,k_{22}} \boxtimes \star) \end{array}$$

Finally, let us suppose that $\emptyset \neq \gamma_{f_1,k_{11}} \subseteq \mathcal{V}al\mathcal{B}$ and that $\emptyset \neq \gamma_{f_2,k_{22}} \subseteq \mathcal{V}al\mathcal{P}$.

Using this information, we are able to illustrate the processing of $D$. First among all the return values $\gamma_{f_i,k_{ij}}$, none is empty. So none is ignored. Second, we must take care of the return values that are spread over $\star$, namely $\gamma_{f_1,k_{12}}$ and $\gamma_{f_2,k_{21}}$. For each, a split demand is emitted that requests that they be split using pattern $\star$. Finally, we take care of non-spreading return values $\gamma_{f_1,k_{11}}$ and $\gamma_{f_2,k_{22}}$. Individually, they are non-spreading but, collectively, they are spread on $\star$. So the set $A$ describing the couples is built:

$$A = \{((c_1,\, v_1),\, \gamma_{c_1,k_{11}}),\, ((c_2,\, v_2),\, \gamma_{c_2,k_{22}})\}$$

The two couples in $A$ are incompatible because their associated return values go through different "holes" of $\star$. The first couple goes through the "Boolean hole" and the second goes through the "pair hole". Since the couples are distinct, at least one of the components must be distinct. In our case, both components differ. The $\textsc{sc}$ function computes an economical way to separate the two couples in $A$ relatively to pattern $\star$. It returns one of the two following splitting strategies:

$$(\{\lambda_\star\},\, \emptyset) \quad \text{or} \quad (\emptyset,\, \{\star\})$$

meaning that either a split should be performed on the first components to separate them based on the closure label or a split should be performed on the second components to separate them based on the type. Splitting both components would be zealous. If we suppose that the first strategy is adopted, then the final result of processing $D$ is:

$\Rightarrow$**split** $\gamma_{f_1,k_{12}}\ \star$
  **split** $\gamma_{f_2,k_{21}}\ \star$
  **split** $\alpha_{l',k}\ \lambda_\star$

**$\lambda$-Expression** Let $e_l = (\lambda_l \mathrm{x}.\ e_{l'})$. When this situation is being considered, we know that it is because $\alpha_{l,k} \boxtimes P$. And since $\alpha_{l,k} = \{\mathsf{cc}(l,\, k)\}$, then $P$ is of the form $\lambda_l\ k'$. So we translate $D$ into a direct model update demand:

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

**Conditional** Let $e_l = (\texttt{if}_l\ e_{l'}\ e_{l''}\ e_{l'''})$. The processing for the conditional expression shares some similarity with that of the call. The evaluation result, $\alpha_{l,k}$, is the union of some sub-evaluations; here, the two branches of the conditional. Each sub-evaluation result spreads on $P$ has to be split first. The non-spreading results are used to select splits on sub-expressions of $e_l$; here, this occurs when both branches are non-spreading but incompatible and a split on the test has to be emitted.

We first consider the case where the evaluation result of at least one of the branches is spread on $P$. Since the evaluation result of $e_l$ remains spread on $P$ as long as the result of at least one branch is, then it is necessary to split the result for each such branch. It is too early to be able to determine if a split on the test is required or not.

If $e_l = (\texttt{if}_l\ e_{l'}\ e_{l''}\ e_{l'''}) \ \wedge\ (\alpha_{l'',k} \boxtimes P \ \vee\ \alpha_{l''',k} \boxtimes P)$:
$\Rightarrow\left\{\textbf{split}\ \alpha_{l^{(n)},k}\ P \;\middle|\; l^{(n)} \in \{l'', l'''\} \ \wedge\ \alpha_{l^{(n)},k} \boxtimes P\right\}$

The other case consists in having the result for both branches not to be spread on $P$. But, since we know that $\alpha_{l,k} \boxtimes P$, these results must be incompatible. To achieve $D$, the necessary and sufficient sub-demand to generate is to ask for both branches not to evaluate in the same contour. So the cases where the test evaluates to a true value must be separated from the cases where the test evaluates to a false value. So a sub-demand is emitted that asks for the split of the result of the test on its type. In fact, this is slightly excessive as a $\mathcal{V}al\mathcal{B}/\mathcal{V}al\mathcal{C}/\mathcal{V}al\mathcal{P}$ distinction is requested when only a $\mathcal{V}al\mathcal{B}/\mathcal{V}al\mathcal{T}rues$ one is required. However, the split pattern syntax that we have chosen cannot express a split coarser than '$\star$'.

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

**Pair Construction** Let $e_l = (\texttt{cons}_l\ e_{l'}\ e_{l''})$. Keeping in mind that $\alpha_{l,k} \boxtimes P$, quick inspection of the different kinds of split patterns allows us to conclude that $P$ is of the form $(P',\ P'')$. One of $P'$ and $P''$ is a split pattern. We process $D$ simply by emitting a sub-demand that asks for the sub-split to be performed on the appropriate sub-expression of $e_l$:

If $e_l = (\mathtt{cons}_l\ e_{l'}\ e_{l''})\ \wedge\ P = (P', P'')\ \wedge\ P' \in \mathrm{SPat}$:
$\Rightarrow$**split** $\alpha_{l',k}\ P'$

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

Note that the proposed processing in the situation where we have a pair construction expression is sufficient. But it is less clear whether it is necessary. To see why, we give an example. Let $P = ((\forall, \forall), \lambda_\star)$. Of course, the value of $e_{l''}$ has to be split using $\lambda_\star$ in a way or another. But is it really $\alpha_{l'',\boxed{k}}$ that should be split? Note that $P$ specifies that additional accuracy is requested only when the CAR-field of the pair contains a pair. Maybe the appropriate processing consists in first splitting $\alpha_{l',k}$ using $\star$ and, when this is done, we have $k$ specialised into, say, $k_\mathcal{B}$, $k_\mathcal{C}$, and $k_\mathcal{P}$. We would then split $\alpha_{l'',k_\mathcal{P}}$ using $\lambda_\star$. That is, we would split the value of $e_{l''}$ only in the contour in which $e_{l'}$ evaluates to pairs. We believe that it is not obvious at all whether this more elaborate way of splitting is easier or more profitable. A split on $\alpha_{l'',k_?}$ has to be made for some $k_?$, anyway.

Since the split sub-pattern in $(P', P'')$, that is, $P'$ or $P''$, has to be propagated to $e_{l'}$ or $e_{l''}$ anyway, the question can be summarised like this: Should more accuracy be requested on the non-splitting side in order to (possibly) facilitate the splitting on the splitting side? We have decided that the answer would be: no. Only the sub-pattern on the splitting side is propagated. No additional accuracy is requested from the non-splitting side.

**CAR-Field Access** Let $e_l = (\mathtt{car}_l\ e_{l'})$. Since $D$ asks for increased accuracy in the representation of the value of $e_l$, then a new demand should be emitted that requests increased accuracy in the representation of the CAR-field of the pairs that come from $e_{l'}$. That is, since $P$ is the split pattern appearing in the request concerning $e_l$, $(P, \forall)$ should be the one appearing in the request concerning $e_{l'}$. However, a verification that the abstract domain $\mathcal{V}al\mathcal{P}$ is accurate enough for $(P, \forall)$ must be done. Indeed, it is pointless to ask for a split of $\alpha_{l',k}$ using $(P, \forall)$ if the abstract pairs are not distinguishable by $(P, \forall)$. If $\mathcal{V}al\mathcal{P}$ is not accurate enough, a direct model update demand is emitted. Otherwise, the normal processing is performed.

If $e_l = (\mathtt{car}_l\ e_{l'})\ \wedge\ \mathcal{V}al\mathcal{P}$ is accurate enough for $(P, \forall)$:

$\Rightarrow$**split** $\alpha_{l',k}$ $(P, \forall)$

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

Verifying that $\mathcal{V}al\mathcal{P}$ is accurate enough for $(P, \forall)$ is relatively simple. The abstract pairs in $\mathcal{V}al\mathcal{P}$ result from projections using $M_{\mathcal{V}}$. The verification proceeds by testing whether there is a modelling pattern in $M_{\mathcal{V}}$ having an intersection with $(P, \forall)$ but for which we cannot decide in which "hole" it passes through. In other words, if there is a modelling pattern in $M_{\mathcal{V}}$ that is spread on $(P, \forall)$:

$$\begin{gathered}\mathcal{V}al\mathcal{P} \text{ is accurate enough for } (P, \forall) \\ \Leftrightarrow \\ \forall\, v \in M_{\mathcal{V}}.\ \neg\ (\{v\} \boxtimes (P, \forall))\end{gathered}$$

This processing for $D$ is an instance of the creation of complex split patterns using simpler ones. Also, it justifies our choice of the meaning of split demands, as presented in Section 5.1.3 that says that values having no intersection with the split pattern are ignored. The split of $\alpha_{l',k}$ using $(P, \forall)$ is concerned only with pairs coming from $e_{l'}$. Non-pairs coming from $e_{l'}$ do not contribute to the value of $e_l$ and, as such, are not concerned by the split pattern $(P, \forall)$. The fact that their presence leads to errors is an independent problem.

**CDR-Field Access** Let $e_l = (\texttt{cdr}_l\ e_{l'})$. The processing of $D$ is completely symmetric to that of a demand concerning a CAR-field access.

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

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

**Pair Membership Test** Let $e_l = (\texttt{pair?}_l\ e_{l'})$. The processing of $D$ is trivial, the split pattern is propagated to the sub-expression without modification:

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

To see why this processing is adequate, each form of split patterns has to be considered. If $P = \star$, then to make the distinction $\mathcal{V}al\mathcal{P}/\mathcal{V}al\mathcal{B}$ on the value of $e_l$, it is necessary to make the distinction $\mathcal{V}al\mathcal{P}/\overline{\mathcal{V}al\mathcal{P}}$ on the value of $e_{l'}$. The split pattern $\star$ is then used. $P$ cannot be $\lambda_\star$ nor be of the form $\lambda_l\ k$, because $\alpha_{l,k} \boxtimes P$. Finally, if $P = (P',\ P'')$, then only the pairs coming from $e_l$ are concerned. Since these pairs are the same as those coming from $e_{l'}$, $P$ itself must be used in the split of $\alpha_{l',k}$.

### Split on $\beta$-Variables

Let $D \equiv$ '**split** $\beta_{\mathrm{x},k,l}\ P$'. Processing $D$ results in a direct model update. However, some information has to be gathered in order to find the appropriate contour pattern-matcher and to produce the right split contour pattern. The first step consists in finding the position of variable 'x' in contour $k$. Recall that $k$ is an abstract version of the lexical environment and that "bounds" on the possible values that each variable can take are listed from the innermost variable to the outermost. Let $e_{l_i} \in \triangle(e_{l_0})$ be the $\lambda$-expression that binds 'x':

$$(\lambda_{l_i}\mathrm{x}.\ (\ldots\ (\lambda_{l_{i+1}}\mathrm{y}_{i+1}.\ (\ldots\ (\lambda_{l_n}\mathrm{y}_n.\ e_{l'_n})\ldots))\ \ldots))$$

where $e_l \in \underline{\triangle}(e_{l'_n})$. In other words, we have that the $e_{l_j}$ are $\lambda$-expressions, for $1 \le j \le n$, and that:

$$\begin{array}{lcl}
(\lambda_{l_1}\mathrm{y}_1.\ e_{l'_1}) & \in & \underline{\triangle}(e_{l_0}) \\
(\lambda_{l_2}\mathrm{y}_2.\ e_{l'_2}) & \in & \underline{\triangle}(e_{l'_1}) \\
 & \ldots & \\
(\lambda_{l_{i-1}}\mathrm{y}_{i-1}.\ e_{l'_{i-1}}) & \in & \underline{\triangle}(e_{l'_{i-2}}) \\
(\lambda_{l_i}\mathrm{x}.\ e_{l'_i}) & \in & \underline{\triangle}(e_{l'_{i-1}}) \\
(\lambda_{l_{i+1}}\mathrm{y}_{i+1}.\ e_{l'_{i+1}}) & \in & \underline{\triangle}(e_{l'_i}) \\
 & \ldots & \\
(\lambda_{l_n}\mathrm{y}_n.\ e_{l'_n}) & \in & \underline{\triangle}(e_{l'_{n-1}}) \\
e_l & \in & \underline{\triangle}(e_{l'_n})
\end{array}$$

So $k = (P_n\ \ldots\ P_{i+1}\ P_i\ P_{i-1}\ \ldots\ P_1)$ and $P_i$ is the bound on the value of 'x' in contour $k$. The intent is to update $k$ such that its $P_i$ pattern is split into specialisations. Note that the pattern-matcher that must be updated is $M_{l_n}$. Updating $M_{l_n}$ using the split contour pattern

$$
\begin{array}{lll}
\multicolumn{3}{l}{\cap : (\mathrm{MPat} \cup \mathrm{SPat}) \times (\mathrm{MPat} \cup \mathrm{SPat}) \rightarrow (\mathrm{MPat} \cup \mathrm{SPat})} \\
P_1 \cap P_2 & \text{is undefined if } P_1, P_2 \in \mathrm{SPat} & \\
\forall \cap P_2 & = \; P_2 & \\
P_1 \cap \forall & = \; P_1 & \\
\star \cap P_2 & = \; \star & \\
P_1 \cap \star & = \; \star & \\
\#\mathbf{f} \cap \#\mathbf{f} & = \; \#\mathbf{f} & \\
\lambda_\forall \cap P_2 & = \; P_2, & \text{if } P_2 \text{ is } \lambda_\forall,\ \lambda_\star, \text{ or } \lambda_l\ k \\
P_1 \cap \lambda_\forall, & = \; P_1, & \text{if } P_1 \text{ is } \lambda_\star \text{ or } \lambda_l\ k \\
\lambda_\star \cap P_2 & = \; \lambda_\star, & \text{if } P_2 \text{ is } \lambda_l\ k \\
P_1 \cap \lambda_\star & = \; \lambda_\star, & \text{if } P_1 \text{ is } \lambda_l\ k \\
\lambda_l\ (P_1\ \ldots\ P_n) \cap \lambda_l\ (P'_1\ \ldots\ P'_n) & = \; \lambda_l\ ((P_1 \cap P'_1)\ \ldots\ (P_n \cap P'_n)) & \\
(P_1,\, P_2) \cap (P'_1,\, P'_2) & = \; (P_1 \cap P'_1,\, P_2 \cap P'_2) &
\end{array}
$$

Figure 5.19: Definition of the intersection operator between patterns

$(P_n\ \ldots\ P_{i+1}\ P\ P_{i-1}\ \ldots\ P_1)$ would almost be what we want except that more than contour $k$ may get updated. Instead, we compute the intersection between $P$ and $P_i$ and use the result in the split contour pattern. That is, we update $M_{l_n}$ using $(P_n\ \ldots\ P_{i+1}\ (P \cap P_i)\ P_{i-1}\ \ldots\ P_1)$. The definition of the intersection is presented in Figure 5.19. This definition is that of a function computing the intersection between patterns. It it different from the $\overset{\exists}{\cap}$ relation whose purpose is simply to determine whether some concrete value is abstracted by both its arguments. The $\cap$ function produce a pattern representing the intersection of the input patterns as long as it makes sense. That is, the patterns must have an intersection (according to $\overset{\exists}{\cap}$) and they must not both be *split* patterns.[4] The result of the processing of $D$ is thus:

$$
\begin{array}{l}
\text{If } k = (P_n\ \ldots\ P_1)\ \wedge \\
\quad (\lambda_{l_1}\mathrm{y}_1.\ e_{l'_1}) \in \underline{\triangle}(e_{l_0})\ \wedge \\
\quad (\lambda_{l_j}\mathrm{y}_j.\ e_{l'_j}) \in \underline{\triangle}(e_{l'_{j-1}}),\ \forall\, 2 \leq j \leq n\ \wedge \\
\quad e_l \in \underline{\triangle}(e_{l'_n})\ \wedge \\
\quad \mathrm{y}_i \textit{ is } \text{in fact x} \\
\Rightarrow \text{Update } M_{l_n} \text{ with } (P_n\ \ldots\ P_{i+1}\ (P \cap P_i)\ P_{i-1}\ \ldots\ P_1)
\end{array}
$$

Note that the value of a variable is generally controlled through many abstract values and contours. In the general case, $M_{l_i}$ must be updated to provide more accurate contours,

[4] The intersection between a split pattern and a modelling pattern may lead to a resulting pattern that is *less* accurate. This is because the split point ($\star$) has priority over the modelling pattern it is intersected with. When this situation occurs, the resulting pattern is a split pattern but it does not cause a real update on the pattern-matcher.

which allows $M_{\mathcal{V}}$ to be updated to provide more accurate closures of the form $\lambda_{l_{i+1}}\ k_{i+1}$, which in turn allows $M_{l_{i+1}}$ to be updated to provide more accurate contours, ..., which allows $M_{l_n}$ to be updated to provide more accurate contours. However, a single update request is emitted and we let the rules that ensure consistency do the rest.

#### Split on $\gamma$-Variables

Let $D \equiv$ '**split** $\gamma_{f,k}$ $P$'. The processing of $D$ appears trivial when we note that the return value of a closure $f$, when its body is evaluated in contour $k$, is precisely the result of the evaluation of the body in contour $k$. The only thing that has to be done is to recover the label of the body of the closure and emit a new split demand:

If $f = \lambda_l\ k' \ \wedge\ e_l = (\lambda_l \mathrm{x}.\ e_{l'})$:
⇒**split** $\alpha_{l',k}$ $P$

### 5.2.5 Call Site Monitoring

As explained in the processing of bad call demands, undesirable invocations are logged into the bad-call log and they are taken care of later. When the invocation of $f$ on $v$, denoted as $(f,\, v)$, is put into the bad-call log for call site $e_l$ and contour $k$, denoted as $L_{\mathrm{BC}}(l,k)$, the call site is flagged for future monitoring. Eventually, the demand-driven analysis goes into a call site monitoring phase and monitors each call site that has been flagged.

We describe the processing of the command $C =$ '**monitor-call** $l$ $k$', that is, the monitoring of call site $e_l$ in contour $k$. We insist on the fact that $C$ is not a demand, but simply a command. Once processed, $C$ cannot be considered as achieved. Even if each demand that results from the processed of $C$ is eventually achieved, $C$ still cannot be considered as achieved. New undesirable invocations occurring at $e_l$ in contour $k$ may be discovered later and a new monitoring would be required.

Let $e_l = ({}_l e_{l'}\ e_{l''})$. Let $A$ be the set of all invocations occurring at $e_l$ in $k$ denoted in the form of couples:

$$A = (\alpha_{l',k} \cap \mathcal{V}al\mathcal{C}) \times \alpha_{l'',k}$$

and $L_{\mathrm{BC}}(l,k)$ contains those that are bad invocations. The first situation that we may face in processing $C$ is that no invocation in $A$ is marked as bad. Then the monitoring trivially succeeds:

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

The second situation is the one in which all invocations in $A$ are marked as bad. None should be allowed to occur. Then the adequate processing consists in requesting a demonstration that $e_l$ does not evaluate in contour $k$:

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

Note that $A$ contains only couples that represent invocations occurring at $e_l$ in $k$. The other couples, i.e. those in:

$$(\alpha_{l',k} \cap (\mathcal{V}al\mathcal{B} \cup \mathcal{V}al\mathcal{P})) \times \alpha_{l'',k}$$

represent illegal invocations as it is not a closure that is to be invoked.

The last situation is the one in which bad invocations and good invocations (invocations not yet considered as bad) appear in $A$. The appropriate processing consists in emitting demands that separate the good from the bad cases. If all these demands are eventually achieved, then the first or second situations will apply in the different specialised contours. Once again, the Split-Couples function is used:

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

### 5.2.6 Split-Couples Function

The Split-Couples function is used in two places in the processing of "demands": in the split of an $\alpha$-variable where the expression involved is a call; in the monitoring of a call site. One might have noted that SC is overloaded. In the first case, it receives a set of couple-result pairs and a split pattern. In the second, it receives two sets of couples. Both type signatures for SC are given here:

$$\begin{array}{llll}\textsc{sc} : 2^{(\mathrm{MPat}\times\mathrm{MPat})\times 2^{\mathrm{MPat}}} & \times\ \mathrm{SPat} & \rightarrow 2^{\mathrm{SPat}} \times 2^{\mathrm{SPat}} \\ \textsc{sc} : 2^{\mathrm{MPat}\times\mathrm{MPat}} & \times\ 2^{\mathrm{MPat}\times\mathrm{MPat}} & \rightarrow 2^{\mathrm{SPat}} \times 2^{\mathrm{SPat}}\end{array}$$

Despite the differences in the uses, the task is essentially the same: couples are grouped into equivalence classes and splits operating on the first or on the second components of the couples must be produced such that all non-equivalent couples have been separated by splits. So we describe the implementation of SC in two steps: computing the equivalence classes, finding splits to separate them.

Let us find the equivalence classes in the first use of Split-Couples. Suppose it is used as $\textsc{sc}(\Sigma, P)$. $\Sigma$ is a set of couple-result pairs like $((f,\,v),\,S)$ where $(f,\,v)$ describes an invocation and $S$ is the result of the invocation. $P$ is a split pattern. By the construction of $\Sigma$, there are no two couple-result pairs that have the same couple. Also, we expect that, in each couple-result pair $((f,\,v),\,S) \in \Sigma$, $S$ is non-empty, has some intersection with $P$, and is not spread on $P$. These conditions ensure that the following definition of relation $R$ on couples is one of an equivalence relation:

$$\begin{array}{c}(f_1,\,v_1)\ R\ (f_2,\,v_2) \quad \Leftrightarrow \quad \neg\Big((S_1 \cup S_2) \boxtimes P\Big) \\ \text{where } ((f_1,\,v_1),\,S_1),((f_2,\,v_2),\,S_2) \in \Sigma\end{array}$$

Basically, $R$ says that two couples are related if their associated return values go through the same "hole" of $P$. The desired equivalence classes are those induced by $R$ on the set $\{(f,\,v) \mid ((f,\,v),\,S) \in \Sigma\}$.

Let us do the same in the second use of Split-Couples. Suppose it is used as $\textsc{sc}(S, T)$. $S$ is the set of invocations that occur. $T$ is the set of undesirable invocations. We define relation $R$ this way:

$$(f_1,\,v_1)\ R\ (f_2,\,v_2) \quad \Leftrightarrow \quad ((f_1,\,v_1),(f_2,\,v_2) \in T) \vee ((f_1,\,v_1),(f_2,\,v_2) \notin T)$$

Basically, $R$ says that two couples are related if they are both good or both bad. The desired equivalence classes are those induced by $R$ on $S$.

From this point on, we can now consider that we have a set of couples and that a *colour* has been assigned to each couple. The number of colours may be much smaller than the

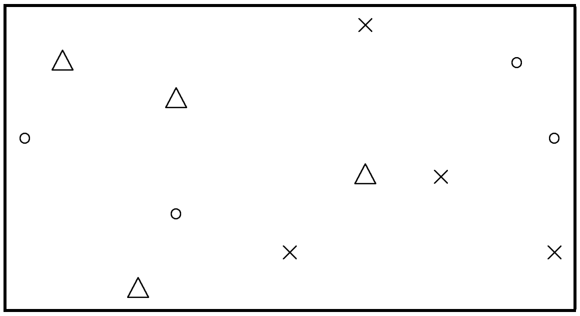

Figure 5.20: Example of couples to separate

number of couples. For example, when the couples have been separated into good and bad calls, there are two colours. To help to understand the task of separating the couples, we choose an illustration that represent couples of different colours. The couples are presented in Figure 5.20 as points on the plan. They are depicted using different symbols to represent different colours. Two couples having the same $x$-coordinate have the same first component but different second components. Similarly for couples having the same $y$-coordinate.

The separation task now consists (in 2D-points terminology) in drawing vertical and horizontal lines (separators) that delimit rectangles in which points of a single colour lie. The simplest separation consists in drawing a complete grid of lines such that each rectangle contains at most one point. However, separations made of fewer separators are desirable because, concretely, each separator translates into a split demand that is emitted on one of the two sub-expressions of a call. Since we cannot presume that any demand is trivial to achieve, demands should be generated with parsimony. A more economical but still naïve method of separation of the couples consists in introducing as many vertical separators as necessary and then to introduce horizontal separators only in the columns that require some. Figure 5.21 presents the separation that is obtained if we proceed this way. It is clearly better than the grid strategy. But it is possible to do better by trying to take advantage of the distribution of the couples. Figure 5.22 presents a more clever separation of the couples. It introduces only 7 separators compared to the 11 introduced by the naïve method.

The illustration using points and colours does not correspond to the couples/equivalence-classes with high fidelity but highlights the main concerns: the separators are uni-dimensional and they should be introduced in small numbers. We can now present the implementation of the process of separation for the classes of couples. Since horizontal and vertical separators

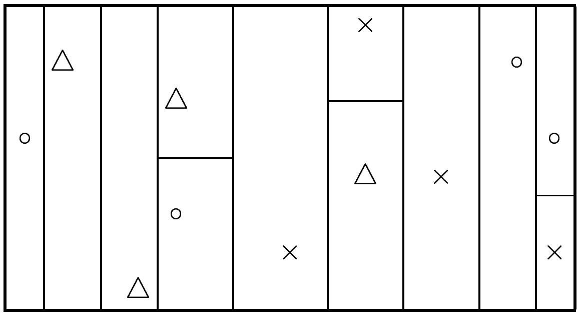

Figure 5.21: Example of a naïve separation

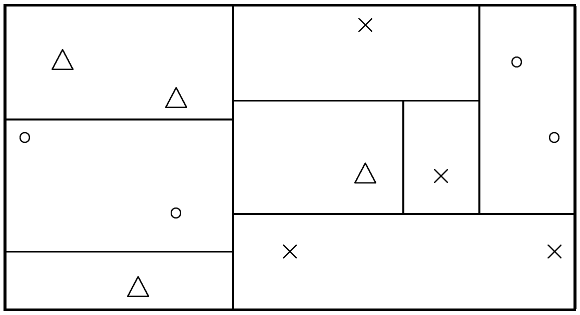

Figure 5.22: Example of a more clever separation

$$
\begin{aligned}
&\textsc{sc}' : 2^{\mathrm{MPat}\times\mathrm{MPat}} \times 2^{(\mathrm{MPat}\times\mathrm{MPat})\times(\mathrm{MPat}\times\mathrm{MPat})} \rightarrow 2^{\mathrm{SPat}} \times 2^{\mathrm{SPat}}\\
&\textsc{sc}'(S, R) = \Big(\{P_1 \mid (P_1,\, P_2) \in D \;\wedge\; P_1 \in \mathrm{SPat}\}\,,\; \{P_2 \mid (P_1,\, P_2) \in D \;\wedge\; P_2 \in \mathrm{SPat}\}\Big)\\
&\qquad\qquad \text{where } A \;\;\; = \Big\{\{(P_1',\, P_2') \in S \mid (P_1',\, P_2')\; R\; (P_1,\, P_2)\}\;\Big|\; (P_1,\, P_2) \in S\Big\}\\
&\qquad\qquad\qquad\;\; B \;\;\; = \Big[\{[\,] \lhd \mathtt{O}\; P_1 \lhd \mathtt{O}\; P_2 \mid (P_1,\, P_2) \in K\}\;\Big|\; K \in A\Big]\\
&\qquad\qquad\qquad\;\; (C,\, \_) = \textsc{sc}'_Q(B)\\
&\qquad\qquad\qquad\;\; D \;\;\; = \Big\{(P_1,\, P_2)\;\Big|\; P_2 \lhd P_1 \lhd [\,] \in C\Big\}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{sc}'_Q, \textsc{sc}'_{\mathtt{O}\,\forall}, \textsc{sc}'_{\mathtt{O}\,\star}, \textsc{sc}'_{\mathtt{C}\,\forall}, \textsc{sc}'_{\mathtt{C}\,\star} : \langle\text{sequence of } 2^{\langle\text{queue of } \{\mathtt{O},\mathtt{C}\}\times\mathrm{MPat}\rangle}\rangle \rightarrow 2^{\langle\text{queue of } \mathrm{MPat}\cup\mathrm{SPat}\rangle} \times \mathbb{N}\\
&\textsc{sc}'_Q([\,]) && = (\emptyset,\, 0)\\
&\textsc{sc}'_Q([K]) && = (\emptyset,\, 0)\\
&\textsc{sc}'_Q([K_1, \ldots, K_{i-1}, \emptyset, K_{i+1}, \ldots, K_n]) && = \textsc{sc}'_Q([K_1, \ldots, K_{i-1}, K_{i+1}, \ldots, K_n])\\
&\textsc{sc}'_Q\Big(\Big[\underbrace{\{[\,]\}, \ldots, \{[\,]\}}_{\geq 2 \text{ times}}\Big]\Big) && = (\emptyset,\, \infty)\\
&\textsc{sc}'_Q([K_1, \ldots, K_n]) && = (N_i,\, n_i), \quad \text{if } \exists\, (\mathtt{O}\; P \lhd q) \in K_1 \;\wedge\; n_i \leq \min(n_1, n_2)\\
& && \quad\; \text{where } (N_1,\, n_1) = \textsc{sc}'_{\mathtt{O}\,\forall}([K_1, \ldots, K_n])\\
& && \qquad\qquad\; (N_2,\, n_2) = \textsc{sc}'_{\mathtt{O}\,\star}([K_1, \ldots, K_n])\\
&\textsc{sc}'_Q([K_1, \ldots, K_n]) && = (N_i,\, n_i), \quad \text{if } \exists\, (\mathtt{C}\; P \lhd q) \in K_1 \;\wedge\; n_i \leq \min(n_1, n_2)\\
& && \quad\; \text{where } (N_1,\, n_1) = \textsc{sc}'_{\mathtt{C}\,\forall}([K_1, \ldots, K_n])\\
& && \qquad\qquad\; (N_2,\, n_2) = \textsc{sc}'_{\mathtt{C}\,\star}([K_1, \ldots, K_n])
\end{aligned}
$$

Figure 5.23: Implementation of the Split-Couples function (to be continued ...)

seem to have a similar cost *a priori*, our approach looks for separators by inspecting both components of the couples level by level. In fact, a breadth-first traversal of *both* components simultaneously is performed in order to have a balance in the complexity of the split patterns that are selected in each dimension. The separation method tries different strategies in a dynamic-programming fashion and selects a shortest separation strategy.

Figure 5.23 presents the implementation of the separation phase of the SC function. The algorithm consists in first taking the (non-empty) equivalence classes among the couples in $S$ induced by relation $R$ and inserting the two components of each couple into a queue. Queues are used for both traversing the components of the couples and for reconstructing split patterns. The split patterns are then extracted from the reconstruction queues. Note that these patterns are intended to split couples, and not just one of the two components. However, as we do in the processing of split demands on $\alpha$-variables related to `cons`-expressions, we keep only the split pattern among the pair of patterns. Costs for the different strategies are returned with the reconstruction queues.

$$
\begin{array}{l}
\textsc{sc}'_{\mathtt{O}\forall}(M) = \Big(\Big\{q \triangleleft \forall \;\Big|\; q \in N\Big\},\, n\Big) \\
\qquad\qquad \text{where } A \qquad = \Big[\{q \mid (\mathtt{O}\ P \triangleleft q) \in K\} \;\Big|\; K \in M\Big] \\
\qquad\qquad\qquad (N,\, n) = \textsc{sc}'_{Q}(A)
\end{array}
$$

$$
\begin{array}{l}
\textsc{sc}'_{\mathtt{O}\star}(M) = \left\{ \begin{array}{ll} (\emptyset,\, \infty), & \text{if } \exists\, K \in M.\ (\mathtt{O}\ \forall \triangleleft q) \in K \\ (\{q_0\} \cup N'_1 \cup N'_2 \cup N'_3,\, 1 + n_1 + n_2 + n_3), & \text{otherwise} \end{array} \right. \\
\qquad \text{where } A \qquad\quad = \Big[\{q \mid (\mathtt{O}\ \#\mathbf{f} \triangleleft q) \in K\} \;\Big|\; K \in M\Big] \\
\qquad\qquad (N_1,\, n_1) = \textsc{sc}'_{Q}(A) \\
\qquad\qquad N'_1 \qquad\;\; = \Big\{q \triangleleft \#\mathbf{f} \;\Big|\; q \in N_1\Big\} \\
\qquad\qquad B \qquad\;\;\; = \Big[\{q \triangleleft \mathtt{C}\ P \mid (\mathtt{O}\ P \triangleleft q) \in K \ \wedge\ (P \text{ is } \lambda_\forall \text{ or } \lambda_l\ k)\} \;\Big|\; K \in M\Big] \\
\qquad\qquad (N_2,\, n_2) = \textsc{sc}'_{Q}(B) \\
\qquad\qquad N'_2 \qquad\;\; = \Big\{q \triangleleft P \;\Big|\; P \triangleleft q \in N_2\Big\} \\
\qquad\qquad C \qquad\;\;\; = \Big[\{q \triangleleft \mathtt{O}\ P_1 \triangleleft \mathtt{O}\ P_2 \mid (\mathtt{O}\ (P_1,\, P_2) \triangleleft q) \in K\} \;\Big|\; K \in M\Big] \\
\qquad\qquad (N_3,\, n_3) = \textsc{sc}'_{Q}(C) \\
\qquad\qquad N'_3 \qquad\;\; = \Big\{q \triangleleft (P_1,\, P_2) \;\Big|\; P_2 \triangleleft P_1 \triangleleft q \in N_3\Big\} \\
\qquad\qquad \{q_0\} \qquad = \Big\{\textsc{sc}''_{Q}(q) \triangleleft \star \;\Big|\; K \in M,\ (\mathtt{O}\ P \triangleleft q) \in K\Big\}
\end{array}
$$

$$
\begin{array}{l}
\textsc{sc}'_{\mathtt{C}\forall}(M) = \Big(\Big\{q \triangleleft \lambda_\forall \;\Big|\; q \in N\Big\},\, n\Big) \\
\qquad\qquad \text{where } A \qquad = \Big[\{q \mid (\mathtt{C}\ P \triangleleft q) \in K\} \;\Big|\; K \in M\Big] \\
\qquad\qquad\qquad (N,\, n) = \textsc{sc}'_{Q}(A)
\end{array}
$$

$$
\begin{array}{l}
\textsc{sc}'_{\mathtt{C}\star}(M) = \left\{ \begin{array}{ll} (\emptyset,\, \infty), & \text{if } \exists\, K \in M.\ (\mathtt{C}\ \lambda_\forall \triangleleft q) \in K \\ (\{q_0\} \cup \bigcup_{l \in L} N'_l,\, 1 + \sum_{l \in L} n_l)\,, & \text{otherwise} \end{array} \right. \\
\qquad \text{where } L \quad = \Big\{l \in \triangle(e_{l_0}) \;\Big|\; e_l \text{ is a } \lambda\text{-expression}\Big\} \\
\qquad\qquad A_l \quad = \Big[\{q \triangleleft \mathtt{O}\ P_1 \triangleleft \ldots \triangleleft \mathtt{O}\ P_j \mid (\mathtt{C}\ \lambda_l\ (P_1\ \ \ldots\ \ P_j) \triangleleft q) \in K\} \;\Big|\; K \in M\Big] \\
\qquad\qquad (N_l,\, n_l) \\
\qquad\qquad\qquad = \textsc{sc}'_{Q}(A_l) \\
\qquad\qquad N'_l \quad = \left\{ q \triangleleft \lambda_l\ (P_1\ \ \ldots\ \ P_j) \;\middle|\; \begin{array}{l} \text{there are } j \text{ visible variables at label } l\ \wedge \\ (P_j \triangleleft \ldots \triangleleft P_1 \triangleleft q) \in N_l \end{array} \right\} \\
\qquad\qquad \{q_0\} = \Big\{\textsc{sc}''_{Q}(q) \triangleleft \lambda_\star \;\Big|\; K \in M,\ (\mathtt{C}\ P \triangleleft q) \in K\Big\}
\end{array}
$$

$$
\begin{array}{l}
\textsc{sc}''_{Q} : \langle\text{queue of } \{\mathtt{O}, \mathtt{C}\} \times \text{MPat}\rangle \to \langle\text{queue of MPat}\rangle \\
\textsc{sc}''_{Q}([\,]) \qquad\quad = [\,] \\
\textsc{sc}''_{Q}(\mathtt{O}\ P \triangleleft q) = \textsc{sc}''_{Q}(q) \triangleleft \forall \\
\textsc{sc}''_{Q}(\mathtt{C}\ P \triangleleft q) = \textsc{sc}''_{Q}(q) \triangleleft \lambda_\forall
\end{array}
$$

Figure 5.23: Implementation of the Split-Couples function (continued ...)

Let us describe the implementation. The main function for the separation of couples is $\textsc{sc}'$, which takes the set of couples passed to $\textsc{sc}$ and the equivalence relation $R$. It acts as an interface in front of the central function $\textsc{sc}'_Q$. Couples are grouped into equivalence classes, represented as sets, and these equivalence classes are grouped into a sequence. We denote sequences using square brackets instead of curly braces and all set operations can be used on the sequences, like definition in comprehension and membership test. We use sequences to contain the equivalence classes instead of sets not that much because they are ordered, but because the same element can appear more than once in a sequence. This feature is useful because, eventually, classes may simply consist of a set containing the empty queue and it is important to be able to distinguish whether there is one or more of these classes.

Central function $\textsc{sc}'_Q$ operates quite similarly to the slicing algorithm that is described in the section on model update. The difference lies in the fact that a sequence of sets of queues is manipulated instead of a single queue and that, at each possible split point, a split may, or may not, be introduced. As in all algorithms performing a breadth-first traversal of patterns, the patterns in the deconstruction queues are marked as either *object* nodes ($\mathtt{O}$) or as *closure* nodes ($\mathtt{C}$). A non-terminal step in the operations of $\textsc{sc}'_Q$ consists in computing a separation strategy for an object node or for a closure node. Note that, for a certain invocation of $\textsc{sc}'_Q$, if one queue in some set in the sequence has length $l$, then all queues have length $l$. Also, if the first element to be extracted from that queue is of the object kind, then it is also the case for all queues. Similarly for the closure kind. Computing a separation strategy for an object node consists in computing one using the blind auxiliary function $\textsc{sc}'_{\mathtt{O}\,\forall}$, computing another using the discriminating auxiliary function $\textsc{sc}'_{\mathtt{O}\,\star}$, and selecting the “best” of both strategies. A strategy has an infinite cost when it does not provide a proper separation. When both strategies have an infinite cost, taking the “best” consists in taking any strategy among the two. Computing a separation strategy for a closure node proceeds in a similar way, using auxiliary functions $\textsc{sc}'_{\mathtt{C}\,\forall}$ and $\textsc{sc}'_{\mathtt{C}\,\star}$.

Blind auxiliary functions $\textsc{sc}'_{\mathtt{O}\,\forall}$ and $\textsc{sc}'_{\mathtt{C}\,\forall}$ elaborate separation strategies by choosing not to insert a split at the current inspection point. Concretely, the first element of each queue is discarded. This means that the information that remains in the equivalence classes for performing the separation is reduced. However, the advantage is that no new separator is introduced at this point. The shortened queues are passed to $\textsc{sc}'_Q$ to let it elaborate a separation strategy based on the remaining information. The splits that it proposes are then updated to allow complete patterns to eventually be reconstructed.

Discriminating auxiliary function $\textsc{sc}'_{\mathtt{0}\star}$ elaborates a separation strategy by choosing to perform a split at the current inspection point. Queues that have a Boolean, a closure, or a pair as their first element are taken separately. So three specialised versions of the equivalence classes are obtained. A separator is introduced. The cost of the resulting strategy is the sum of the costs of the sub-strategy for each specialised partition, plus one for the additional separator. Since a split is done, the sub-patterns of the inspected patterns become apparent and queues are updated accordingly at deconstruction and at reconstruction. Discriminating auxiliary function $\textsc{sc}'_{\mathtt{C}\star}$ proceeds in a similar manner with closure inspection nodes. However, instead of making three versions of the partition based on the type, $|L|$ specialised versions are made, where $L$ is the set of labels of $\lambda$-expressions. For $\textsc{sc}'_{\mathtt{0}\star}$ and $\textsc{sc}'_{\mathtt{C}\star}$, an immediate split may be impossible if there is a queue that contains an "ambiguous" pattern. That is, if a queue contains '$\forall$' or '$\lambda_\forall$', respectively. In such a case, the separation strategy is marked as having an infinite cost. It is then rejected by upper levels in the separation strategy selection.

We come back to the description of the different cases in $\textsc{sc}'_Q$. The first terminal cases are the success of a separation strategy. The equivalence classes are successfully separated if there is at most one class left. No separator is required and the cost of the separation strategy is 0. The other terminal case is the failure of a separation strategy. The separation fails if there remains at least two equivalence classes containing empty queues. This means that no information remains about the original couples and incompatible ones cannot be distinguished. An infinitely costly strategy is returned. Such a failure is not an extraordinary event. It simply means that insufficient separators are selected in upper stages of the separation strategy selection. Note that the complete selection process cannot fail as introducing separators at every inspection point is guaranteed to produce a successful strategy. Finally, there is a "clean-up" non-terminal case. It removes empty classes from the sequence. An empty class occurs when no representative of a certain type (or closure label) can be found among the queues of a certain class during a previous specialisation.

This completes the description of the implementation of the Split-Couples function. Since its internal operations are slightly complex, we present a short example illustrating the computations it makes. Let us consider the couples formed by the invocations of $\lambda_3$ () and $\lambda_5$ () on $\#\mathbf{f}$ and $\lambda_\forall$. Note that, normally, modelling pattern $\lambda_\forall$ is not supposed to be manipulated directly as a value. But we need to split very simple couples in order to keep the example to a reasonable size. Suppose that the couple ($\lambda_5$ (), $\#\mathbf{f}$) is marked as bad.

The main computations that are made to split the couples are the following:

$$
\begin{array}{l}
\textsc{sc}(\{(\lambda_3\ (),\ \#\mathbf{f}), (\lambda_3\ (),\ \lambda_\forall), (\lambda_5\ (),\ \#\mathbf{f}), (\lambda_5\ (),\ \lambda_\forall)\},\ \{(\lambda_5\ (),\ \#\mathbf{f})\}) \\
\textsc{sc}'(\{(\lambda_3\ (),\ \#\mathbf{f}), (\lambda_3\ (),\ \lambda_\forall), (\lambda_5\ (),\ \#\mathbf{f}), (\lambda_5\ (),\ \lambda_\forall)\},\ R) \\
\quad \text{where } R = \{(\lambda_3\ (),\ \#\mathbf{f}), (\lambda_3\ (),\ \lambda_\forall), (\lambda_5\ (),\ \lambda_\forall)\}^2 \cup \{(\lambda_5\ (),\ \#\mathbf{f})\}^2 \\
\left[\begin{array}{l}
\textsc{sc}'_Q([\{[\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \lambda_\forall],\ [\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \lambda_\forall]\}, \{[\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \#\mathbf{f}]\}]) \\
\ldots \\
\Rightarrow\ \ (\{[\forall,\ \star], [\star,\ \lambda_\forall], [\#\mathbf{f},\ \lambda_\star]\},\ 3)
\end{array}\right. \\
\Rightarrow\ \ (\{\star, \lambda_\star\},\ \{\star\})
\end{array}
$$

The computations made by central function $\textsc{sc}'_Q$ are shown in Figure 5.24. Despite the smallness of the input to $\textsc{sc}$, an impressive amount of computations has to be performed.

In the trace of the computations performed by $\textsc{sc}'_Q$, the main ideas are illustrated. The trace of each use of the central function or of an auxiliary function is presented in a separate box. With the notable exception that blind auxiliary functions only use $\textsc{sc}'_Q$ once and no separate box is depicted for these uses of $\textsc{sc}'_Q$. Function $\textsc{sc}'_Q$ uses either auxiliary functions $\textsc{sc}'_{\mathtt{O}\forall}$ and $\textsc{sc}'_{\mathtt{O}\star}$ when the next pattern in the queues is of the object kind, and $\textsc{sc}'_{\mathtt{C}\forall}$ and $\textsc{sc}'_{\mathtt{C}\star}$ when the next pattern is of the closure kind. Each time, the best of both resulting strategies is returned. Blind auxiliary functions $\textsc{sc}'_{\mathtt{O}\forall}$ and $\textsc{sc}'_{\mathtt{C}\forall}$ simply consume the first pattern in each queue, sometimes leading to equivalence classes containing only empty queues. Discriminating auxiliary function $\textsc{sc}'_{\mathtt{O}\star}$ separates its input queues into those that start with a Boolean, those that start with a closure, and those that start with a pair. Sub-strategies are elaborated for each new partitions of queues. They are then combined together with the addition of a queue containing parts of a new split pattern performing the discrimination directly introduced by $\textsc{sc}'_{\mathtt{O}\star}$ itself. Similarly, $\textsc{sc}'_{\mathtt{C}\star}$ separates its input queues into those that start with a closure having 3 as a label and those that start with a closure having 5 as a label. Note how the reconstruction queues are modified depending on which type or which label they are the result for. Unfortunately, not all cases appearing in the implementation of $\textsc{sc}'$ are illustrated in the example. But a complete one would likely result in a huge trace. We tried to keep a balance between completeness and comprehensibility.

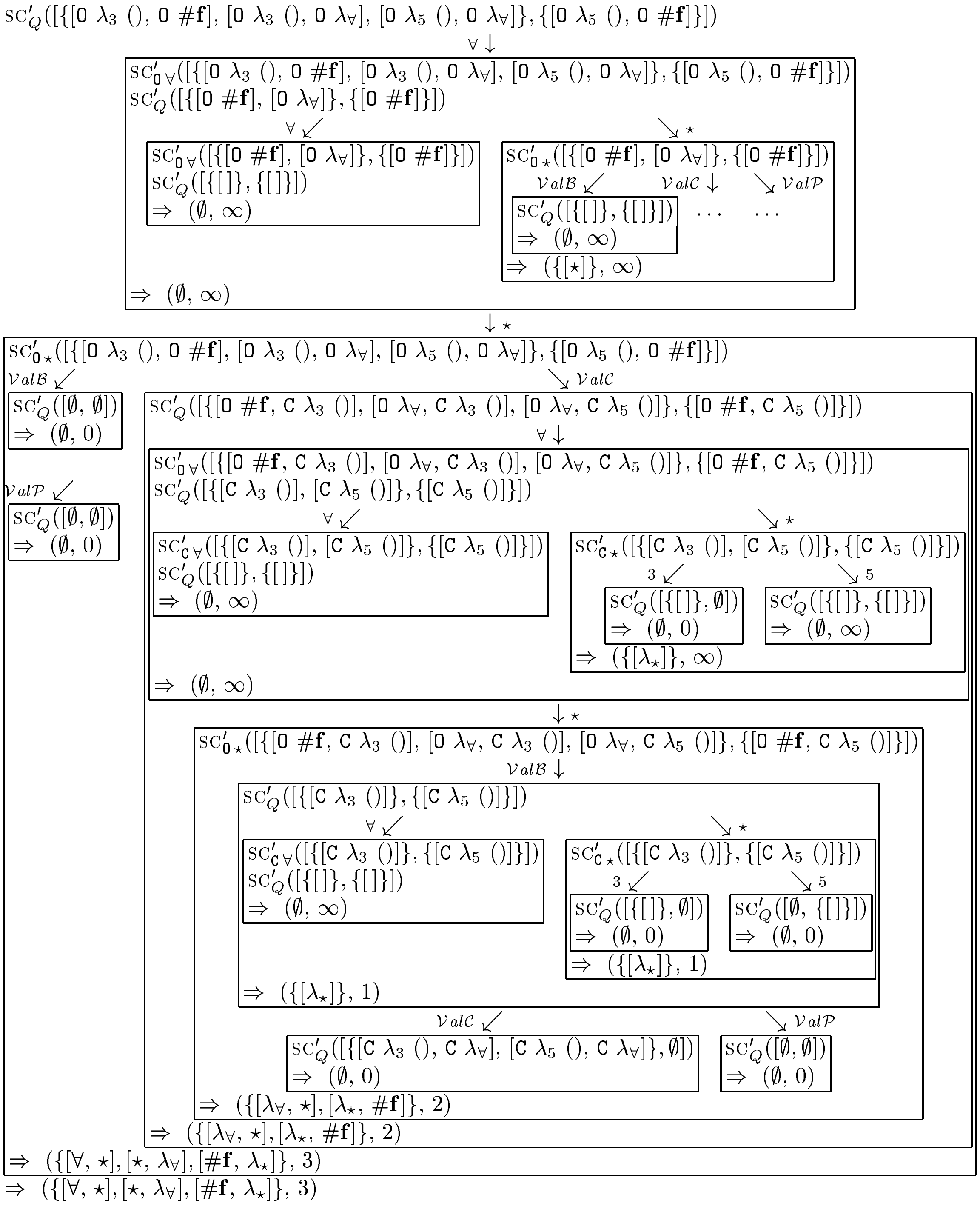
$\textsc{sc}'_Q([\{[\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \lambda_\forall],\ [\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \#\mathbf{f}]\}])$
$\forall\downarrow$
$\textsc{sc}'_{\mathtt{O}\,\forall}([\{[\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \lambda_\forall],\ [\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \#\mathbf{f}]\}])$
$\textsc{sc}'_Q([\{[\mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \#\mathbf{f}]\}])$
$\forall\swarrow$
$\searrow\star$
$\textsc{sc}'_{\mathtt{O}\,\forall}([\{[\mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \#\mathbf{f}]\}])$
$\textsc{sc}'_Q([\{[\,]\},\{[\,]\}])$
$\Rightarrow\ (\emptyset,\ \infty)$
$\textsc{sc}'_{\mathtt{O}\,\star}([\{[\mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \#\mathbf{f}]\}])$
$\mathcal{V}al\mathcal{B}\swarrow$
$\mathcal{V}al\mathcal{C}\downarrow$
$\searrow\mathcal{V}al\mathcal{P}$
$\textsc{sc}'_Q([\{[\,]\},\{[\,]\}])$
$\ldots\quad\ldots$
$\Rightarrow\ (\emptyset,\ \infty)$
$\Rightarrow\ (\{[\star]\},\ \infty)$
$\Rightarrow\ (\emptyset,\ \infty)$
$\downarrow\star$
$\textsc{sc}'_{\mathtt{O}\,\star}([\{[\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \#\mathbf{f}],\ [\mathtt{O}\ \lambda_3\ (),\ \mathtt{O}\ \lambda_\forall],\ [\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \lambda_\forall]\},\{[\mathtt{O}\ \lambda_5\ (),\ \mathtt{O}\ \#\mathbf{f}]\}])$
$\mathcal{V}al\mathcal{B}\swarrow$
$\searrow\mathcal{V}al\mathcal{C}$
$\textsc{sc}'_Q([\emptyset,\ \emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\mathcal{V}al\mathcal{P}\swarrow$
$\textsc{sc}'_Q([\emptyset,\emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\textsc{sc}'_Q([\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_5\ ()]\}])$
$\forall\downarrow$
$\textsc{sc}'_{\mathtt{O}\,\forall}([\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_5\ ()]\}])$
$\textsc{sc}'_Q([\{[\mathtt{C}\ \lambda_3\ ()],\ [\mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$\forall\swarrow$
$\searrow\star$
$\textsc{sc}'_{\mathtt{C}\,\forall}([\{[\mathtt{C}\ \lambda_3\ ()],\ [\mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$\textsc{sc}'_Q([\{[\,]\},\{[\,]\}])$
$\Rightarrow\ (\emptyset,\ \infty)$
$\textsc{sc}'_{\mathtt{C}\,\star}([\{[\mathtt{C}\ \lambda_3\ ()],\ [\mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$3\swarrow$
$\searrow 5$
$\textsc{sc}'_Q([\{[\,]\},\emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\textsc{sc}'_Q([\{[\,]\},\{[\,]\}])$
$\Rightarrow\ (\emptyset,\ \infty)$
$\Rightarrow\ (\{[\lambda_\star]\},\ \infty)$
$\Rightarrow\ (\emptyset,\ \infty)$
$\downarrow\star$
$\textsc{sc}'_{\mathtt{O}\,\star}([\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_3\ ()],\ [\mathtt{O}\ \lambda_\forall,\ \mathtt{C}\ \lambda_5\ ()]\},\{[\mathtt{O}\ \#\mathbf{f},\ \mathtt{C}\ \lambda_5\ ()]\}])$
$\mathcal{V}al\mathcal{B}\downarrow$
$\textsc{sc}'_Q([\{[\mathtt{C}\ \lambda_3\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$\forall\swarrow$
$\searrow\star$
$\textsc{sc}'_{\mathtt{C}\,\forall}([\{[\mathtt{C}\ \lambda_3\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$\textsc{sc}'_Q([\{[\,]\},\{[\,]\}])$
$\Rightarrow\ (\emptyset,\ \infty)$
$\textsc{sc}'_{\mathtt{C}\,\star}([\{[\mathtt{C}\ \lambda_3\ ()]\},\{[\mathtt{C}\ \lambda_5\ ()]\}])$
$3\swarrow$
$\searrow 5$
$\textsc{sc}'_Q([\{[\,]\},\emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\textsc{sc}'_Q([\emptyset,\ \{[\,]\}])$
$\Rightarrow\ (\emptyset,\ 0)$
$\Rightarrow\ (\{[\lambda_\star]\},\ 1)$
$\Rightarrow\ (\{[\lambda_\star]\},\ 1)$
$\mathcal{V}al\mathcal{C}\swarrow$
$\searrow\mathcal{V}al\mathcal{P}$
$\textsc{sc}'_Q([\{[\mathtt{C}\ \lambda_3\ (),\ \mathtt{C}\ \lambda_\forall],\ [\mathtt{C}\ \lambda_5\ (),\ \mathtt{C}\ \lambda_\forall]\},\emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\textsc{sc}'_Q([\emptyset,\emptyset])$
$\Rightarrow\ (\emptyset,\ 0)$
$\Rightarrow\ (\{[\lambda_\forall,\ \star],[\lambda_\star,\ \#\mathbf{f}]\},\ 2)$
$\Rightarrow\ (\{[\lambda_\forall,\ \star],[\lambda_\star,\ \#\mathbf{f}]\},\ 2)$
$\Rightarrow\ (\{[\forall,\ \star],[\star,\ \lambda_\forall],[\#\mathbf{f},\ \lambda_\star]\},\ 3)$
$\Rightarrow\ (\{[\forall,\ \star],[\star,\ \lambda_\forall],[\#\mathbf{f},\ \lambda_\star]\},\ 3)$

Figure 5.24: Example of computation made by Split-Couples

### 5.2.7 Remarks

We conclude the section on demand processing with a few remarks. The first one is the observation that we took care of choosing processing rules that emit sufficient and necessary sub-demands. In the presentation of certain processing rules, we mentioned that some of the emitted split demands are more aggressive than what is really needed. For example, it is the case in the processing of bound demands, in the processing of split demands on $\alpha$-variables where the expression is a pair construction, and with the split demands prescribed by function $\textsc{sc}$. However, these demands cannot be designed as *unnecessary* since the property they express is indeed true. In fact, all split demands *are necessary.* This may seem surprising but it should be noted that the abstract interpretation tries to be a simplified representative of the concrete interpretation. And in concrete interpretation, at most one value is the result of each evaluation of an expression in a concrete contour. That concrete value, taken alone, cannot be spread on any split pattern. Since the abstract evaluation of an expression in an abstract contour represents a (usually infinite) union of (non-spreading) concrete evaluations, it is legitimate to ask for a split of this evaluation into non-spreading abstract evaluations. The split demand may not be achievable but, at least, the property it expresses is true.

We have chosen split demands to be the main tool in the translation of the needs of the optimiser into model update prescriptions. They are the basic operations that are performed to prepare the analysis results for adequate processing of bound, never, and bad call demands. However, in most of the cases, we could proceed otherwise and bound demands could be processed and transformed mostly into new bound demands. For example, the processing of $D \equiv$ '**show** $\alpha_{l,k} \subseteq B$', for $B$ being some modelling pattern, or union of modelling patterns, could easily be done by emitting new bound and never demands when $e_l$ is $\#\mathtt{f}_l$, $(\mathtt{if}_l\ e_{l'}\ e_{l''}\ e_{l'''})$, $(\mathtt{cons}_l\ e_{l'}\ e_{l''})$, $(\mathtt{car}_l\ e_{l'})$, $(\mathtt{cdr}_l\ e_{l'})$, or $(\mathtt{pair?}_l\ e_{l'})$. By having an update of the model additionally, expressions $\mathrm{x}_l$ and $(\lambda_l \mathrm{x}.\ e_{l'})$ could easily be processed too. Also, never demands, which can be seen as special variants of bound demands operating on $\delta$-variables, could be processed by emitting new bound, never, and bad call demands. However, the processing of bad call demands and that of $D$, where $e_l = ({}_l e_{l'}\ e_{l''})$, would be problematic. Consider processing $D$ knowing that:

$$\alpha_{l',k} \quad = \quad \{f\}$$

$$\begin{array}{lcl}
\alpha_{l'',k} & = & \{v\} \\
\mathsf{call}(l,f,v,k) & = & k' \\
\gamma_{f,k'} & \not\subseteq & B
\end{array}$$

Should the new bound demand '**show** $\gamma_{f,k'} \subseteq B$' be emitted, blaming $f$ for the violation of the bound? Or should a demand be emitted that asks for a demonstration that the call does not occur at all? If so, by emitting '**show** $\delta_{l',k} = \emptyset$' or by emitting '**show** $\delta_{l'',k} = \emptyset$'? At least one of the three properties expressed in these demands has to be true. But which one? Always choosing the right one would require an oracle. And emitting three "or-related" demands seems, if not impossible, far from obvious. So it seems that split demands are unavoidable if adequate processing of demands like $D$ is desired. And since split demands and their complex processing is necessary, we chose to use them extensively and simplify the processing of the other demands.

The final remark about our processing rules is that the rules always propose a single "plan" to achieve the processed demands. As we mention in the previous remark, in some situations, it would be useful to be able to express things like this set of properties or that one needs to be verified to achieve the processed demand. Since we chose not to allow the execution of alternate plans, only a single plan is allowed and consequently it must include only necessary demands. This may unduly delay the achievement of the processed demands. Indeed, if it were possible to propose two plans, the normal, necessary plan could coexist with an alternate, aggressive plan that would immediately try to show a property that is *only probably* true. The knowledge that a property is probably true could come from profiling statistics on the program, for example. The processed demand would be achieved as soon as one of the plans is completed. Typically, the aggressive plan would "have guessed right" and succeed quickly. But sometimes it would result in the launch of unfeasible demands that could cause a considerable waste of analysis efforts. It would certainly be interesting to investigate on the value of allowing alternate plans in the future.

## 5.3 Complete Approach

Now that all the necessary tools have been presented, we can describe the complete demand-driven analysis approach. As mentioned in the previous chapter, the demand-driven analysis is divided in two parts. A *preliminary analysis* is first performed and then the demand-driven *cycle* is entered.

The preliminary analysis simply consists in analysing the program $e_{l_0}$ using the *initial model*. The cycle needs these preliminary results in order to start. The initial model is relatively coarse. There is one abstract Boolean, one abstract pair, one abstract closure per $\lambda$-expression, and one abstract contour for the body of each $\lambda$-expression plus the main contour $()$. More formally, the initial model $\mathcal{M}_0$ is built on the following pattern-matchers:

$$\begin{array}{lcl} M_{\mathcal{V}} & = & \{\#\mathbf{f}, (\forall, \forall)\} \cup \{\lambda_l\ (\underbrace{\forall\ \ldots\ \forall}_{n(l)\text{ times}}) \mid l \in L\} \\ M_l & = & \{(\ \underbrace{\forall\ \ldots\ \forall}_{n(l)+1\text{ times}}\ )\}, \quad l \in L \\ \text{where } L & = & \{l \in \triangle(l_0) \mid e_l \text{ is a } \lambda\text{-expression}\} \\ n(l) & = & \text{number of variables visible at label } l \end{array}$$

In short, $\mathcal{M}_0$ is the simplest model that does not mix the three types of values and the closures coming from different $\lambda$-expressions. We believe that $\mathcal{M}_0$ is a good compromise between simplicity and accuracy. If $\mathcal{M}_0$ were coarser, the quality of the preliminary analysis results would be too low. Also, extra mechanisms would have to be added in the set of demands and the demand processing rules to take anonymous closures or values into account. On the other hand, if $\mathcal{M}_0$ were more accurate, more time would be spent in the preliminary analysis without evidence that this extra accuracy is useful at all. The demand-driven cycle is better informed to choose which part of the abstract model ought to be made more accurate.

The demand-driven cycle is the repetition of the *model-update* and *re-analysis* phases. The cycle ends when there is no time left or there are no more dynamic safety types tests to remove. The model-update phase consists in making a modification to the abstract model through demand processing. The re-analysis phase simply performs an analysis of the program using the newly updated model. Hopefully, the modification to the model makes the new analysis results more precise. Note that there is no guarantee that the modification leads to more precise results. Note also that what we mean by “more precise” is not having analysis results expressed using more precise abstract values, but having analysis results that are more informative, or, stated differently, less overly conservative. For example, suppose that among the analysis results, we have that:

$$\alpha_{l,k} = \{(\forall,\, \forall)\}$$

and that, after a model update and a re-analysis (assuming that $k$ has not been specialised):

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

These new results are expressed using more precise abstract values but they are not more precise themselves. What we know is that $e_l$, when evaluated in contour $k$, can produce any pair. These new results are not less conservative. However, if the new results are:

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

we can say that they are more precise, or more informative.

The model-update phase proceeds by *generating* and *processing* demands and then *selecting* a particular model update. The idea is that the initial demands directly reflect the needs of the optimiser and that the processing of demands is a kind of translation from the needs of the optimiser to prescriptions of model updates. All suggestions of model update that can be obtained from the current analysis results are gathered and the selection occurs among the suggestions. In order to gather the suggestions of model update, the demands that are normally processed by modifying the model are kept apart without being processed. Only those that do not modify the model are processed.

The execution of the model-update phase consists in maintaining a set of demands to process. When there are no more demands to process, a selection occurs among the model-modifying demands that have been gathered. The demands that are put in the set initially are those reflecting the needs of the optimiser. These initial demands correspond exactly to the constraints that would be violated if the safety constraints for the program using the current model were generated and confronted to the analysis results. Formally, these demands are:

$$\left\{\textbf{show } \alpha_{l',k} \subseteq \mathcal{V}al\mathcal{C} \;\middle|\; ({}_{l}e_{l'}\ e_{l''}) \in \triangle(l_0) \ \wedge\ k \in \mathcal{C}ont \ \wedge\ \alpha_{l',k} \not\subseteq \mathcal{V}al\mathcal{C}\right\} \cup$$
$$\left\{\textbf{show } \alpha_{l',k} \subseteq \mathcal{V}al\mathcal{P} \;\middle|\; \begin{array}{l} \big((\texttt{car}_l\ e_{l'}) \in \triangle(l_0) \ \vee\ (\texttt{cdr}_l\ e_{l'}) \in \triangle(l_0)\big) \ \wedge \\ k \in \mathcal{C}ont \ \wedge \\ \alpha_{l',k} \not\subseteq \mathcal{V}al\mathcal{P} \end{array}\right\}$$

Once the initial demands are inserted in the set, demand processing starts. A demand is extracted from the set and processed provided that it is not a model-modifying demand. Otherwise it is inserted in the set of model-modifying demands. The processing of an ordinary demand usually causes the emission of new demands. So processing continues until the set of demands to process is empty. Of course, verifications are done to ensure that a demand is not processed more than once. If the set of demands to process becomes empty, but there are call sites to monitor, the monitoring of all those sites is triggered. The monitoring usually causes new demands to be emitted. If there is no site to monitor, then the demand processing has completed. If the allotted time expires during demand processing, the processing is stopped and the selection is done immediately.

The model-modifying demands are of the form:

$$\begin{array}{l} \textbf{split } \mathcal{V}al\mathcal{C}\ P \\ \textbf{split } \mathcal{V}al\mathcal{P}\ P \\ \textbf{split } \beta_{\mathrm{x},k,l}\ P \end{array}$$

The selection of the model update is done on a space consumption basis. In our prototype, the data structures for the abstract model and the analysis results use a considerable amount of space. So the criterion that is used to select the “best” model update consists in trying to minimise the amount of space used by the model and the results. Despite the fact that this criterion is relatively naïve, it is quite effective. A model update that leads to more precise analysis results is favoured because the number of abstract values propagated during the analysis using the proposed model has a tendency to decrease. However, including the size of the abstract model in the criterion is crucial because it ensures that the gains in the size of the results are not obtained by causing the model to expand too much. The inconvenience associated to this criterion is that a re-analysis has to be performed for each model update proposal.

A summary of the complete demand-driven approach is presented in Figure 5.25. Many operations are only informally specified. They are italicised to indicate that their definition can found elsewhere. Here is the meaning of each variable of the algorithm. The current abstract model is $\mathcal{M}$. The current analysis results are $\mathcal{R}$. The set of demands to process is $S$. The demands already seen in this period of the cycle are in $T$. The model-modifying demands are kept in $U$. Variable $F$ contains the call sites flagged for future monitoring. Naturally, the couples describing the bad invocations (closure and argument) are kept in the bad-call log $L_{\mathrm{BC}}$. Variables $D$ and $S'$ act as temporaries and contain a demand and a set of demands, respectively.

## 5.4 Example of Demand-Driven Analysis

We illustrate the demand-driven analysis algorithm by analysing a small program. Despite its small size, it is designed to be relatively intricate. At least, for an analyser. A trace of the execution of the demand-driven analysis is given. The processing of each demand and its effects are presented. The trace includes the set of demands to process, markers to distinguish the model-modifying demands, the bad-call log and the flagged call sites. The evolution of the abstract model through the updates is presented. Also, excerpts of the current analysis results are shown in order to bring some justification to the presented demand processing. Let us begin the example.

The program to analyse is the following:

$$
\begin{array}{l}
\texttt{(}_{1}\texttt{(}\lambda_{2}\texttt{swap.}\\
\qquad\texttt{(}_{3}\texttt{swap}_{4}\ \texttt{(car}_{5}\ \texttt{(}_{6}\texttt{swap}_{7}\ \texttt{(cons}_{8}\ \texttt{(}\lambda_{9}\texttt{x. x}_{10}\texttt{)}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{(cons}_{11}\ \texttt{(}\lambda_{12}\texttt{y. \#f}_{13}\texttt{)}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \texttt{\#f}_{14}\texttt{))))))}\\
\quad\texttt{(}\lambda_{15}\texttt{p. (cons}_{16}\ \texttt{(cdr}_{17}\ \texttt{p}_{18}\texttt{)}\ \texttt{(car}_{19}\ \texttt{p}_{20}\texttt{))))}
\end{array}
$$

Essentially, a function ‘swap’ is defined and used by the “main program”. ‘Swap’ takes a pair and returns a new pair where the CAR- and CDR-fields have been swapped. The main program builds a #f-terminated list containing the identity function and a constant function. It then calls ‘swap’ on the list and extracts the CAR-field from the result. This is equivalent to dropping the head of the list. Finally, it calls ‘swap’ on this shortened list. It is easy for a human reader to convince himself that this program does not lead to an error when it is evaluated. Consequently, it is natural to hope that the demand-driven analysis

```
$\mathcal{M} := \mathcal{M}_0$;   $\mathcal{R} := \mathrm{FW}(e_{l_0}, \mathcal{M})$                /* preliminary analysis */
while there is time                                           /* demand-driven cycle */
    $S := \{$initial demands$\}$;   $T := S$;   $U := \emptyset$;   $F := \emptyset$;
    $L_{\mathrm{BC}}(l, k) = \emptyset$, $\forall\, l \in \triangle(l_0), k \in \mathcal{C}ont$;
    if $S = \emptyset$ then exit
    do
        while there is time and $S \neq \emptyset$                       /* demand processing */
            let $D \in S$;   $S := S - \{D\}$
            if D is model-modifying then
                $U := U \cup \{D\}$
            else if $D \equiv$ 'bad-call $l\ f\ v\ k$' then
                process D with flagged couple $(l,\, k)$ put in $F$, if necessary
            else
                process D with emitted demands in $S'$
                $S := S \cup (S' - T)$;   $T := T \cup S'$
            end if
        end while
        while there is time and $F \neq \emptyset$                     /* call site monitoring */
            let $(l,\, k) \in F$;   $F := F - \{(l,\, k)\}$
            process 'monitor-call $l\ k$' with emitted demands in $S'$
            $S := S \cup (S' - T)$;   $T := T \cup S'$
        end while
    while there is time and $S \neq \emptyset$
    if $U = \emptyset$ then
        exit
    else
        let D be the best demand in U                          /* selection of a ... */
        process D with modified model in $\mathcal{M}$                 /* ... model update */
    end if
    $\mathcal{R} := \mathrm{FW}(e_{l_0}, \mathcal{M})$                                   /* re-analysis */
end while
```

Figure 5.25: Algorithm for the demand-driven analysis

will be able to eliminate all dynamic type tests. We will see that it indeed does so.

The abstract model used to perform the preliminary analysis is based on the following pattern-matchers:

$$\begin{aligned}
M_{\mathcal{V}} &= \left\{\begin{array}{l} \#\mathbf{f}, \\ \lambda_2\ (),\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ \lambda_{15}\ (), \\ (\forall,\ \forall) \end{array}\right\} \\
M_2 &= \{(\forall)\} \\
M_9 &= \{(\forall\ \forall)\} \\
M_{12} &= \{(\forall\ \forall)\} \\
M_{15} &= \{(\forall)\}
\end{aligned}$$

Note how the value pattern-matcher contains one abstract value per type, except for the closure type where there is one abstract closure per $\lambda$-expression. The contour pattern-matchers for the invocation of each kind of closures are the trivial ones.

Here is an excerpt of the results collected by the preliminary analysis:

$$\begin{array}{rllll}
\mathcal{R}: & \alpha_{2,()} & = \{\lambda_2\ ()\}, & \alpha_{4,(\forall)} & = \{\lambda_{15}\ ()\}, \\
& \alpha_{6,(\forall)} & = \{(\forall,\ \forall)\}, & \alpha_{7,(\forall)} & = \{\lambda_{15}\ ()\}, \\
& \alpha_{18,(\forall)} & \multicolumn{3}{l}{= \{\#\mathbf{f},\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ (\forall,\ \forall)\},} \\
& \alpha_{20,(\forall)} & \multicolumn{3}{l}{= \{\#\mathbf{f},\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ (\forall,\ \forall)\}}
\end{array}$$

Only the results that are pertinent for the example are presented.

Now that the preliminary analysis has been performed, the demand-driven cycle can start. We show a trace of the first model-update phase. We add comments throughout the presentation of the different model-update phases. Comments are indicated similarly to footnotes. A sign like $\textcircled{99}$ is put on top of the arrows separating the numerous steps of the model-update phases. The corresponding comment is given in the text. Here is the trace of the first model-update phase:

$$\left\{\begin{array}{l} \textbf{show}\ \alpha_{18,(\forall)} \subseteq \mathcal{V}al\mathcal{P} \\ \textbf{show}\ \alpha_{20,(\forall)} \subseteq \mathcal{V}al\mathcal{P} \end{array}\right\} \overset{\textcircled{1}}{\Rightarrow} \left\{\begin{array}{l} \textbf{show}\ \alpha_{20,(\forall)} \subseteq \mathcal{V}al\mathcal{P} \\ \textbf{split}\ \alpha_{18,(\forall)}\ \star \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \textbf{split}\ \alpha_{18,(\forall)}\ \star \\ \textbf{split}\ \alpha_{20,(\forall)}\ \star \end{array}\right\}$$

$$\overset{\textcircled{2}}{\Rightarrow} \left\{\begin{array}{l} \textbf{split}\ \alpha_{20,(\forall)}\ \star \\ [\textbf{split}\ \beta_{\mathrm{p},(\forall),18}\ \star] \end{array}\right\} \overset{\textcircled{3}}{\Rightarrow} \left\{\begin{array}{l} [\textbf{split}\ \beta_{\mathrm{p},(\forall),18}\ \star] \\ [\textbf{split}\ \beta_{\mathrm{p},(\forall),20}\ \star] \end{array}\right\}$$

① The first set contains the initial demands. A quick examination of the program reveals that there are six expressions that may require a dynamic safety test. However, the results of the preliminary analysis indicate that four of the expressions do not really need a test. So the remaining two tests are taken care of by the emission of these two initial demands. The arrow indicates that an elementary step of the demand-driven algorithm is performed. In this case, there exists a demand to process, so the arrow indicates that the first demand is processed. In all the traces, we take the convention that the first demand to process is taken care of and that the eventual new demands are added at the end of the set. Normally, we will not describe the processing of the demands themselves. The processing rules are quite precise and the information that they need about the analysis results that is needed is presented in the corresponding result excerpt. ② The newly emitted demand is a model-modifying demand. To indicate that it should not be processed, we enclose it into square brackets. ③ The demand-processing ends because there is no more demand to process. Also, there is no call site to monitor.

The demand processing of this first model-update phase has produced two model update suggestions. Next, a selection is made to choose the update to perform on the abstract model. In this case, both demands have exactly the same effect on the model. The update on the model causes pattern-matcher $M_{15}$ to be updated. Here is its new definition:

$$M_{15} \;=\; \{(\#\mathbf{f}),\ (\lambda_\forall),\ ((\forall, \forall))\}$$

Now, when ‘swap’ is invoked, its body is not always evaluated in the same contour. The contour depends on the type of the argument that is passed to ‘swap’. Intuitively, this first update makes sense as it is necessary to know whether ‘p’ is a pair or not before we can do a CAR- or CDR-field extraction on it.

Using this new, updated model, a re-analysis of the program is performed. Here is an excerpt of the new analysis results:

$$
\begin{array}{llll}
\mathcal{R}: & \alpha_{4,(\forall)} = \{\lambda_{15}\ ()\}, & & \\
 & \alpha_{5,(\forall)} = \{\#\mathbf{f},\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ (\forall,\ \forall)\}, & & \\
 & \alpha_{18,(\#\mathbf{f})} = \{\#\mathbf{f}\}, & \alpha_{20,(\#\mathbf{f})} & = \{\#\mathbf{f}\}, \\
 & \alpha_{18,(\lambda_\forall)} = \{\lambda_9\ (\forall),\ \lambda_{12}\ (\forall)\}, & \alpha_{20,(\lambda_\forall)} & = \{\lambda_9\ (\forall),\ \lambda_{12}\ (\forall)\}, \\
 & \alpha_{18,((\forall,\forall))} = \{(\forall,\ \forall)\}, & \alpha_{20,((\forall,\forall))} & = \{(\forall,\ \forall)\}, \\
 & \kappa_{(\#\mathbf{f})} = \{(3,\ (\lambda_{15}\ ()),\ \#\mathbf{f},\ (\forall))\}, & & \\
 & \kappa_{(\lambda_\forall)} = \left\{\begin{array}{l}(3,\ (\lambda_{15}\ ()),\ (\lambda_9\ (\forall)),\ (\forall)), \\ (3,\ (\lambda_{15}\ ()),\ (\lambda_{12}\ (\forall)),\ (\forall))\end{array}\right\} & &
\end{array}
$$

Note that we do not include information on the value of expressions $e_2$, $e_6$, and $e_7$ again since it was already determined at the beginning of the first cycle that they did not need a dynamic safety test. However, that on $e_4$ is needed for the next model-update phase and is mentioned nevertheless. Note that an updated model cannot lead to worse analysis results. This is why we consider the cases of $e_2$, $e_4$, $e_6$, and $e_7$ to be closed.

Based on these new analysis results, a second demand-processing phase can start. Note how the remaining initial demands are expressed in more precise terms because of the updated model. Also, demands are still necessary in only two of the three contours since contour '$((\forall,\ \forall))$' means that 'p' cannot contain anything else than pairs. Which is perfectly satisfactory for the extraction of the field of a pair. Here is the trace of the second demand-processing phase:

$$
\left\{\begin{array}{l}
\textbf{show}\ \alpha_{18,(\#\mathbf{f})} \subseteq \mathcal{V}al\mathcal{P} \\
\textbf{show}\ \alpha_{18,(\lambda_\forall)} \subseteq \mathcal{V}al\mathcal{P} \\
\textbf{show}\ \alpha_{20,(\#\mathbf{f})} \subseteq \mathcal{V}al\mathcal{P} \\
\textbf{show}\ \alpha_{20,(\lambda_\forall)} \subseteq \mathcal{V}al\mathcal{P}
\end{array}\right\}
\Rightarrow \overset{\text{4 steps}}{\cdots} \Rightarrow
\left\{\begin{array}{l}
\textbf{show}\ \delta_{18,(\#\mathbf{f})} = \emptyset \\
\textbf{show}\ \delta_{18,(\lambda_\forall)} = \emptyset \\
\textbf{show}\ \delta_{20,(\#\mathbf{f})} = \emptyset \\
\textbf{show}\ \delta_{20,(\lambda_\forall)} = \emptyset
\end{array}\right\}
\Rightarrow \overset{\text{4 steps}}{\cdots}
$$

$$
\Rightarrow
\left\{\begin{array}{l}
\textbf{show}\ \delta_{17,(\#\mathbf{f})} = \emptyset \\
\textbf{show}\ \delta_{17,(\lambda_\forall)} = \emptyset \\
\textbf{show}\ \delta_{19,(\#\mathbf{f})} = \emptyset \\
\textbf{show}\ \delta_{19,(\lambda_\forall)} = \emptyset
\end{array}\right\}
\Rightarrow \overset{\text{4 steps}}{\cdots} \Rightarrow
\left\{\begin{array}{l}
\textbf{show}\ \delta_{16,(\#\mathbf{f})} = \emptyset \\
\textbf{show}\ \delta_{16,(\lambda_\forall)} = \emptyset
\end{array}\right\}
$$

$$
\overset{\textcircled{4}}{\Rightarrow}
\left\{\begin{array}{l}
\textbf{show}\ \delta_{16,(\lambda_\forall)} = \emptyset \\
\textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ \#\mathbf{f}\ (\forall)
\end{array}\right\}
\overset{\textcircled{5}}{\Rightarrow}
\left\{\begin{array}{l}
\textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ \#\mathbf{f}\ (\forall) \\
\textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ (\lambda_9\ (\forall))\ (\forall) \\
\textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ (\lambda_{12}\ (\forall))\ (\forall)
\end{array}\right\}
$$

$$\overset{\textcircled{6}}{\Rightarrow} \left\{\begin{array}{l} \textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ (\lambda_{9}\ (\forall))\ (\forall) \\ \textbf{bad-call}\ 3\ (\lambda_{15}\ ())\ (\lambda_{12}\ (\forall))\ (\forall) \\ \hline L_{\mathrm{BC}}(\underline{3,(\forall)}) = \{{\scriptstyle (\lambda_{15}\ (), \#\mathbf{f})}\} \end{array}\right\} \Rightarrow \overset{2\ \text{steps}}{\cdots} \Rightarrow \left\{L_{\mathrm{BC}}(\underline{3,(\forall)}) = \left\{\begin{array}{l}{\scriptstyle (\lambda_{15}\ (), \#\mathbf{f}),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{9}\ (\forall)),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{12}\ (\forall))}\end{array}\right\}\right\}$$

$$\overset{\textcircled{7}}{\Rightarrow} \left\{\begin{array}{l} \textbf{monitor-call}\ 3\ (\forall) \\ \hline L_{\mathrm{BC}}(3,(\forall)) = \left\{\begin{array}{l}{\scriptstyle (\lambda_{15}\ (), \#\mathbf{f}),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{9}\ (\forall)),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{12}\ (\forall))}\end{array}\right\} \end{array}\right\} \overset{\textcircled{8}}{\Rightarrow} \left\{\begin{array}{l} \textbf{split}\ \alpha_{5,(\forall)}\ \star \\ \hline L_{\mathrm{BC}}(3,(\forall)) = \left\{\begin{array}{l}{\scriptstyle (\lambda_{15}\ (), \#\mathbf{f}),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{9}\ (\forall)),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{12}\ (\forall))}\end{array}\right\} \end{array}\right\}$$

$$\overset{\textcircled{9}}{\Rightarrow} \left\{\begin{array}{l} [\textbf{split}\ \mathcal{V}al\mathcal{P}\ (\star, \forall)] \\ \hline L_{\mathrm{BC}}(3,(\forall)) = \left\{\begin{array}{l}{\scriptstyle (\lambda_{15}\ (), \#\mathbf{f}),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{9}\ (\forall)),} \\ {\scriptstyle (\lambda_{15}\ (), \lambda_{12}\ (\forall))}\end{array}\right\} \end{array}\right\}$$

④ The processing of this never demand consists in finding all invocation circumstances leading to the selection of contour '(#**f**)' and involving a closure originating from parent $\lambda$-expression $e_{15}$. It appears that the single circumstance logged in $\kappa_{(\#\mathbf{f})}$ involves a closure originating from $e_{15}$, so it becomes a bad call demand. ⑤ Similarly, the two circumstances become bad call demands. ⑥ The bad call demand is not trivially achieved so it must be inserted into the bad-call log. We denote this insertion by indicating the state of the log at the bottom of the set. Also, we flag the concerned call site by underlining its appearance as an index in the log. ⑦ There is no more demand to process. However, there is a flagged call site. A monitor command is emitted and the flag is removed from the call site. ⑧ The call site only implicates function 'swap' and arguments of all types. Only the pair is allowed as an argument. Consequently, a split demand is emitted to request a separation of the good and the bad cases. ⑨ Finally, a model-modifying demand is emitted and there are no more demand to process and no call site to monitor.

The unique model-modifying demand is necessarily selected. It requests an update on the representation of the pairs. Pattern-matcher $M_{\mathcal{V}}$ is updated and becomes:

$$M_{\mathcal{V}} = \left\{\begin{array}{l} \#\mathbf{f}, \\ \lambda_2\ (),\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ \lambda_{15}\ (), \\ (\#\mathbf{f}, \forall),\ (\lambda_{\forall}, \forall),\ ((\forall, \forall), \forall) \end{array}\right\}$$

With the new model, a re-analysis is performed and we can observe these new analysis results:

$$\begin{array}{rllll}
\mathcal{R}: & \alpha_{4,(\forall)} & = \{\lambda_{15}\ ()\}, & \alpha_{5,(\forall)} & = \{\#\mathbf{f},\ (\lambda_{\forall},\forall)\}, \\
& \alpha_{6,(\forall)} & = \{(\#\mathbf{f},\forall),\ ((\forall,\forall),\forall)\}, & & \\
& \alpha_{7,(\forall)} & = \{\lambda_{15}\ ()\}, & \alpha_{8,(\forall)} & = \{(\lambda_{\forall},\forall)\}, \\
& \alpha_{16,((\forall,\forall))} & = \{(\#\mathbf{f},\forall),\ ((\forall,\forall),\forall)\}, & & \\
& \alpha_{17,((\forall,\forall))} & = \{\#\mathbf{f},\ (\lambda_{\forall},\forall)\}, & \alpha_{18,(\#\mathbf{f})} & = \{\#\mathbf{f}\}, \\
& \alpha_{18,(\lambda_{\forall})} & = \emptyset, & \alpha_{20,(\#\mathbf{f})} & = \{\#\mathbf{f}\}, \\
& \alpha_{20,(\lambda_{\forall})} & = \emptyset, & \gamma_{\lambda_{15}\ (),((\forall,\forall))} & = \{(\#\mathbf{f},\forall),\ ((\forall,\forall),\forall)\}, \\
& \kappa_{(\#\mathbf{f})} & = \{(3,\ (\lambda_{15}\ ()),\ \#\mathbf{f},\ (\forall))\} & &
\end{array}$$

The results reveal that, in fact, 'swap' is not called on any closure. However, there is no evidence that it is not called on #f and the two remaining initial demands try to remedy to the situation in the next demand-processing phase:

$$\left\{\begin{array}{l}\mathbf{show}\ \alpha_{18,(\#\mathbf{f})}\subseteq\mathcal{V}al\mathcal{P}\\ \mathbf{show}\ \alpha_{20,(\#\mathbf{f})}\subseteq\mathcal{V}al\mathcal{P}\end{array}\right\}\Rightarrow\overset{\text{2 steps}}{\cdots}\Rightarrow\left\{\begin{array}{l}\mathbf{show}\ \delta_{18,(\#\mathbf{f})}=\emptyset\\ \mathbf{show}\ \delta_{20,(\#\mathbf{f})}=\emptyset\end{array}\right\}\Rightarrow\overset{\text{2 steps}}{\cdots}$$

$$\Rightarrow\left\{\begin{array}{l}\mathbf{show}\ \delta_{17,(\#\mathbf{f})}=\emptyset\\ \mathbf{show}\ \delta_{19,(\#\mathbf{f})}=\emptyset\end{array}\right\}\Rightarrow\overset{\text{2 steps}}{\cdots}\Rightarrow\left\{\mathbf{show}\ \delta_{16,(\#\mathbf{f})}=\emptyset\right\}\Rightarrow\left\{\mathbf{bad\text{-}call}\ 3\ (\lambda_{15}\ ())\ \#\mathbf{f}\ (\forall)\right\}$$

$$\Rightarrow\left\{L_{\mathrm{BC}}(\underline{3,(\forall)})=\{(\lambda_{15}\ (),\#\mathbf{f})\}\right\}\Rightarrow\left\{\begin{array}{l}\mathbf{monitor\text{-}call}\ 3\ (\forall)\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}$$

$$\Rightarrow\left\{\begin{array}{l}\mathbf{split}\ \alpha_{5,(\forall)}\ \star\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}\Rightarrow\left\{\begin{array}{l}\mathbf{split}\ \alpha_{6,(\forall)}\ (\star,\ \forall)\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}$$

$$\Rightarrow\left\{\begin{array}{l}\mathbf{split}\ \gamma_{\lambda_{15}\ (),((\forall,\forall))}\ (\star,\ \forall)\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}\Rightarrow\left\{\begin{array}{l}\mathbf{split}\ \alpha_{16,((\forall,\forall))}\ (\star,\ \forall)\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}$$

$$\Rightarrow\left\{\begin{array}{l}\mathbf{split}\ \alpha_{17,((\forall,\forall))}\ \star\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}\Rightarrow\left\{\begin{array}{l}[\mathbf{split}\ \mathcal{V}al\mathcal{P}\ (\forall,\ \star)]\\ \hline L_{\mathrm{BC}}(3,(\forall))=\{(\lambda_{15}\ (),\#\mathbf{f})\}\end{array}\right\}$$

Only one model-modifying demand is generated and it is automatically selected. It asks for another improvement in the representation of the pairs. Once again, pattern-matcher $M_{\mathcal{V}}$ is updated and it now includes abstract pairs that are uniformly specified one level deep:

$$M_{\mathcal{V}} \;=\; \left\{\begin{array}{lll} \#\mathbf{f}, & & \\ \lambda_2\ (),\ \lambda_9\ (\forall),\ \lambda_{12}\ (\forall),\ \lambda_{15}\ (), & & \\ (\#\mathbf{f},\,\#\mathbf{f}), & (\lambda_\forall,\,\#\mathbf{f}), & ((\forall,\,\forall),\,\#\mathbf{f}), \\ (\#\mathbf{f},\,\lambda_\forall), & (\lambda_\forall,\,\lambda_\forall), & ((\forall,\,\forall),\,\lambda_\forall), \\ (\#\mathbf{f},\,(\forall,\,\forall)), & (\lambda_\forall,\,(\forall,\,\forall)), & ((\forall,\,\forall),\,(\forall,\,\forall)) \end{array}\right\}$$

A re-analysis with the new model leads to the following results:

$$\begin{array}{llll} \mathcal{R}: & \alpha_{4,(\forall)} = \{\lambda_{15}\ ()\}, & \alpha_{5,(\forall)} = \{\#\mathbf{f},\ (\lambda_\forall,\,\#\mathbf{f})\}, \\ & \alpha_{6,(\forall)} = \{(\#\mathbf{f},\,\lambda_\forall),\ ((\forall,\,\forall),\,\lambda_\forall)\}, & \\ & \alpha_{7,(\forall)} = \{\lambda_{15}\ ()\}, & \alpha_{8,(\forall)} = \{(\lambda_\forall,\,(\forall,\,\forall))\}, \\ & \alpha_{16,((\forall,\forall))} = \{(\#\mathbf{f},\,\lambda_\forall),\ ((\forall,\,\forall),\,\lambda_\forall)\}, & \\ & \alpha_{17,((\forall,\forall))} = \{\#\mathbf{f},\ (\lambda_\forall,\,\#\mathbf{f})\}, & \alpha_{18,(\#\mathbf{f})} = \{\#\mathbf{f}\}, \\ & \alpha_{18,((\forall,\forall))} = \{(\lambda_\forall,\,\#\mathbf{f}),\ (\lambda_\forall,\,(\forall,\,\forall))\}, & \alpha_{20,(\#\mathbf{f})} = \{\#\mathbf{f}\}, \\ & \gamma_{\lambda_{15}\ (),((\forall,\forall))} = \{(\#\mathbf{f},\,\lambda_\forall),\ ((\forall,\,\forall),\,\lambda_\forall)\}, & \kappa_{(\#\mathbf{f})} = \{(3,\,(\lambda_{15}\ ()),\,\#\mathbf{f},\,(\forall))\} \end{array}$$

Unfortunately, they do not allow the removal of the last two safety tests, yet. The same two initial demands are emitted for the next demand-processing phase:

$$\left\{\begin{array}{l} \textbf{show}\ \alpha_{18,(\#\mathbf{f})} \subseteq \mathcal{V}al\mathcal{P} \\ \textbf{show}\ \alpha_{20,(\#\mathbf{f})} \subseteq \mathcal{V}al\mathcal{P} \end{array}\right\} \Rightarrow \overset{\text{14 steps}}{\cdots} \Rightarrow \left\{\frac{\textbf{split}\ \alpha_{17,((\forall,\forall))}\ \star}{L_{\mathrm{BC}}(3,(\forall)) = \{(\lambda_{15}\ (),\#\mathbf{f})\}}\right\}$$

$$\Rightarrow \left\{\frac{\textbf{split}\ \alpha_{18,((\forall,\forall))}\ (\forall,\,\star)}{L_{\mathrm{BC}}(3,(\forall)) = \{(\lambda_{15}\ (),\#\mathbf{f})\}}\right\} \Rightarrow \left\{\frac{[\textbf{split}\ \beta_{\mathrm{p},((\forall,\forall)),18}\ (\forall,\,\star)]}{L_{\mathrm{BC}}(3,(\forall)) = \{(\lambda_{15}\ (),\#\mathbf{f})\}}\right\}$$

Only one model-modifying demand is generated by the demand-processing phase. Its application to the model causes the update of pattern-matcher $M_{15}$:

$$M_{15} \;=\; \{(\#\mathbf{f}),\ (\lambda_\forall),\ ((\forall,\,\#\mathbf{f})),\ ((\forall,\,\lambda_\forall)),\ ((\forall,\,(\forall,\,\forall)))\}$$

Before this modification, the analysis of the behaviour of 'swap' was confusing both invocations of 'swap'. Remember that the first invocation involves the whole list and the second, the shortened list. Each time, the abstract invocation of 'swap' sees a pair coming as an argument. So the values for both invocations were blended together. With this last update, the analysis no longer confuses both invocations and now each invocation has its own return value. The second invocation involves only the shortened list originating from the return value of the first invocation.

A re-analysis using this last model provides the desired results. Namely, these contain:

$$\mathcal{R}: \quad \alpha_{18,(\#\mathbf{f})} = \emptyset, \qquad \alpha_{20,(\#\mathbf{f})} = \emptyset$$

which completes the demonstration that no safety test is required for the whole program.

The presented example illustrates the execution of the demand-driven analysis on a simple program. But also, it is remarkable to see how the algorithm did it all without coming close to "understand" the program or being intelligent in a human sense. The only expertise in type analysis is present in the design of the global approach and mainly in the design of the processing rules. But even in the processing rules, there is no long body of domain-specific knowledge; only relatively short, sensible tests and transformations. Nevertheless, the whole approach is remarkably intelligent. Empirical evaluation of its performances are presented in Chapter 6.

## 5.5 Development of the Prototype

The presented prototype is not a first attempt that has happened to immediately work well. Many previous prototypes have been built and tried. The attentive reader may have noticed some details that suggest that previous approaches were used: the (SUCCESS) and (FAILURE) comments that are ignored; many kinds of demands that can never happen to be trivially achieved, namely bad call demands; the split demands with $\mathcal{V}al\mathcal{C}$ as a splittee that can never be emitted.

### 5.5.1 Resolution-Like Processing of Demands

The first prototypes did not proceed with a model-update re-analysis cycle but were doing a kind of request resolution *à la* Prolog. That explains the presence of the (SUCCESS) and (FAILURE) comments. Reaching (SUCCESS) meant that the current demand was trivially achieved and reaching (FAILURE) meant that the current demand could not be achieved. When many sub-demands were emitted by the processing of the current demand, they were considered to be linked by a logical-and operator, i.e. the current demand was achieved if all its sub-demands were achieved.

This resolution-like approach had many problems. For example, the natural processing for a bound demand is first to separate the good cases from the bad cases and then to show the impossibility of the bad cases. This processing requires an ordering in time that cannot be expressed using simple Boolean operators. So sequencing operators were introduced. Their task consisted in triggering the processing of a certain demand, waiting for its achievement, and then emitting another demand. In fact, a complete system of package of things to do, called *wills*, were implemented to take care of the prescriptions issued by the processing of demands. Wills could include the emission of groups of demands, sequences of other wills, and other commands that we mention below. Wills were intended to implement all the mechanisms needed for performing the resolution of the demands in a resolution-like fashion. They were pretty complex.

Another problem with the resolution-like approach was that of the model updates being performed *during* the resolution process. The execution of a will doing a sequencing operation typically consists in waiting for a model update to cause the first sub-demand to succeed in order to trigger the processing of the next one. This particular will is specifically designed to deal with such updates. However, the processing of other demands might be affected by the model update. For example, calls are expressions with a very complex interpretation and they can be affected by almost any model update. So, a demand concerning a call that is processed at the beginning of the demand-driven analysis usually does not lead to the same set of sub-demands as if it were processed later. It typically becomes easier to process as the model evolves. So the prototypes had two mechanisms to deal with the processing of difficult demands. The first was that the demand could be re-emitted by its will. For example, when a demand cannot be completely processed (typically because a separation of good and bad cases has to be performed first), its will consists in a sequencing operator that first emitted split demands for the separation and then emitted the original demand again. If the splits are eventually achieved, then the new processing of the demand can happen within new, separated results.

The problem with this re-emission after the separation is completed is that, often, the requested separation is too complex. Indeed, at the beginning of the demand-driven analysis, the analysis results may be too inaccurate and the processing of a demand concerning a call may produce a will that asks for a separation that is excessively ambitious. So the split demands involved in the separation may never be achieved. However, the analysis results typically become more accurate as the demand-driven analysis progresses. The processing of the same demand, if done later, would lead to the request of a much more sober separation step, increasing the chances of its realisability. To take advantage of the progressive improvement of the analysis results, we added another mechanism: a *wake-up call* for the complex demands. If there is a wake-up call that is set for a certain demand and that the demand is not achieved within a certain amount of time, then it is automatically re-emitted. In order to determine which demands have been processed, since when, whether they are achieved or not, etc., we added a demand log. It was a complex data structure with fast access and in which all existing demands were noted along with their related informations. Another problem that the demand log helped to deal with was that of the *cyclic demands.* Cyclic demands often appear when, for example, two functions are mutually recursive and the result of each one depends on that of the other. A split demand on the result of the first leads to the emission of a split demand (among other) on the result of the other, which in turn leads to the same first demand. The demand log allowed to verify if a demand was already in the waiting queue to be processed or has already been processed and possesses a will.

Periodically reprocessing a demand could be expensive as a new will could possibly be created, which lead to the possible emission of similar demands as before. So, for certain kinds of demands, we instead performed periodic *checks* to see if they now happened to be achieved, due to some update of the model. When a demand was discovered to be achieved during a check, we would delete the whole "search" tree that represented its resolution process and send a success signal to its parent demands. One can imagine the complexity of such an operation because of the wills, wake-up calls, demand log that are involved in the resolution process.

All these mechanisms were introduced in the successive prototypes in order to try to make the demand-driven analysis work. All of this was terribly complex and, on top of that, it did not work satisfactorily. The main problems that we finally identified through extensive experiments were: the processing of a demand rarely benefits from the most up to date analysis results; many demands continued to be "resolved" while it could be established from the current analysis results that they were now useless (not to confuse with "achieved").

### 5.5.2 Model-Update Selection and Re-Analysis Cycle

From these observations, we decided to make a major change in the demand-driven analysis procedure and decided that, each time the model was changed, demand processing had to be restarted from scratch. At first glance, it seems like a terrible waste of resources. Indeed, the initial demands have to be generated each time, many similar demands have to be processed each time. The reader could witness that redundancy in the example of demand-driven analysis in the previous section. But the benefits clearly outweigh the inconveniences: only the demands that are needed according to the current analysis resources are generated and processed. As soon as new results indicate that such or such property no longer needs to be verified, the corresponding demand does not get emitted. The model-modifying demands that are proposed by the demand processing now have a very high degree of pertinence.

The new problem is that the demand-processing phase of the cycle usually proposes more than one model-modifying demands. Our first strategy consisted in selecting all of them. We immediately saw an improvement in the intelligence of the prototype. It could discover facts that stayed completely unsuspected by the previous prototypes. However, it caused a massive expansion of the abstract model. During the demand-driven analysis, the analysis results were rapidly improving in quality but they were expressed in so many precise values that they were expanding very quickly, too. After only a few minutes of execution, the prototype needed more than a gigabyte of memory space.

So we decided to use a selection criterion. The first one simply consisted in measuring the increase in size of the abstract model and selecting the model-modifying demand that caused the smallest increase. It succeeded in keeping the model to a reasonable size but it had the tendency to choose demands that do not really help in making the results more informative. Consequently, the results quickly expanded as they were always denoting the same information but in ever finer terms. Nevertheless, for some benchmarks, this control on the size of the model, plus the high pertinence of the proposed model-modifying demands resulted in successful analyses, where previous prototypes stagnated or exploded.

We changed the criterion for a slightly more clever one: its measures the increase in the size of both the model and the results and selects the least increase-causing demand. Despite the fact that this criterion is not much more clever than the previous one, it happens to be really useful. It is the one that is used in the current prototype and allows the latter to analyse perfectly well many of the benchmarks that we submit to it.

In conclusion, we say that it is practically the most naïve design of demand-driven analysis that allowed us to obtain a working prototype. The "more clever" approach of having a resolution of demands *à la* Prolog did not work properly. Although the cyclic approach of performing demand processing from scratch and re-analysing after each model update seems to imply some resource waste, it turns out to be economical in the demands that it insists on processing.

## 5.6 Discussion

To conclude with the pattern-based demand-driven analysis, we make a few comments. We believe that the pattern-based analysis is the simplest instance of demand-driven analysis that has a reasonably high power. The meaning of the patterns as abstract values is straightforward. There are only a few kinds of demands that need to be manipulated and the rules to process them are relatively intuitive. Moreover, the pattern-based instance respects the intent expressed in the presentation of demand-driven analyses in general that we should avoid creating an expert system with an extensive knowledge base to obtain a good type analysis.

The pattern-based approach has somehow a reduced power compared to the concept of demand-driven analysis in general. Not necessarily theoretically, but in practice. Theoretically, the modelling of the concrete values and evaluation contexts using patterns is not less powerful than the generic modelling allowed by the analysis framework: a correctly terminating program still can be analysed perfectly well using a model based on patterns. Indeed, the correctly terminating program runs only for a certain time; so it creates values and manipulates environments that have only a certain depth (if written as syntax trees); so choosing pattern-matchers that only project details that are beyond this depth would allow to simulate with perfect accuracy the concrete computations.

In practice, however, the program runs for an unknown time and *a priori* manipulates arbitrarily big and deep values. The pattern-based modelling is intrinsically *myopic* and fails to capture many kinds of properties applying to the values. For example, for long enough lists, the difference between lists having an even length and those having an odd length cannot be made. By inspection of the first few levels of a list, it is clear that it is impossible to determine how many other pairs are linked in the list. On the other hand, with the general modelling provided by the analysis framework, it is easy to choose $\mathcal{V}al\mathcal{P}$ and pc such that pairs that start lists of even and odd length are distinguished.

Despite its myopia, the pattern-based modelling, in combination with the analysis results, may sometimes discover non-superficial properties of the values manipulated by a program. Consider a simple program that manipulates two kinds of lists: lists of closures and lists of pairs. Suppose that both kinds of lists are terminated by #f. Let us pretend that we are the analyser ourselves and that an oracle told us that the program only manipulates those two kinds of lists. Then we could identify which of the two kinds of lists we are manipulating in a myopic fashion: if the first pair contains a closure, then it is the head of a closure list; otherwise, it is the head of a pair list. The real analyser can discover the same invariant (without the help of an oracle, of course) by exploiting the contents of the log of pair creation circumstances, i.e. the $\pi$ matrix. First, let us observe what the log contains when there is only one abstract pair, i.e. when $\mathcal{V}al\mathcal{P} = \{(\forall, \forall)\}$:

$$\pi_{(\forall,\forall)} = \left\{ \begin{array}{llll} (\_, & \lambda_{\_}\,(\ldots), & \#\mathbf{f}, & \_), \\ (\_, & \lambda_{\_}\,(\ldots), & (\forall, \forall), & \_), \\ (\_, & (\forall, \forall), & \#\mathbf{f}, & \_), \\ (\_, & (\forall, \forall), & (\forall, \forall), & \_), \\ \ldots & & & \end{array} \right\}$$

These circumstances illustrate the best possible case for the analysis results. Note that we intentionally omitted to give the labels and contours where the pairs are created and the details from the closures stored into the pairs. The omitted details are not useful to the example. The information in the log only indicates that lists are #f-terminated and that they contain closures and pairs. But if we now observe the contents of the log if the model were updated to have pairs that are distinguished by the type of the value in their CAR-field, i.e. $\mathcal{V}al\mathcal{P} = \{(\#\mathbf{f}, \forall), (\lambda_\forall, \forall), ((\forall, \forall), \forall)\}$:

$$\begin{array}{ll} \pi_{(\#\mathbf{f},\forall)} & = \emptyset \\ \pi_{(\lambda_\forall,\forall)} & = \{(\_, \lambda_{\_}\,(\ldots), \#\mathbf{f}, \_), (\_, \lambda_{\_}\,(\ldots), (\lambda_\forall, \forall)\_), \ldots\} \\ \pi_{((\forall,\forall),\forall)} & = \{(\_, (\_, \forall), \#\mathbf{f}, \_), (\_, (\_, \forall), ((\forall, \forall), \forall), \_), \ldots\} \end{array}$$

Again, these results also illustrate the best possible case for the analysis results. Observe that pairs having a closure in the CAR-field are either terminated by #f or have a closure list in the CDR-field; pairs having a pair in the CAR-field are either terminated by #f or have a pair list in the CDR-field. This property of the lists can be discovered by the analyser because the different kinds of lists have a superficial difference that is sufficient to distinguish them.

On the contrary, if there are two kinds of lists and that these two kinds are only differentiated deeply, the analyser cannot find the distinction. For example, suppose that the program manipulates these two kinds of lists: both are lists of Booleans, but one kind is #f-terminated and the other is terminated by a closure. Then, if we consider two sufficiently long lists, one that is #f-terminated and the other not, then there is no superficial difference between them. So they have to be represented by the same abstract value (by the projection through the value pattern-matcher). The best the pattern-based analyser could do is to determine that the lists contain Booleans and are terminated by a Boolean or a closure. Only short lists could be classified correctly. However, by directly using generic models accepted by the analysis framework, a model can be chosen such that pairs are different depending on the type of the value that terminates the list they are the head of.

We finish by asking, and answering, the following question: Since any correctly terminating program can be analysed perfectly well using an appropriate pattern-based model, is the pattern-based demand-driven *analysis* always able to eventually analyse the program perfectly well? Unfortunately, the answer is: No. Intuitively, it is relatively easy to accept this answer. It is because the analyser starts with a coarse model, may only obtain obscure analysis results, and may not be able to discover the appropriate model updates before there is no more useful information it can extract from the results. However, strictly speaking, this explanation is not sufficient. But in Chapter 6, experiments show that our prototype is not able to analyse perfectly well some of the benchmarks.

# Chapter 6

# Experimental Results

We have run some experiments on a prototype implementation of the pattern-based demand-driven analysis. But before we present the results of these experiments, we first give some details on the implementation of the prototype. And then we describe the method used to measure the effectiveness of the demand-driven analysis. Finally, we present the results and make comments.

## 6.1 Current Implementation

The prototype is implemented in a rather naïve way. No special effort has been made to make it particularly efficient, in time and in space. The abstract values and abstract contours are implemented almost as we have presented them in the previous chapter. They are represented using simple, easy to read Scheme data made of lists, symbols and numbers. For example, here is the representation of two abstract values:

$$\begin{array}{lcl} (\#\mathbf{f},\ \lambda_{\forall}) & \mapsto & \texttt{(pair (bool) (clos any))} \\ \lambda_{12}\ (\forall\ \#\mathbf{f}\ \forall) & \mapsto & \texttt{(clos 12 (vals (bool) vals))} \end{array}$$

This representation is quite space consuming and could certainly be reduced to a more compact form.

During the analyses, values and contours of this kind are created and projected using the pattern-matchers. Their projection involves their deconstruction using a queue. Consequently, this process is also time consuming. However, a feature of the implementation of the pattern-matchers reduces the space consumption. During a single analysis, the abstractions that result from the projections using the pattern-matchers are not created from scratch. Instead, they are taken from the leaves of the pattern-matchers. So the values stored in the $\alpha$, $\beta$, $\gamma$, and $\delta$ matrices are the same objects (in the sense of `eq?`) as those already present in the pattern-matchers. However, the tuples found in the $\chi$, $\pi$, and $\kappa$ matrices are built during the analysis even if their contents are already existing objects. Naturally, the data structures implementing the matrices themselves must also be created.

The considerable amount of data structures that are needed in order to perform the analyses causes a loss of time efficiency due to the stress on memory management. Also, the repeated projections of abstractions add to the inefficiency.

The implementation of the sets that hold the values of each matrix entry is efficient. However, operations on the sets rely on an ordering relation between abstractions that is quite heavy. To determine the relative order of two abstractions, the relation traverses the lists and atoms until a difference is found. These comparisons cause a large consumption of time.

In fact, the major source of time consumption in the prototype is the need to re-analyse the program from scratch each time a model-modifying demand is evaluated by the selection criterion. With models that are increasingly complex during the whole demand-driven analysis, the repeated analyses incur a tremendous cost. When one watches the trace that is produced by the prototype, it is perfectly obvious that almost all the time is spent in the demand selection step. Even in the prototype that used only the size of the model as a criterion, almost all of the time was spent in the analyses. The demand generation and processing steps are faster by orders of magnitude.

Because of that, our current measure of the amount of resources to invest in the demand-driven analysis is not reliable. The amount of resources is measured in the number of processed demands. Since the processing of demands is far from being the major cost, the measure does not represent very accurately the amount of resources that are available. Using a measure like the CPU time would be preferable. At least, it is so from the point of view of the user of the system. From our current point of view, the advantage of the current measure is that it measures the amount of *reasoning* the demand-driven analysis can do. Indeed, the cleverness of the approach comes mostly from the processing of demands.

## 6.2 Test Methodology

### 6.2.1 What is Measured?

For each benchmark, we count the number of safety tests that can be removed from the program. It is important to note the distinction between the fact that these tests are *dynamic* and the fact that we remove their static occurrences in the *program text*. We count the number of static occurrences of the tests, not the number of dynamic uses of the tests.

One might object that "counting the static occurrences of the tests is farther from measuring the concrete improvement in the execution time of the program than counting their dynamic uses". We agree, but we answer that "it is not farther by much". Let us give our reasons.

The number of dynamic uses does not have a relation to the execution time of the program that is as tight as we may expect at first. Many other factors impact on the execution time: the "useful" computations made by the program, the hidden run-time tasks such as memory management, the interaction with the operating system, the particular machine on which the program runs, etc. In general, it is hazardous to predict what is the impact on the execution time of the program when it has been determined that only 50% of the uses of dynamic tests were required. In some situations, the savings on the safety tests are overshadowed by the remainder of the program tasks and little improvement of the execution time is observed. On the other hand, the frequency of the dynamic tests during the computations might be so high that the reduction in the execution time could be close to that of the number of uses of safety tests. In exceptional cases, the improvement could even be over 50% if the optimizations help the code to be smaller and to behave more favourably in relation to the cache memory and if they improve the branch prediction success rate in the processor.

For the exact same reasons, directly taking the improvement of the execution time of the programs as a measure of the effectiveness of the analysis is not representative.

Using the number of static occurrences of safety tests in the program text has many advantages. It exclusively depends on the analysis and the program. No external factors can influence the measure. The success of our analysis in the removal of the different safety tests depends more on the intrinsic difficulty of the program. Consequently, we believe it gives a more stable measure of the effectiveness of the analysis. For instance, it cannot happen to obtain very good results on a particular benchmark because it eliminated a few very frequently used tests and, the next time, to obtain poor results because it eliminated many rarely used tests. A measure of the dynamic uses of the safety tests is more sensitive to "luck". Moreover, there is no reason to believe that frequently used tests are harder or easier to eliminate than rarely used ones. Our measure is insensitive to the inputs of the program while it is executed. Or course, our mini-language does not include input/output operations, but a concrete language for which the analysis could eventually be implemented should include input/output.

Finally, counting the number of static occurrences of safety tests is common in the field of static analyses. Also, it is compatible with the goal we gave ourselves at the beginning of the document: to try to remove as many safety tests as possible.

### 6.2.2 Benchmarks

The effectiveness of the analysis is evaluated using a variety of benchmarks. They vary from small to medium size. There are a few toy programs, adaptations of some of the Gabriel benchmarks, and other programs. Many benchmarks involve numerical computations. Some have a more symbolic nature. Most are written or translated, completely or partly, by hand from Scheme. Some are automatically compiled from a subset of Scheme into the syntax of the mini-language.

Before we present each benchmark, we need to discuss a few issues regarding their translation. The most important issue concerns the use of `letrec`-expressions. As we know, the mini-language does not include `letrec`-expressions (it does not even include `let`-expressions). In order to obtain benchmarks written in the mini-language, `letrec`s are reduced into `let`s plus uses of the well known "Y" combinator. For each benchmark, we used two different translations. One in which variable Y is first bound to an appropriate function and in which each recursive function gets created by calling Y on a partially recursive function. The other in which each recursive function is created using a *private* Y combinator. Clearly, having one global Y combinator makes the program harder to analyse because every recursive function is created using the same $\lambda$-expression coming from Y. Naturally, the returned closure remembers its associated partially recursive function but

```
(letrec_1 foo = (λ_2 x. (_3 foo_4 x_5))
   (_6 foo_7 #f_8))
```

(a) Original program

```
(let_1 Y = (λ_2 f. (let_3 g = (λ_4 h. (λ_5 z. (_6 (_7 f_8 (_9 h_10 h_11)) z_12)))
                      (_13 g_14 g_15)))
   (let_16 foop = (λ_17 foof. (λ_18 x. (_19 foof_20 x_21)))
      (let_22 foo = (_23 Y_24 foop_25)
         (_26 foo_27 #f_28))))
```

(b) Translation with global Y

```
(let_1 foo = (let_2 f = (λ_3 foo2. (λ_4 x. (_5 foo2_6 x_7)))
                (let_8 g = (λ_9 h. (λ_10 z. (_11 (_12 f_13 (_14 h_15 h_16)) z_17)))
                   (_18 g_19 g_20)))
   (_21 foo_22 #f_23))
```

(c) Translation with private Y

Figure 6.1: Translation of `letrec`-expressions

the analyser has to discover that by itself. On the other hand, private Y combinators allow recursive functions from distinct `letrec`s to be created from distinct $\lambda$-expressions. Certainly, this does not make the task as easy as if the analyser knew how to handle `letrec`-expressions directly but nevertheless it helps a lot. Figure 6.1 shows both kinds of translation for the little benchmark `loop`.

Many benchmarks involve numerical computations. But we know that the mini-language does not include numbers. Consequently, a reduction step used in the elaboration of the benchmarks consists in getting rid of the numbers by replacing them by lists of Booleans. Only the naturals and a few arithmetic operations are supported. The numbers are encoded in unary. Thus, the constant '3' appearing in the program is translated into:

```
(cons #f (cons #f (cons #f #f)))
```

The "numerical lists" do not have any special status and are manipulated as ordinary values by the mini-language.

The benchmarks written in the Scheme subset may include empty lists, both Booleans, numbers (naturals), pairs, vectors, closures of any (non-variable) arity, and symbols. A subset of the standard library is provided. The extra special forms are `letrec` and `let`, provided that they include only one binding. Also, the expression to which the variable is bound in `letrec`-expressions must be a $\lambda$-expression. Programs written in this subset of Scheme are translated into the mini-language plus `let`, `letrec`, and unary numbers. Each Scheme object is represented by a pair of the mini-language. The pair contains a type tag (a small number) and the value encoded in a type-dependent way. The necessary library functions are included. The Scheme type discipline is enforced and a Scheme type error leads to the evaluation of the mini-language expression `(car #f)`. Wrapping and unwrapping instrumentation is added throughout the translated program. Programs thus translated tend to expand considerably. Figure 6.2 shows the translation of a very small expression. The code expansion is evident.

We now describe each benchmark:

**cdr-safe** Definition and use of a secure version of the `cdr` function. Written in the extended mini-language.

**loop** An infinite loop. Written in the extended mini-language.

**2-1** Computes the indicated subtraction. Written in the extended mini-language.

**map-easy** Two uses of `map` on the same list using two different functions. Written in the extended mini-language.

**map-hard** A use of `map` on two different lists using two different functions. Each function can only be applied on the elements of its corresponding list. Otherwise, an error would occur. This simple benchmark is reported in [37] as being impossible to analyse perfectly well by the $k$-cfa analysis, no matter how big $k$ is. Written in the extended mini-language.

**fib** Computes the 7th Fibonacci number. Adapted from a Gabriel benchmark. Written in the extended mini-language.

**gcd** Computes the greatest common divisor of 3 and 5. Written in the extended mini-language.

```
(if (= 2 3) #f '(32 a))
```

$$\mapsto$$

```
(let_{1} true = (cons_{2} 4_{3} (cons_{4} #f_{5} #f_{6}))
 (let_{7} false = (cons_{8} 4_{9} #f_{10})
  (let_{11} wrap-num = (λ_{12}n. (cons_{13} 2_{14} n_{15}))
   (let_{16} null = (cons_{17} 0_{18} #f_{19})
    (let_{20} wrap-clos =
      (λ_{21}n. (λ_{22}c. (cons_{23} 1_{24} (λ_{25}m. (if_{26} (=_{27} n_{28} m_{29}) c_{30} (car_{31} #f_{32}))))))
     (let_{33} test = (λ_{34}x. (if_{35} (=_{36} (car_{37} x_{38}) 4_{39}) (cdr_{40} x_{41}) x_{42}))
      (let_{43} call = (λ_{44}x. (if_{45} (=_{46} (car_{47} x_{48}) 1_{49}) (cdr_{50} x_{51}) (car_{52} #f_{53})))
       (let_{54} wrap-sym = (λ_{55}l. (cons_{56} 6_{57} l_{58}))
        (let_{59} dummy = #f_{60}
         (let_{61} = =
           (_{62}(_{63}wrap-clos_{64} 2_{65})
             (λ_{66}x. (λ_{67}y. (if_{68} (=_{69} (car_{70} x_{71}) 2_{72})
                                   (if_{73} (=_{74} (car_{75} y_{76}) 2_{77})
                                    (if_{78} (=_{79} (cdr_{80} x_{81}) (cdr_{82} y_{83})) true_{84} false_{85})
                                    (car_{86} #f_{87}))
                                   (car_{88} #f_{89})))))
          (let_{90} cons = (_{91}(_{92}wrap-clos_{93} 2_{94})
                             (λ_{95}x. (λ_{96}y. (cons_{97} 3_{98} (cons_{99} x_{100} y_{101})))))
           (if_{102} (_{103}test_{104} (_{105}(_{106}(_{107}(_{108}call_{109} =_{110}) 2_{111})
                                        (_{112}wrap-num_{113} 2_{114}))
                                     (_{115}wrap-num_{116} 3_{117})))
            false_{118}
            (_{119}(_{120}(_{121}(_{122}call_{123} cons_{124}) 2_{125}) (_{126}wrap-num_{127} 32_{128}))
               (_{129}(_{130}(_{131}(_{132}call_{133} cons_{134}) 2_{135})
                    (_{136}wrap-sym_{137} (cons_{138} 97_{139} #f_{140})))
                 null_{141})))))))))))))))
```

Figure 6.2: Translation from the Scheme subset to the extended mini-language

**tak** Computes the Takeuchi function on 18, 12, and 6. Adapted from a Gabriel benchmark. Written in the extended mini-language.

**n-queens** Counts the number of solutions to the problem of the $n$-queens, for $n = 4$. Written in the extended mini-language.

**ack** Computes the Ackermann function on 4 and 0. Adapted from a Gabriel benchmark. Written in the extended mini-language.

**SKI** Interpreter for programs written using the well known S, K, and I combinators. The SKI program is that of an infinite loop. Written in the extended mini-language.

**change** Computes the optimal strategy for returning the change using coins taken from unlimited supplies of coins of 25¢, 17¢, 4¢, 3¢, and 1¢. The optimal change return consists in minimising the number of coins. The result of the computation is a vector of pairs. Each pair contains the optimal strategy for making change for the amount corresponding to its position in the vector. The strategy is expressed by a pair containing the optimal number of coins and the most valuable coin needed by this strategy. For amounts greater than the length of the vector, the most valuable coin must be selected until the remaining amount is handled by the vector. Written in the Scheme subset.

**interp** Interpreter for the Scheme subset. The program it interprets is:

```
(letrec ((foo (lambda () (foo)))) (foo))
```

The interpreter does not check whether the operations performed by the program it interprets are valid. So an illegal operation in the interpreted program causes the interpreter to do an illegal operation itself. Written in the Scheme subset.

**cps-QS-s** Generation and sort of a list of numbers. The list contains the numbers 1 to 28 in “random” order. The numbers are generated by the successive powers of 2 modulo 29. The list is then sorted using the Quicksort algorithm. The program is written in *continuation-passing style* (CPS) except for the initial definition of the CPS versions of the library functions. Written in the Scheme subset.

**cps-QS-m** The same program but translated by hand in the extended mini-language. Indeed, apart from the empty lists terminating the lists of numbers, the other values are directly present in the extended mini-language.

Appendix A presents the listing of each benchmark.

## 6.3 Results

We present the results of the experiments on the benchmarks in Table 6.1. Each benchmark has been translated into the mini-language in two versions: one with a global Y combinator and one with a Y combinator for each `letrec`-expression. A limit of 10000 "work units" has been allowed for the analysis of each benchmark. The machine running the benchmarks is a PC with a 1.5 GHz Athlon CPU, 2 GByte RAM, and running RH Linux kernel 2.4.18-5smp. Gambit-C 4.0 was used to compile the demand-driven analysis.

The meaning of each column is the following. The column labelled **Y** indicates whether the benchmark is the version with one Global Y combinator or with Private Y combinators. The column labelled **size** indicates the size of the benchmark, as measured by the number of expressions. The columns labelled **total**, **pre**, and **post** indicate the number of occurrences of safety tests present in the non-optimised program, in the optimised program based on the preliminary analysis results, and in the optimised program after demand-driven analysis, respectively. The column labelled **during** gives a trace of the evolution of the number of safety tests through the analysis. An item of the form **n**@$t$ indicates that $n$ safety tests are still necessary after $t$ work units have been consumed. The columns labelled **units** and **time** indicate how many work units and how much CPU time, respectively, were consumed by the whole analysis process.

Only partial results could be obtained for the benchmarks written in the Scheme subset and for the global Y version of `cps-QS-m`. The execution of the demand-driven analysis on these consumed too much memory and it had to be stopped. Consequently, they are analysed using the 0-cfa only. No **post** information is available for them. Nevertheless, we insist on mentioning the benchmarks as they could serve as a basis for comparison if future improvements of the implementation of the demand-driven analysis eventually allows these to be analysed. The size of the `interp` benchmark may seem particularly impressive, but it is mainly due to the expressions that create the "Scheme symbols".

Looking at the results of the experiments on the other benchmarks, we easily note

| | **Y** | **size** | **total** | **pre** | **during** | **post** | **units** | **time(s)** |
|---|---|---|---|---|---|---|---|---|
| `cdr-safe` | G | 17 | 4 | 1 | | 0 | 5 | 0.04 |
| | P | 17 | 4 | 1 | | 0 | 5 | 0.03 |
| `loop` | G | 32 | 11 | 0 | | 0 | 1 | 0.04 |
| | P | 26 | 9 | 0 | | 0 | 1 | 0.03 |
| `2-1` | G | 48 | 15 | 2 | **1**@7 | 0 | 47 | 0.51 |
| | P | 42 | 13 | 2 | **1**@7 | 0 | 48 | 0.42 |
| `map-easy` | G | 82 | 26 | 6 | **4**@19 | 0 | 134 | 3.12 |
| | P | 76 | 24 | 6 | **4**@19 | 0 | 134 | 2.83 |
| `map-hard` | G | 96 | 33 | 9 | **6**@38 **5**@254 **3**@305 **1**@520 | 0 | 1399 | 115.54 |
| | P | 101 | 35 | 4 | **2**@118 | 0 | 284 | 8.42 |
| `fib` | G | 141 | 40 | 12 | | 12 | 10000 | 2204.95 |
| | P | 168 | 50 | 5 | **4**@16 **3**@29 **2**@39 **1**@46 | 0 | 358 | 13.87 |
| `gcd` | G | 257 | 77 | 8 | **7**@25 **6**@47 **5**@66 **4**@82 **3**@95 **2**@105 **1**@112 | 1 | 10000 | 11482.90 |
| | P | 328 | 103 | 6 | **5**@19 **4**@35 **3**@48 **2**@58 **1**@65 | 0 | 8509 | 1633.34 |
| `tak` | G | 202 | 46 | 9 | | 9 | 10000 | 2967.36 |
| | P | 218 | 52 | 4 | **3**@13 **2**@23 **1**@30 | 0 | 240 | 18.22 |
| `n-queens` | G | 372 | 121 | 51 | | 51 | 10000 | 23028.97 |
| | P | 454 | 151 | 11 | **10**@34 **9**@65 **8**@93 **7**@118 **6**@140 **5**@1750 | 5 | 10000 | 2667.07 |
| `ack` | G | 162 | 49 | 5 | **4**@16 **3**@29 **2**@39 **1**@46 | 1 | 10000 | 5786.97 |
| | P | 189 | 59 | 3 | **2**@10 **1**@17 | 0 | 200 | 12.51 |
| `SKI` | G | 285 | 46 | 19 | **15**@91 **13**@173 **11**@323 **9**@397 **7**@473 **6**@543 **5**@1474 **4**@3584 | 4 | 10000 | 1238.40 |
| | P | 290 | 48 | 17 | **13**@52 **11**@98 **9**@138 **8**@212 **5**@249 **4**@358 **3**@567 **1**@673 | 0 | 899 | 98.90 |
| `change` | G | 2371 | 717 | 377 | | [377] | [0] | 3227.67 |
| | P | 2519 | 771 | 329 | | [329] | [0] | 1944.26 |
| `interp` | G | 42056 | 1348 | 762 | | [762] | [0] | 17251.09 |
| | P | 42292 | 1434 | 678 | | [678] | [0] | 9597.56 |
| `cps-QS-s` | G | 2042 | 584 | 277 | | [277] | [0] | 11273.67 |
| | P | 2157 | 626 | 242 | | [242] | [0] | 7705.23 |
| `cps-QS-m` | G | 693 | 211 | 58 | | [58] | [0] | 71.47 |
| | P | 808 | 253 | 16 | **14**@49 **13**@92 **12**@132 **11**@169 **10**@203 **9**@234 **8**@262 **7**@287 **6**@309 **5**@328 **4**@344 **3**@357 **2**@444 **1**@1121 | 1 | 10000 | 3356.97 |

Table 6.1: Experimental results

that having private Y combinators make the analysis much simpler. It is the case for the preliminary analysis and for the complete demand-driven analysis.

In fact, the demand-driven analysis is able to remove all safety tests when private Y combinators are used except in the cases of the `n-queens` and of the `cps-QS-m` benchmarks. In these two cases, the demand-driven analysis is nevertheless able to improve on the results obtained by the preliminary analysis. These results are remarkable, given that the Y combinator is quite intricate. Also, in the benchmarks that use the subtraction, a pretty difficult property has to be demonstrated. The property says that, when an expression such as $(-\ x\ y)$ is evaluated, $y$ is never greater than $x$. In fact, subtraction is implemented using a call to a function that is inserted during the reduction that removes the numbers from the extended mini-language. The function *assumes* that its arguments respect the property. If they do not, the function eventually attempts to extract the CDR-field of the Boolean #f that ends the unary representation of $x$. In the `2-1`, `fib`, `gcd`, `tak`, and `ack` benchmarks, the property necessarily had to be demonstrated since these rely on the subtraction.

When a global Y combinator is used, on the other hand, the analysis obtains results of a pretty variable quality. The problem is that all recursive functions are blended together by Y and when there is no easily detectable difference in the behaviour of these functions, then the analysis does not realise that a “good move” consists in creating distinct recursive functions for uses of Y on distinct partially recursive functions.

Note that there is no clear relation between the size of the benchmarks and the success of the demand-driven analysis on them. Certainly, the style of programming has a much bigger impact, as demonstrated by the difference introduced by global and private Y combinators. On top of the difficulty created by their size, we expect the benchmarks written in Scheme to be difficult to analyse because of their style, also. Eventually, their style may even have a bigger impact than their size. The main reason is that all Scheme values are encapsulated in pairs using special purpose functions and that this encapsulation may produce a masking effect similar to that of the global Y combinator. As an instance, two Scheme functions introduced by distinct `letrec`-expressions become very difficult to distinguish: they both are represented as pairs; their CAR-field contains the same “closure” type tag and their CDR-field contains functions created by the wrap-num function, whose task is to check the number of arguments that are to be passed; the check function then contains a reference to the distinct “raw” recursive functions. If a global Y combinator is to be used on top

| **unrolling** | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **units** | 176 | 280 | 532 | 1276 | 3724 |
| **time(s)** | 10.59 | 24.34 | 79.97 | 374.34 | 2325.77 |

Table 6.2: The effect of the size of a program on the analysis

of that, the difference between both raw recursive functions is even more difficult to make. The difference appears only in the references to the partially recursive functions the closure from Y has captured. It is not clear if even a very improved demand-driven analysis could ever be discriminating enough for these benchmarks.

Note that, when the demand-driven analysis has some success in removing safety tests, it usually is able to find the opportunities rather quickly. This suggests that relatively modest investments in analysis time can usually be fruitful. At least, provided that the program is "analysable". When it is not, it seems that considerable time investments in the analysis do not help. This is a happy result since it means that the demand-driven analysis should be used with a rather limited amount of resources, which tends to make it more practical.

We ran another kind of experiment. We wanted to obtain a measure of the time required by the demand-driven analysis on a family of programs that have exactly the same programming style. To do so, we have modified the `ack` benchmark and unrolled the recursive function by various factors. Figure 6.3 shows the aspect of the resulting programs. For each unrolling level $i$, the number of safety tests in the resulting program is $43 + 19i$ if no optimisation is done. There remain 3 after the preliminary analysis. And the demand-driven analysis removes the remaining tests. The times required by the complete analysis for different unrolling levels are presented in Table 6.2. The measures indicate that the total time required by the complete analysis grows between quadratically and cubically with the level of unrolling. This is certainly better than the exponential behaviour expected of a type analysis that uses lexical-environment contours.

We also ran experiments concerning the *inputs* used in some of the benchmarks. Members of the jury of this dissertation have expressed the concern that some benchmarks used very small input values. For example, the `ack` benchmark contains the computation of the Ackermann function on arguments 4 and 0, which produces only 13 as a result. It is obvious that it is cheaper by orders of magnitude to evaluate this benchmark than to analyse it. Analysing such a program does not seem very worthwhile. Consequently, we present a few

```
(letrec_1 ack =
 (λ_2m. (λ_3n. (let_4 ack = (λ_5m. (λ_6n. (if_7 (=_8 m_9 0_10)
                                            (+_11 n_12 1_13)
                                            (if_14 (=_15 n_16 0_17)
                                             (_18(_19ack_20 (-_21 m_22 1_23)) 1_24)
                                             (_25(_26ack_27 (-_28 m_29 1_30))
                                               (_31(_32ack_33 m_34) (-_35 n_36 1_37)))))))
                (let_38 ack = (λ_39m. (λ_40n. (if_41 (=_42 m_43 0_44)
                                                (+_45 n_46 1_47)
                                                (if_48 (=_49 n_50 0_51)
                                                 (_52(_53ack_54 (-_55 m_56 1_57)) 1_58)
                                                 (_59(_60ack_61 (-_62 m_63 1_64))
                                                   (_65(_66ack_67 m_68) (-_69 n_70 1_71)))))))
                 ...
                 (_?(_?ack_? m_?) n_?)···))))
 (_?(_?ack_? 4_?) 0_?))
```

Figure 6.3: Unrolling of the `ack` benchmark

| test | time(s) | test | time(s) | test | time(s) |
|---|---|---|---|---|---|
| `ack` 4 0 | 18.5 | `fib` 7 | 20.5 | `gcd` 3 5 | 2375.8 |
| `ack` 4 4 | 19.0 | `fib` 50 | 25.2 | `gcd` 3 6 | 4167.5 |
| `ack` 10 10 | 19.8 | | | | |

Table 6.3: The effect of the inputs on the analysis times

experiments in Table 6.3 that show the impact of the programs inputs on the analysis times. These experiments were run on a different machine and at a different time. They were run on a PC with a 1.2 GHz Athlon CPU, 1 GByte RAM, and running RH Linux kernel 2.4.9.

Clearly, the time required to run the first benchmarks is longer than the time required to analyse them. The measures show that, roughly, benchmarks `ack` and `fib` remain as difficult to analyse, no matter what the input numbers are. On the other hand, the `gcd` benchmark becomes much harder to analyse when one of its inputs is only increased by one. This may seem surprising at first since the numbers manipulated by the first two benchmarks are gigantic (even delirious in the case of `ack`) while those manipulated by the last one are very small. However, the difference comes from the fact that the demonstration of the safety of the first benchmarks only has to partition the naturals into $\{0\}$, $\{1\}$, $\{2\}$, and the rest, while the demonstration for the last benchmark has to distinguish *each* number involved in the computations. It is easy to realise the difficulty of such demonstrations when one remembers that numbers are encoded as lists.

A last experiment was conducted to prove an affirmation made in the previous chapter: that the pattern-based demand-driven analysis may fail to analyse perfectly well some programs, even if the amount of resources is unlimited. We let the analyser do its work on the `SKI` benchmark (with global Y) until it stopped by itself. After the consumption of 29560 work units, it stopped by lack of proposal of model-modifying demands.

# Chapter 7

# Conclusions

## 7.1 Contributions

Our goal was to obtain a type analysis of very high quality that is not prohibitively expensive. We think that we have reached our goal by proposing the demand-driven analysis: the program is repetitively analysed using abstract models that are increasingly specialised for the task at hand; the updates of the abstract model are directed by the processing of demands, which constitutes the means to translate the needs of the optimiser into proposals of updates to the abstract model.

Static analysis of programs is a classical domain in the field of compilation (see [3]). However, all proposed static analyses share the characteristic that their underlying abstract model is constant. Even if some compilers offer a spectrum of analyses of varying strength, it remains the responsibility of the user to select himself the desired analysis. In any case, the analysis certainly does not adapt to the given program while the compilation occurs.

To improve static analysis: we proposed an analysis where the abstract model is modifiable through the use of an analysis framework; we proposed and realised an implementation of abstract models based on patterns such as those used in many programming languages such as ML, Haskell, or Prolog; we introduced the concept of demands that are requests for the achievement of desirable tasks; the demands are generated according to the needs of the optimiser, are translated following precise rules—the demand processing rules—and result in specific proposals of update of the abstract model so that the analyser becomes better equipped to analyse successfully the program. Most of the theoretical basis behind the approach has been proved in the dissertation. Finally, the approach has been implemented and tested. It exhibits an impressive cleverness in the difficulty of the facts that it is able to discover in order to enable the optimisations.

Expect for the concepts of static analysis, abstract interpretation (see [19]), and parameterisable analysis (at compiler implementation time, though, not at compile time, see [11]), our whole work is original. Two papers present parts of our contributions [21, 22].

## 7.2 Related Work

As we underline in the exposition of our contributions, we had to propose ourselves almost everything that we have presented, so it is not surprising to find that there is virtually no related work. In fact, the most closely related work is so more by the name than by the ideas.

Demand-driven analyses are presented by Duesterwald et al [23, 24], by Agrawal [1, 2], and by Heintze and Tardieu [31]. The analyses that are presented are a data-flow analysis, a simultaneous data-flow and call graph analysis, and a pointer analysis, respectively. In essence, these works consist in taking classical static analyses and turning them into lazy versions. That is, the presented analyses are able to produce only parts of the results that the classical ones compute *and* to reduce the necessary amount of computations accordingly. The demands represent the need for a specific part of the results. Demand processing rules are used to determine the minimal subset of computations that is necessary to produce only those parts. In each case, the original analysis is very simple and, not too surprisingly, the demand processing rules turn out to be quite simple, too.

Other work also shares similar names. But they are used in the compilation of languages featuring lazy evaluation. They have a completely different purpose: they are *normal* analyses that compute informations about demands on the suspended computations of the programs. They are usually referred to as strictness analyses. For the sake of information, such works are presented in [13], [47], and [52], for example.

## 7.3 Future Work

As our work is not exactly a polished, refined solution to a well-delimited problem but more a bold leap into a whole new methodology in static type analysis, it brings with it a lot of new questions, problems, and things to try. We briefly mention some.

### 7.3.1 On the Pattern-Based Analysis

A lot of additional work ought to be done on the pattern-based demand-driven analysis itself.

**Speeding Up the Analysis**

In order to make the demand-driven type analysis really practical, its speed must be improved. We propose some means to make it faster.

First, the approach would be much faster if the numerous re-analyses were not always computed from scratch. Indeed, a single modification to the abstract model does not necessarily imply that the new analysis results completely change. A kind of *incremental re-analysis* could be implemented. That is, given a model $\mathcal{M}$, the corresponding analysis results $\mathcal{R}$, and an updated model $\mathcal{M}'$, the new analysis results $\mathcal{R}'$ could be obtained more efficiently than by performing a re-analysis from scratch. A way to do it consists in having a mechanism that allows the analyser to retract from $\mathcal{R}$ the abstractions that have been refined (and only these) and then to propagate the refined values instead. At the beginning of the process of analysing the program, we expect the model to be so coarse that any update would concern a major fraction of the abstractions but, as the model becomes more refined, model updates should involve only a very small fraction of the abstractions and the retraction and propagation sweep should become minor.

Second, the direct manipulation of the naïvely represented abstractions during the analyses is costly and more efficient representations should be considered. Indices for the abstractions instead of the abstractions themselves would be more lightweight. Bit vectors are often employed to implement set operations, also.

Third, the representation of the contours could be optimised and they could be restricted to contain only the environment variables to which there is a reference. In most of the closure bodies, only a fraction of all the visible variables are really referenced. The corresponding contours should only list the values of these.

### Aggressive, Risky Strategies

In the current approach, the demand processing rules produce a single set of new demands and these demands are restricted to be *necessary* and sufficient. The uniqueness of the strategy could be abandoned. The rules could still produce the same conservative strategy but, additionally, more aggressive and risky strategies. These would not need to be made of necessary demands, but of sufficient ones. The multiplicity of strategies would make the analyser tolerant to the failure of the aggressive strategies and allow it to fall back to the conservative ones when necessary.

### Better Selection of Model-Modifying Demands

The current criterion for the selection of the "best" model-modifying demand is very naïve. A more appropriate criterion should measure the *quality* of the information contained in the analysis results. Sometimes, good (informative) analysis results need to be verbose.

Also, the current method consists in selecting a model-modifying demand after the other and accumulating the updates without considering other sequences of updates. This sequence of updates can be viewed as a search for an ideal model. Now, single-threaded searches have the inconvenience of being easy to trap in "local optima". Browsing through elementary AI references for search methods could be profitable. For example, a kind of best-first search could be more effective than our greedy search.

### Extension to Scheme

Our demand-driven type analysis is intended for the mini-language but should be extended to cover a dynamically-typed functional language such as Scheme. We expect the greatest challenge to come partly from separate compilation (not a standard feature but a part of most Scheme implementations) and from continuations but, most of all, from the side-effects created by `define`, `set!`, and a few standard library functions. Indeed, the heart of the pattern-based approach relies on the absence of side-effects. Contours, by definition, represent the value of the variables in the lexical environment. But what does it mean to be in a contour where, say, 'x' is constrained to contain a pair and then a side-effect mutates its contents to a vector? Does the contour stays the same and we allow vectors to be contained in variable 'x' despite the fact that the contours says that the bound on the possible values of 'x' ought to be the pairs? Or does the contour instantaneously changes when the side-effect occurs?

### 7.3.2 Alternate Modelling

The pattern-based modelling of values and evaluation contexts is just a choice of ours and a different modelling could be used while maintaining the fact that the demand-driven analysis uses abstract interpretation.

#### The Use of Labels

We should try a modelling of the pairs that produce abstractions that remember the label of the `cons`-expressions that created them. However, recall that we argued that pairs are never discriminated on the basis of their origins in the concrete interpretation. So they should not be in the abstract interpretation either. Also, abstract pairs without labels help in avoiding a proliferation of abstractions having the same meaning. But the point of creation may carry a lot of information as the programmer may have different plans for pairs created in different parts of the programs.

#### Regular Trees

Patterns, and even patterns that include creation site labels, are shallow representations of concrete values. Of course, we showed that deep invariants could sometimes be discovered through the use of the information kept in the log matrices of the analysis framework. Regular trees, on the contrary, naturally express deep invariants of the concrete values. A sound mathematical basis comes along with them. Analyses using regular trees should be considered. They have been used by Aiken (and collaborators) in [5, 4, 6] and presented by Courcelle in [18]. The results by Aiken showed an impressive representation power but did not seem to be efficient enough.

### 7.3.3 Extensions

#### Other Languages

Although we explicitly aim at analysing dynamically-typed languages, we believe that the type analysis could be useful in some statically-typed languages, too. Indeed, statically-typed languages such as ML and Haskell feature algebraic types. The particular choice of a constructor is not determined at compile time. In many situations (such as prior to the extraction of the first element of a list), a dynamic test must be performed to ensure that an appropriate constructor is being manipulated. These dynamic tests are perfectly analogous to the safety types tests made in Scheme, for example. And they incur similar run-time penalties, too.

In fact, we can consider the typing system of Scheme to be implemented as a single algebraic type that includes many different constructors. The main type means "Scheme object" and the constructors mean number, character, etc. To push the point further, we say that even if Scheme programs do not include type annotations, they usually respect an implicit type discipline that is much stricter than the full dynamism that Scheme allows. We believe that Scheme programs and ML and Haskell programs often have very similar data structures with comparable type signatures, even if no static verification of the types is done in the first case.

#### Profiling

Having profiling statistics about the program to analyse would be very useful to the demand-driven analysis. It would put a realistic *price* on the safety tests or, conversely, a realistic *profit estimation* on the eventual removal of these tests. It is folklore in computer science that execution occurs 90% of the time in only 10% of the program. The work units invested in the demand-driven analysis would be used in a more profitable way if they enabled optimisations on more frequently executed code.

### Different Sources of Initial Demands

We are able to say that our demands express reasonable requests because they correspond to necessary properties of the program. The basis of this necessity is that run-time errors probably will not occur. Consequently, initial demands are only generated from expressions where run-time errors could occur. In an extended system, initial demands could be generated from different sources. However, various degrees of reliability should be attributed to these sources. That is, the confidence that the properties must be true varies from a source to another. For example, during the compilation of a complete program along with the necessary library functions, a higher degree of reliability should be granted to the demands originating from expressions in the library functions. Indeed, these are normally written with extreme care while it is doubtful that the program should be considered to be as secure as the library.

If profiling were used, a whole family of optimiser needs could be taken care of by the analyser. For example, the information needed by the optimiser to perform inlining is not related to safety issues at all. But if profiling statistics show that, at certain call sites, the same closures are always invoked, then some kind of credibility could be granted to a demand requesting the demonstration of the conjectured (but desirable) property.

### Certainty Analysis

If a future extension of the demand-driven analysis allows the demand processing rules to speculatively generate aggressive non-necessary strategies, it would be useful to know which strategies are more likely to fail or, even, which are sure to fail. Profiling information helps in deciding which are likely to fail. But in order to know that an aggressive strategy is sure to fail, we have to know that a particular non-necessary property is certainly false. For example, let us suppose that the processing of a demand $D$ would be greatly simplified if it could be shown that closure $c$ does not get invoked at $e_l$. Suppose also that the stated property is not a necessary one. An aggressive strategy might try to achieve the desired demonstration. However, if we knew that $c$ is indeed invoked at $e_l$ in at least one occasion, the analyser would avoid to make a useless attempt with this aggressive strategy.

That kind of information is knowledge that something *does* occur. Analyses used for optimisation purposes never gather that kind of knowledge. They are conservative analyses and they gather a superset of all that happens. The information that we need in the present case is of the opposite nature: it is a subset of all that happens. All the facts reported by such an analysis are sure to happen. We call such an analysis a *certainty analysis*. Its results would be useful for the evaluation of the pertinence of various strategies.

#### Other Kinds of Analyses

The general approach of generating and processing demands that express necessary properties could be tried on other analyses than type analysis. Its natural applications are the analyses related to safety issues. It should adapt well to numerical range analysis, for instance. Such an analysis determines in which range all the numerical values contained in a variable must lie. This information is then used to optimise accesses to arrays since one or both bound checks may possibly be dropped.

By using profiling statistics to obtain suggestions of plausible properties, the (non-type) analysis need not necessarily be related to safety issues. It appears that most of the optimisations are not related to safety. For example, inlining (see [9, 39]), eager evaluation in lazy languages (with the help of strictness analyses, see [14, 13, 47, 52]), register allocation (with the help of liveliness analysis and pointer or alias analysis, see [3, 17, 65, 31]), stack allocation to replace heap allocation (see [25, 53]), selection of efficient representation for the values (see [32, 33, 54]), recycling of heap objects (see [35, 36]), elimination of dead code (see [3, 40]), static branch prediction (using numerical analysis, though, see [48]), etc.

### 7.3.4 Demand Propagation Calculus

We merely make an allusion to this subject as it is no more than a vague idea by ours. A demand-driven type analysis could be based on a pure demand propagation calculus and not relying on abstract interpretation of the programs at all. We imagine that the result would be a kind of reverse abstract interpretation where bounds on acceptable values are propagated backward in the program instead of sets of possibles values being propagated forward. However, we are not able to guess what would be the power of such an approach or whether it would be equivalent to something that is already known.

# Bibliography

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

[2] Gagan Agrawal, Jinqian Li, and Qi Su. Evaluating a demand-driven technique for call graph construction. In *Computational Complexity*, pages 29–45, 2002.

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

[4] Alexander Aiken and Brian Murphy. Implementing regular tree expressions. In *Functional Programming and Computer Architecture*, pages 427–447, aug 1991.

[5] Alexander Aiken and Brian Murphy. Static type inference in a dynamically typed language. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pages 279–290, jan 1991.

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

[7] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, jan 1994.

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. Technical Report DEC-SRC-62, Digital Equipment Corporation, Systems Research Centre, aug 1990.

[9] J. Michael Ashley. The effectiveness of flow analysis for inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[10] J. Michael Ashley and Charles Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. In *ACM Transactions on Programming Languages and Systems*, pages 1431–1448, sep 1994.

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

[12] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference of Functional Programming*, pages 1–10, jun 1997.

[13] Sandip K. Biswas. A demand-driven set-based analysis. In *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, jan 1997.

[14] Adrienne Bloss and Paul Hudak. Variations on strictness analysis. In *1986 ACM Symposium on Lisp and Functional Programming*, pages 132–142, 1986.

[15] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[16] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the 1993 ACM Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[17] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Conference on Programming Language Design and Implementation*, volume 30, pages 130–138, jun 1995.

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

[19] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28:324–328, jun 1996.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, jan 1977.

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

[22] Danny Dubé and Marc Feeley. A demand-driven adaptive type analysis. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, pages 84–97, oct 2002.

[23] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Symposium of Principles of Programming Languages*, pages 37–48, jan 1995.

[24] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering*, pages 575–586, mar 1996.

[25] Benjamin Goldberg and Young Gil Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 152–160, may 1990.

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

[27] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, may 1977.

[28] The Haskell language. `http://www.haskell.org/`.

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

[30] Nevin Heintze. Control-flow analysis and type systems. *Lecture Notes in Computer Science*, 983:189–206, 1995.

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

[32] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215. ACM, aug 1992.

[33] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Portland, Oregon*, pages 213–226, jan 1994.

[34] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata, languages and computations.* Addison-Wesley, Reading, MA, 1979.

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

[36] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, oct 1988.

[37] Suresh Jagannathan and Stephen 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.

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

[39] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 193–205, 1996.

[40] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, 1994.

[41] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[42] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pages 419–428, 1993.

[43] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[44] The ML language. `http://cm.bell-labs.com/cm/cs/what/smlnj/sml97.html`.

[45] Patrick O'Keefe and Mitchell Wand. Type inference for partial types is decidable. In *ESOP'92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 408–417, feb 1992.

[46] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, apr 1995.

[47] Dirk Pape. Higher order demand propagation. *Lecture Notes in Computer Science*, 1595:153–168, 1999.

[48] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 67–78, jun 1995.

[49] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 305–315, jan 1992.

[50] The Prolog language. `http://www.logic-programming.org/prolog_std.html`.

[51] The Scheme language. `http://www.scheme.org/`.

[52] R. Sekar and I. V. Ramakrishnan. Fast strictness analysis based on demand propagation. *ACM Transactions on Programming Languages and Systems*, 17(6):896–937, nov 1995.

[53] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 1996.

[54] Zhong Shao. Flexible representation analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, jun 1997.

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

[56] Olin Shivers. Cps data-flow analysis example. Technical report, Carnegie Mellon University, may 1990.

[57] Olin Shivers. Data-flow analysis and type recovery in Scheme. Technical Report CMU-CS-90-115, Carnegie Mellon University, mar 1990.

[58] Olin Shivers. Super-$\beta$: Copy, constant, and lambda propagation in Scheme. Technical report, Carnegie Mellon University, may 1990.

[59] Olin Shivers. Useless-variable elimination. Technical report, Carnegie Mellon University, apr 1990.

[60] Olin Shivers. *Control-flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, may 1991.

[61] Olin 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.

[62] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. To be published by ACM.

[63] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, pages 686–701, apr 1993.

[64] Adam Brooks Webber. Program analysis using binary relations. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–260, jun 1997.

[65] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, volume 30, pages 1–12, jun 1995.

[66] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Conference on Lisp and Functional Programming*, pages 250–262, jun 1994.

# Appendix A

# Benchmarks

The source of each benchmark is presented next. All benchmarks except **change**, **interp**, and **cps-QS-s** are written in the syntax of the extended mini-language. These benchmarks have to be reduced to the basic mini-language and $\alpha$-converted before the demand-driven analysis can operate on them. On the other hand, the **change**, **interp**, and **cps-QS-s** benchmarks are written in Scheme syntax. Before they can be processed by the demand-driven analysis, they first have to be translated from Scheme to the extended mini-language and then undergo the same reductions as the other benchmarks.

## A.1 Source of the cdr-safe Benchmark

```
(let$_1$ cdr-safe = ($\lambda_2$l. (if$_3$ (pair?$_4$ l$_5$) (cdr$_6$ l$_7$) #f$_8$))
   ($_9$cdr-safe$_{10}$ ($_{11}$cdr-safe$_{12}$ (cons$_{13}$ #f$_{14}$ ($\lambda_{15}$x. x$_{16}$)))))
```

## A.2 Source of the loop Benchmark

```
(letrec$_1$ foo = ($\lambda_2$x. ($_3$foo$_4$ x$_5$))
   ($_6$foo$_7$ #f$_8$))
```

## A.3 Source of the 2-1 Benchmark

```
(−_1 2_2 1_3)
```

## A.4 Source of the map-easy Benchmark

```
(letrec_1 map =
    (λ_2op. (λ_3l. (if_4 l_5 (cons_6 (_7op_8 (car_9 l_{10})) (_{11}(_{12}map_{13} op_{14}) (cdr_{15} l_{16}))) #f_{17})))
  (let_{18} d = (cons_{19} (λ_{20}x. x_{21}) (cons_{22} #f_{23} #f_{24}))
    (let_{25} list = (cons_{26} d_{27} (cons_{28} d_{29} (cons_{30} d_{31} #f_{32})))
      (let_{33} op1 = (λ_{34}y. (_{35}(car_{36} y_{37}) #f_{38}))
        (let_{39} op2 = (λ_{40}z. (car_{41} (cdr_{42} z_{43})))
          (cons_{44} (_{45}(_{46}map_{47} op1_{48}) list_{49}) (_{50}(_{51}map_{52} op2_{53}) list_{54})))))))
```

## A.5 Source of the map-hard Benchmark

```
(letrec_1 map =
    (λ_2op. (λ_3l. (if_4 l_5 (cons_6 (_7op_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}))))))
```

## A.6 Source of the fib Benchmark

```
(letrec_1 fib = (λ_2n. (if_3 (<=_4 n_5 1_6) n_7 (+_8 (_9fib_{10} (−_{11} n_{12} 1_{13})) (_{14}fib_{15} (−_{16} n_{17} 2_{18})))))
  (_{19}fib_{20} 7_{21}))
```

## A.7 Source of the gcd Benchmark

```
(letrec_{1} mod = (λ_{2}x. (λ_{3}d. (-_{4} x_{5} (*_{6} (/_{7} x_{8} d_{9}) d_{10}))))
  (letrec_{11} gcd = (λ_{12}b. (λ_{13}s. (if_{14} (=_{15} s_{16} 0_{17}) b_{18} (_{19}(_{20}gcd_{21} s_{22})
                                                                                        (_{23}(_{24}mod_{25} b_{26}) s_{27})))))
    (let_{28} gcd = (λ_{29}x. (λ_{30}y. (if_{31} (>=_{32} x_{33} y_{34}) (_{35}(_{36}gcd_{37} x_{38}) y_{39})
                                                                           (_{40}(_{41}gcd_{42} y_{43}) x_{44}))))
      (_{45}(_{46}gcd_{47} 3_{48}) 5_{49}))))
```

## A.8 Source of the tak Benchmark

```
(letrec_{1} tak = (λ_{2}x. (λ_{3}y. (λ_{4}z. (if_{5} (<=_{6} x_{7} y_{8})
                                                        z_{9}
                                                        (_{10}(_{11}(_{12}tak_{13} (_{14}(_{15}(_{16}tak_{17} (-_{18} x_{19} 1_{20})) y_{21}) z_{22}))
                                                                    (_{23}(_{24}(_{25}tak_{26} (-_{27} y_{28} 1_{29})) z_{30}) x_{31}))
                                                              (_{32}(_{33}(_{34}tak_{35} (-_{36} z_{37} 1_{38})) x_{39}) y_{40}))))))
  (_{41}(_{42}(_{43}tak_{44} 18_{45}) 12_{46}) 6_{47}))
```

## A.9 Source of the n-queens Benchmark

```
(letrec_{1} make-list =
  (λ_{2}n. (λ_{3}v. (if_{4} (=_{5} n_{6} 0_{7}) #f_{8} (cons_{9} v_{10} (_{11}(_{12}make-list_{13} (-_{14} n_{15} 1_{16})) v_{17})))))
 (letrec_{18} list-ref =
   (λ_{19}l. (λ_{20}n. (if_{21} (=_{22} n_{23} 0_{24}) (car_{25} l_{26}) (_{27}(_{28}list-ref_{29} (cdr_{30} l_{31}))
                                                                                 (-_{32} n_{33} 1_{34})))))
  (letrec_{35} list-set =
    (λ_{36}l. (λ_{37}n. (λ_{38}v. (if_{39} (=_{40} n_{41} 0_{42})
                                      (cons_{43} v_{44} (cdr_{45} l_{46}))
                                      (cons_{47} (car_{48} l_{49})
                                                 (_{50}(_{51}(_{52}list-set_{53} (cdr_{54} l_{55})) (-_{56} n_{57} 1_{58})) v_{59}))))))
   (letrec_{60} nq =
     (λ_{61}n.
      (λ_{62}i.
       (λ_{63}sw.
        (λ_{64}s.
         (λ_{65}se.
          (if_{66} (=_{67} i_{68} 0_{69})
           1_{70}
           (letrec_{71} loop =
             (λ_{72}j.
              (if_{73} (=_{74} j_{75} n_{76})
               0_{77}
               (+_{78}
                (if_{79} (_{80}(_{81}list-ref_{82} sw_{83}) j_{84})
                 (if_{85} (_{86}(_{87}list-ref_{88} s_{89}) j_{90})
                  (if_{91} (_{92}(_{93}list-ref_{94} se_{95}) j_{96})
                   (let_{97} sw = (cdr_{98} (_{99}(_{100}(_{101}list-set_{102} sw_{103}) j_{104}) #f_{105}))
                    (let_{106} s = (_{107}(_{108}(_{109}list-set_{110} s_{111}) j_{112}) #f_{113})
                     (let_{114} se = (cons_{115} (cons_{116} #f_{117} #f_{118})
                                     (_{119}(_{120}(_{121}list-set_{122} se_{123}) j_{124}) #f_{125}))
                      (_{126}(_{127}(_{128}(_{129}(_{130}nq_{131} n_{132}) (−_{133} i_{134} 1_{135})) sw_{136}) s_{137})
                         se_{138}))))
                   0_{139})
                  0_{140})
                 0_{141})
                (_{142}loop_{143} (+_{144} j_{145} 1_{146})))))
            (_{147}loop_{148} 0_{149}))))))))
    (let_{150} nqueens =
      (λ_{151}n. (let_{152} flags = (_{153}(_{154}make-list_{155} (*_{156} 2_{157} n_{158})) (cons_{159} #f_{160} #f_{161}))
             (_{162}(_{163}(_{164}(_{165}(_{166}nq_{167} n_{168}) n_{169}) flags_{170}) flags_{171}) flags_{172})))
     (_{173}nqueens_{174} 4_{175}))))))
```

## A.10 Source of the `ack` Benchmark

```
(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}))
```

## A.11 Source of the `SKI` Benchmark

```
(letrec_{1} append =
 (λ_{2}l1. (λ_{3}l2. (if_{4} (pair?_{5} l1_{6}) (cons_{7} (car_{8} l1_{9}) (_{10}(_{11}append_{12} (cdr_{13} l1_{14})) l2_{15}))
                         l2_{16})))
 (letrec_{17} eval =
  (λ_{18}exp. (if_{19} (car_{20} (car_{21} exp_{22}))
          (_{23}eval_{24} (_{25}(_{26}append_{27} (car_{28} exp_{29})) (cdr_{30} exp_{31})))
          (let_{32} c = (car_{33} exp_{34})
           (let_{35} rest = (cdr_{36} exp_{37})
            (if_{38} (pair?_{39} rest_{40})
             (let_{41} arg1 = (car_{42} rest_{43})
              (let_{44} rest = (cdr_{45} rest_{46})
               (if_{47} (pair?_{48} (cdr_{49} c_{50}))
                (if_{51} (pair?_{52} rest_{53})
                   (let_54 arg2 = (car_55 rest_56)
                    (let_57 rest = (cdr_58 rest_59)
                     (if_60 (pair?_61 (cdr_62 (cdr_63 c_64)))
                      (if_65 (pair?_66 rest_67)
                       (let_68 arg3 = (car_69 rest_70)
                        (let_71 rest = (cdr_72 rest_73)
                         (_74eval_75 (cons_76 (cons_77 arg1_78 (cons_79 arg3_80 #f_81))
                                            (cons_82 (cons_83 arg2_84 (cons_85 arg3_86 #f_87))
                                                     rest_88)))))
                       exp_89)
                      (_90eval_91 (cons_92 arg1_93 rest_94)))))
                   exp_95)
                  (_96eval_97 (cons_98 arg1_99 rest_100)))))
               exp_101)))))
(_102
 eval_103
 (cons_104
  (cons_105
   (cons_106
    (cons_107 #f_108 (cons_109 #f_110 (cons_111 #f_112 #f_113)))
    (cons_114
     (cons_115
      (cons_116 #f_117 (cons_118 #f_119 #f_120))
      (cons_121 (cons_122 (cons_123 (cons_124 #f_125 (cons_126 #f_127 (cons_128 #f_129 #f_130)))
                                    (cons_131 (cons_132 #f_133 #f_134) #f_135))
                          (cons_136 (cons_137 #f_138 #f_139) #f_140))
                #f_141))
     #f_142))
   (cons_143
    (cons_144
     (cons_145
      (cons_146 #f_147 (cons_148 #f_149 (cons_150 #f_151 #f_152)))
      (cons_153
       (cons_154
        (cons_155
         (cons_156 #f_157 (cons_158 #f_159 (cons_160 #f_161 #f_162)))
         (cons_163
          (cons_164 (cons_165 #f_166 (cons_167 #f_168 #f_169))
                    (cons_170 (cons_171 #f_172 (cons_173 #f_174 (cons_175 #f_176 #f_177))) #f_178))
          #f_179))
        (cons_180
         (cons_181
          (cons_182 (cons_183 #f_184 (cons_185 #f_186 (cons_187 #f_188 #f_189)))
                    (cons_190 (cons_191 (cons_192 #f_193 (cons_194 #f_195 #f_196))
                                        (cons_197 (cons_198 #f_199 (cons_200 #f_201 #f_202)) #f_203))
                              #f_204))
          (cons_205 (cons_206 #f_207 #f_208) #f_209))
         #f_210))
       #f_211))
     (cons_212
      (cons_213
       (cons_214 #f_215 (cons_216 #f_217 #f_218))
       (cons_219 (cons_220 (cons_221 (cons_222 #f_223 (cons_224 #f_225 (cons_226 #f_227 #f_228)))
                           (cons$_{229}$ (cons$_{230}$ #f$_{231}$ #f$_{232}$) #f$_{233}$))
                    (cons$_{234}$ (cons$_{235}$ #f$_{236}$ #f$_{237}$) #f$_{238}$))
             #f$_{239}$))
      #f$_{240}$))
    #f$_{241}$))
   (cons$_{242}$ (cons$_{243}$ #f$_{244}$ #f$_{245}$) #f$_{246}$)))))
```

## A.12 Source of the change Benchmark

```
(let ((queue-empty
       (cons '() '())))
 (let ((queue-insert
        (lambda (q x)
          (if (null? (car q))
            (cons (cons x '()) '())
            (cons (car q) (cons x (cdr q)))))))
  (let ((queue-top
         (lambda (q)
           (car (car q)))))
   (let ((queue-pop
          (lambda (q)
            (let ((head (cdr (car q))))
             (if (null? head)
               (cons (reverse (cdr q)) '())
               (cons head (cdr q)))))))
    (let ((queue->list
           (lambda (q)
             (append (car q) (reverse (cdr q))))))
     (let ((stratv->stratf
            (lambda (v)
              (let ((len (vector-length v)))
               (let ((c (cdr (vector-ref v 0))))
                (lambda (M)
                  (if (< M len)
                    (vector-ref v M)
                    (let ((n (quotient (+ (- M (- len 1)) (- c 1)) c)))
                     (let ((cpl (vector-ref v (- M (* n c)))))
                      (cons (+ n (car cpl)) c))))))))))
      (letrec
          ((ret
            (lambda (coins)
              (if (null? (cdr coins))
                (list->vector (cons (cons 0 1) '()))
                (let ((c (car coins)))
                 (let ((rest (cdr coins)))
                  (let ((v (ret rest)))
                   (let ((f (stratv->stratf v)))
                    (let ((initq (queue-insert queue-empty (cons 0 c))))
                     (letrec
                         ((loop2
                           (lambda (M wq nb-c sq)
                             (if (= nb-c c)
                               (list->vector (queue->list sq))
                               (let ((strat-hi (queue-top wq)))
                                (let ((wq (queue-pop wq)))
                                 (let ((nb-c (if (= (cdr strat-hi) c)
                                                 (- nb-c 1)
                                                 nb-c)))
                                  (let ((sq (queue-insert sq strat-hi)))
                                   (let ((strat-lo (f M)))
                                    (if (< (+ (car strat-hi) 1)
                                           (car strat-lo))
                                      (let ((strat
                                             (cons (+ (car strat-hi) 1)
                                                   c)))
                                       (let ((wq (queue-insert wq strat)))
                                        (loop2 (+ M 1) wq (+ nb-c 1) sq)))
                                      (let ((wq
                                             (queue-insert wq strat-lo)))
                                       (loop2 (+ M 1)
                                              wq
                                              nb-c
                                              sq))))))))))))
                      (letrec
                          ((loop1
                            (lambda (M wq nb-c)
                              (if (< M c)
                                (loop1 (+ M 1)
                                       (queue-insert wq (f M))
                                       nb-c)
                                (loop2 M wq nb-c queue-empty)))))
                       (loop1 1 initq 1)))))))))))
       (ret (cons 25 (cons 17 (cons 4 (cons 3 (cons 1 '()))))))))))))))))
```

## A.13 Source of the interp Benchmark

```
(letrec ((zip
         (lambda (l1 l2)
           (if (null? l1)
               '()
               (cons (cons (car l1) (car l2)) (zip (cdr l1) (cdr l2)))))))
(let ((capply
       (lambda (f)
         (lambda (args) (apply f args)))))
 (let
   ((std-alist
     (append
      (append
       (append
        (append (append (cons (cons 'null? (capply null?)) '())
                        (cons (cons 'boolean? (capply boolean?)) '()))
                (append (cons (cons 'number? (capply number?)) '())
                        (cons (cons 'equal? (capply equal?)) '())))
        (append (append (cons (cons '= (capply =)) '())
                        (cons (cons '< (capply <)) '()))
                (append (cons (cons '<= (capply <=)) '())
                        (cons (cons '+ (capply +)) '()))))
       (append (append (append (cons (cons '- (capply -)) '())
                               (cons (cons '* (capply *)) '()))
                       (append (cons (cons 'quotient (capply quotient))
                                     '())
                               (cons (cons 'cons (capply cons)) '())))
               (append (append (cons (cons 'car (capply car)) '())
                               (cons (cons 'cdr (capply cdr)) '()))
                       (append (cons (cons 'cadr (capply cadr)) '())
                               (cons (cons 'caddr (capply caddr)) '())))))
      (append
       (append
        (append (append (cons (cons 'cadddr (capply cadddr)) '())
                        (cons (cons 'length (capply length)) '()))
                (append (cons (cons 'reverse (capply reverse)) '())
                        (cons (cons 'append (capply append)) '())))
        (append
         (append (cons (cons 'assoc (capply assoc)) '())
                 (cons (cons 'vector-length (capply vector-length)) '()))
         (append (cons (cons 'vector-ref (capply vector-ref)) '())
                 (cons (cons 'list->vector (capply list->vector)) '()))))
       (append
        (append
         (append
          (cons (cons 'map (lambda (args)
                             (let ((f (car args)))
                               (map (lambda (x) (f (cons x '())))
                                    (cadr args)))))
```

```
              '())
        (cons (cons 'apply (lambda (args) ((car args) (cadr args))))
              '()))
       (append (cons (cons 'symbol? (capply symbol?)) '())
               '()))
      '()))))))
(let ((standard-environment
       (lambda (v)
         (cdr (assoc v std-alist)))))
 (letrec ((ev
           (lambda (exp env)
            (if (boolean? exp)
              exp
              (if (number? exp)
                exp
                (if (symbol? exp)
                  (env exp)
                  (let ((kw (car exp)))
                   (if (equal? kw 'quote)
                     (cadr exp)
                     (if (equal? kw 'lambda)
                       (let ((fpars (cadr exp)))
                        (let ((body (caddr exp)))
                         (lambda (apars)
                          (let ((alist (zip fpars apars)))
                           (ev body
                               (lambda (v)
                                 (let ((a (assoc v alist)))
                                   (if a (cdr a) (env v)))))))))
                       (if (equal? kw 'if)
                         (if (ev (cadr exp) env)
                           (ev (caddr exp) env)
                           (ev (cadddr exp) env))
                         (if (equal? kw 'let)
                           (let ((binding (car (cadr exp))))
                            (let ((var (car binding)))
                             (let ((val (ev (cadr binding) env)))
                              (ev (caddr exp)
                                  (lambda (v)
                                    (if (equal? v var) val (env v)))))))
                           (if (equal? kw 'letrec)
                             (let ((binding (car (cadr exp))))
                              (let ((var (car binding)))
                               (let ((l-e (cadr binding)))
                                (letrec ((env2 (lambda (v)
                                                 (if (equal? v var)
                                                     (ev l-e env2)
```

```
                                                        (env v)))))
                                    (ev (caddr exp) env2)))))
                                ((ev (car exp) env)
                                 (map (lambda (e) (ev e env))
                                      (cdr exp)))))))))))))))
     (let ((eval (lambda (exp) (ev exp standard-environment))))
      (eval
       (cons 'letrec
             (cons
              (cons (cons 'foo
                          (cons (cons 'lambda
                                      (cons '()
                                            (cons (cons 'foo '()) '())))
                                '()))
                    '())
              (cons (cons 'foo '()) '()))))))))))
```

## A.14 Source of the cps-QS-s Benchmark

```
(let ((CPS-=
       (lambda (x y k) (k (= x y)))))
 (let ((CPS-if
        (lambda (res k1 k2) (if res (k1) (k2)))))
  (let ((CPS-*
         (lambda (x y k) (k (* x y)))))
   (let ((CPS-modulo
          (lambda (x y k) (k (modulo x y)))))
    (let ((CPS-cons
           (lambda (x y k) (k (cons x y)))))
     (let ((CPS-null?
            (lambda (x k) (k (null? x)))))
      (let ((CPS-car
             (lambda (x k) (k (car x)))))
       (let ((CPS-cdr
              (lambda (x k) (k (cdr x)))))
        (let ((CPS-<
               (lambda (x y k) (k (< x y)))))
         (let ((CPS-<=
                (lambda (x y k) (k (<= x y)))))
          (let ((CPS-append
                 (lambda (x y k) (k (append x y)))))
           (let ((CPS-k
                  (lambda (res) res))
```

```
(let ((gen-list
       (lambda (g p k1)
        (letrec ((loop
                  (lambda (n acc k2)
                   (CPS-= n 1
                    (lambda (tmp1)
                     (CPS-if tmp1
                      (lambda ()
                       (k2 acc))
                      (lambda ()
                       (CPS-* n g
                        (lambda (tmp2)
                         (CPS-modulo tmp2 p
                          (lambda (tmp3)
                           (CPS-cons n acc
                            (lambda (tmp4)
                             (loop tmp3 tmp4
                              k2))))))))))))))
         (CPS-cons 1 '()
          (lambda (tmp5)
           (loop g tmp5
            k1)))))))
 (letrec ((filter
           (lambda (pred? l k3)
            (CPS-null? l
             (lambda (tmp6)
              (CPS-if tmp6
               (lambda ()
                (k3 '()))
               (lambda ()
                (CPS-car l
                 (lambda (tmp7)
                  (pred? tmp7
                   (lambda (tmp8)
                    (CPS-if tmp8
                     (lambda ()
                      (CPS-car l
                       (lambda (tmp9)
                        (CPS-cdr l
                         (lambda (tmp10)
                          (filter pred? tmp10
                           (lambda (tmp11)
                            (CPS-cons tmp9 tmp11
                             k3))))))))
                     (lambda ()
                      (CPS-cdr l
                       (lambda (tmp12)
                      (filter pred? tmp12
                       k3))))))))))))))))
(letrec ((quicksort
          (lambda (l k4)
           (CPS-null? l
            (lambda (tmp13)
             (CPS-if tmp13
              (lambda ()
               (k4 '()))
              (lambda ()
               (CPS-car l
                (lambda (pivot)
                 (CPS-cdr l
                  (lambda (rest)
                   (filter
                    (lambda (n k5)
                     (CPS-< n pivot
                      k5))
                    rest
                    (lambda (lows)
                     (filter
                      (lambda (n k6)
                       (CPS-<= pivot n
                        k6))
                      rest
                      (lambda (highs)
                       (quicksort lows
                        (lambda (tmp14)
                         (quicksort highs
                          (lambda (tmp15)
                           (CPS-cons pivot tmp15
                            (lambda (tmp16)
                             (CPS-append tmp14 tmp16
                              k4))))))))))))))))))))))
 (gen-list 2 29
  (lambda (tmp17)
   (quicksort tmp17
    CPS-k))))))))))))))))))
```

## A.15 Source of the cps-QS-m Benchmark

(let$_1$ CPS-= = ($\lambda_2$x. ($\lambda_3$y. ($\lambda_4$k. ($_5$k$_6$ (=$_7$ x$_8$ y$_9$)))))
 (let$_{10}$ CPS-if = ($\lambda_{11}$res. ($\lambda_{12}$k1. ($\lambda_{13}$k2. (if$_{14}$ res$_{15}$ ($_{16}$k1$_{17}$ #f$_{18}$) ($_{19}$k2$_{20}$ #f$_{21}$)))))

```
(let_22 CPS-* = (λ_23x. (λ_24y. (λ_25k. (_26k_27 (*_28 x_29 y_30)))))
 (let_31 CPS-modulo = (λ_32x. (λ_33y. (λ_34k. (_35k_36 (-_37 x_38 (*_39 (/_40 x_41 y_42) y_43))))))
  (let_44 CPS-cons = (λ_45x. (λ_46y. (λ_47k. (_48k_49 (cons_50 x_51 y_52)))))
   (let_53 CPS-null? = (λ_54x. (λ_55k. (_56k_57 (if_58 x_59 #f_60 (cons_61 x_62 x_63)))))
    (let_64 CPS-car = (λ_65x. (λ_66k. (_67k_68 (car_69 x_70))))
     (let_71 CPS-cdr = (λ_72x. (λ_73k. (_74k_75 (cdr_76 x_77))))
      (let_78 CPS-< = (λ_79x. (λ_80y. (λ_81k. (_82k_83 (<_84 x_85 y_86)))))
       (let_87 CPS-<= = (λ_88x. (λ_89y. (λ_90k. (_91k_92 (<=_93 x_94 y_95)))))
        (let_96 CPS-append =
         (λ_97x. (λ_98y. (λ_99k. (letrec_100 loop =
                                  (λ_101l. (if_102 l_103 (cons_104 (car_105 l_106)
                                                              (_107loop_108 (cdr_109 l_110)))
                                                   y_111))
                                  (_112k_113 (_114loop_115 x_116))))))
        (let_117 CPS-k = (λ_118res. res_119)
         (let_120 gen-list =
          (λ_121g.
           (λ_122p.
            (λ_123k1.
             (letrec_124 loop =
              (λ_125n.
               (λ_126acc.
                (λ_127k2.
                 (_128(_129(_130CPS-=_131 n_132) 1_133)
                      (λ_134tmp1.
                       (_135(_136(_137CPS-if_138 tmp1_139) (λ_140dummy. (_141k2_142 acc_143)))
                            (λ_144dummy.
                             (_145(_146(_147CPS-*_148 n_149) g_150)
                                  (λ_151tmp2.
                                   (_152(_153(_154CPS-modulo_155 tmp2_156) p_157)
                                        (λ_158tmp3.
                                         (_159(_160(_161CPS-cons_162 n_163) acc_164)
                                              (λ_165tmp4. (_166(_167(_168loop_169 tmp3_170) tmp4_171)
                                                                       k2_172))))))))))))))
               (_173(_174(_175CPS-cons_176 1_177) #f_178)
                    (λ_179tmp5. (_180(_181(_182loop_183 g_184) tmp5_185) k1_186)))))))
          (letrec_187 filter =
           (λ_188pred?.
            (λ_189l.
             (λ_190k3.
              (_191(_192CPS-null?_193 l_194)
                   (λ_195tmp6.
                    (_196(_197(_198CPS-if_199 tmp6_200) (λ_201dummy. (_202k3_203 #f_204)))
                         (λ_205dummy.
                          (_206(_207CPS-car_208 l_209)
                               (λ_210tmp7.
                                (_211(_212pred?_213 tmp7_214)
                                     (λ_215tmp8.
                                      (_216(_217(_218CPS-if_219 tmp8_220)
                                                (λ_221dummy.
                                                 (_222(_223CPS-car_224 l_225)
                                                      (λ_226tmp9.
                                                       (_227(_228CPS-cdr_229 l_230)
                                                  (λ_231 tmp10.
                                                   (_232 (_233 (_234 filter_235 pred?_236) tmp10_237)
                                                        (λ_238 tmp11.
                                                         (_239 (_240 (_241 CPS-cons_242 tmp9_243)
                                                                  tmp11_244)
                                                             k3_245)))))))))
                                 (λ_246 dummy.
                                  (_247 (_248 CPS-cdr_249 l_250)
                                       (λ_251 tmp12. (_252 (_253 (_254 filter_255 pred?_256) tmp12_257)
                                                             k3_258))))))))))))))))
(letrec_259 quicksort =
  (λ_260 l.
   (λ_261 k4.
    (_262
     (_263 CPS-null?_264 l_265)
     (λ_266 tmp13.
      (_267 (_268 (_269 CPS-if_270 tmp13_271) (λ_272 dummy. (_273 k4_274 #f_275)))
           (λ_276 dummy.
            (_277 (_278 CPS-car_279 l_280)
                 (λ_281 pivot.
                  (_282 (_283 CPS-cdr_284 l_285)
                       (λ_286 rest.
                        (_287 (_288 (_289 filter_290 (λ_291 n. (λ_292 k5. (_293 (_294 (_295 CPS-<_296 n_297)
                                                                                   pivot_298)
                                                                              k5_299))))
                                 rest_300)
                             (λ_301 lows.
                              (_302 (_303 (_304 filter_305
                                             (λ_306 n. (λ_307 k6.
                                                         (_308 (_309 (_310 CPS-<=_311 pivot_312) n_313)
                                                             k6_314))))
                                       rest_315)
                                   (λ_316 highs.
                                    (_317 (_318 quicksort_319 lows_320)
                                         (λ_321 tmp14.
                                          (_322 (_323 quicksort_324 highs_325)
                                               (λ_326 tmp15.
                                                (_327 (_328 (_329 CPS-cons_330 pivot_331) tmp15_332)
                                                     (λ_333 tmp16.
                                                      (_334 (_335 (_336 CPS-append_337 tmp14_338)
                                                                tmp16_339)
                                                           k4_340)))))))))))))))))))))
 (_341 (_342 (_343 gen-list_344 2_345) 29_346)
      (λ_347 tmp17. (_348 (_349 quicksort_350 tmp17_351) CPS-k_352)))))))))))))))))
```