PARALLEL IMPLEMENTATION OF LAZY FUNCTIONAL LANGUAGES USING ABSTRACT DEMAND PROPAGATION

by

Guy Tremblay

School of Computer Science
McGill University, Montréal
Québec, Canada
November 1994

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

This thesis investigates the implementation of lazy functional programming languages on parallel machines using a novel approach to strictness analysis called *abstract demand propagation* and examines the impact of laziness on parallelism.

Although some work on strictness analysis using demand propagation has been done before, the present work is original in that it gives a precise interpretation of the notions of demands and demand propagation using non-standard denotational semantics. The intuition behind the notion of (abstract) demand propagation is that it can be seen as a form of (abstract) inverse computation.

This form of strictness analysis has been incorporated into a prototype compiler for a lazy variant of (a subset of) the Id functional programming language targeted for the TTDA dataflow machine, which makes it the first implementation of a lazy language on a dataflow architecture. The overall structure of that compiler is described together with some novel features incorporated in the compiler.

The last part of the thesis examines the question of whether sufficient parallelism can be obtained from programs written in a lazy functional language, which is done by comparing various implementation strategies: purely lazy, optimized lazy and lenient. As little parallelism can effectively be obtained using lazy strategies, the notion of *bounded speculative laziness* is introduced in order to tentatively improve the parallelism of the programs while preserving the key characteristic of laziness, i.e., the ability to manipulate potentially infinite data objects.

Résumé

Cette thèse traite de la mise en oeuvre des langages fonctionnels paresseux sur des machines parallèles via l'utilisation d'une nouvelle approche d'analyse de la stricticité appelée *propagation abstraite de demandes* et examine l'impact de l'approche paresseuse sur le parallélisme.

Bien qu'il existe déjà certaines approches d'analyse de stricticité utilisant la propagation de demandes, le présent travail est original en ce qu'il fournit une interprétation précise et formelle des notions de demandes et de propagation de demandes, interprétation obtenue via une sémantique dénotationnelle non-standard. L'intuition derrière cette notion de propagation (abstraite) de demandes est que celle-ci peut être vue comme une forme (abstraite) de calcul d'un inverse.

Cette forme d'analyse de stricticité a été incorporée dans un prototype de compilateur pour une variante paresseuse (d'un sous-ensemble) du langage Id destinée à l'exécution sur la machine à flux de données TTDA, ce qui en fait la première mise en oeuvre d'un langage complètement paresseux sur une machine à flux de données. La structure générale du compilateur est décrite ainsi qu'un nombre d'éléments additionnels ayant été incorporés dans le compilateur.

La dernière partie de la thèse tente de déterminer si suffisament de parallélisme peut être extrait de programmes écrits dans un langage paresseux, ce qui est fait en comparant diverses stratégies de mise en oeuvre: purement paresseuse, paresseuse optimisée et "lenient". Parce que peu de parallélisme peut effectivement être obtenu, la notion de *spéculation paresseuse (bornée)* est introduite dans le but de tenter d'améliorer le parallélisme tout en préservant la principale caractéristique de la paresse, à savoir, la capacité de manipuler des structures de données potentiellement infinies.

Acknowledgements/ Remerciements

First of all, I would like to thank Prof. G.R. Gao, my supervisor, for his guidance and for his patience for a thesis which has been quite long to come. He always had good questions, comments and suggestions that really helped improve the scope and the overall organization of the thesis.

I wish to thank Prof. L. Hendren for helping me straightening up my initial domain of demands, for showing me how to view the fixed point computation process in a much simpler way and for explaining how the McCAT compiler performed inter-procedural analysis. Thanks to Prof. P. Panangaden, which through his comment on an initial version of the demand propagation model — *How do you know this is not science-fiction?* — got me into a (long!) search to relate the abstract model with the *real* world, which finally lead to the notion of inverse as a model for demand propagation, thus ensuring that demand propagation *is not* science-fiction.

I also wish to thank the CSG group from MIT which developed and made available the Id World environment and the Id compiler, on top of which I was able to build my own lazy compiler: without this great piece of software, I would still be hacking up code. Special thanks to R.P. Johnson, who answered many of my questions concerning Id World and the Id compiler, and to J. Hicks, who fixed an important bug in the handling of L-structures, thus ensuring that the parallelism results I obtained were correct.

Ensuite, je désire remercier l'Université du Québec à Montréal, plus précisément mes collègues du département de Mathématiques et Informatique qui m'ont fait confiance en m'engageant comme professeur alors que j'avais à peine entamé mes études doctorales et qui m'ont fourni le support pour les terminer: congés de perfectionnement, congés sans solde, dégrèvement. Sans le 9.06 (bien que ressenti parfois comme un épée de Damoclès, comme un cadeau empoisonné) je ne me serais probablement jamais rendu jusqu'au bout.

Thanks to all the members of the ACAPS group that bear with my silent presence. And the various system administrators that answered the questions I had over my long stay at McGill, including Jacek S., from the EE dept., which took care of the two machines of which I was the sole user for over a year.

Je désire aussi remercier mes amis — François et Michel — et ma famille, qui ont connu avec moi, durant ces 9 années (eh oui!) les hauts et les (nombreux) bas de la vie d'un étudiant de doctorat. Remerciements, aussi, à ceux qui, professionnellement, m'ont aidé à passer à travers les périodes difficiles, tant moralement — Douglas Long, Catherine Pichard, Lucie Roy — que physiquement — Claude-Emile Racette. Dans ce dernier cas, comme le dit le dicton: *Quand l'appétit va tout va!* Prochain objectif: me faire reprendre les kilos perdus!

Pour terminer, je dois remercier trois personnes qui m'ont toujours connu en train de vivre les affres du doctorat: François, Michelle et Anne-Marie. Plus de 7 ans à me voir m'acharner, dans un premier temps, sur la recherche d'un sujet de thèse, puis, sur la thèse elle-même, et à vivre avec moi, surtout depuis 2 ans, des années de vache maigre. Dans le cas de Michelle et François, je les ai à coup sûr découragés de faire un doctorat, mais j'espère ne pas les avoir découragés de faire des études parce que cela peut être, malgré tout, fascinant. Quant à Anne-Marie, je lui souhaite un mémoire moins pénible que ma thèse!

Dédicace

Au Dr. Paul David, dont l'intérêt ne s'est jamais démenti ...

A Anne-Marie, qui m'a accompagné tout au long de ce voyage. Un voyage qui ne fut pas "facile, confortable à tout instant. Mais un voyage fascinant ... un voyage fascinant."

Contributions

The major contributions of this thesis are:

- A novel approach to the (backward) strictness analysis of lazy functional programming languages based on the notion of abstract demand propagation, together with a formalization of the notions of demand and demand propagation as (abstract) computation of inverse.
- A prototype implementation of a compiler based on the model of abstract demand propagation for a lazy functional programming language targeted for a dataflow machine, making this the first implementation of a lazy language on a dataflow architecture.
- An investigation of the impact of laziness on parallelism through the examination of the potential parallelism that can be extracted from programs written in a lazy language.
- A new notion of laziness — *bounded speculative laziness* — which is better suited to parallel machines than the traditional (sequential) notion and which can be used to improve the efficiency of the resulting program while still preserving the key characteristic of laziness, i.e., the ability to deal with potentially infinite data structures.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The major objectives of this thesis are to examine the implementation of lazy functional programming languages on fine-grain parallel (dataflow) architectures, to propose a novel and intuitively simpler approach to strictness analysis and the parallel implementation of lazy languages and to investigate the impact of laziness on parallelism, i.e., to examine whether sufficient parallelism can be obtained from lazy programs.

The following two sections present background information concerning lazy functional programming languages and fine-grain parallel architectures based on the dataflow model of computation. The next section presents some related work; a more detailed presentation of related work on strictness analysis is presented in Section. 3.4. An overview of the major contributions of the thesis is presented in Section 1.4. This is followed, in Section 1.5, by a general presentation of the organization of the thesis.

1.1 Lazy functional programming languages

Functional programming languages and parallel processing

Currently, the dominant programming paradigm is the *imperative* one. This paradigm is based essentially on an abstraction of the sequential uniprocessor machine. However, even for such simple machines, the field of programming and software development has been encountering numerous difficulties and problems — which has come to be known as the *software crisis*.

In order to deal intelligently with the different types of parallel machines which are being proposed and built — whose complexity are much greater due to the large number of processors and the associated synchronization/communication problems — it is becoming imperative (!) that other programming paradigms be explored. One possible alternative is functional programming [Bac78, Hud89].

Functional programming languages possess the important property of *referential transparency*, which implies that the value of an expression does not depend on the history of the computation. In turn, this brings another interesting property: the *Church-Rosser property* [Hud89]. This latter property is especially important, as it ensures that any order of evaluation of expressions within a functional program always produces the same result, including any order of parallel evaluation.

Parallelism can be extracted from functional programs based on the data dependencies induced by the programs. Such parallelism — which is said to be *implicit*, as it does not need to be explicitly indicated by the programmer, thus facilitating the programming task — can be detected rather easily and can generate much parallelism to be exploited on a multiprocessing machine [Nik90]. Functional languages can thus be good vehicles for expressing parallelism and making it available for exploitation by parallel machines.

Lazy vs. non-lazy functional programming languages

There are three major classes of functional programming languages, classes which can be distinguished based on the function invocation discipline they use:

- Strict evaluation: The arguments to function calls are evaluated *before* the function body is entered and executed. This has been called *call-by-value*.
- Lazy evaluation: The function body is executed and the arguments are evaluated *only if* and *when* they become needed. Moreover, once computed, the value of an argument is saved, so that it does not need to be recomputed. [1] This has been called *call-by-need*.
- Lenient evaluation: No *a priori* ordering is imposed on the evaluation of arguments vs. evaluation of the function body. The only requirement is that when an argument's value becomes needed within the function, execution of

[1] This can be done because of referential transparency.

the function body must be suspended until the argument's value becomes available [Tra88].

The major advantage of lazy evaluation over the other two approaches is that it ensures that no useless computation are ever performed. For example, when a strict discipline is used, there are programs which will not terminate but would have if lazy evaluation had been used. As for lenient evaluation, there are two different cases: (i) sequential implementation: the lenient approach suffers from the same problem as the strict approach, viz. some programs may not produce any results because of non-termination; (ii) parallel implementation: a lenient approach will always produce a result when the lazy approach can produce one [AN87]; however, the lenient approach may nevertheless trigger an unbounded amount of useless computation, thus wasting machine resources.

This characteristic of lazy evaluation (no useless computation) brings a number of other interesting properties:

- Lazy evaluation always preserves equational reasoning. For example, given f defined by $f\ x = 0$, then for any a, $(f\ a)$ can always be replaced by 0. This is not true for the other approaches, e.g., if a is the expressions $1/0$.
- Lazy evaluation can be seen as a new kind of *modularity glue*, i.e., as a new mechanism for supporting modularity [Hug84]. A particular example is that lazy evaluation makes it possible to deal with data structures which are *potentially infinite*.

This latter property is important in that it allows many problems to be solved using programs which are much simpler than if they had been expressed in a non-lazy language. A small example is presented below. This program computes the square root of a number using the Newton-Raphson algorithm. The program creates a (potentially) infinite sequence of approximations to the square root (function `mkApproximations`), sequence which is then (lazily) searched until the difference between two successive approximations satisfies some criteria specified by the function `within`. In this case, the modularity obtained from laziness leads to a clear separation between the generation of the approximations and the search through them; a different search criteria could easily be adopted simply by using a different `within` function.

```
Def nextApprox x a =
    (a + (x / a)) / 2;
Def mkApproximations x0 =
    { as = ((x0 + 1) / 2) : (map (nextApprox x0) as) In as };
Def within eps (a0 : a1 : rest) =
    If abs(a0-a1) <= eps Then a1 Else within eps (a1 : rest);
Def sqrt x eps =
    within eps (mkApproximations x);
```

Lazy evaluation and strictness analysis

The implementation of *lazy* evaluation requires a lot of overhead: a data structure has to be created to represent the expression to be delayed; when the expression is finally needed and evaluated, its associated value has to be remembered for subsequent uses, etc. [Hen80]. Fortunately, it is often possible to avoid this overhead by using a form of compile-time analysis called *strictness analysis*. The intuitive idea is the following: if it is possible to determine, at compile-time, that an argument is sure to be needed, then the overhead of lazy evaluation can be avoided and the argument can be passed by value, i.e., without being delayed.[2] This form of compile-time analysis, which comes under various forms as we will see in Section 3.4, is surely the most important factor allowing for the efficient implementation of lazy functional programming languages, as it can reduce significantly the overhead incurred by lazy evaluation.

1.2 Dataflow and parallel architectures

For the purpose of this thesis, we chose dataflow graphs as our framework to express parallelism [DK82, Den85]. Dataflow graphs only specify a *partial order* of execution for the instructions of programs [AI87]; thus, they do not impose any unnecessary sequentialization and can express parallelism down to the level of simple instructions. This is in contrast with imperative programs which specify a total order of execution.

Dataflow architectures are a possible means for executing programs represented by dataflow graphs [AC86a, Den85]. The major characteristics of such machines

[2] Formally, an argument can be strict even if it is not sure to be needed.

are that they can deal with fine-grain parallelism, and that their execution is based on the *data-driven* principle, i.e., the decision to fire an instruction is based only on the availability of its operands. This type of machine has been proposed to address the two fundamental issues of multiprocessing, i.e., memory latency and synchronization [AI87].

Nowadays, it is being recognized that laziness is essentially *a compilation problem* [Bur89a]. In this thesis, we are interested, through an appropriate compilation strategy, in implementing lazy languages — which are based on a high-level *demand-driven* model of execution — on dataflow machines — which are based on a *data-driven* model of execution. The objective is to execute lazy programs under the maximum parallelism, for which dataflow machines are well-suited. The compiler we will develop will have a lazy variant of (a subset of) the Id functional programming language [Nik88] as its source language and will be targeted to the TTDA dataflow machine [AN87]. Interestingly, although these two models seem, at first, contradictory, it will be seen (cf. Section 4.1) that it is possible for a data-driven architecture to provide some form of architectural support for the implementation of the demand-driven semantics required by laziness.

Another reason motivating the choice of dataflow machines as target for the compilation of lazy languages is that is has been shown that certain types of non-lazy functional programs, when executed on dataflow machines, can lead to such an abundance of parallelism that it has to be lowered down through some form of throttling mechanism [ACM88]. One of our initial goal was thus to investigate whether laziness could be used as a form of throttling; as we will soon see, this is effectively the case, but in such an extreme way that the opposite problem is encountered, i.e., too little parallelism is obtained.

In recent years, there has also been great interest toward *multi-threaded* architectures [CSS$^+$89, DG92, NPA92]. Although we do not study or use such architectures in this thesis, we conjecture that our work could easily be extended to such type of architectures, as dataflow graphs have been shown to be an appropriate intermediate form for generating code for such machines [Tra91, TCS92]. In fact, dataflow graphs can even be used for generating code for more classical architectures [BP89].

1.3 Related work

This section examines some of the previous work related to the parallel implementation of lazy functional languages, more precisely, work on parallel graph reduction, previous work on the implementation of laziness on dataflow architectures and work examining quantitative aspects of lazy functional languages implementation. Work related to the more specific problem of strictness analysis will be presented in Section 3.4.

1.3.1 Parallel graph reduction

In the last ten years, there has been a lot of work done on parallel graph reduction [AJ89, CP86, Geo89, GH87, HR87, HG85, Kel85, Pey87, WW87]. Most of the approaches to parallel graph reduction have assumed, as the underlying machine architecture, processing elements of the von Neumann type. In most cases, the parallel abstract machine models used for compilation were simply *extensions* of existing sequential abstract machines, extensions obtained by adding some primitives for parallel processing, e.g., *spark* [AJ89], *spawn* [Bur91b].

A positive aspect of those approaches is that they can build on top of all the previous work on sequential graph reduction [Joh83, FW87, BPR88] because they use the exact same framework, i.e., compilation of the source language into an abstract graph reduction machine. Furthermore, being based on an underlying sequential model of execution, they also lend themselves easily to implementation on von Neumann type of multi-processing machines.

Using a von Neumann type of architecture as the underlying machine model, however, inevitably leads to difficulties in dealing with the two fundamental issues of multiprocessing [AI87]. For example, a major limitation of the Alfalfa project [GH87, HG85, Gol88b] is that all synchronization has to be done through the use of a run-time *software* system, thus leading to poor efficiency in handling synchronization events. Other approaches completely sidestep the issue of dealing with long/unpredictable memory latencies by assuming a shared memory machine model [PCS87, PCS89, AJ89, Gol88a]. [3]

The approach we propose in this thesis will be radically different: it will start from a fully parallel model (dataflow graphs) and it will assume an underlying

[3] For a more detailed presentation of the different approaches to graph reduction, see [Tre89].

architecture that can efficiently handle synchronization events and tolerate long memory latencies (data-driven fine-grain parallel architecture).

1.3.2 Lazy evaluation and dataflow architectures

There are few papers discussing the implementation of lazy functional languages for dataflow machines. In fact, as far as we know, no complete implementation of lazy languages exists for such machines. Nonetheless, some interesting work on this problem has been done.

K. Pingali [Pin86] was the first to study the problem of demand-driven evaluation of a *stream* language on a dataflow architecture. His approach, like ours, implemented the lazy semantics while preserving the data-driven firing rules of dataflow.

The major limitations of Pingali's work were the following:

- User-defined functions were not handled.
- The exclusive use of demand signals, as proposed by Pingali, leads to the *implicit* creation of thunks. In turn, this can create memory management problem (*frame dragging* [Tre90]): the deallocation of a function frame can not be performed until all actors within this frame have fired. Thus, whole function frames may remain allocated simply because a single actor within the function has not yet fired (and may never fire if its result is not needed).
- Pingali's initial approach [Pin86] required the merging of (possibly complex) demand structures to be performed at run-time: this could become costly, e.g., when done within some loop.

  Pingali's later work on lazy evaluation eliminated the need for explicit merging of demand structures [Pin88]. However, one drawback of this approach is that it is not possible to represent demands for objects of arbitrary length without allocating the objects themselves. Thus, demands such as "*all tails are needed*" or "*all heads and all tails are needed*" can not be expressed, and neither can the notion of head-strictness. In addition, the frame dragging problem still remain unsolved.
- Pingali proposed only a limited form of optimization of the demand propagation code: it applied exclusively to the interconnection of stream operators containing *no tests or conditionals*. Moreover, his optimization could result in

a certain amount (although bounded) of unneeded computation, thus leading to an approach which was not purely lazy.

The above problems will be avoided, in our approach, by using a *delay-and-force* style of implementation [Hen80], i.e., explicit thunks, coupled with a compile-time analysis which will try to remove as many of these delays. In addition, the compile-time analysis, through the use of *cyclic demands*, will have a more expressive domain of demands; furthermore, the optimization approach will be more general as it will not be restricted to streams and will not ignore conditionals; moreover, it will be possible for it to be purely lazy, although a form of *bounded speculation* will also be possible.

Gaudiot and Wei [WG88] have discussed an approach to realize a demand-driven implementation of the FP language [Bac78] on a dataflow architecture. Their approach is based on the explicit introduction of demand propagation code followed by its subsequent optimization. The major drawbacks of their approach are the following:

- The expressiveness of their domain of demands is weak, so they can not express neither absence nor head strictness. Absence is interesting not only in itself — if a demand for a value is *sure* to be *absent*, i.e., the value is sure not to be needed, then its associated code can be eliminated — but also because it makes it possible to propagate some strictness information inside data structures, so that delayed expressions can contain subexpressions which are *not* delayed.

- Their approach requires all objects (values, functions and operators) to be transformed into higher-order functions. Because of this, it is hard to see how they could obtain an effective implementation on dataflow machines which would be efficient. (Their description of a possible implementation on dataflow machines is far from precise and complete.)

S. Heller, in his thesis [Hel89], presented a scheme for the efficient implementation of lazy data structures on the MIT tagged-token dataflow architecture [AN87]. The interesting aspect of his approach, which we intend to use, is the provision of a hardware mechanism to support the evaluation of suspensions. In his approach, a suspension is *implicitly forced* as soon as an attempt is made to fetch its value. The instruction requesting the fetched value is then suspended until the value becomes available.

The limitations of Heller's work are the following:

- It is restricted to the implementation of lazy data structures: data structure components can be delayed, but the same can not be done for the arguments of functions. This means it is not possible to enclose lazy data structure constructors within function definitions and still ensure that the lazy semantics will be preserved.
- It is based on the use of *programmer's annotations*: it is the programmer which has to decide which expressions have to be delayed and which should not. This may sometimes be a difficult and tricky task — as exemplified by some of the programs presented by Heller [Hel89] — especially since passing a lazy data structure as argument to a function will trigger its evaluation.

Contrary to Heller's approach, we will support laziness for the whole of the source language and without the need for any programmer's annotations.

1.3.3 Empirical evaluation of lazy functional programming languages implementation

In [HKL92], a parallel implementation of a purely lazy functional language is described. The identification of potential parallelism is done using the evaluation transformer model of [Bur87, Bur91b], model which can not deal with the notion of head-strictness (cf. Chapter 2 and Section 3.4.1). Because their target machine is an architecture without fine-grain parallelism (transputers system with distributed memory), a major question they need to examine is the effect of the size of the parallel tasks on the efficiency of the resulting program. Various strategies are examined and compared, the conclusion being that, for their particular architecture, only complete (super-)combinator applications should be evaluated in parallel. The full potential parallelism that could be obtained on a fine-grain parallel machine is not examined.

The work of [KPR+92] describes a parallel implementation of a purely lazy functional language on a shared memory multi-processing machine (Sequent Symmetry). The exploitable parallelism is identified using the analysis described in [SPR90], analysis which is based on a form of demand propagation, but restricted to only two types of demand: "evaluate to WHNF" (called *dd-analysis*) and "evaluate all heads and tails" (*ee-analysis*). Based on this analysis, multiple (at most 2!) versions of a function can be created while optimizing the intermediate code, before generation of the target (C) code.

In that paper, the speedups obtained from their analysis for various test programs on a parallel machine with a small number of processors (1 to 6) are examined. Their conclusion is that they can obtain almost linear speedup, i.e., their approach can identify all of the interesting parallelism. However, there is an important weakness in their conclusion: *none of their program really require lazy evaluation*! For example, they present the classical example of finding prime numbers using the sieve of Eratosthenes' approach. More precisely, they sieve through a finite interval of integers to identify the primes:

$$\mathit{sieve}\ (\mathit{fromTo}\ 1\ 1000)$$

This is quite different from the following, which identifies the first 1000 prime numbers:

$$\mathit{take}\ 1000\ (\mathit{sieve}\ (\mathit{from}\ 1))$$

In the first case, lazy evaluation is *not* required, as the program sieves through a *finite* list of numbers, whereas in the second case the sieve must go through a potentially infinite list of numbers in order to obtain the 1000 primes which are required. All of their examples are of this type, i.e., none of them really exploits the major characteristic of lazy evaluation, which is the possibility of dealing with potentially infinite object. This will not be the case with the experiments we present in Chapter 5.

In [FB93], an assessment of the evaluation transformer model of reduction [Bur87] on the performance of lazy programs targeted for the Spineless G-machine is presented. Although various parameters are examined, e.g., heap allocation, number of memory accesses, etc., parallelism *per se* is not examined. Note that one of their interesting conclusion is that "it is worth investigating more complicated patterns of evaluation for nested data structures" (p. 334); this will be made possible by our own approach, which will use a domain of cyclic demands with a better expressive power than evaluation transformers.

Finally, [HGW91] and [HL93] examine the impact of various compile-time analyses and implementation choices on the efficiency of a sequential implementation of a lazy functional language. Although a large number of factors are examined, the impact on parallelism is totally ignored because all the implementations which are compared are purely sequential.

1.3.4 Strictness analysis

The goal of strictness analysis it to determine, at compile-time, those arguments of functions which are sure to be strict, so that the simpler mechanism of call-by-value can be used instead of the call-by-need mechanism of lazy evaluation. This topic will be examined in more detail in Chap. 3, after our own approach has been presented.

1.4 Major contributions

The first major contribution of this thesis is to propose a novel approach to strictness analysis. This approach is based on the notion of *abstract demand propagation*, which will be seen to be a form of *backward* strictness analysis. Although it is still limited to the analysis of first-order functional programming languages, this approach can effectively deal with any kind of user-defined algebraic data types, which is not the case for existing forward strictness analysis approaches. Furthermore, this novel approach can also express the important notion of *head-strictness*, which is not the case with the forward approaches and most of the existing backward approaches. Finally, an additional interesting characteristic of the proposed approach is that its effective interpretation can formally be given by relating it to the standard semantics of the language, where it can be seen to be an *abstract* way of computing the *inverse* of functions within a properly defined abstract domain.

The next and equally important contribution of the thesis is to present an implementation of a first-order lazy functional programming language on a dataflow architecture. Although there exists a number of implementation of lazy functional programming languages on parallel machines, to our knowledge, this is the first implementation for a complete lazy language for a dataflow-like machine model. Although this implementation is based on the proposed abstract demand propagation approach, it also introduces a number of features not present in the basic model. Many of these features could easily be added to the basic model because of the particular framework in which it was set, i.e., dataflow graphs. The importance and influence of these various features on the precision of the strictness analysis process is also examined and conclusion about their usefulness is presented (cf. App. F).

Another important contribution of the thesis is its examination of the impact of lazy evaluation on the exploitable parallelism, i.e., on the *quantity* of parallelism which can be generated from functional programs with lazy semantics. It is generally agreed that the most promising use of functional languages is for programming parallel machines [Hud89, Nik90]. It has been shown that programs written in functional languages with an eager non-strict (*lenient*) semantics can provide an abundance of fine-grain parallelism [ACM88, Nik90]. However, nothing similar has been proven, or disproved, for lazy functional languages. Thus, a major contribution of the thesis is to give a first answer to the question: Can *lazy* functional languages provide sufficient parallelism to be exploited by fine-grain parallel machines? Now, anticipating on the results to be presented in Chap. 5, we can say that the answer will be negative; another contribution of thesis is thus the introduction of the notion of *speculative laziness* in an attempt to improve the parallelism and the efficiency of the resulting programs while still preserving the main characteristic of laziness, viz., the ability to deal with potentially infinite data structures.

Finally, another contribution, discussed in Chapter 2, is to show that the important notion of head-strictness can not be expressed by one of the main approach to strictness analysis, i.e., the framework of BHA-style forward strictness analysis [BHA85]. Although a similar result has been presented before [Kam92], our result, which has been derived independently, is easier to understand and provides a better intuition of how the main approach to forward strictness analysis works and why it fails to deal with head-strictness.

1.5 Synopsis

The thesis is organized as follows:

- Chapter 2 presents the important notion of *head-strictness* and compares it with other notions of strictness. It then examines why the major forward approach to strictness analysis can not express this concept.
- Chapter 3 presents our own framework for strictness analysis, which is based on the notion of *abstract demand propagation*. First, using a small language dealing only with scalar values, an exact model of demand propagation is presented; this model is shown to be *sound* by relating it with the standard semantics using the notion of *inverse*. Next, an abstraction of this exact

model, appropriate for a compile-time analysis, is presented; the soundness of this abstraction, i.e., its safety, is also examined. The source language is then extended to include data structures and both an exact and abstract model of demand propagation are presented, together with their appropriate soundness properties. The section ends with a brief survey of related work on strictness analysis.

- Chapter 4 presents the prototype compiler for a lazy variant of (a subset of) the Id language, targeted for the TTDA dataflow machine, that has been implemented using the concepts introduced in Chapter 3. It first presents an overview of the compiler substrate on top of which the lazy compiler was built, followed by brief descriptions of the major phases of the lazy compilation process. It also presents a number of additional features implemented by the compiler, e.g., abstract conditions, multiple function versions, run-time demands, etc., features which are not part of the model presented in Chap. 3.

- Chapter 5 examines the impact of laziness on the exploitable parallelism, i.e., whether sufficient parallelism can be extracted. First, the major factors influencing the precision of the strictness analysis process are briefly presented. Then, the implemented prototype compiler is compared with other approaches, e.g., an unoptimized lazy approach, an approach based on the evaluation transformer model of [Bur91b] and a lenient approach. Because little parallelism can effectively be obtained from lazy programes, a notion of *speculative laziness* is introduced, in a tentative to improve the parallelism while still preserving laziness.

- Finally, Chapter 6 reviews the major contributions of the thesis and briefly examines areas for further research.

Chapter 2

On head-strictness and the limits of forward strictness analysis

Strictness analysis is an important technique to obtain efficient implementation of lazy functional languages, both for sequential and parallel machines. There are two general strategies for performing strictness analysis:

- Forward strictness analysis [BHA85, Wad87, Bur89b]: Given a function and an *abstract* description of the input to the function, the idea is to compute, at compile-time, an abstract description (an *approximation*) of the output. By choosing an appropriate domain of computation, it is then possible to infer information about the strictness properties of the function.
- Backward strictness analysis [Hug85, Lin86, Hal87, HW87]: given a function and a description of the output required from the function, the idea is to compute a description of the input required to produce this output. By analyzing how the *demands* for the output are transformed into demands for the input, strictness properties of the function can be inferred.

The initial research effort on strictness analysis focused on the use of *forward abstract interpretation* [Myc80, BHA85, HY86, CP85, Wad87]. Although these approaches were powerful enough to deal with both first-order and higher-order functional languages, they have had less success with the analysis of data structures; so far, backward approaches have had more success with data structures [Hal87, HW87], although their treatment of higher-order functions has been weaker.

Strengths and weaknesses of various approaches to strictness analysis will be presented in more detail in Section 3.4, after our own approach has been introduced. The goal of the present chapter — which can be skipped on first reading without loss of continuity — is to examine one important property related with the strictness analysis of data structures — the property of *head-strictness*, introduced in [HW87] — and to formalize the following idea: the main approach to forward strictness analysis based on abstract interpretation [BHA85, Wad87, Bur89b] — the so-called BHA-style of abstract interpretation — can not express/recognize *head-strictness*, although it can be recognized by some of the existing backward approaches. Thus, it is a response to an open question in [Bur90a] where some relationships between the forward and backward approaches were examined, but where the question of whether or not the notion of head-strictness of [HW87] could be recognized using abstract interpretation was not answered.

Finally, note that since the research presented in the current section was initially done ([Tre91]), other researchers have proven similar or stronger results [Kam92, EM91]. For example, [Kam92] has proven the stronger result that head-strictness can not be expressed by any forward abstract interpretation framework using a monotonic abstraction function, the BHA-style being simply a special case of this. However, it is felt that the proof presented below is still interesting, as it presents a simpler, more intuitive explanation of strictness properties in terms of *révélateurs* (explaining more clearly how forward strictness analysis works) and showing why the general notion of head-strictness can not be expressed, although a (much) weaker form, introduced in [Bur90a], can effectively be expressed and analyzed.

2.1 What is a forward abstract interpretation?

In the following, we assume we are dealing with a first-order functional language with data structures. The abstract syntax of such a language is presented in Figure 2.1.

An abstract interpretation for this language is defined by the following [Bur91a]:

- For each base type A with associated domain D_A, an abstraction map is defined:

  $$abs_A \colon D_A \to D_A^{ab}$$

  This abstraction map is defined such that:

  - abs_A is strict and continuous;

$p \in Prog$	Programs where
	$Prog ::= \quad \{f_i\ x_1\ \cdots\ x_{n_i} = e_i\}$
$e \in Expr$	Expressions where
	$Expr ::= \quad c \mid x_i \mid p\ e_1\ \cdots\ e_n \mid f\ e_1\ \cdots\ e_n$
$c \in Const$	Constants, i.e., Booleans, Integers
$x \in VarName$	Variable names
$p \in Prim$	Primitive operations names, e.g., *if*, *cons*, *case*, $+$, $*$, etc.
$f \in FuncName$	Function names

Figure 2.1: Abstract syntax of the source language

  – D_A^{ab} is a finite complete lattice.

- For each primitive function p, an abstract version of the primitive (p^{ab}) is defined such that the following safety requirement holds:

$$\forall x_1\ \cdots\ x_n \cdot abs(p\ x_1\ \cdots\ x_n) \sqsubseteq p^{ab}\ abs(x_1) \cdots abs(x_n)$$

The abstract interpretation of an arbitrary expression is then obtained from the standard semantic by using, as the interpretation of each constant value or primitive, its associated abstract interpretation, i.e., $abs(c)$ or p^{ab}. [1]

The characteristic of finiteness of the abstract domain is crucial in the following development. In fact, our main result is that head-strictness can not be recognized using such an abstract domain unless the domain is *infinite*.

[1]An abstract interpretation is just a special case of an *interpretation*. An interpretation for a language can be defined as a pair $I = (\{D_A^I\}, \{c_\sigma^I\})$ where the D_A^I are the domains associated with the base types (one domain for each such base type) and c_σ^I is the interpretation of the constant c of type σ, i.e., $c_\sigma^I \in D_\sigma^I$. Each such interpretation naturally induces a semantic function:

$$E^I : Exp \rightarrow Env^I \rightarrow \bigcup D_\sigma^I$$

which is defined in the same manner as the standard semantics, but using the appropriate constants, i.e., those of the specified interpretation. This definition of an interpretation means that abstraction, application and fixpoint calculation are interpreted in the same way in all interpretations. Although it is also possible to parameterize these components of the interpretation, this is not necessary or useful in the case of strictness analysis.

2.2 How is strictness analysis done in an abstract interpretation framework?

The flat domain case

An interesting characteristic of the abstraction function is that it gives rise to an associated *concretization* function. This function associates, to each abstract element $\overline{s}$ from domain D_A^{ab}, a set of concrete elements (i.e., from $\mathcal{P}(D_A)$), more precisely, the set of values whose abstraction is at most as defined as $\overline{s}$ ($abs_A(x) \sqsubseteq \overline{s}$). Formally, such a concretization is defined as follows: [2]

$$conc_A : D_A^{ab} \rightarrow \mathcal{P}(D_A)$$
$$conc_A\, \overline{s} \;\triangleq\; \{x \mid abs_A(x) \sqsubseteq \overline{s}\}^{\circ}$$

The intuition behind the abstraction function and its associated (induced) concretization function is that an abstract value represents some *property* that we want to analyze/recognize. In other words, a property can be represented by the set of values which satisfy that property [AH87]. In the special case we examine — BHA style of strictness analysis — the important characteristic of these sets is that they are *Scott-closed*, that is, the following hold:

1. If $Y \subseteq X$ and Y is directed, then $\bigsqcup Y \in X$.
2. If $y \sqsubseteq x \in X$ then $y \in X$.

In strictness analysis, the general goal is to determine whether a given function is strict in a particular argument. Formally, we have the following definition:

Definition 2.1 $f\ x_1\ \cdots\ x_n = e$ is *strict* in x_i if the following holds:

$\forall e_1, \ldots, e_n \cdot$
$\quad f\ e_1\ \cdots\ e_{i\text{-}1}\ \bot\ e_{i+1}\ \cdots\ e_n = \bot$

In this case, the property of interest — *simple* strictness — is a property of functions. However, this property can also be represented as *the set of values* which can *reveal* that some arbitrary function is indeed strict. This leads to the following equivalent definition of strictness, where the set $\{\bot\}$ is the *révélateur* for simple strictness:

[2] X° denotes the least Scott-closed set containing X. The definition of Scott-closed subset is given in the next paragraph.

Definition 2.2 $f\ x_1\ \cdots\ x_n = e$ is strict in x_i if the following holds:

$\forall x \in \{\perp\} \cdot \forall e_1,\ \ldots,\ e_n \cdot$
$f\ e_1\ \cdots\ e_{i\text{-}1}\ x\ e_{i+1}\ \cdots\ e_n = \perp$

In other words, the set $\{\perp\}$ can be used to reveal the simple strictness of an arbitrary function. In fact, this is effectively the set which is used in the standard approach to strictness analysis on flat domains [Myc80, CP85], as it corresponds to the concretization set of the abstract value 0.

Let's illustrate this in the case of a flat domain. Our set of concrete values will be the integers together with $\perp$ using the standard flat ordering:

$D_{Int} = \{\perp, 0, 1, 2, \ldots, \text{-}1, \text{-}2, \text{-}3, \ldots\}$

Assuming that we want to perform strictness analysis of first-order functions on the domain of natural numbers, the abstract domain that would be used is the following:

$D^{ab}_{Int} = \{0, 1\}$ with the ordering $0 \sqsubseteq 1$.

The abstraction function abs_{Int} would be defined as follows:

$abs_{Int} : D_{Int} \rightarrow D^{ab}_{Int}$

$abs_{Int}(x) \quad \triangleq \quad \mathsf{if}\ x = \perp\ \mathsf{then}\ 0\ \mathsf{else}\ 1$

The concretization function induced by this abstraction function would then be the following:

$conc_{Int} : D^{ab}_{Int} \rightarrow \mathcal{P}(D_{Int})$

$conc_{Int}(\overline{s}) \quad \triangleq \quad \mathsf{if}\ \overline{s} = 0\ \mathsf{then}\ \{\perp\}\ \mathsf{else}\ \mathbb{Z}_\perp$

Finally, here are the abstractions associated with (some of) the primitives:

$\overline{+} = \lambda x_1, x_2 \cdot x_1 \sqcap x_2 \quad$ (* And similarly for the other strict binary primitives *)
$\overline{if} = \lambda x_0, x_1, x_2 \cdot x_0 \sqcap (x_1 \sqcup x_2)$

In order to determine whether a given function g is strict or not, an abstract version of it would be constructed (denoted by $\overline{g}$). For example, given

$$g\ x\ =\ \mathit{if}\ x == 0\ \mathit{then}\ 0\ \mathit{else}\ 2 * x$$

the associated abstract function would be

$$\overline{g}\, x \;=\; (x \sqcap 1) \sqcap (1 \sqcup (1 \sqcap x))$$

which could be simplified to

$$\overline{g}\, x \;=\; x$$

Then, it would be *safe* to say that g is strict only when $\overline{g}\,0 = 0$, which is effectively the case for this example. As can be seen, this is equivalent to the alternative formulation of strictness used in Definition 2.2, as the concretization set associated with 0 is effectively $\{\perp\}$.

The non-flat domain case

Generalizing the notion of strictness

The first point to note, when dealing with data structures, is that the basic notion of strictness is not sufficient. When dealing with scalars, there are only two levels of evaluation which are meaningful: fully evaluated vs. no evaluation done yet. On the other hand, a list can be evaluated to various degrees: no evaluation done yet, fully evaluated, all the tails evaluated, some heads and tails evaluated, etc.

Below are some useful generalizations of strictness for function dealing with lists. Given a function $f\ l = e$, where l is some linear list:[3]

- f is simply *strict* if $f\ \perp = \perp$;
- f is *tail-strict* if $f\ l = \perp$ whenever l is an infinite list or some approximation to such an infinite list (i.e., a list with a tail equal to $\perp$);
- f is *head-and-tail-strict* if $f\,l = \perp$ whenever l is an infinite list or some approximation to such an infinite list, or l is a list which contains some head equal to $\perp$;
- Otherwise, f is non-strict.

From an operational point of view, these definitions of strictness lead to the following possible optimizations:

[3]We assume the function takes a single argument, although the definitions generalize in a straightforward manner to multiple arguments.

- Simple strictness implies that it is safe to evaluate the outermost cell of a list, although its content — head and tail — must remain unevaluated. In other words, there is no need to create a thunk for the actual argument expression.
- Tail-strictness implies that a function can safely evaluate the tail of every cell it allocates. Thus, it means there is no need to create a thunk for any of the tails.
- Head-and-tail-strictness implies that the function can safely evaluate the head *and* the tail of every cell it allocates, i.e., there is no need to create any thunks and the list can be fully evaluated.

The associated *révélateurs*

In the flat domain case, the notion of *simple* strictness could be identified and characterized by defining a set of values R (the *révélateur*), such that f is strict when:

$\forall v \in R \cdot f\ v = \perp$

In the approach of [Wad87, Bur87], exactly the same idea is used when dealing with data structures. In order to explain this idea, we adopt the notation of [Bur89b], where sets of list values (the concretization sets, i.e., the *révélateurs*) are represented by *regular expressions*. For example, let $\mathbb{Z}$, $\mathbb{Z}_\perp$ and $\mathbb{Z}^0_\perp$ represent respectively the set of all integers, all the integers together with $\perp$, all the integers *except* 0 together with $\perp$. Then, $\mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^* . \mathsf{nil} \cup \mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^* . \perp \cup \mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^\omega$ would represent the set of all lists — finite (*) or infinite ($^\omega$) — whose first head *is not* 0. [4]

The domains which have been proposed for strictness analysis of lists using the abstract interpretation framework [Wad87, Bur87] lead to the following (Scott-closed) concretization sets:

- `WHNF` $= \{\perp\}$: Used to identify simple strictness;

[4]In more detail, here are the various terms of the regular expression:

- $\mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^* . \mathsf{nil}$: Any list whose first head is not 0, followed by a finite number of arbitrary elements, and terminated by nil, i.e., finite lists not starting with 0.
- $\mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^* . \perp$: Similar to the previous case except the last tail is $\perp$, i.e., partial lists not starting with 0.
- $\mathbb{Z}^0_\perp . {\mathbb{Z}_\perp}^\omega$: Any infinite list not starting with 0.

- $\mathtt{TS} = {\mathbb{Z}_\perp}^* . \perp \cup {\mathbb{Z}_\perp}^\omega$: Used to identify tail-strictness, which can be revealed by any non-finite list (i.e., partial or infinite list);
- $\mathtt{HTS} = \mathtt{TS} \cup \mathbb{Z}^* . \perp . {\mathbb{Z}_\perp}^* . \mathsf{nil}$: Used to identify head-and-tail-strictness, which can be revealed by any non-finite list (TS) or any finite list containing some undefined head.

Thus, in order to determine whether an arbitrary function f is $\mathcal{R}$-strict ($\mathcal{R}$ = *Simple, Tail, Head-and-tail*), we have to determine whether the following holds, where R is the associated *révélateur* (WHNF, TS, HTS):

$\forall x \in R \cdot f\ x = \perp$

2.3 What is head-strictness?

The notion of head-strictness introduced in [HW87] is different from the notions of tail- and head-and-tail-strictness. Tail-strictness means that, whenever a cell is allocated, it is safe to evaluate its tail. Clearly, the implication is then that *all tails* need to be evaluated, as evaluating the next tail implies allocating the next cell and evaluating its tail, etc. The same holds for head-and-tail-strictness, which implies that whenever a cell is allocated, it is safe to evaluate both its head and its tail.

On the other hand, head-strictness is different. Analogously to the previous cases, it means that whenever a cell is allocated, it is safe to evaluate its head, implying that no thunk needs to be built for any of the heads, thus saving one thunk per cell. However, it does not mean that *all* heads need to be evaluated: evaluating all heads would also require all tails to be evaluated, in which case we would simply have head-and-tail-strictness! [5]

It can easily be seen that this notion of head-strictness is related to the process that occurs when a data structure is being printed by a *sequential* interpreter, e.g., a list: in order to sequentially print the list, the first head must be fetched; when it is finally available, it can then be printed; then the next cell can be obtained, by evaluating the tail of the current cell, and the head of that new cell can be evaluated and printed, etc.

[5] Alternatively, head-strictness can be intuitively characterized by the fact that whenever a function has to examine any prefix of its argument, then all the elements of that prefix must also be evaluated [Kam92].

Let's examine another example of a head-strict function. Suppose we have the following function:

$$\begin{aligned}
has_0\ l \ \triangleq\ & \mathsf{cases}\ l\ \mathsf{of} \\
& \mathsf{nil} \qquad\ \to \mathsf{false} \\
& (x{:}\ xs) \to \mathsf{if}\ x = 0\ \ \mathsf{then\ true\ \ else}\ has_0\ xs \\
& \mathsf{end}
\end{aligned}$$

This function determines whether a list contains the value 0 as one of its elements. Clearly, until a 0 is found, the heads all have to be examined. However, as soon as one is found, then the function can terminate without examining the remaining heads. So, if a list l only contains a $\perp$ element *after* the appearance of the first 0, then $(has_0\ l)$ *will not be* $\perp$.

It is there that head-strictness differs from tail- or head-and-tail- strictness: in the latter two cases, the concrete set of values that can be used to *reveal* the associated property is totally independent of the exact content of the list; on the other hand, in the head-strict case, there exists a dependence on the elements: a specific head will or will not be evaluated depending on the value of the previously encountered heads.

2.4 Head-strictness can not be recognized by the BHA-style of strictness analysis

The concrete set of values for the head-strictness property of has_0

Suppose that the property of head-strictness of the function has_0 could be identified in the same manner as tail- and head-and-tail-strictness, i.e., with some appropriate Scott-closed set. This would mean that, in the abstract domain, there would exist an abstract value α_0 for which the associated concretization set — the *révélateur* — would be some set $\mathtt{HS}_0$ such that the following holds:

$\forall v \in \mathtt{HS}_0 \cdot has_0\ v = \perp$

Definition 2.3 *Let* $\mathtt{HS}_0$ *be the following set:*

$$\mathtt{HS}_0 = {\mathbb{Z}_\perp^0}^* \,.\, \perp \cup {\mathbb{Z}_\perp^0}^\omega \cup {\mathbb{Z}_\perp^0}^* \,.\, \perp \,.\, {\mathbb{Z}_\perp}^* \,.\, \perp \cup {\mathbb{Z}_\perp^0}^* \,.\, \perp \,.\, {\mathbb{Z}_\perp}^\omega \cup {\mathbb{Z}_\perp^0}^* \,.\, \perp \,.\, {\mathbb{Z}_\perp}^* \,.\, \mathsf{nil}$$

The set $\mathtt{HS}_0$ is composed of all the non-finite lists which do not contain a 0, together with all the lists (partial, infinite or finite) which, if they contain a 0, also contain a head which is $\perp$ before the 0 is encountered. The first useful fact concerning this set is the following, which indicates that $\mathtt{HS}_0$ is effectively an appropriate Scott-closed set:

Fact 2.1 $\mathtt{HS}_0$ is a Scott-closed set.

Proof: The proof is quite straightforward from the definition and will be omitted. In fact, $\mathtt{HS}_0$ can easily be shown to be a c.p.o. □

The major lemma is Lemma 2.2, which claims that $\mathtt{HS}_0$ is effectively the appropriate *révélateur* of head-strictness for the function has_0, i.e., any value in that set is sure to lead has_0 into non-termination:

Lemma 2.2 $x \in \mathtt{HS}_0 \;\Leftrightarrow\; has_0\, x = \perp$

Proof: See App. H.1, p. 23. □

The concrete set of values for the head-strictness property of has_k

Let's now examine the following function, similar to has_0, except that it determines whether a list contains a 1 as one of its element:

$$
\begin{array}{lll}
has_1\, l & \triangleq & \mathsf{cases}\ l\ \mathsf{of} \\
 & & \mathsf{nil} \quad\quad \to \mathsf{false} \\
 & & (x\!: xs) \to \mathsf{if}\ x = 1\ \ \mathsf{then\ true\ \ else}\ has_1\, xs \\
 & & \mathsf{end}
\end{array}
$$

The associated set of concrete values required to characterize the head-strictness property of has_1 would be the following, which is identical to $\mathtt{HS}_0$ except for 0 which has been changed to 1:

$$\mathtt{HS}_1 = \mathbb{Z}_\perp^{1\,*} . \perp \cup \mathbb{Z}_\perp^{1\,\omega} \cup \mathbb{Z}_\perp^{1\,*} . \perp . \mathbb{Z}_\perp{}^* . \perp \cup \mathbb{Z}_\perp^{1\,*} . \perp . \mathbb{Z}_\perp{}^\omega \cup \mathbb{Z}_\perp^{1\,*} . \perp . \mathbb{Z}_\perp{}^* . \mathsf{nil}$$

If we want to be able to recognize the head-strictness of has_1, then our abstract domain must include an abstract value corresponding to the set $\mathtt{HS}_1$. That is, the abstract domain would have to include an abstract value α_1 such that its associated concretization set would be $\mathtt{HS}_1$, i.e., such that the following holds:

$\mathtt{HS}_1 = conc(\alpha_1)$

Similarly, for each function has_k with $k \in \mathbb{Z}$, to recognize the head-strictness of has_k, we would need an abstract value α_k such that $\mathtt{HS}_k = conc(\alpha_k)$, with $\mathtt{HS}_k$ similar to $\mathtt{HS}_0$ but with k used in place of 0.

The abstract domain must be infinite!

We now show that, in order to recognize the head-strictness of all the functions has_k, the abstract domain would have to include an infinite number of abstract values. First, we clearly have that the révélateurs for the different has_k functions must be distinct.

Lemma 2.3 $\forall i, j \in \mathbb{Z} \cdot i \neq j \;\Rightarrow\; \mathtt{HS}_i \neq \mathtt{HS}_j$

Proof: See App. H.1, p. 274. □

The next step is to show that whenever two abstract values have different concretization sets, then they represent different abstract values.

Lemma 2.4 *For any two abstract values a and b:*

$conc(a) \neq conc(b) \;\Rightarrow\; a \neq b$

Proof: Straightforward, from the definition of *conc*, using the contraposite. See App. H.1, p. 274. □

From these two facts, we then have the following lemma:

Lemma 2.5 *To recognize the head-strictness property of all of the functions* has_k *($k \in \mathbb{Z}$) using the BHA-Style of strictness analysis, the abstract domain must contain an infinite number of distinct abstract values (α_k with $k \in \mathbb{Z}$).*

Proof: Straightforward, from Lemmas 2.3 and 2.4. □

2.5 A weaker notion of head-strictness

In [Bur90a], a weaker notion of head-strictness is introduced and an abstract interpretation powerful enough to analyze this new notion is presented. As we have seen earlier, the general notion of head-strictness expresses the idea that, in the case of lists, whenever a cell is going to be needed, then its head is also going to be needed. On the other hand, the weaker notion of head-strictness presented in [Bur90a] simply expresses the idea that, whenever the outermost level of the list is going to be needed, then the first head will also be needed. This property, which is not as powerful as the general notion of head-strictness, can be identified using the following *révélateur*, which simply represents the set of all lists whose first head *is* $\perp$:

$$\mathtt{HS} = \perp . \mathbb{Z}_\perp{}^* . \mathsf{nil} \cup \perp . \mathbb{Z}_\perp{}^* . \perp \cup \perp . \mathbb{Z}_\perp{}^\omega$$

Interestingly, the following equivalence can easily be seen to hold:

Fact 2.6 $\mathtt{HS} = \bigcap_{j \in \mathbb{Z}} \mathtt{HS}_j$

In other words, this weaker form of head-strictness can be seen as the intersection of all the more general head-strictness properties introduced earlier.

2.6 Conclusion

This chapter has examined the notion of head-strictness introduced in [HW87] and has explained why it is different from the other notions of strictness used in approaches based on forward abstract interpretation [Wad87, BHA85, Bur87]. Within this latter framework — forward abstract interpretation where abstract values correspond to Scott-closed subset of values — we have shown that it is not possible to identify head-strictness of arbitrary functions *unless* the abstract domain contains an infinite number of elements. In other words, using the *usual* approach to strictness analysis based on finite domains of Scott-closed subset of values, then given an arbitrary function f, the strictness analyzer will not be able to determine whether f is head-strict or not.

As we will see in Section F.6, recognizing head-strictness can be important to obtain efficient implementation of lazy languages, especially when used to express an initial head-strict (printing) demand. For example, in the case of linear lists, this can intuitively be seen by the fact that head-strictness allows to save one thunk *per* cell, i.e., per list element. The approach we present in the next chapter, which is a backward approach, will be able to recognize head-strictness.

Of course, what we have shown does not imply that head-strictness can not be expressed by any forward abstract interpretation. For example, forward abstract interpretations are proposed in [Hun90, Kam92] that can identify head-strictness. In these approaches, however, abstract values do not correspond anymore to closed subset of values but instead are defined in terms of *partial equivalence relations*. The resulting interpretations seem much more difficult to understand. Furthermore, the abstraction functions are not monotonic anymore, which may clearly create problems when dealing with fixed point computations. In fact, it has even been conjectured that although these new approaches can *express* head-strictness, they will not be able to effectively discover it [HL91]. Thus, the usefulness of these new forward approaches still remains to be seen.

Chapter 3

Abstract demand propagation

This chapter presents the approach to strictness analysis which forms the heart of our compiler and which is based on the notion of *abstract demand propagation*.

Section 3.1 begin by presenting the intuition behind our backward approach to strictness analysis, which can be seen as a form of *approximate inverse computation*. Then, in the following sections, we formalize these various concepts, first for scalars, then for data structures.

In Section 3.2, the major concepts of our approach are first illustrated and formalized using a small functional language dealing only with scalars. First, we introduce a notion of *demand* and present a *non-standard* denotational semantics which, we claim, represents the notion of *exact demand propagation*. Two (2) types of demands will be introduced: *positive* vs. *negative* demands; intuitively, these will represent, respectively, the fact that a certain subexpression is *sure to be needed* vs. is *sure* not *to be needed*. The *soundness* of this non-standard semantics is then examined by relating it with the standard semantics, showing that the non-standard semantics represents a certain form of *inverse* of the standard semantics. Next, an *abstraction* of the non-standard semantics for demand propagation is presented and a notion of *safety* is introduced, relating the exact non-standard semantics with the abstract one. [1] Combining these two relationships then allows us to present a

[1]Note that this relationship between the abstract and exact non-standard semantics (safety) can also be viewed as a form of soundness, i.e., soundness of the abstract semantics relative to the exact one. In the present chapter, in order to avoid any confusion, the term *safety* (which is often used in the terminology of abstract interpretation [AH87, Hug89]) will be used to refer to this relationship between the abstract and exact non-standard semantics, whereas *soundness* will always be used to refer to the relationships between the non-standard semantics and the standard one.

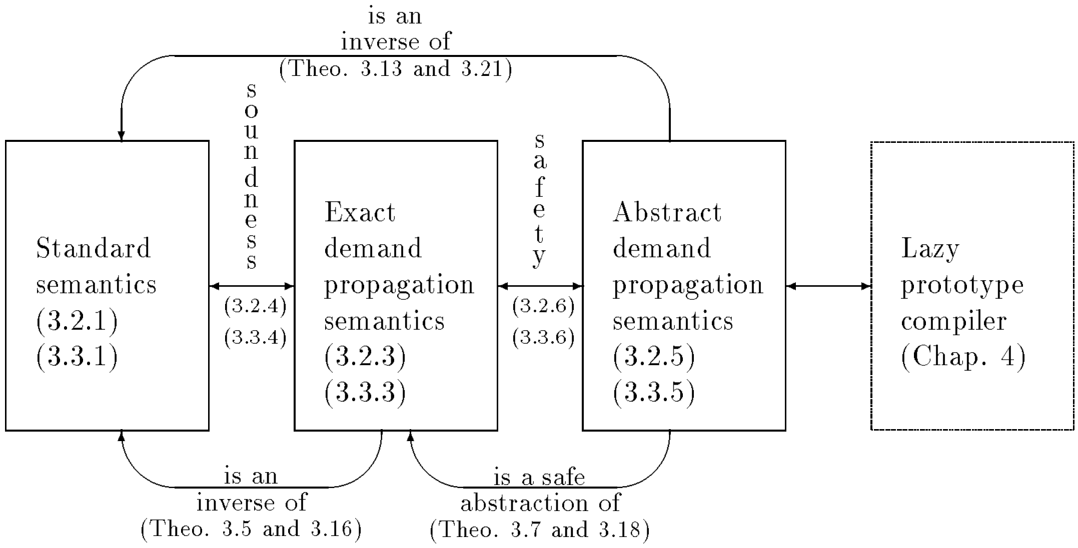


Figure 3.1: Relationships between the various semantics and the compiler

number of interesting properties, namely, that a positive demand effectively reflects strictness, a negative demand reflects non-neededness and that the *bottom* element of the abstract domain reflects non-termination.

An overview of the relationships between the various semantics to be defined and the compiler we have implemented is presented in Figure 3.1 [2] The major theorems to be presented, i.e., those formally establishing the relationships between the various semantics, are also indicated: the first theorem, in the case of scalars (resp. data structures), is Theorem 3.5 (resp. 3.16) which formally establishes that the exact demand propagation semantics is an *inverse* of the standard semantics; as for Theorem 3.7 (resp. 3.18), it formally establishes that the abstract demand propagation semantics is a safe approximation of the exact demand propagation; finally, Theorem 3.13 (resp. 3.21) combines these two facts to show that the abstract demand propagation can also be seen as an appropriate inverse of the standard semantics.

Section 3.3 presents the same ideas as in Section 3.2 but this time for a language allowing data structures (user-defined algebraic types). The appropriate exact and abstract semantics are introduced and the soundness and safety properties are examined. The termination property of a compiler using the proposed abstract domain of demands for data structures is also examined.

[2] The numbers between parentheses refer to section numbers: the first number is for the language subset dealing only with scalars whereas the second one is for the whole language dealing with data structures.

Finally, Section 3.4 discusses related work, i.e., briefly presents a number of alternative approaches to the problem of strictness analysis.

3.1 The intuition behind abstract demand propagation

In this section, we present an alternative way of looking at the notion of strictness and show how it can be related to the notion of inverse computation, notion which is then related to the concepts of demand and demand propagation.

3.1.1 Strictness: A different way to look at it

The classical definition of strictness is the following, assuming for simplicity that we have a function of a single argument:

Definition 3.1
f is strict $\triangleq\ x = \bot \ \Rightarrow\ f\ x = \bot$

This definition expresses the idea that a function is strict in its argument if and only if, when given an undefined value as argument, the result is sure to be undefined. This definition of strictness naturally leads to a *forward* view of the process of strictness analysis, as exemplified by the work on strictness analysis and abstract interpretation [CP85, BHA85, Hug85]: to determine, at compile-time, whether a function f is strict, an abstract value representing $\bot$ is fed to (an abstract representation of) f; if the result which is obtained is equal to the abstract value for $\bot$, then f is determined to be strict.

Alternatively, by using the contrapositive of the implication, the definition of strictness can be expressed as follows:

Definition 3.2
f is strict $\triangleq\ f\ x \neq \bot \ \Rightarrow\ x \neq \bot$

When f is a function over a flat domain such as the integers $\mathbb{Z}_\bot$, so that $x \neq \bot$ implies $x \in \mathbb{Z}$, we thus have the following equivalent definition of strictness:

Definition 3.3
f is strict $\triangleq\ f\ x \in \mathbb{Z} \ \Rightarrow\ x \in \mathbb{Z}$

In this case, we are expressing the idea that a function is strict in its argument if, in order to obtain a fully defined result, we have to feed in a fully defined argument. This leads to an alternative way of viewing strictness analysis as a *backward* process [Hal87, HW87, Dyb88]: to decide if a function f is strict, determine whether (an abstract representation of) f, in order to produce a fully defined result, must be fed with inputs which are fully defined.

It can easily be seen that the previous definition is equivalent to the following one:

Definition 3.4

f is strict $\triangleq \{x \mid f\ x \in \mathbb{Z}\}\ \subseteq\ \mathbb{Z}$

Here is a small example. Suppose we have the following function definition:

```
Def fact x = if x == 0 then 1 else x * (fact (x-1));
```

In this case, we effectively have that `fact` is a strict function:

$\{x \mid \mathit{fact}\ x \in \mathbb{Z}\} = \mathbb{N} \subseteq \mathbb{Z}$

The interest of Definition 3.4 is that it can easily be modified to express properties other than simple strictness. For example, the fact that a function is *sure not to terminate* can be expressed as follows:

Definition 3.5

f is sure not to terminate $\triangleq \{x \mid f\ x \in \mathbb{Z}\}\ =\ \{\,\}$

In words, this says that a function is sure not to terminate if there is no element which can ensure that the result will be fully defined. Here is a simple example: [3]

```
Def loop x = loop (x+1);
```

Here, we can conclude that `loop` is sure not to terminate; but note that we could also (correctly) conclude that loop is strict:

$\{x \mid \mathit{loop}\ x \in \mathbb{Z}\} = \{\,\} \subseteq \mathbb{Z}$

[3] The property of a function not needing its argument, to be introduced in the next section, can not be expressed using only the above notions. At first, it may seem that $\{\perp\} \subseteq \{x \mid f\ x \in \mathbb{Z}\}$ means f does not need its argument. This is not the case, as the `loop` example shows: $\{x \mid \mathit{loop}\ x \in \mathbb{Z}\} = \{\,\} \not\supseteq \{\perp\}$. But, clearly, `loop` does not need its argument!

3.1.2 Strictness, other properties and the notion of inverse

We have just seen how, intuitively, the process of strictness analysis can be performed in a backward manner. Let's now examine how these ideas can be related to the notion of *inverse* by defining the following predicate:

Definition 3.6

$$\textit{is-inverse}_1(f, R, A) \quad \triangleq \quad \forall r \in R \cdot f\ x = r \ \Rightarrow \ x \in A$$

This predicate expresses the following idea: a set A is an inverse of f relative to R when for any $r \in R$, if x can be used as an argument to f to obtain r, then A contains x.

From the examples presented earlier, we have:

- $\textit{is-inverse}_1(\textit{fact}, \mathbb{Z}, \mathbb{N})$
- $\textit{is-inverse}_1(\textit{loop}, \mathbb{Z}, \{\,\})$

We are now going to modify the definition of $\textit{is-inverse}_1$ to allow the specification of additional properties expressing what could be called *generalized* forms of strictness. First, it can be seen that Defn. 3.6 is equivalent to the following one:

Definition 3.7

$$\textit{is-inverse}_2(f, R, A) \quad \triangleq \quad \forall r \in R \cdot f\ x = r \ \Rightarrow \ \exists a \in A \cdot a = x \ \wedge \ f\ a = r$$

Defn. 3.7 can now be modified, to make it more general, based on the following motivations: *i*) given a result $r \in R$ to be obtained, it is quite acceptable if the result we obtain is *more defined* than the required one; *ii*) given r obtained by applying f to x, it is also fine if that same result can be obtained using inputs which are *less defined* than x. In other words, we modify the definition so that A is an inverse for R relative to f if results at least as defined as those in R can be obtained using inputs from A which are as little defined as possible.

Definition 3.8

$$\textit{is-inverse}(f, R, A) \quad \triangleq \quad \forall r \in R \cdot f\ x \sqsupseteq r \ \Rightarrow \ \exists a \in A \cdot a \sqsubseteq x \ \wedge \ f\ a \sqsupseteq r$$

Using this new *is-inverse* predicate it is now possible to express additional properties, e.g., the fact that a function is sure *not to need its argument*:

Definition 3.9

f is sure *not* to need its argument $\triangleq$ $\textit{is-inverse}(f, \mathbb{Z}, \{\perp\})$

In other words, a function is sure not to need its argument if, for any fully defined result r that can be obtained by applying f to some x, it is possible to obtain this same result r simply by feeding $\perp$ as the argument in place of x, i.e., the value of x is irrelevant. Here is a small example, the constant function `zero`:

```
Def zero x = 0;
```

In this case, we clearly do not have that `zero` is strict, as $\{x \mid f\ x \in \mathbb{Z}\} = \mathbb{Z}_\perp$ and $\mathbb{Z}_\perp \sqsubseteq \mathbb{Z}$ does not hold. However, the following effectively holds: $\textit{is-inverse}(\textit{zero}, \mathbb{Z}, \{\perp\})$. For any argument x, the value of f is always 0. Given any x, we always have $\perp \sqsubseteq x$; furthermore, we also have $f \perp = 0$!

Using the *is-inverse* predicate, the three forms of *generalized* strictness properties that have been introduced so far can be expressed as follows:

$$\textit{is-inverse}(f, \mathbb{Z}, A) \wedge A \subseteq \mathbb{Z} \quad \Rightarrow \quad f \text{ is strict} \tag{3.1}$$

$$\textit{is-inverse}(f, \mathbb{Z}, \{\,\}) \quad \Rightarrow \quad f \text{ is sure to loop} \tag{3.2}$$

$$\textit{is-inverse}(f, \mathbb{Z}, \{\perp\}) \quad \Rightarrow \quad f \text{ is sure not to need its argument} \tag{3.3}$$

There is an important property which must be mentioned, property which will be vital when we discuss safety. This property simply indicates that any superset of an inverse is also an inverse, which is quite obvious from the definition of *is-inverse* (Defn. 3.8, p. 31):

Fact 3.1

$\textit{is-inverse}(f, R, A) \wedge A \subseteq A' \;\Rightarrow\; \textit{is-inverse}(f, R, A')$

From this property, we have the following implications:

- $\textit{is-inverse}(f, \mathbb{Z}, \{\,\}) \;\Rightarrow\; \textit{is-inverse}(f, \mathbb{Z}, \{\perp\})$: If f is sure to loop, then surely f does not need its argument.
- $\textit{is-inverse}(f, \mathbb{Z}, \{\,\}) \;\Rightarrow\; \textit{is-inverse}(f, \mathbb{Z}, \mathbb{Z})$: If f is sure to loop, then f is strict.

Additional properties can also be expressed:

- $\textit{is-inverse}(f, \{\bot\}, \{\bot\})$: If any result is acceptable, then any argument can be fed in.
- $\textit{is-inverse}(f, \{\,\}, \{\,\})$: If no result is acceptable, then no argument is needed.

3.1.3 Approximate inverse computation and safety

As we just saw, it is possible to express strictness and other similar properties of a function in terms of an appropriate form of inverse of that function. Thus, this leads to the idea that performing (generalized) strictness analysis could be done by computing the *inverse* of functions and checking whether the computed inverse satisfies some appropriate predicate. But what does *Computing the inverse* mean? From Defn. 3.8, it can be seen that it essentially involves determining *the least amount of information* needed to produce *at least the demanded result*; that is, given some result r we want to obtain, we must determine the minimal amount of information required to produce a result at least as defined as r. So, let's formally define the inverse f^{-1} of a function f as follows: [4]

Definition 3.10

$$f^{-1}(R) \;\triangleq\; \iota\, A \cdot \textit{is-inverse}(f, R, A) \land (\forall A' \cdot \textit{is-inverse}(f, R, A') \;\Rightarrow\; A \subseteq A')$$

The reason we would like to obtain the minimal set is to allow us to infer strictness properties which are as precise as possible (cf. the implications presented in Eqns. 3.1, 3.2 and 3.3). Of course, computing the exact inverse of a function as specified in Defn. 3.10 is not possible; if it was, then the halting problem could be solved: given some function f, it would suffice to check if $f^{-1}(\mathbb{Z}) = \{\,\}$! Therefore, a compiler based on the idea of inverse computation would only compute some (safe) approximation of the exact result.

The key to a safe abstraction can be obtained from Fact 3.1: any superset of the exact inverse is safe, because it is still an appropriate inverse, as defined by *is-inverse* (Defn. 3.8). This can be re-expressed as the following fact:

Fact 3.2

$$f^{-1}(R) \subseteq A \;\Rightarrow\; \textit{is-inverse}(f, R, A)$$

[4] In the VDM notation, "$\iota\, x \cdot p(x)$" is the *iota-function* which returns the unique value satisfying the predicate p.

So, suppose we could define a function *inv*, to be implemented by some compiler, having the following property:

$$\forall R \cdot f^{-1}(R) \subseteq inv(f, R) \tag{3.4}$$

Then, given Fact 3.2 and Equations 3.1, 3.2 and 3.3, the following inferences would be valid:

$$\begin{aligned} inv(f, \mathbb{Z}) \subseteq \mathbb{Z} \quad &\Rightarrow \quad f \text{ is strict} &(3.5)\\ inv(f, \mathbb{Z}) = \{\,\} \quad &\Rightarrow \quad f \text{ is sure to loop} &(3.6)\\ inv(f, \mathbb{Z}) = \{\perp\} \quad &\Rightarrow \quad f \text{ is sure not to need its argument} &(3.7) \end{aligned}$$

In other words, this means that as long as we use approximations which produce a *superset* of the exact answer, then no unsafe optimizations will be performed. This can be explained by the fact that:

- For simple strictness: If we use a superset which contains $\perp$, then we will not conclude that the function is strict, thus no optimization will be done.
- When deciding whether a function is sure to loop, if the superset is *not* the empty set, then we will not conclude that the function is sure to loop.
- When deciding whether a function is sure not to need its argument, if the superset is *not* the singleton set $\{\perp\}$, then no optimization will be done either.

However, there are two cases where using a superset might change the exact inference that would be made, both cases being when a superset of the empty set would be used. However, in both cases, the inference that would be done would still be valid:

- The superset is exactly $\{\perp\}$: clearly, if the function is sure not to terminate, it is quite safe to infer that its argument will not be used.
- The superset does not contain $\perp$, from which we would conclude that the function is strict. From the definition of strictness, this is also perfectly safe, as a non-terminating function is indeed strict.

3.1.4 Approximate inverse computation as abstract demand propagation

The notion of inverse computation as a form of generalized strictness analysis is not totally new as it has been explored before, albeit in a simplified form, in [Dyb88].

Another notion which is not new is the notion of *demand* and the related notion of *demand propagation*; it has been used before either as a run-time concept — e.g., demand-driven evaluation [DK82, TBH82, Pin86] — or as a compile-time concept — e.g., backward form of strictness analysis [Hal87, Lin88, LM91]. What has never been done before is to formally relate these two notions, i.e., inverse computation and demand propagation; this will be done in the following sections using the notion of *abstract demand propagation*, notion which will formally be defined through various forms of non-standard denotational semantics.

Essentially, abstract demand propagation will realize a function *inv*, as introduced in Equation 3.4; from Fact 3.2, such a function, given some function f to analyze, will then produce a safe approximation of the inverse of f, thus allowing us to use the inferences described in Equations 3.5, 3.6 and 3.7 (p. 34).

This *inv* function will be defined using the notion of demands and demand propagation. Informally, a *demand* will represent a set of possible/acceptable results expected from a function (the sets R and A). As for the process of *demand propagation*, it will represent an *approximation* of the process of *inverse computation* (function *inv*), i.e., trying to determine which arguments are required (A) in order to produce the expected/required result (R).

This new way of viewing demand propagation will allow us to formally specify what demands represent [5] and will explain why our domain of demands is different from previous ones. This will also allow us to have a simple interpretation of the notion of safety and, furthermore, will motivate the claim that the demand propagation semantics to be defined *is not science-fiction* (using an appropriate soundness theorem).

3.2 Abstract demand propagation: Scalars

In this section, we introduce the notions of *exact* vs. *abstract* demand propagation in the context of a simple functional language dealing only with scalars. These notions are defined using *non-standard* denotational semantics definitions. These semantics, together with the standard semantics, will be related through the notions of *soundness* and *safety*, notions which will be explained in terms of inverse

[5] In earlier approaches using demand propagation, it was not clear what demands represented. In one case, this lead to difficulties when analyzing recursive function definitions and computing least fixed point. For more detail, cf. Section 3.4.2 p. 100.

$p \in Prog$	Programs where	
	$p ::=$	$\{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$
$e, e_i \in Expr$	Expressions where	
	$e ::=$	$c \mid x_i \mid e_1 + e_2 \mid$
		$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid f_i(e_1, \ldots, e_{n_i})$
$c \in Const$	Constants, i.e., integers	
$x_i \in VarName$	Variable names	
$f, f_i \in FuncName$	Function names	

Figure 3.2: Abstract syntax of the source language (scalars only)

computation.

3.2.1 The source language and its standard semantics

The source language

The source language is a simple functional language without higher-order functions and without data structures. Its abstract syntax is presented in Figure 3.2. We assume that we are dealing with lambda-lifted function definitions without free variables, i.e., combinators [Joh87]; each variable x_j can thus be uniquely identified, within the function f_i in which it is used, by an integer j comprised between 1 and n_i, where n_i represents the number of arguments of the function f_i. Also, note that $+$ really stands for any strict primitive, e.g., $-$, $/$, $*$, $=$, $\neq$, etc., as these would all be handled in the same manner. [6] We also assume that, as in the C programming language [KR78], integers play the role of booleans, where 0 stands for false and any non-0 value stands for true.

The standard semantics

The standard semantics is presented in Figure 3.3. In the following, *bve*/*Bve* stand for *Bound Variables Environment* whereas *fve*/*Fve* stand for *Function Variables Environment*. Note that because we are dealing with lambda-lifted first-order functions without free variables, bound variables environments can simply be represented as

[6]Non-strict primitives, e.g., `and`, `or`, are not predefined; these can be defined with user-defined functions containing *if* expressions.

$$Value = \mathbb{Z}_\perp$$

$$Bve = Value^*$$

$$Fve = FuncName \xrightarrow{m} Value^* \rightarrow Value$$

$$\mathcal{M} : Prog \rightarrow Fve$$

$$\mathcal{M}[\![p]\!] \;\triangleq\; \boldsymbol{\mu}\, fve \cdot \{f_i \mapsto \lambda(z_1, \ldots, z_{n_i}) \cdot \mathcal{E}[\![e_i]\!][z_j \mid 1 \leq j \leq n_i]\, fve\}$$
$$\textsf{where } p \;=\; \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$$

$$\mathcal{E} : Expr \rightarrow Bve \rightarrow Fve \rightarrow Value$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{E}[\![e]\!]\; bve\; fve \;\;\triangleq} \\
\multicolumn{2}{l}{\quad \textsf{cases } [\![e]\!] \textsf{ of}} \\
\quad [\![c]\!] & \rightarrow c \\
\quad [\![x_i]\!] & \rightarrow bve(i) \\
\quad [\![e_1 + e_2]\!] & \rightarrow (\mathcal{E}[\![e_1]\!]\; bve\; fve) + (\mathcal{E}[\![e_2]\!]\; bve\; fve) \\
\quad [\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & \rightarrow \textsf{if } \mathcal{E}[\![e_1]\!]\; bve\; fve \neq 0 \\
 & \quad\; \textsf{then } \mathcal{E}[\![e_2]\!]\; bve\; fve \\
 & \quad\; \textsf{else } \mathcal{E}[\![e_3]\!]\; bve\; fve \\
\quad [\![f_i(e_1, \ldots, e_{n_i})]\!] & \rightarrow fve[\![f_i]\!](\mathcal{E}[\![e_1]\!]\; bve\; fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\; bve\; fve) \\
\multicolumn{2}{l}{\quad \textsf{end}}
\end{array}
$$

Figure 3.3: Standard semantics (scalars only)

sequences; fetching the value associated with an identifier x_i then simply requires accessing the i^{th} component of the appropriate bound variables environment.[7]

Note that there is a fixed point computation made explicit in the definition of $\mathcal{M}[\![p]\!]$ using the $\boldsymbol{\mu}$ fixpoint operator. In the following, we will denote $\mathcal{M}[\![p]\!]$ by $fve_{[\![p]\!]}$, i.e., the function environment which is obtained for a given program p defined as follows:

$$p = \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$$

[7]This representation of bound variables environment as sequences has also been chosen because the result of the non-standard semantics, to be presented in Section 3.2.3, will produce a sequence of demands, sequence which will be of the same *shape* (i.e., length) as the sequence representing the bound variables environment.

3.2.2 The basic notion of demands

The first notion of demand that we introduce is that of *positive* demand, which we denote by \$. Such a demand indicates a definite need for a value, i.e., it requires the demanded value to be fully defined. In the case of scalar values, this simply corresponds to evaluation to WHNF. All previous approaches to demand propagation had a similar notion of demand [Hal87, Lin88, LM91]; thus, the domain of demands used by these approaches, when restricted to scalar values, looked like the following:

$$\begin{tikzcd} \$ \arrow[d, dash] \\ \perp \end{tikzcd}$$

Our own initial investigation into the use of demands, described in [Tre90], led us to the realization that another type of demand was needed: [8] in addition to positive demands, a notion of *negative* demand was needed, demand which indicates that some subexpression is sure *not to be needed*. This notion of negative demand — which we will denote by @ — is present in our abstract demand propagation approach. This gives rise to the following basic domain of demands:

$$\begin{tikzcd} \$ \arrow[dr, dash] & & @ \arrow[dl, dash] \\ & \perp & \end{tikzcd}$$

The ordering indicates that a negative demand provides as much definite information as a positive demand: knowing that a value is sure *not* to be needed (@) provides as much information as knowing that it is sure to be needed (\$) and more information than not knowing anything ($\perp$). [9]

The notion of negative demand distinguishes our approach to demand propagation from earlier ones and provides a number of interesting advantages. First, it makes it possible to infer information such as *an argument is sure not be needed.*

[8]In [Tre90], these were called respectively *green* and *red* signals.

[9]Note that in our demand propagation semantics $\perp$ will not be present in the basic domain of demands; instead, only definite demands (@, \$) will be used. This will be explained in more detail in Section 3.2.3.

Also, when the abstract domain of demands is introduced, it will make it possible to deal with conditionals without having to infer that nothing is known about an argument when that argument appears in only one branch of a conditional. Here is a small example:

```
Def f x y = If x == 0 Then 0 Else y;
```

In this case, it is known that `y` is sure either to be needed ($) or not needed (@), from which it can be inferred that `y` is not sure to be needed, but if it ever becomes needed, then it will have to be evaluated to WHNF. On the other hand, using the approach presented in [Hal87], it could simply be inferred that nothing is known about `y`.

3.2.3 A non-standard denotational semantics for demand propagation

Intuitive overview of the exact demand propagation semantics

The goal of the exact non-standard semantics which we are going to describe is to represent the process of demand propagation using the notions of positive/negative demands introduced in the previous section. This semantics is *exact* and *non-standard* in the following sense. First, it is non-standard in that it does not compute the standard result — e.g., given a function and some inputs, determine what the output of the function would be — but instead manipulates demands and works in a backward manner: given a demand for the result of a function, it determines the demands required for the arguments in order to satisfy the result's demand. Second, it is exact in that it refers to the standard semantics to deal with conditionals and determine the appropriate demands depending on which branch is effectively taken; of course, this would not make sense in a real compiler, which is why an abstraction of the non-standard semantics will be introduced (cf. Section 3.2.5).

Note that the strategy that we use is along the lines of the work done at Yale by P. Hudak and his students. In a number of papers, they presented a number of compile-time analysis, all expressed as abstractions of some non-standard semantics: sharing analysis [Gol88a], compile-time reference counting [Hud87], path analysis for in-place array updates [BH86, BH88]. More generally, these were all forms of *abstract interpretation* [AH87, PC87]. The particularity of the one presented in this section is that it is defined from a *backward* non-standard semantics, whereas abstract interpretations are generally *forward* [Hug88]. Furthermore, we also motivate the fact that our non-standard semantics *is not science-fiction* by formally relating it with the standard semantics (Section 3.2.4) using the notion of inverse. [10]

A final point before presenting the formal definitions. What the exact (non-standard) demand propagation semantics will compute, given a demand for the result of a function, is the demands which will have been required of the arguments when the result computed by the function *has been fully produced and evaluated*. Once this result is fully produced, we can then know exactly, for each argument, whether it was needed or not. Thus, although the domain used by the demand propagation semantics will include a bottom element, all demands effectively computed for the arguments will be definite (positive or negative) demands. In other words, the interpretation of $\perp$ will be that the function's result has not/can not be fully computed, thus nothing can be inferred about the arguments' demands.

The domain of demands for the exact demand propagation semantics

Given a function f with n arguments, we want to compute the demand for each of the argument when the function terminates and effectively produces a fully defined value; on the other hand, if the function does not terminate, i.e., can not be fully produced, then we want to define the demand semantics so that it produces $\perp$. In order to represent this idea, the domain of demands used to represent the result of the semantic functions will consist of a flat domain, where each non-$\perp$ result will effectively be a *vector* (sequence) of non-$\perp$ (positive or negative) demand; given a function f with n arguments, the demand propagation semantics will then associate with f a vector of demands with n elements, each component indicating a demand for one of the argument, or $\perp$ if f's result can not be fully computed. This domain is (partially) illustrated in Figure 3.4.[11] The semantic domains are presented in Figure 3.5. [12]

[10]Another technical difference is in our explicit reference to the standard semantics, which is done in order to formally relate the non-standard semantics with the standard one. In most of the papers mentioned above, an *oracle* is used instead. Although this simplifies the relationship between the exact and abstract non-standard semantics, in our case, this would have precluded the formalization of the relationship between the standard and exact non-standard semantics.

[11]This domain can be viewed as the *coalesced* sum of a number of subdomains. Note that, in practice, if within the program to be analyzed the maximum number of arguments for any function is n, then we only need to include the sequences of length $\leq n$.

[12]Domains which are not specified have the same definition as they had in Fig. 3.3.

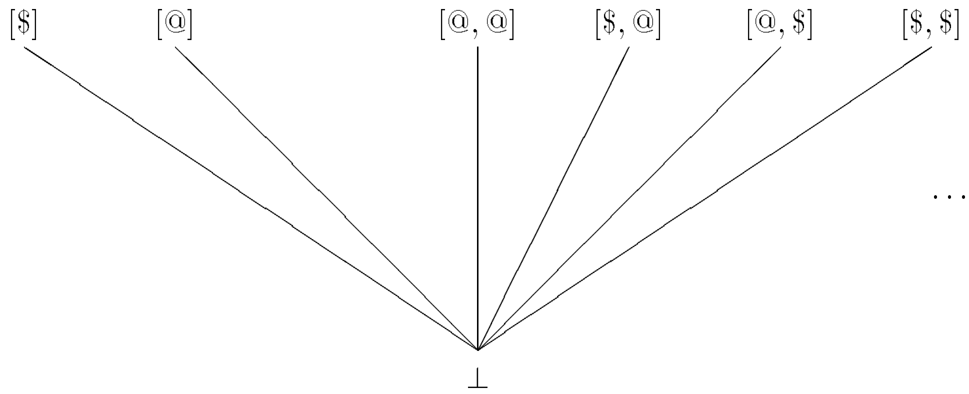


Figure 3.4: A partial illustration of the domain of demands for the exact demand propagation semantics (scalars only)

$$Dmd = \{\$, @\}$$

$$DmdVec_k = \{[d_1, \ldots, d_k] \mid d_i \in Dmd\}_\perp$$

$$DmdVec = \textstyle\sum_{k \geq 1} DmdVec_k \quad \text{(coalesced sum)}$$

Figure 3.5: The domains for the exact demand propagation semantics (scalars only)

The merge ($\oplus$) operation

Before presenting the exact demand propagation semantics, a major element needs to be introduced: the operation used to combine together various demands made for a single value — the merge ($\oplus$) operator. This operator is defined such that whenever some part of the program which is sure to be executed needs the argument and the program terminates then the argument is sure to be needed. Conversely, we can be sure that an argument is not needed only if all parts of the program that could use it are sure not to need it (and the program terminates).

The definition of the $\oplus$ operator for basic definite demands is presented in Table 3.1. Note that an alternative definition, more general and meaningful, will be presented in Section 3.2.4 when concretization sets associated with demands are introduced.

$\oplus$	@	$
@	@	$
$	$	$

Table 3.1: Merge operator for definite demands (scalars only)

To work properly within the demand propagation semantics which deals with vectors of demands, the basic merge operator has to be extended to also deal with vectors. This is presented below, where $\overline{\oplus}$ effectively represents a *family* of (overloaded) operators, i.e., there is a $\overline{\oplus}$ for each $k \geq 1$.

Definition 3.11 *(Merge of vectors)*

$$\overline{\oplus} : DmdVec_k \to DmdVec_k \to DmdVec_k$$

$$\begin{array}{l} \overline{d} \mathbin{\overline{\oplus}} \overline{e} \quad \triangleq \\ \qquad \textsf{let } [d_1 \ \cdots \ d_k] = \overline{d} \textsf{ in} \\ \qquad \textsf{let } [e_1 \ \cdots \ e_k] = \overline{e} \textsf{ in} \\ \qquad [d_j \oplus e_j \mid 1 \leq j \leq k] \end{array}$$

Note that, from the semantics of the pattern-matching operator on sequences (cf. Section A.7), $\overline{\oplus}$ is effectively a strict operator; thus, if either $\overline{d}$ or $\overline{e}$ is $\perp$, then so will the result of the merge. This definition can be motivated by the following interpretation, which should become clear when the semantics function is presented:

$$\overline{Fve} = FuncName \xrightarrow{m} Dmd \rightarrow Value^* \rightarrow DmdVec$$

$$\overline{\mathcal{M}} : Prog \rightarrow \overline{Fve}$$

$$\overline{\mathcal{M}}[\![p]\!] \;\triangleq\; \boldsymbol{\mu}\,\overline{fve} \cdot \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \overline{\mathcal{E}}_{n_i}[\![e_i]\!]\, d\,[z_j \mid 1 \leq j \leq n_i]\, fve_{[\![p]\!]}\, \overline{fve}\}$$
$$\textsf{where}\ p \;=\; \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$$

Figure 3.6: Exact demand propagation semantics for programs

if some part of the program which is sure to be needed does not terminate, then neither will the whole program.

The exact demand propagation semantics

Within the exact (non-standard) demand propagation semantics, we are not interested in computing the standard result. However, in order to resolve conditionals, the standard semantics must still be used. For this purpose, we will use — within the non-standard semantics — the function $\mathcal{E}$ from the standard semantics together with the function environment produced by this semantics ($fve_{[\![p]\!]}$).

The definitions of the exact semantic functions are presented in Figures 3.6 and 3.7. As in the standard semantics, $\overline{\mathcal{M}}[\![p]\!]$ will be denoted by $\overline{fve}_{[\![p]\!]}$. Note that we are effectively defining a family of function $\overline{\mathcal{E}}_k$, for any integer $k \geq 1$, where $\overline{\mathcal{E}}_k$ produces sequences of length k used to analyze the body of functions having exactly k arguments. (Remember that $p = \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$.)

Note that, in order to ease the reading of the definitions, the following notational convention is used:

- $d, d_1, d_2 \in Dmd$
- $\overline{d}, \overline{d_1}, \overline{d_2} \in DmdVec$

Let us first briefly explain the demand propagation semantics for expressions; the one for programs will be illustrated below with a number of examples. The non-standard semantics presented in Fig. 3.7 computes, given an expression e and a demand d, a vector of demands for the function arguments (formals) of which e is a sub-expression (we assume that this function has $k \geq 1$ arguments).

First, when the initial demand is a negative demand (@), then we can immediately infer that none of the arguments will be needed ($[@ \mid 1 \leq j \leq k]$). [13] Similarly,

[13]Clearly, in this semantics, $d \neq @ \;\Rightarrow\; d = \$$. Although the semantics could be simplified to

when the expression is a constant value ($[\![c]\!]$), then we can be sure that none of the arguments will be needed. When the expression is an identifier denoting one of the argument ($[\![x_i]\!]$), then we can be sure that no demand will be made for the identifiers but x_i itself: thus, a vector of demands consisting of d in the i^{th} position can be produced, with a negative demand (@) in the remaining positions.

In the case of a strict binary operator ($[\![e_1 + e_2]\!]$), the global demands are obtained by combining the demands made while computing each expression independently. From the definition of $\overline{\oplus}$, it can be seen that: a) if any of the sub-expression produces $\perp$, i.e., does not produce a fully defined result, then so will the whole expression; b) any argument will be needed for sure ($) if it is demanded for sure in either one (or both) of the branches (because $\$ \oplus @ = \$$).

The case dealing with conditionals ($[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]$) is the one that effectively refers to the standard semantics (so it will be this case which will be handled differently in the abstract demand propagation version). If the predicate (e_1) turns out to be true (*à la C*, i.e., $e_1 \neq 0$), then the final demands are obtained by combining the demands made while evaluating the predicate with those made by evaluating the then-branch, and similarly for the false case ($e_1 = 0$) and else-branch. Note that the predicate expression has to be evaluated, which is why it is effectively demanded with $.

The last case ($[\![f_i(e_1, \ldots, e_{n_i})]\!]$) deals with function calls (an example is presented below). In order to understand how such calls are handled, it must be kept in mind that demand propagation works in the *backward* direction. In the standard semantics, when encountering a call to a function f_i, we first evaluate the actual argument expressions and use the computed value to call f_i, in order to produce its output result. In the demand propagation semantics, things go backward: given some demand d for the *result* of f_i, we first compute the demands made for the actual argument expressions used in the call — these are the $[d_j \mid 1 \leq j \leq n_i] = \overline{fve}[\![f_i]\!]\ d\ (\ldots)$. [14] Once these demands (for the actual argument expressions) have been determined, we can then use them to compute the effect of the evaluation of each of the actual argument expressions ($\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve}$), combining them in order to obtain the global effect on the free variables relative to which $\overline{\mathcal{E}}_k$ is

account for this, it is defined this way to *pave* the way for the next section, where data structures will be introduced; for data structures, $d \neq @$ will not imply that $d = \$$.

[14]Note that, because we are dealing with the exact demand propagation semantics, the computation of these demands also requires the standard values of the actual arguments to be computed.

$$\overline{\mathcal{E}}_k : Expr \rightarrow Dmd \rightarrow Bve \rightarrow Fve \rightarrow \overline{Fve} \rightarrow DmdVec_k$$

$$\begin{array}{l}
\overline{\mathcal{E}}_k[\![e]\!]\, d\ bve\ fve\ \overline{fve} \quad \triangleq \\
\quad \mathsf{if}\ d = @ \\
\quad \mathsf{then}\ [@ \mid 1 \leq i \leq k] \\
\quad \mathsf{else\ cases}\ [\![e]\!]\ \mathsf{of} \\
\qquad \begin{array}{ll}
[\![c]\!] & \rightarrow [@ \mid 1 \leq i \leq k] \\
[\![x_i]\!] & \rightarrow [@ \mid 1 \leq j < i] \curvearrowright [d] \curvearrowright [@ \mid i < j \leq k] \\
[\![e_1 + e_2]\!] & \rightarrow (\overline{\mathcal{E}}_k[\![e_1]\!]\ d\ bve\ fve\ \overline{fve})\ \overline{\oplus}\ (\overline{\mathcal{E}}_k[\![e_2]\!]\ d\ bve\ fve\ \overline{fve}) \\
[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!] & \rightarrow \mathsf{if}\ \mathcal{E}[\![e_1]\!]\ bve\ fve \neq 0 \\
 & \quad \mathsf{then}\ (\overline{\mathcal{E}}_k[\![e_1]\!]\ \$\ bve\ fve\ \overline{fve})\ \overline{\oplus}\ (\overline{\mathcal{E}}_k[\![e_2]\!]\ d\ bve\ fve\ \overline{fve}) \\
 & \quad \mathsf{else}\ (\overline{\mathcal{E}}_k[\![e_1]\!]\ \$\ bve\ fve\ \overline{fve})\ \overline{\oplus}\ (\overline{\mathcal{E}}_k[\![e_3]\!]\ d\ bve\ fve\ \overline{fve}) \\
[\![f_i(e_1,\ \ldots,\ e_{n_i})]\!] & \rightarrow \mathsf{let}\ [d_j \mid 1 \leq j \leq n_i] = \overline{fve}[\![f_i]\!]\ d(\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\ bve\ fve)\ \mathsf{in} \\
 & \quad \overline{\bigoplus}\ \{\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mid 1 \leq j \leq n_i\}
\end{array} \\
\quad \mathsf{end}
\end{array}$$

Figure 3.7: Exact demand propagation semantics for expressions (scalars only)

computed ($\overline{\bigoplus}\{\ldots \mid 1 \leq j \leq n_i\}$).

Below are some examples illustrating the demand propagation semantics for programs together with the handling of function calls. Suppose we are given a program made of the following function definitions: [15]

```
Def fac x = If x == 0 Then 1 Else x * (fac (x-1));
Def g x y z = If x == 0 Then 0 Else y;
Def h u v = v + (g u v (v+1));
```

The non-standard meaning associated with that program would be

$$\overline{fve}_{[\![p]\!]} = \{fac \mapsto f', g \mapsto g', h \mapsto h'\}$$

with f', g' and h' defined as follows:

$$\begin{array}{l}
f' = \lambda d, (x) \cdot \mathsf{if}\ d = @\ \mathsf{then}\ [@]\ \mathsf{else\ if}\ x \geq 0\ \mathsf{then}\ [\$]\ \mathsf{else}\ \perp \\
g' = \lambda d, (x, y, z) \cdot \mathsf{if}\ d = @\ \mathsf{then}\ [@, @, @]\ \mathsf{else\ if}\ x = 0\ \mathsf{then}\ [\$, @, @]\ \mathsf{else}\ [\$, \$, @] \\
h' = \lambda d, (u, v) \cdot \mathsf{if}\ d = @\ \mathsf{then}\ [@, @]\ \mathsf{else\ if}\ u = 0\ \mathsf{then}\ [\$, \$]\ \mathsf{else}\ [\$, \$]
\end{array}$$

[15]For clarity of presentation, the examples use identifiers which are not necessarily of the form f_j or x_j.

The function environment $\overline{fve}_{[\![p]\!]}$ would thus lead, for example, to the following demand being computed:

$$\begin{array}{lll}
\overline{fve}_{[\![p]\!]}[\![f]\!] \ @ \ (\bot) = [@] & \overline{fve}_{[\![p]\!]}[\![g]\!] \ \$ \ (\bot, 2, \bot) = \bot & \overline{fve}_{[\![p]\!]}[\![h]\!] \ @ \ (\bot, \bot) = [@, @] \\
\overline{fve}_{[\![p]\!]}[\![f]\!] \ \$ \ (3) = [\$] & \overline{fve}_{[\![p]\!]}[\![g]\!] \ \$ \ (0, \bot, \bot) = [\$, @, @] & \overline{fve}_{[\![p]\!]}[\![h]\!] \ \$ \ (\bot, 0) = \bot \\
\overline{fve}_{[\![p]\!]}[\![f]\!] \ \$ \ (\text{-}1) = \bot & \overline{fve}_{[\![p]\!]}[\![g]\!] \ \$ \ (1, \bot, 2) = [\$, \$, @] & \overline{fve}_{[\![p]\!]}[\![h]\!] \ \$ \ (0, \bot) = [\$, \$] \\
 & & \overline{fve}_{[\![p]\!]}[\![h]\!] \ \$ \ (1, \bot) = [\$, \$]
\end{array}$$

Let's briefly examine how a function call is handled. Suppose we are analyzing the call to g within h, i.e., $(g\ u\ v\ (v+1))$, with h's result demanded with \$. The first step is to determine the demands made for the actual argument expressions, i.e., the demands made respectively for u, v, $v+1$. This means computing the following expression:

$$\overline{fve}_{[\![p]\!]}[\![g]\!] \ \$ \ (\mathcal{E}[\![u]\!] \ bve \ fve, \mathcal{E}[\![v]\!] \ bve \ fve, \mathcal{E}[\![v+1]\!] \ bve \ fve)$$

Suppose, for the sake of the example, that $u \neq 0$. In this case, the previous expression would return the vector $[\$, \$, @]$. In other words, the expressions u, v and $v+1$ would be demanded respectively with \$, \$ and @. In turn, determining the demands relative to the free variables of h for each of these expressions would result in the following vectors of demands : u under demand $\$ = [\$, @]$; v under $\$ = [@, \$]$ and $v+1$ under $@ = [@, @]$. Combining these three vectors with $\overline{\oplus}$ would thus produce the vector $[\$, \$]$ as the demand for the whole function call to g. (Note that this vector, which is the result of the demand evaluation for $+$'s second operand, would then be merged with the one produced for the evaluation of $+$'s first operand (v) which would be $[@, \$]$.)

3.2.4 Soundness of the exact demand propagation semantics

It was mentioned in Section 3.1.4 that the notions of demand and demand propagation could also be viewed in terms of inverse computation. This will now be done by relating the standard semantics with the exact demand propagation semantics using a notion of inverse similar to the one introduced in Section 3.1.2. But first, we have to give a precise interpretation of demands, which will be done through a *concretization function*. Fig. 3.8 shows a pictorial representation of the relationships between the standard semantics and the exact demand propagation one when

$$
\begin{tikzcd}[column sep=huge, row sep=large]
\mathit{Value}^* \arrow[d, "\text{Standard semantics}"'] \arrow[dr, phantom, "\text{Concretization function}"] & \mathit{DmdVec} \arrow[l] \\
\mathit{Value} & \mathit{Dmd} \arrow[l] \arrow[u, "\text{Exact demand propagation semantics}"']
\end{tikzcd}
$$

Figure 3.8: Using a concretization function to relate the standard and non-standard (demand propagation) semantics

viewed through the concretization function to be defined. Note that the intuition about the exact demand propagation semantics being an inverse of the standard one is illustrated from the reversed direction of the associated arrow, which maps a demand for the result (*Dmd*) to demands for the arguments (*DmdVec*), instead of mapping some actual arguments (Value^*) to a result (*Value*).

Demands as sets of values

So far, basic demands ($, @) have only been given an intuitive interpretation, where, for example, @ was said to represent a *negative* demand (sure not to be needed) and $ represented a *positive* demand (sure to be needed). In order to formally relate the exact non-standard semantics with the standard one, we now give a formal interpretation of demands in terms of *sets of concrete values*. We thus define the following *conc*retization function: [16]

Definition 3.12 *(Concretization of exact demands)*

$$\mathit{conc} : \mathit{Dmd} \to \mathcal{P}(\mathit{Value})$$

$$
\begin{array}{lll}
\mathit{conc}(d) & \triangleq & \textsf{cases } d \textsf{ of} \\
& & @ \to \{\bot\} \\
& & \$ \to \mathbb{Z} \\
& & \textsf{end}
\end{array}
$$

[16] Note that it is the *powerset* constructor ($\mathcal{P}(\mathit{Value})$) which has been used to defined the range of the concretization function *conc*. This constructor allows for infinite sets, contrary to the *finite* set constructor of VDM (Value-**set**) which only allows for sets with a finite number of elements.

The motivation behind this definition is that we want a demand to represent a set of acceptable values, or, put another way, to represent the set of values which can *satisfy* the demand. Furthermore, we want to represent this using as little information as possible. In the case of a positive demand — i.e., when we know that we will need for sure the result/value implying we need a value which is not $\perp$ — then the required value must be some well-defined integer value, so all such values have to be included in the concretization set. As for a negative demand, it indicates that we don't really care about the value — any value is acceptable, can satisfy the demand — thus, any value at least as defined as $\perp$ — i.e., anything! — must be acceptable. The notion of a value *satisfying* a demand can be formalized as follows (where $\sqsupseteq$ refers to the ordering within the standard domain, i.e., between *Values*):

Definition 3.13 *(Demand satisfaction)*

$$satisfy : Value \rightarrow Dmd \rightarrow \mathbb{B}$$

$$satisfy(v, d) \quad \triangleq \quad \exists u \in conc(d) \cdot v \sqsupseteq u$$

The previous definition shows, for example, why it is sufficient for $conc(@)$ to be $\{\perp\}$ instead of $\mathbb{Z}_\perp$: given the previous definition of $conc(@)$, we effectively have that $\{v \mid satisfy(v, @)\} = \mathbb{Z}_\perp$, i.e., anything satisfy @.

An alternative characterization of $\oplus$

The *satisfy* definition together with the concretization function of the previous subsection are interesting as they allow to motivate the definition of $\oplus$ which was given earlier in Table 3.1. The $\oplus$ operator is used to combine demands made from different definite uses of a given identifier. The demand for the combined use must thus be such that any value which is acceptable for the combined use should also be acceptable for the independent uses. In other words, $d_1 \oplus d_2$ must be defined so that the following property holds:

$$satisfy(v, d_1 \oplus d_2) \;\Leftrightarrow\; satisfy(v, d_1) \wedge satisfy(v, d_2) \tag{3.8}$$

The definition presented in Table 3.1 clearly has this property. What is interesting is that it is possible to give a formal characterization of $\oplus$ in terms of its concretization set which will ensure that property 3.8 holds; furthermore, this will also ensure another important property which will later be required to prove the soundness of the demand propagation semantics relative to the standard one. First, let's define the following predicate:

Definition 3.14 *(Compatibility of values)*

$$\sim : Value \rightarrow Value \rightarrow \mathbb{B}$$

$$v_1 \sim v_2 \quad \triangleq \quad v_1 \sqcup v_2 \text{ exists}$$

Using the predicate $v_1 \sim v_2$, pronounced v_1 *is compatible with* v_2, we can then characterize $d_1 \oplus d_2$ in terms of its resulting concretization set as follows:

Fact 3.3

$\forall d_1, d_2 \cdot$

$$conc(d_1 \oplus d_2) = \{v_1 \sqcup v_2 \mid v_1 \in conc(d1) \land v_2 \in conc(d_2) \land v_1 \sim v_2\}$$

Proof: Straightforward, from a simple examination of the various possibilities — $@ \oplus @ = @$, $@ \oplus \$ = \$$ and $\$ \oplus \$ = \$$ — using the definitions of $\oplus$ and *conc*. □

This characterization of $\oplus$ satisfies the requirement that values which satisfy both demands also have to satisfy the resulting merged demand; furthermore, by using only appropriate least-upper bounds, when they exist, we are trying to obtain the *smallest* set satisfying this requirement.

Using Fact 3.3, it is now possible to state and prove the following fact:

Fact 3.4

$$\forall v, d_1, d_2 \cdot satisfy(v, d_1) \land satisfy(v, d_2) \;\Leftrightarrow\; satisfy(v, d_1 \oplus d_2)$$

Proof: See App. H.2.1, p. 274. □

Formal definition of inverse

The definitions we now present are based on Defn. 3.8 presented in Section 3.1.2. The general idea is that we are trying to determine *the least amount of information needed to produce at least the demanded result*. The first predicate we define formalizes the notion of inverse of an expression relative to an appropriate bound variables environment. More precisely, given an expression e and a demand d, the predicate specifies, as a sequence of demands $\overline{d}$, the values required for the arguments of the function of which e is a sub-expression. Furthermore, it also expresses the idea that we are trying to infer constraints as weak as possible for the arguments (the least amount of information), i.e., there exists some values u_j — not more defined that the values v_j (i.e., $u_j \sqsubseteq v_j$) which make it possible to obtain the required result — with which a result at least as defined can be obtained.

Definition 3.15 *(Inverse of an expression)*

$$
\begin{array}{l}
\textit{is-inverse}_{bve,fve} : \textit{Expr} \rightarrow \textit{Dmd} \rightarrow \textit{DmdVec} \rightarrow \textit{Bve} \rightarrow \textit{Fve} \rightarrow \mathbb{B} \\
\textit{is-inverse}_{bve,fve}(e, d, \overline{d}) \quad \triangleq \\
\qquad bve = [v_j \mid 1 \leq j \leq k] \wedge \overline{d} = [d_j \mid 1 \leq j \leq k] \\
\qquad \Rightarrow \\
\qquad \forall v \in conc(d) \cdot \\
\qquad\qquad \mathcal{E}[\![e]\!] \; bve \; fve \sqsupseteq v \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \exists u_1 \in conc(d_1), \ldots, u_k \in conc(d_k) \cdot \\
\qquad\qquad\quad \mathcal{E}[\![e]\!] [u_j \mid 1 \leq j \leq k] \; fve \sqsupseteq v \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad u_j \sqsubseteq v_j \;\; (j = 1, \ldots, k)
\end{array}
$$

In words, the above definition indicates that a vector of demands $\overline{d}$ is an inverse of d relative to e (and the appropriate environment) if, whenever it is possible to evaluate e in environment bve and obtain a value v acceptable for d, then values associated with the concretization set of $\overline{d}$ also make it possible to obtain a result at least as defined as v using values which are not more defined that those used in bve.

Below are some examples, letting $bve = [0, 1, 2]$ (and ignoring fve):

- $\textit{is-inverse}_{bve,fve}(\llbracket f_1(x_2)\rrbracket, @, [@, @, @])$

  (* For any function f_1, if the result of a call to f_1 is not demanded, then neither are any of the free variables. *)

- $\textit{is-inverse}_{bve,fve}(\llbracket 0\rrbracket, \$, [@, @, @])$

  (* Demanding a constant requires no demand of the free variables. *)

- $\textit{is-inverse}_{bve,fve}(\llbracket x_2\rrbracket, \$, [@, \$, @])$

  (* Demanding a variable x_2 only imposes a demand for x_2. *)

- $\textit{is-inverse}_{bve,fve}(\llbracket x_1 + x_2\rrbracket, \$, [\$, \$, @])$

  (* A positive demand for a strict operator propagates directly to each of the subexpressions. *)

- $\textit{is-inverse}_{bve,fve}(\llbracket \texttt{if}\ x_1 == 0\ \texttt{then}\ x_2\ \texttt{else}\ x_3\rrbracket, \$, [\$, \$, @])$

  (* The demand propagates to the predicate and to the selected branch. *)

The definition can next be extended to function environments as presented below in Defn. 3.16: a non-standard function environment $\overline{fve}$ is an inverse of a standard environment fve if, for any function f_i, the result obtained from the demand semantics under demand d and using some arbitrary values $v_1, \ldots, v_{n_i}$ ($\overline{fve}\llbracket f_i\rrbracket\ d\ (v_1, \ldots, v_{n_i})$) is an inverse under d of the expression consisting of a function call to f_i ($\llbracket f_i(x_1, \ldots, x_{n_i})\rrbracket$) relative to a bound variable environment consisting of the chosen $v_1, \ldots, v_{n_i}$ ($bve = [(v_1, \ldots, v_{n_i})]$) and relative to fve.

Definition 3.16 *(Inverse of a function environment)*

$$\textit{is-inverse} : Fve \rightarrow \overline{Fve} \rightarrow \mathbb{B}$$

$$\begin{array}{l} \textit{is-inverse}(fve, \overline{fve}) \ \ \triangleq \\ \quad \forall f_i \in \mathsf{dom}\, fve \cdot \\ \quad\quad \forall d \in Dmd, v_1, \ldots, v_{n_i} \in Value \cdot \\ \quad\quad\quad \textit{is-inverse}_{bve,fve}(\llbracket f_i(x_1, \ldots, x_{n_i})\rrbracket, d, \overline{fve}\llbracket f_i\rrbracket\ d\ (v_1, \ldots, v_{n_i})) \\ \quad\quad\quad\quad \text{where } bve = [v_j \mid 1 \le j \le n_i] \end{array}$$

Below is a simple example illustrating two function environments which are inverse of each other, each environment consisting simply in the appropriate meaning for the factorial function (bound to f):

$$\begin{array}{l} fve = \{f \mapsto \lambda(x) \cdot \mathsf{if}\ x \ge 0\ \mathsf{then}\ x!\ \mathsf{else}\ \bot\} \\ \overline{fve} = \{f \mapsto \lambda d, (x) \cdot \mathsf{if}\ d = @\ \mathsf{then}\ [@]\ \mathsf{else\ if}\ x \ge 0\ \mathsf{then}\ [\$]\ \mathsf{else}\ \bot\} \end{array}$$

Let $d = \$$, v_1 be some arbitrary value and $bve = [v_1]$. We then have:

$$\overline{fve}[\![f]\!]\ \$\ (v_1) = \textsf{if}\ v_1 \geq 0\ \textsf{then}\ [\$]\ \textsf{else}\ \bot$$

If $v_1 < 0$ then $\overline{fve}[\![f]\!]\ \$\ (v_1) = \bot$, so the following holds vacuously: [17]

$$\textit{is-inverse}_{bve,fve}([\![f(x_1)]\!], \$, \bot)$$

On the other hand, suppose $v_1 \geq 0$, so that $\overline{fve}[\![f]\!]\ \$\ (v_1) = [\$]$. We must then have:

$$\textit{is-inverse}_{bve,fve}([\![f(x_1)]\!], \$, [\$])$$

Now, evaluating $\mathcal{E}[\![f(x_1)]\!]$ in environments bve and fve clearly will produce a well-defined result ≥ 0 (as $bve(1) = v_1 \geq 0 \Rightarrow v_1! \geq 0$). If we then take some arbitrary value $w_1 \in conc(\$) = \mathbb{Z}$ such that $\mathcal{E}[\![f(x_1)]\!]\ bve\ fve \sqsupseteq w_1$, we can clearly obtain the same result with some value $u_1 \in conc(\$) = \mathbb{Z}$, by taking $u_1 = v_1$. Thus, $\overline{fve}[\![f]\!]\ \$\ (v_1) = [\$]$ is effectively an inverse of the expression $[\![f(x_1)]\!]$ relative to the environments $bve = [v_1]$ and fve under demand $\$$. [18]

The exact demand propagation semantics as an inverse of the standard semantics

The claims about the relationships between the standard and exact demand propagation semantics can be expressed by Theorem 3.5, presented below, which asserts that for any program the function environment produced by the exact demand semantics is an appropriate inverse of the one obtained from the standard semantics.

Theorem 3.5 (Soundness of exact demand propagation semantics)

For any program p:

$\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$

Proof: The proof of this theorem is by fixed-point induction on $\overline{fve}^{(k)}_{[\![p]\!]}$. The details are presented in App. H.2.1, p. 275. The key lemma used by the proof is presented

[17] Remember that evaluating the expression $[\![x_i]\!]$ in environment bve returns the i^{th} element of bve, i.e., $bve(i)$.

[18] Note that under this definition, $\overline{fve} = \{f_i \mapsto \lambda d, (x_1, \ldots, x_{n_i}) \cdot \bot \mid 1 \leq i \leq k\}$ is effectively an inverse of any fve such that $\textsf{dom}\,fve = \{f_i \mid 1 \leq i \leq k\}$. This is perfectly consistent with the view that constraints are imposed on the arguments only when the non-standard semantics effectively produces a vector of demands as result. Also note that $\textsf{dom}\,\overline{fve} \supset \textsf{dom}\,fve$ will not create any problem: the constraints on being an inverse must hold only when $f_i \in \textsf{dom}\,fve$.

below in Lemma 3.6, which establishes a similar *is-inverse* relationship between the standard and non-standard semantics for expressions, i.e., it expresses the fact that $\overline{\mathcal{E}}_k$ effectively computes the inverse of an expression e relative to an appropriate environment. □

Lemma 3.6 *(Soundness of expression semantics)*

$$
\begin{array}{l}
\forall\, e, d, bve, fve, \overline{fve} \cdot \\
\quad \textit{is-inverse}(fve, \overline{fve}) \\
\qquad \Rightarrow \\
\quad \textit{is-inverse}_{bve,fve}(e, d, \overline{\mathcal{E}}_k[\![e]\!]\; d\; bve\; fve\; \overline{fve}) \\
\qquad \text{where} \\
\qquad\quad k = \mathsf{len}\; bve
\end{array}
$$

Proof: The proof is by structural induction (cf. App. H.2.1, p. 277). The key of the proof resides in Fact 3.3, which ensures that for any $v_1 \in conc(d_1)$, $v_2 \in conc(d_2)$ with $v_1 \sim v_2$, we always have that $v_1 \sqcup v_2 \in conc(d_1 \oplus d_2)$. □

At this point, we now have defined a notion of demand propagation as a form of inverse computation, notion which has been substantiated through the definition of an exact (non-standard) demand propagation semantics, a precise notion of inverse and a soundness theorem which relates the standard semantics and the demand propagation one. However, as currently defined, this latter semantics can not be used as a model of demand propagation that could be incorporated directly in a prototype compiler; the reason is that its execution does not always terminate, because of its direct dependence on the exact semantics to resolve conditionals. This problem will be resolved, in the next section, by introducing an abstraction of exact demand propagation, i.e., by defining an abstract demand propagation semantics that always terminate and by showing that it provides only safe results.

3.2.5 An abstraction of the exact (non-standard) demand propagation semantics

We now define an abstraction of the non-standard semantics presented in Section 3.2.3. This *abstract non-standard* semantics is an abstraction in the sense that information is lost when dealing with conditionals. In the exact non-standard semantics, the predicate of a conditional was computed using the standard semantics. Using the standard semantics to evaluate an expression means evaluation of the expression may never terminate, i.e., it could be $\perp$; this is not acceptable for the abstract semantics, as we want to be sure the analysis always terminates, so it can be implemented by a compiler. Thus, when dealing with conditionals, an approximation will be performed: the result of both branches of the conditionals will be *combined* to obtain a less precise yet safe result. Note that as abstract demands will effectively correspond to *sets* of demands, this combination will simply consist in the union of the corresponding sets.

The abstract domain of demands

As in the exact demand propagation semantics, we will be computing vectors; of course, these vectors will now be composed of abstract demands. The domain for demand vectors, which is not a flat domain anymore although it is still of finite height, is defined in Fig. 3.9. The ordering on basic abstract demands ($ADmd$) is illustrated in Fig. 3.10 (the ordering on vectors of demands is simply the pointwise extension of the basic ordering).

$$ADmd = \{\phi, @, \$, \hat{}\$\}$$

$$ADmdVec_k = \{[d_1,\ \ldots,\ d_k] \mid d_i \in ADmd\}$$

$$ADmdVec = \textstyle\sum_{k \geq 1} ADmdVec_k \quad \text{(separated sum)}$$

Figure 3.9: Abstract domain of demands (scalars only)

Although the basic domain of abstract demands still contains @ and \$, these represent overloaded uses of the symbols used in the previous section. Abstract demands really correspond to sets of exact demands; the correspondence between abstract demands and sets of exact demands can be defined using the following concretization function:

Definition 3.17 *(Concretization of abstract demands)*

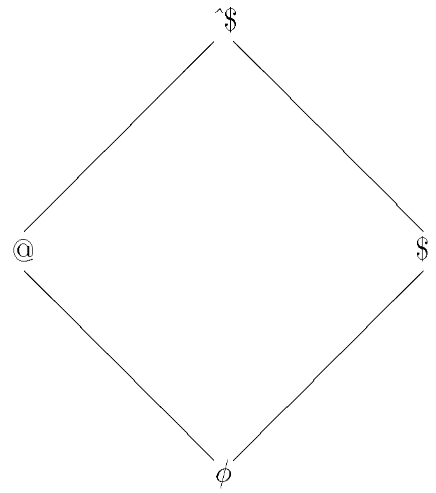


Figure 3.10: The ordering for basic abstract demands (scalars only)

$$conc : ADmd \rightarrow Dmd\text{-set}$$

$$
\begin{aligned}
conc(\delta) \;\; \triangleq \;\; & \textsf{cases } \delta \textsf{ of} \\
& \phi \rightarrow \{\,\} \\
& @ \rightarrow \{@\} \\
& \$ \rightarrow \{\$\} \\
& \hat{\$} \rightarrow \{@, \$\} \\
& \textsf{end}
\end{aligned}
$$

In the following sections, the symbols from the left will often be used as abbreviations for the sets on the right, while the original symbol for (exact) positive and negative demands will also be used; the context should always make clear which use is intended. Thus, an expression such as "$@ \in @$" will be used without ambiguity — the first @ will denote an exact negative demand whereas the second one will denote the abstract demand containing only the negative demand.

The loss of precision related to the abstract domain of demands is due to the presence of ^$ (called an *uncertain positive* demand) as it is associated with a set containing multiple possibilities. Such a demand, which represents an uncertain demand for the value of a scalar, is obtained by combining positive and negative demand information coming from distinct branches of a conditional. For example, suppose we are given the following function definition:

```
Def f x y = If x == 0 Then 0 Else y;
```

The exact demand for `y` would be either @ or \$ depending on which branch of the conditional gets executed. In the abstract semantics, to avoid the dependence on the exact value of the predicate, we will account for both possibilities: this will be expressed by associating `y` with the demand ˆ\$, the "ˆ" (called the *uncertainty* indicator) signaling the relative *uncertainty* of the demand.

The abstract operations on demands

The merge ($\oplus$) operator The merge ($\oplus$) operator for abstract demands is presented in Table 3.2.

$\oplus$	ϕ	@	\$	ˆ\$
ϕ	ϕ	ϕ	ϕ	ϕ
@	ϕ	@	\$	ˆ\$
\$	ϕ	\$	\$	\$
ˆ\$	ϕ	ˆ\$	\$	ˆ\$

Table 3.2: Merge operator for abstract demands (scalars only)

Like the exact merge operator, a certain positive demand always takes precedence over a negative demand (\$ $\oplus$ @ = \$); similarly, a certain positive demand also takes precedence over an uncertain one (\$ $\oplus$ ˆ\$ = \$). As for the bottom element (ϕ), it acts as a zero of the $\oplus$: if nothing is acceptable for one of the certain use of a variable, then nothing can be acceptable for the global use. A clearer definition of $\oplus$, the bottom element and uncertainty can be given by the function below, which uses the set representation of the abstract demands: [19] When viewed in terms of sets of elements, $\oplus$ combines (merges) all possible concrete demands coming from the two sets; clearly, then, if one of the set is empty, then so will the whole result.

Definition 3.18 *(Abstract demand merge)*

$$\oplus : ADmd \rightarrow ADmd \rightarrow ADmd$$

$$ad_1 \oplus ad_2 \quad \triangleq \quad \{d_1 \oplus d_2 \mid d_1 \in ad_1 \wedge d_2 \in ad_2\}$$

[19] We will see in Section 3.3.2 that the abstract $\oplus$ operation should really be defined as $\{d_1 \oplus d_2 \mid d_1 \in ad_1 \wedge d_2 \in ad_2 \wedge d_1 \sim d_2\}$, with a definition of compatible ($\sim$) appropriate for basic demands. However, it turns out that, for scalar demands, $d_1 \sim d_2$ is always true, so this element is redundant in the definition below.

The operator $\oplus$ has to be extended to deal with vectors of abstract demands, denoted by $\tilde{\oplus}$. It is defined as the element-wise extension of $\oplus$, with one minor caveat: we assume the vector constructor to be strict in ϕ, i.e., if any of the component is ϕ, then the result is a vector completely composed of ϕ elements, i.e.:

$$[x_1, \ldots, x_{i\text{-}1}, \phi, x_{i+1}, \ldots, x_k] = [\phi \mid 1 \leq j \leq k]$$

In other words, the domain of abstract demands for vectors of size k identifies all vectors having one of their component equal to ϕ. Thus, assuming strictness in ϕ, we have the following definition:

Definition 3.19 *(Merge of abstract demand vectors)*

$$\tilde{\oplus} : ADmdVec_k \rightarrow ADmdVec_k \rightarrow ADmdVec_k$$

$$[\delta_1, \ldots, \delta_k] \,\tilde{\oplus}\, [\epsilon_1, \ldots, \epsilon_k] \quad \triangleq \quad [\delta_j \oplus \epsilon_j \mid 1 \leq j \leq k]$$

The $\cup$ operator The $\cup$ operator is used to combine demands coming from different branches of conditionals. Given the ordering defined above, it is effectively equivalent to the l.u.b. operation and really corresponds to the union of the associated exact demands. The table defining the $\cup$ operator is presented in Table 3.3.

$\cup$	ϕ	@	\$	^\$
ϕ	ϕ	@	\$	^\$
@	@	@	^\$	^\$
\$	\$	^\$	\$	^\$
^\$	^\$	^\$	^\$	^\$

Table 3.3: Union operator for abstract demands (scalars only)

The following function, which defines the operator using the set representation, makes it clear that we are dealing with union:

Definition 3.20 *(Abstract demand union)*

$$\cup : ADmd \rightarrow ADmd \rightarrow ADmd$$

$$ad_1 \oplus ad_2 \quad \triangleq \quad \{d \mid d \in ad_1 \vee d \in ad_2\}$$

The definition is extended element-wise to vectors in the following way:

Definition 3.21 *(Union of abstract demand vectors)*

$$\tilde{\cup} : ADmdVec_k \rightarrow ADmdVec_k \rightarrow ADmdVec_k$$

$$[\delta_1, \ldots, \delta_k] \,\tilde{\cup}\, [\epsilon_1, \ldots, \epsilon_k] \;\triangleq\; [\delta_j \cup \epsilon_j \mid 1 \leq j \leq k]$$

The abstract demand propagation semantics

The definitions for the abstract semantics are presented in Fig. 3.11 and 3.12. Again, the term $\widetilde{\mathcal{M}}[\![p]\!]$ will be denoted by $\widetilde{fve}_{[\![p]\!]}$. The following notational convention is used:

- $\delta, \delta_1, \delta_2 \in ADmd$
- $\tilde{\delta}, \tilde{\delta}_1, \tilde{\delta}_2 \in ADmdVec$

The abstract function environment associated with a program p is obtained in a way similar to the one produced by the exact demand semantics (cf. Fig. 3.6), except that the abstract one is simpler as it does not need to refer to the standard semantics and does not need to have arguments from the standard domain, i.e., only a demand argument is required. As for the definition of the abstract semantics for expressions (Fig. 3.12), it is also similar to the one for the exact demand semantics (cf. Fig. 3.7, p. 45) with a number of small differences:

- No reference to the standard semantics is made, as we want the abstract semantics to be computable at compile-time. Thus, when dealing with conditionals, instead of referring to the standard semantics to determine which branch is effectively taken, the results from the then- and the else-branches are combined using the $\cup$ operator, so that both possibilities (true vs. false predicate) are accounted for.
- The heart of function $\tilde{\mathcal{E}}_k$ (the cases expression) really deals with the case where $\delta = \$$, i.e., δ only contains a definite positive demand. If @ is one of the possibility expressed by δ (i.e., $@ \in \delta$), then a recursive call to $\tilde{\mathcal{E}}_k$ is done, using a demand not containing @ ($\delta - \{@\}$); when the demand for this is produced, the fact that @ was one of the initial possibility is then accounted for by adding @ to the resulting demand ($\tilde{\delta}(j) \cup @$). When the demand denotes

$$\widetilde{Fve} = FuncName \xrightarrow{m} ADmd \rightarrow ADmdVec$$

$$\widetilde{\mathcal{M}} : Prog \rightarrow \widetilde{Fve}$$

$$\widetilde{\mathcal{M}}[\![p]\!] \;\triangleq\; \boldsymbol{\mu}\, \widetilde{fve} \cdot \{f_i \mapsto \lambda\delta \cdot \widetilde{\mathcal{E}}_{n_i}\, [\![e_i]\!]\, \delta\, \widetilde{fve}\}$$
$$\textsf{where } p \;=\; \{f_i(x_1,\, \ldots,\, x_{n_i}) = e_i; \ldots\}$$

Figure 3.11: Abstract demand propagation semantics for programs

an empty set of possibilities ($d = \phi$), then no values are acceptable for the arguments, so $[\phi \mid 1 \leq j \leq k]$ is returned.

The examples presented earlier will again be used, this time to illustrate the abstract semantics. The function definitions are:

```
Def fac x = If x == 0 Then 1 Else x * (fac (x-1));
Def g x y z = If x == 0 Then 0 Else y;
```

The abstract meaning associated with the program would then be $\widetilde{fve}_{[\![p]\!]} = \{fac \mapsto f', g \mapsto g'\}$ with f' and g' defined as follows:

$$f' = \lambda d \cdot [d]$$

$$\begin{array}{l} g' = \lambda d \cdot \textsf{cases } d \textsf{ of} \\ \qquad\qquad \phi \rightarrow [\phi, \phi, \phi] \\ \qquad\qquad @ \rightarrow [@, @, @] \\ \qquad\qquad \$ \rightarrow [\$, \hat{}\$, @] \\ \qquad\qquad \hat{}\$ \rightarrow [\hat{}\$, \hat{}\$, @] \\ \qquad\qquad \textsf{end} \end{array}$$

More generally, it is easy to see that the following properties hold:

$$\widetilde{\mathcal{E}}_k[\![e]\!]\, \phi\, \widetilde{fve} = [\phi \mid 1 \leq j \leq k] \tag{3.9}$$

$$\widetilde{\mathcal{E}}_k[\![e]\!]\, @\, \widetilde{fve} = [@ \mid 1 \leq j \leq k] \tag{3.10}$$

$$\widetilde{\mathcal{E}}_k[\![e]\!]\, \$\, \widetilde{fve} = [\delta_j \mid 1 \leq j \leq k] \quad \Rightarrow \quad \widetilde{\mathcal{E}}_k[\![e]\!]\, \hat{}\$\, \widetilde{fve} = [\delta_j \cup @ \mid 1 \leq j \leq k] \tag{3.11}$$

In words, these properties can be paraphrased as follows:

(3.9) If no result is acceptable, then there is no argument which will make it possible to produce an acceptable result.

$$\widetilde{\mathcal{E}}_k : Expr \rightarrow ADmd \rightarrow \widetilde{Fve} \rightarrow ADmdVec_k$$

$$
\begin{array}{l}
\widetilde{\mathcal{E}}_k [\![e]\!]\, \delta\, \widetilde{fve} \quad \triangleq \\
\quad \textsf{if } @ \in \delta \\
\quad \textsf{then let } \widetilde{\delta} = \widetilde{\mathcal{E}}_k [\![e]\!]\, (\delta - \{@\})\, \widetilde{fve} \;\; \textsf{in} \\
\qquad [\widetilde{\delta}(j) \cup @ \mid 1 \leq j \leq k] \\
\quad \textsf{else if } \delta = \phi \\
\qquad \textsf{then } [\phi \mid 1 \leq j \leq k] \\
\qquad \textsf{else cases } [\![e]\!] \textsf{ of} \\
\qquad\quad \begin{array}{ll}
[\![c]\!] & \rightarrow [@ \mid 1 \leq i \leq k] \\
[\![x_i]\!] & \rightarrow [@ \mid 1 \leq j < i] \curvearrowright [\delta] \curvearrowright [@ \mid i < j \leq k] \\
[\![e_1 + e_2]\!] & \rightarrow (\widetilde{\mathcal{E}}_k [\![e_1]\!]\, \delta\, \widetilde{fve})\; \widetilde{\oplus}\; (\widetilde{\mathcal{E}}_k [\![e_2]\!]\, \delta\, \widetilde{fve}) \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & \rightarrow \textsf{let } \widetilde{\delta}_1 = \widetilde{\mathcal{E}}_k [\![e_1]\!]\, \$\, \widetilde{fve} \;\; \textsf{in} \\
 & \ \textsf{let } \widetilde{\delta}_2 = \widetilde{\mathcal{E}}_k [\![e_2]\!]\, \delta\, \widetilde{fve} \;\; \textsf{in} \\
 & \ \textsf{let } \widetilde{\delta}_3 = \widetilde{\mathcal{E}}_k [\![e_3]\!]\, \delta\, \widetilde{fve} \;\; \textsf{in} \\
 & \ (\widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2)\; \widetilde{\cup}\; (\widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_3) \\
[\![f\ e_1\ \cdots\ e_n]\!] & \rightarrow \textsf{let } [\delta_j \mid 1 \leq j \leq n] = \widetilde{fve} [\![f]\!]\, \delta \;\; \textsf{in} \\
 & \ \widetilde{\bigoplus} \{ \widetilde{\mathcal{E}}_k [\![e_j]\!]\, \delta_j\, \widetilde{fve} \} \\
\textsf{end} &
\end{array}
\end{array}
$$

Figure 3.12: Abstract demand propagation semantics for expressions (scalars only)

(3.10) If the result is sure not to be needed, then neither will any of the argument.

(3.11) If the result is not sure to be needed, then neither are any of the arguments.

3.2.6 Safety of the abstract demand propagation semantics

Definition of safety

The goal of our safety theorem is to relate the exact demand semantics with the abstract one. The intuition behind the notion of safety is that whenever the exact semantics produces a well-defined answer $\overline{d}$ ($\neq \bot$), then this answer is sure to be contained in the set of answers $\widetilde{\delta}$ produced by the abstract semantics.

First, let's define a notion of an exact demand being safely approximated by an abstract one ($d \lhd \delta$), definition which simply expresses the idea that the concretization set associated with the abstract demand δ contains d:

Definition 3.22 *(Safe demand approximation)*

$$\lhd : Dmd \rightarrow ADmd \rightarrow \mathbb{B}$$

$$d \lhd \delta \quad \triangleq \quad d \in conc(\delta)$$

We next extend this definition to vectors of demands. Given $\overline{d} \in DmdVec_k$ and $\widetilde{\delta} \in ADmdVec_k$, we say that $\overline{d}$ is safely approximated by $\widetilde{\delta}$ ($\overline{d} \lhd \widetilde{\delta}$) whenever the following holds, i.e., if the exact result is well-defined ($\overline{d} \neq \bot$), then each of its component is safely approximated by the corresponding abstract component:

Definition 3.23 *(Safe demand vector approximation)*

$$\lhd : DmdVec_k \rightarrow ADmdVec_k \rightarrow \mathbb{B}$$

$$\overline{d} \lhd \widetilde{\delta} \quad \triangleq \quad \overline{d} \neq \bot \;\Rightarrow\; \forall j \in \{1, \ldots, k\} \cdot \overline{d}(j) \lhd \widetilde{\delta}(j)$$

The definition of $\lhd$ can then be extended to function environment. We say that $\overline{fve}$ is safely approximated by $\widetilde{fve}$ ($\overline{fve} \lhd \widetilde{fve}$) if the following holds, i.e., given a safe approximation δ to d, the result for any function f_i obtained from the exact $\overline{fve}$ is safely approximated by the result obtained from the abstract $\widetilde{fve}$:

Definition 3.24 *(Safe function environment)*

$$\lhd : \overline{Fve} \rightarrow \widetilde{Fve} \rightarrow \mathbb{B}$$

$$\begin{aligned}
&\overline{fve} \lhd \widetilde{fve} \quad \triangleq \\
&\quad \forall f_i \cdot \\
&\qquad \forall v_1, \ldots, v_{n_i}, d, \delta \cdot \\
&\qquad\quad d \lhd \delta \;\Rightarrow\; \overline{fve}[\![f_i]\!]\; d\, (v_1, \ldots, v_{n_i}) \lhd \widetilde{fve}[\![f_i]\!]\; \delta
\end{aligned}$$

Safety theorem

We now state the major theorem concerning the safety of the abstract demand propagation analysis, which is that the abstract function environment produced for a program p safely approximates the exact function environment for that same program:

Theorem 3.7 (Safety of abstract demand propagation semantics)

For any program p:

$\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$

Proof: The proof is by fixed point (computational) induction. The details are given in App. H.2.1, p. 281. □

The key lemma used in the proof of Theorem 3.7 is presented below. Essentially, it indicates that, given some expression e, the result produced by the exact expression semantic function ($\overline{\mathcal{E}}_k$) is safely approximated by the abstract one ($\widetilde{\mathcal{E}}_k$), assuming, of course, that the corresponding arguments are safe relative to one another (i.e., $\overline{fve} \lhd \widetilde{fve}$ and $d \lhd \delta$).

Lemma 3.8 *(Safety of expression semantics)*

$$\begin{aligned}
&\forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd \widetilde{fve} \cdot \\
&\quad \forall d, \delta \cdot d \lhd \delta \cdot \\
&\qquad \forall k, bve \cdot \mathsf{len}\; bve = k \cdot \\
&\qquad\quad \overline{\mathcal{E}}_k[\![e]\!]\; d\; bve\; fve\; \overline{fve} \;\lhd\; \widetilde{\mathcal{E}}_k[\![e]\!]\; \delta\; \widetilde{fve}
\end{aligned}$$

Proof: The proof is by structural induction (cf. App. H.2.1, p. 283). It uses a number of auxiliary facts (presented in the Appendix) which indicate that the two major operators ($\widetilde{\oplus}$ and $\widetilde{\cup}$) effectively lead to safe results. □

3.2.7 Putting it all together: Take 1

So, how does all of the above fit together? Let's examine a number of properties which can be inferred from the major theorems presented in the previous sections.

Bottom as indication of non-termination

One interesting characteristic of our abstract demand propagation approach is that it can recognize certain functions which are always *sure* to lead to non-terminating computations. Note that this does not mean we are solving the halting problem. All we are saying is that, even though the general problem is unsolvable, there are cases where we can effectively identify the possibility of non-termination.

Intuitively, in terms of demand propagation, such cases can be recognized when the demand made for the result of a recursive function call is certain and is the same as the one made for the initial call of the function. Here are some examples:

```
Def f1 x = f1 (x + 1);

Def f2 x = 2 * (f2 x) - 1;

Def f3 x = If (f3 x) == 0 Then ... Else ...;

Def f4 x = If ... Then (f4 x) Else (f4 x);

Def f5 x = g (f5 x);
Def g x = x + 1;
```

This possibility of being able to identify certain non-terminating programs is expressed by the definition and theorem below. Definition 3.25 indicates that a function is sure to loop when it always produces $\perp$, independently of which arguments are fed in:

Definition 3.25

f is sure to loop when:

$\forall v_1, \ldots, v_n \cdot f(v_1, \ldots, v_n) = \perp$

Theorem 3.9 states that the abstract demand propagation semantics presented earlier can effectively detect certain non-terminating programs. More precisely, if the abstract demand propagation semantics infers that there are no argument (ϕ) which lead f_i to produce a well-defined result ($\$$), then f_i is a function which is sure to loop.

Theorem 3.9 *(Correct detection of non-termination)*

$$\forall p \cdot \quad \widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\phi \mid 1 \le j \le n_i] \Rightarrow f_i \text{ is sure to loop}$$

Proof: Cf. App. H.2.1, p. 286. □

Positive demands as indication of strictness

Similarly, using the definition and theorem below, it is possible to infer, when a positive demand is computed for an argument, that this argument is effectively strict.

Definition 3.26

f is strict in its l^{th} argument when:

$$\forall v_1, \ldots, v_n \cdot f(v_1, \ldots, v_n) \neq \bot \;\Rightarrow\; v_l \neq \bot$$

Theorem 3.10 states that the abstract demand propagation semantics presented earlier can correctly infer that certain arguments are strict. More precisely, if it is inferred that some argument has to be well-defined ($\delta_l = \$$) in order for f_i to produce a well-defined result (\$), then f_i is effectively strict in that argument.

Theorem 3.10 *(Correct detection of strictness)*

$$\forall p \cdot \quad \widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\delta_j \mid 1 \le j \le n_i] \wedge \delta_l = \$ \Rightarrow f_i \text{ is strict in its } l^{th} \text{ argument}$$

Proof: Straightforward using the fact that $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$ and *is-inverse*$(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$. Cf. App. H.2.1, p. 292. □

Negative demands as indication of not being needed

Finally, a similar claim can be made about the detection of arguments which are sure not to be needed. As presented in Defn. 3.27, an argument is sure not to be needed if the function produces a well-defined result whether the argument is defined or not:

Definition 3.27

f does not need its l^{th} argument when:

$$\forall v_1, \ldots, v_n \cdot f(v_1, \ldots, v_n) \neq \bot \Rightarrow f(v_1, \ldots, v_{l\text{-}1}, \bot, v_{l+1}, \ldots, v_n) \neq \bot$$

As for Theorem 3.11, it essentially states that if the abstract semantics infers that some argument does not need to be well-defined ($\delta_l = @$) in order for f_i to produce a well-defined result (\$), then that argument is sure not to be needed.

Theorem 3.11 *(Correct detection of non-needed arguments)*

$$\forall p \cdot$$
$$\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\delta_j \mid 1 \leq j \leq n_i] \wedge \delta_l = @$$
$$\Rightarrow$$
$$f_i \text{ does not need its } l^{th} \text{ argument}$$

Proof: Straightforward using the fact that $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$ and $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$. Cf. App. H.2.1, p. 293. □

3.2.8 Putting it all together: Take 2

An alternative strategy to tie everything together is to show that the abstract semantics is effectively an appropriate inverse of the standard semantics. The intuition behind this is motivated by Fact 3.1 (Section 3.1.2, p. 32), where it was seen that any superset of an inverse is also an inverse. From Theorem 3.5, we know that the answer produced by the exact demand semantics is an appropriate inverse; furthermore, we also know, from Theorem 3.7, that the exact demand semantics' result, when it is defined, is always contained in the answer produced by the abstract semantics, thus leading to the required result.

To formalize this intuition, we define revised versions of the *is-inverse* predicates relating directly the standard and abstract (non-standard) demand propagation semantics. In order to do this, we must first define a *conc*retization function for abstract demands not in terms of demands, i.e., of the non-standard semantics, but instead in terms of standard values, i.e., of the standard semantics. This is straightforward: the concretization set associated with abstract demand δ is simply the union of the concretization sets of all the demands d which are safely approximated by δ:

Definition 3.28 *(Abstract demand concretization)*

$$conc : ADmd \rightarrow \mathcal{P}(Value)$$
$$conc(\delta) \;\triangleq\; \bigcup_{d \lhd \delta} conc(d)$$

Clearly, the following property then holds, i.e., the concretization set of an exact result is contained in the concretization set of any safe abstract approximation of it:

Fact 3.12

$$[d_1, \ldots, d_k] \lhd [\delta_1, \ldots, \delta_k] \;\Rightarrow\; conc(d_j) \subseteq conc(\delta_j)$$

Proof: Straightforward, from the fact that $d_j \lhd \delta_j$ (by defn. of $\lhd$ for vectors). □

Using this new concretization function, we can now define revised versions of the *is-inverse* predicates appropriate for the abstract semantics (cf. Defn. 3.15 and 3.16, p. 50 for the original definitions). These are presented in Definitions 3.29 and 3.30. The first of these definitions, which determines whether an abstract demand $\widetilde{\delta}$ is an inverse relative to expression e when demanded with exact demand d, is really identical to the original definition, except for the *conc* function which now maps abstract demands to sets of (standard) values.

Definition 3.29

$$is\text{-}inverse_{bve,fve} : Expr \rightarrow Dmd \rightarrow ADmdVec \rightarrow Bve \rightarrow Fve \rightarrow \mathbb{B}$$

$$\begin{array}{l}
is\text{-}inverse_{bve,fve}(e, d, \widetilde{\delta}) \quad \triangleq \\
\quad bve = [v_j \mid 1 \leq j \leq k] \wedge \widetilde{\delta} = [\delta_j \mid 1 \leq j \leq k] \\
\quad \Rightarrow \\
\quad \forall v \in conc(d) \cdot \\
\quad\quad \mathcal{E}[\![e]\!]\; bve\; fve \sqsupseteq v \\
\quad\quad \Rightarrow \\
\quad\quad \exists u_1 \in conc(\delta_1), \ldots, u_k \in conc(\delta_k) \cdot \\
\quad\quad\quad \mathcal{E}[\![e]\!][u_j \mid 1 \leq j \leq k]\; fve \sqsupseteq v \\
\quad\quad\quad \wedge \\
\quad\quad\quad u_j \sqsubseteq v_j \;\; (j = 1, \ldots, k)
\end{array}$$

As for the second definition (Defn. 3.30), which relates standard and abstract non-standard function environments, the only difference with the original definition is that the singleton set $\{d\}$ is used as the demand for the result of the abstract semantics. Intuitively, the abstract function environment $\widetilde{fve}$ is an inverse of the standard environment fve if, given an appropriate f_i, the result obtained from $\widetilde{fve}$ for f_i using a (abstract) demand corresponding to the set containing only the exact demand d is an inverse of the expression consisting in a function call to f_i, when the result of that call is demanded with d.

Definition 3.30

$$is\text{-}inverse : Fve \rightarrow \widetilde{Fve} \rightarrow \mathbb{B}$$

$$\begin{array}{l}
is\text{-}inverse(fve, \widetilde{fve}) \quad \triangleq \\
\quad \forall f_i \in \mathsf{dom}\, fve \cdot \\
\quad\quad \forall d \in Dmd, v_1, \ldots, v_{n_i} \in Value \cdot \\
\quad\quad\quad is\text{-}inverse_{bve,fve}([\![f_i(x_1, \ldots, x_{n_i})]\!], d, \widetilde{fve}[\![f_i]\!]\,\{d\}) \\
\quad\quad\quad\quad \text{where } bve = [v_j \mid 1 \leq j \leq n_i]
\end{array}$$

From these new definitions, the following theorem can then be stated and proved, i.e., the abstract semantics is effectively an inverse of the standard semantics:

Theorem 3.13 *(Soundness of abstract demand propagation)*

For any program p:

$$is\text{-}inverse(fve_{[\![p]\!]}, \widetilde{fve}_{[\![p]\!]})$$

Proof: The proof is relatively straightforward, using the fact that $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$ and $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$ hold, together with Theorem 3.9 (Detection of non-termination). For more detail, cf. App. H.2.1, p. 293. □

Using the above, it would then be a simple exercise (left to the reader) to show that a positive, resp. negative, demand computed for an argument means that the argument is effectively strict, resp. not needed, as was done in Theorems 3.10 and 3.11.

3.3 Abstract demand propagation: Data structures

In this section, we present once more the concepts that were introduced in Section 3.2 but this time for a language allowing for arbitrary user-defined algebraic types (data structures). Of course, the resulting domain of demands and associated semantics will be slightly more complex, especially the abstract version as it will introduce a notion of *cyclic* demands whose representation will require the introduction of a global state (*demands environment*) into the specification. Thus, although the exact demand propagation semantics and the associated soundness theorem will be presented in detail, the details for the abstraction of the demand propagation semantics will only be presented in App. B, where the notions of demand environments and demand environment transformers will be introduced. However, Section 3.3.5 will still present the major theorem concerning the safety of the abstraction.

3.3.1 The source language and its standard semantics

The source language

The source language is a functional language without higher-order functions but with arbitrary user-defined algebraic types. Its abstract syntax is presented in Figure 3.13. Like for the scalar language of Section 3.2, we assume we are dealing with lambda-lifted function definitions.

The language uses a simplified form of case-expressions in order to better illustrate the idea of demand propagation through data structure selectors. In this simplified form of case-expressions, no pattern-matching is allowed. Instead, the

$p \in Prog$	Programs	where
	$p ::=$	$\{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$
$e, e_i \in Expr$	Expressions	where
	$e ::=$	$c \mid$
		$x_i \mid$
		$e_1 + e_2 \mid$
		$f_i(e_1, \ldots, e_{n_i}) \mid$
		$K_{ij}\, e_1 \ldots e_{a_{ij}} \mid$
		$K\_l(e) \mid$
		$\{\texttt{case}\ e_0\ \texttt{of}\ K_j \Rightarrow e_j; \ldots\}$
$c \in Const$	Scalar constants, i.e., integers	
$x_i \in VarName$	Variable names	
$f, f_i \in FuncName$	Function names	
$K, K_{ij} \in ConsName$	Constructor names	

Figure 3.13: Abstract syntax of the source language

case is given an expression e_0 used to select among a number of branches. Each branch consists of a guard and an associated expression; a guard simply indicates the name of a data structure constructor (K_j) and it matches when expression e_0 evaluates to a data structure built with the indicated constructor. Selectors of the form $K\_l(e_i)$ can then be used within the branch to select the required component, in this case the l^{th} component. Note that such selectors can be used anywhere within the program, not only within properly guarded case-branches. However, unless they are protected by the appropriate guard, there is always the possibility that a run-time variant conflict occurs, i.e., trying to select a component from an improper variant (an example of this is presented below).

Note that if-expressions are not used, contrary to our earlier language. Instead, we assume that case-expressions can have guards which use the predefined constructors *True* and *False*, an example of which is also presented below (*fromTo*).

Type definitions

Although this is not indicated in the abstract syntax of Fig. 3.13, we assume that new data types can be defined and used by a program. A type definition, which defines an *algebraic type*, explicitly specifies a set of constructors for objects of the type, where each constructor is composed from 0 or more elements. For example, suppose we are given the following type definition:

$$\texttt{Type}\ t_i = K_{i1} t_{i1}^1 \ldots t_{i1}^{a_{i1}} \ \mid \ \ldots \ \mid \ K_{ij} t_{ij}^1 \ldots t_{ij}^{a_{ij}} \ \mid \ \ldots \ \mid \ K_{im_i} t_{im_i}^1 \ldots t_{im_i}^{a_{im_i}}; \tag{3.12}$$

This definition introduces a new type t_i having m_i variants ($m_i \geq 1$) with constructors $K_{i1}, \ldots, K_{im_i}$. Constructor K_{ij} is of arity a_{ij}, i.e., is made of a_{ij} components ($a_{ij} \geq 0$), whose types are given by $t_{ij}^1, \ldots, t_{ij}^{a_{ij}}$. When such a type is defined, a number of related functions and constants are automatically introduced for use within the program:

- Constructor function K_{ij} used to construct values of variant j of type t_i.
- Selector function $K_{ij}\_l(\_)$ which, given a value of variant j of type t_i, returns its l^{th} component (otherwise it returns $\perp$).
- Discriminator K_{ij} to be used in guards of case-expressions.

Some examples are presented in Fig. 3.14. Function *fromTo* illustrates the use of a case-expression to represent a simple conditional. As for function *sum* and *sum_inf*, they both compute the sum of the elements of a list. However, there is a major difference between the two: in the case of *sum*, no run-time variant conflict can occur; on the other hand, in the case of *sum_inf*, the argument l must definitively be an infinite list, otherwise a run-time error will occur when the program reaches the end of the list and tries to fetch the first and second *Cons* component of a *Nil* variant. For example, a call to *sum_inf*(*fromTo*(1, 100)) would produce such an error.

The standard semantics

Within the standard semantics, non-$\perp$ data structures will be represented by values from the domain *Struct*, which is presented in Fig. 3.15. A non-$\perp$ data structure will be represented by a composite object whose *cons* slot will indicate the associated variant (tag) and whose *components* slot will be a sequence of values representing the components of the data structure object. Note that for this to work correctly, we assume that the source program has been properly type-checked and, furthermore, that any given constructor name is used only within a single type definition.

The domains and functions defining the standard semantics are presented in Fig. 3.16. The ordering on the *Value* domain is given by Defn. 3.31: it is the standard

$$
\begin{array}{l}
\text{Type } list \ = \ Nil \mid Cons\ N\ list; \\
\text{Def } fromTo(l,\ h) = \\
\quad \{\texttt{case}\ l > h\ \texttt{of} \\
\quad\quad True \Rightarrow Nil; \\
\quad\quad False \Rightarrow Cons\ l\ fromTo(l+1,\ h); \}; \\
\text{Def } sum(l) = \\
\quad \{\texttt{case}\ l\ \texttt{of} \\
\quad\quad Nil \Rightarrow 0; \\
\quad\quad Cons \Rightarrow Cons\_1(l) + sum(Cons\_2(l)); \}; \\
\text{Def } sum\_inf(l) = \\
\quad Cons\_1(l) + sum\_inf(Cons\_2(l));
\end{array}
$$

Figure 3.14: Some examples of programs illustrating the source language with data structures

$$
\begin{array}{lll}
Struct :: cons & : & ConsName \\
\quad\quad\ \ components & : & Value^*
\end{array}
$$

Figure 3.15: Representation of data structures within the standard semantics

ordering for data structures, i.e., $\perp$ is less defined than anything, an integer value is less defined itself, and a data structure is less defined than another one if they have the some constructor and the corresponding components are less defined. [20]

Definition 3.31 *(Ordering between standard values)*

$$\sqsubseteq\ : Value \rightarrow Value \rightarrow \mathbb{B}$$

$$
\begin{array}{l}
v_1 \sqsubseteq v_2 \quad \triangleq \\
\quad v_1 = \perp \\
\quad \vee \\
\quad v_1 \in \mathbb{Z} \wedge v_2 \in \mathbb{Z} \wedge v_1 = v_2 \\
\quad \vee \\
\quad v_1 \in Struct \wedge v_2 \in Struct \wedge cons(v_1) = cons(v_2) \wedge \\
\quad\quad \forall j \in \mathsf{inds}\ components(v_1) \cdot components(v_1)(j) \sqsubseteq components(v_2)(j)
\end{array}
$$

Except for the handling of data structure constructs, the semantics functions are essentially the same as those for scalars presented in Fig. 3.3 (p. 37). As for

[20] Given a sequence s, the operator $\mathsf{inds}\ s$ returns the set of indices of s, i.e., the set $\{1, \ldots, \mathsf{len}\ s\}$. As for the *components* function, it is implicitly introduced by the record definition presented in Fig. 3.15 to select the indicated record component.

data structure constructs, their handling is rather straightforward. In the case of a constructor $[\![K_{ij}\, e_1 \cdots e_{a_{ij}}]\!]$, an appropriate value is constructed using *mk-Struct*. For a selector $[\![K\_l(e)]\!]$, the tag is checked and, if it matches, the required component is selected, otherwise $\perp$ is returned. Finally, for a case-expression $[\![\{\texttt{case}\ e_0\ \texttt{of}\ K_{ij} \Rightarrow e_j; \ldots\}]\!]$, the selector expression (e_0) is evaluated and a dispatch is made based on the associated tag $(cons(v))$, returning $\perp$ if none of the guard matches. Note that we assume that the *cons* selector, when given an integer value, returns the *False* tag when the value is 0, otherwise it returns *True* (i.e., *à la C*).

3.3.2 The domain of exact demands for data structures

The domain of demands for data structures, much like the one for scalar demands, will include both negative and positive demands. However, positive demands will be much more varied, in order to account for the various data types that can be defined and used by the program.

The domain of exact demands for data structures is presented in Fig. 3.17. The names used in that figure stand for the following abbreviations:

- *EDmd*: Exact Demand;
- *ENDmd*: Exact Negative Demand;
- *EPDmd*: Exact Positive Demand.

Positive demands are constructed from a constructor name and a sequence of exact demands; we assume that all demands are constructed using sub-demands of the proper type, as specified by the original type definition in which the constructor is defined. Like for scalar values, we assume \$ denotes an appropriate constructor name for scalars, i.e., it denotes a demand for an arbitrary integer. As an abbreviation, we will also use \$ to denote the positive demand *mk-EPDmd*(\$, [ ]).

Note the domain of exact demands is a discrete domain, as any two distinct demands are incomparable. Also note that infinite demand values are not allowed, i.e., all demands are finite. This stems from the interpretation that we want to give to the exact demand semantics: it should compute the demands that have to be made for the arguments *when* the demanded result has been *fully produced*. [21]

[21] This does not mean potentially infinite data structures will not be represented: an infinite structure will be handled, in the abstract semantics, using an abstract demand representing an infinite set containing all the possible finite *prefix* of this structure.

$$Value = \{\perp\} + \mathbb{Z} + Struct$$

$$\begin{array}{lll} Struct :: & cons & : ConsName \\ & components & : Value^* \end{array}$$

$$Bve = Value^*$$

$$Fve = FuncName \xrightarrow{m} Value^* \to Value$$

$$\mathcal{M} : Prog \to Fve$$

$$\mathcal{M}[\![p]\!] \;\triangleq\; \mu\, fve \cdot \{f_i \mapsto \lambda(z_1, \ldots, z_{n_i}) \cdot \mathcal{E}[\![e_i]\!][z_j \mid 1 \leq j \leq n_i]\, fve\}$$
$$\text{where } p = \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$$

$$\mathcal{E} : Expr \to Bve \to Fve \to Value$$

$$\begin{array}{l}
\mathcal{E}[\![e]\!]\; bve\; fve \;\triangleq \\
\quad \begin{array}{ll}
\textsf{cases } [\![e]\!] \textsf{ of} & \\
[\![c]\!] & \to c \\
[\![x_i]\!] & \to bve(i) \\
[\![e_1 + e_2]\!] & \to (\mathcal{E}[\![e_1]\!]\; bve\; fve) + (\mathcal{E}[\![e_2]\!]\; bve\; fve) \\
[\![f_i(e_1, \ldots, e_{n_i})]\!] & \to fve[\![f_i]\!](\mathcal{E}[\![e_1]\!]\; bve\; fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\; bve\; fve) \\
[\![K_{ij}\, e_1 \cdots e_{a_{ij}}]\!] & \to mk\text{-}Struct(K_{ij}, [\mathcal{E}[\![e_l]\!]\; bve\; fve \mid 1 \leq l \leq a_{ij}]) \\
[\![K_l(e)]\!] & \to \textsf{let } v = \mathcal{E}[\![e]\!]\; bve\; fve \textsf{ in} \\
& \quad \textsf{if } cons(v) = K \\
& \quad \textsf{then } components(v)(l) \\
& \quad \textsf{else } \perp \\
[\![\{\texttt{case}\; e_0\; \texttt{of}\; K_{ij} \Rightarrow e_j; \ldots\}]\!] & \to \textsf{cases } cons(\mathcal{E}[\![e_0]\!]\; bve\; fve) \textsf{ of} \\
& \quad K_{ij} \quad \to \mathcal{E}[\![e_j]\!]\; bve\; fve \\
& \quad \ldots \\
& \quad \textsf{otherwise} \to \perp \\
& \quad \textsf{end} \\
& \\
\textsf{end} &
\end{array}
\end{array}$$

Figure 3.16: Standard semantics for language with data structures

$$EDmd = ENDmd + EPDmd$$

$$ENDmd = \{@\}$$

$$\begin{array}{lll} EPDmd :: & cons & : ConsName \\ & components & : EDmd^{*} \end{array}$$

Figure 3.17: The domains for exact demands (data structures)

The merge ($\oplus$) operation

The definition of the $\oplus$ operator for data structures is presented in Defn. 3.32. Note that it requires, as a pre-condition, that the demands to be merged be *compatible* (cf. below for the definition of $\sim$). [22] As before, @ acts as an identity of $\oplus$, whereas the merge of data structures is done component-wise, the fact that both data structures have the same constructor being ensured by their compatibility.

Definition 3.32 *(Merge of exact demands)*

$$\oplus : EDmd \rightarrow EDmd \rightarrow EDmd$$

$$\begin{array}{l} ed_1 \oplus ed_2 \quad \triangleq \\ \quad \textsf{if } ed_1 = @ \\ \quad \textsf{then } ed_2 \\ \quad \textsf{else if } ed_2 = @ \\ \qquad \textsf{then } ed_1 \\ \qquad \textsf{else let } k = \textsf{len } components(ed_1) \textsf{ in} \\ \qquad\quad \textsf{let } comps = [components(ed_1)(j) \oplus components(ed_2)(j) \mid 1 \leq j \leq k] \textsf{ in} \\ \qquad\quad mk\text{-}EPDmd(cons(ed_1), comps) \end{array}$$

$$\textsf{pre } ed_1 \sim ed_2$$

The notion of compatibility between exact demands is defined in Defn. 3.33: $ed_1 \sim ed_2$ indicates whether two exact demands are compatible, i.e., can be merged without any conflict. Essentially, a negative demand is compatible with any other demands, as it imposes no constraints; as for two positive demands, they are compatible when they both describe demands for data structures having the same constructor and for which the subcomponents are also compatible.

[22] And see Section A.8 (p. 202) for a description of the use of pre- and post- conditions in the specification of functions.

Definition 3.33 *(Compatibility between exact demands)*

$$\sim\, : EDmd \to EDmd \to \mathbb{B}$$

$$\begin{aligned}
&ed_1 \sim ed_2 \quad \triangleq \\
&\qquad ed_1 = @ \vee \\
&\qquad ed_2 = @ \vee \\
&\qquad cons(ed_1) = cons(ed_2) \wedge \\
&\qquad\quad \forall j \in \mathsf{inds}\, components(ed_1) \cdot components(ed_1)(j) \sim components(ed_2)(j)
\end{aligned}$$

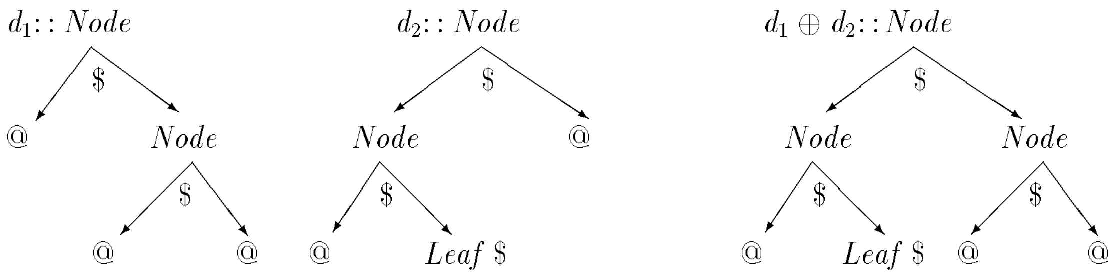
Below is a small example illustrating demands for data structures and demand compatibility. Suppose we have the following type definition that defines a binary tree type where each variant may contain an integer component:

```
Def tree = Leaf N | Node tree N tree;
```

Then, demands d_1 and d_2 presented below would effectively be compatible, i.e., $d_1 \sim d_2$ would hold, so that they could effectively be merged:

$$d_1 = mk\text{-}EPDmd(Node, [@, \$, mk\text{-}EPDmd(Node, [@, \$, @])])$$
$$d_2 = mk\text{-}EPDmd(Node, [mk\text{-}EPDmd(Node, [@, \$, mk\text{-}EPDmd(Leaf, [\$])]), \$, @])$$

Graphically, these two demands, together with their merged result, could be represented by the following:



Auxiliary functions

Two auxiliary functions will be useful, within the exact demand propagation semantics, to construct new demands. The first function ($mkVariant_{ED}$) creates a demand appropriate for a case-expression guard. When a such a guard is known to apply, we can then be sure that the associated object must have the indicated tag, although nothing is yet known about its components. Thus, given a tag K, the function $mkVariant_{ED}$ presented in Defn. 3.34 will construct the appropriate demand, returning a scalar demand (\$) in the special case of a *True* or *False* tag:

Definition 3.34 *(Demand for case-expression guard)*

$$mkVariant_{ED} : ConsName \rightarrow EDmd$$

$$\begin{array}{l} mkVariant_{ED}\ K \quad \triangleq \\ \quad \textsf{if}\ K \in \{True, False\} \\ \quad \textsf{then}\ mk\text{-}EPDmd(\$, [\,]) \\ \quad \textsf{else}\ mk\text{-}EPDmd(K, [@ \mid 1 \le j \le arity(K)]) \end{array}$$

The second function ($mkComponent_{ED}$) is used when a selector is applied to an expression. Supposing that the l^{th} component of a K variant is demanded with d, then the resulting demand for the whole expression must surely have the required tag (K) and its l^{th} slot must satisfy d, whereas no constraints are put on the remaining components. Defn. 3.35 presents function $mkComponent_{ED}$:

Definition 3.35 *(Demand for data structure selector)*

$$mkComponent_{ED} : ConsName \rightarrow \mathbb{N}_1 \rightarrow EDmd \rightarrow EDmd$$

$$\begin{array}{l} mkComponent_{ED}\ K\ l\ d \quad \triangleq \\ \quad \textsf{let}\ ds = [@ \mid 1 \le j < l] \curvearrowright [d] \curvearrowright [@ \mid l < j < arity(K)]\ \textsf{in} \\ \quad mk\text{-}EPDmd(K, ds) \end{array}$$

$$\textsf{pre}\ 1 \le l \le arity(K)$$

3.3.3 An exact non-standard semantics for demand propagation

The domain of demand vectors for the exact demand propagation semantics

As for the non-standard semantics presented in Section 3.2.3, the semantics for data structures also deals with vectors of (exact) demands. The semantic domains for such vectors are presented in Figure 3.18.

$$EDmdVec_k = \{[d_1, \ldots, d_k] \mid d_i \in EDmd\}_\perp$$

$$EDmdVec = \textstyle\sum_{k \geq 1} EDmdVec_k \quad \text{(coalesced sum)}$$

Figure 3.18: The domains for vectors of demands used in the exact demand propagation semantics

As before, the basic merge operator has to be extended to deal with vectors of demands. This is presented in Defn. 3.36, where the fact that non-compatible demands must not be used as arguments to $\oplus$ is taken care of: if such incompatible demands are present, i.e., no result could satisfy the two sets of constraints represented by the two vectors of demands, then $\perp$ is returned as the result. [23]

Definition 3.36 *(Merge of vectors of exact demands)*

$$\overline{\oplus} : DmdVec_k \rightarrow DmdVec_k \rightarrow DmdVec_k$$

$$\begin{array}{l} \overline{d} \mathbin{\overline{\oplus}} \overline{e} \quad \triangleq \\ \qquad \mathsf{let}\ [d_1\ \cdots\ d_k] = \overline{d}\ \mathsf{in} \\ \qquad \mathsf{let}\ [e_1\ \cdots\ e_k] = \overline{e}\ \mathsf{in} \\ \qquad \mathsf{if}\ \forall j \in \{1, \ldots, k\} \cdot d_j \sim e_j \\ \qquad \mathsf{then}\ [d_j \oplus e_j \mid 1 \leq j \leq k] \\ \qquad \mathsf{else}\ \perp \end{array}$$

The exact demand propagation semantics

The definitions of the exact non-standard semantic functions are presented in Figures 3.19, 3.20 and 3.21. The semantic function for programs is the same as for

[23] Again, remember that, from the semantics of pattern-matching and let-expressions, the result will also be $\perp$ if either one of the argument is $\perp$.

$$\overline{Fve} = FuncName \xrightarrow{m} Dmd \rightarrow Value^* \rightarrow EDmdVec$$

$$\overline{\mathcal{M}} : Prog \rightarrow \overline{Fve}$$

$$\overline{\mathcal{M}}[\![p]\!] \;\triangleq\; \mu\, \overline{fve} \cdot \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \overline{\mathcal{E}}_{n_i}[\![e_i]\!]\, d\, [z_j \mid 1 \le j \le n_i]\, fve_{[\![p]\!]}\, \overline{fve}\}$$
$$\textsf{where } p \;=\; \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}$$

Figure 3.19: Exact demand propagation semantics for programs with data structures

$$\overline{\mathcal{E}}_k : Expr \rightarrow EDmd \rightarrow Bve \rightarrow Fve \rightarrow \overline{Fve} \rightarrow EDmdVec_k$$

$$\overline{\mathcal{E}}_k[\![e]\!]\; d\; bve\; fve\; \overline{fve} \;\;\triangleq$$
$$\textsf{if } d = @$$
$$\textsf{then } [@ \mid 1 \le i \le k]$$
$$\textsf{else } \overline{\mathcal{E}}^+_k[\![e]\!]\; d\; bve\; fve\; \overline{fve}$$

Figure 3.20: Exact demand propagation semantics for expressions #1 (data structures)

scalars; only the functions for expressions differ. First of all, note that, for layout reasons (i.e., in order that the definition can fit properly on a single page) the semantic function for expressions has been splitted into two function: $\overline{\mathcal{E}}_k$ and $\overline{\mathcal{E}}^+_k$. The first function, $\overline{\mathcal{E}}_k$ (Fig. 3.20), really represents the top-level function for expressions; it can receive an arbitrary demand, positive or negative, and explicitly handles the case where the demand is negative, otherwise, i.e., when the demand is known to be positive, it dispatches to $\overline{\mathcal{E}}^+_k$. This latter function (Fig. 3.21) is only used when the demand is known to be positive. It is quite similar to the part of the definition for scalars which deals with a positive demand (presented in Fig. 3.7) except for additional elements related to the handling of data structures (described below) and for the fact that the demand made for a scalar constant or expression ($[\![c]\!]$ or $[\![e_1 + e_2]\!]$) must definitely be a scalar demand ($), otherwise this is considered unacceptable and $\perp$ is returned. Note that, within $\overline{\mathcal{E}}^+_k$, all subexpressions of expression e are always analyzed using $\overline{\mathcal{E}}_k$; although in some cases $\overline{\mathcal{E}}^+_k$ could be called directly because the demand d is already known to be positive, e.g., when analyzing the e_j of a binary expression or case-expression, recursively calling the top-level function $\overline{\mathcal{E}}_k$ simplifies the inductive argument used for the soundness proof (cf. Section 3.3.4).

$$\overline{\mathcal{E}}_k^+ : Expr \to EDmd \to Bve \to Fve \to \overline{Fve} \to EDmdVec_k$$

$$\overline{\mathcal{E}}_k^+ [\![e]\!]\; d\; bve\; fve\; \overline{fve} \quad \triangleq$$

$$\begin{array}{ll}
\textsf{cases } [\![e]\!] \textsf{ of} & \\
[\![c]\!] & \to \textsf{if } d = \$ \\
& \quad \textsf{then } [@ \mid 1 \le i \le k] \\
& \quad \textsf{else } \bot \\
[\![x_i]\!] & \to [@ \mid 1 \le j < i] \curvearrowright [d] \curvearrowright [@ \mid i < j \le k] \\
[\![e_1 + e_2]\!] & \to \textsf{if } d = \$ \\
& \quad \textsf{then let } \overline{d}_l = \overline{\mathcal{E}}_k [\![e_l]\!]\; d\; bve\; fve\; \overline{fve} \quad (l = 1, 2) \textsf{ in} \\
& \qquad \overline{\bigoplus} \{\overline{d}_1, \overline{d}_2\} \\
& \quad \textsf{else } \bot \\
[\![f_i(e_1, \ldots, e_{n_i})]\!] & \to \textsf{let } [d_1, \ldots, d_{n_i}] = \overline{fve} [\![f_i]\!]\; d\; (\mathcal{E}[\![e_1]\!]\; bve\; fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\; bve\; fve) \textsf{ in} \\
& \quad \textsf{let } \overline{d}_l = \overline{\mathcal{E}}_k [\![e_l]\!]\; d_l\; bve\; fve\; \overline{fve} \quad (l = 1, \ldots, n_i) \textsf{ in} \\
& \quad \overline{\bigoplus} \{\overline{d}_l \mid 1 \le l \le n\} \\
[\![K_{ij}\, e_1 \cdots e_{a_{ij}}]\!] & \to \textsf{if } cons(d) = K_{ij} \\
& \quad \textsf{then let } mk\text{-}EPDmd(K_{ij}, [d_1, \ldots, d_{a_{ij}}]) = d \textsf{ in} \\
& \qquad \textsf{let } \overline{d}_l = \overline{\mathcal{E}}_k [\![e_l]\!]\; d_l\; bve\; fve\; \overline{fve} \quad (l = 1, \ldots, a_{ij}) \textsf{ in} \\
& \qquad \overline{\bigoplus} \{\overline{d}_l \mid 1 \le l \le a_{ij}\} \\
& \quad \textsf{else } \bot \\
[\![K_{ij}_l(e)]\!] & \to \textsf{let } d' = mkComponent_{ED}\; K_{ij}\; l\; d \textsf{ in} \\
& \quad \overline{\mathcal{E}}_k [\![e]\!]\; d'\; bve\; fve\; \overline{fve} \\
[\![\{\texttt{case } e_0 \texttt{ of } K_{ij} \Rightarrow e_j; \ldots\}]\!] & \to \textsf{cases } cons(\mathcal{E}[\![e_0]\!]\; bve\; fve) \textsf{ of} \\
& \quad K_{ij} \quad \to \textsf{let } d^v = mkVariant_{ED}\; K_{ij} \textsf{ in} \\
& \qquad\qquad \textsf{let } \overline{d^{e_0}} = \overline{\mathcal{E}}_k [\![e_0]\!]\; d^v\; bve\; fve\; \overline{fve} \textsf{ in} \\
& \qquad\qquad \textsf{let } \overline{d^{e}} = \overline{\mathcal{E}}_k [\![e_j]\!]\; d\; bve\; fve\; \overline{fve} \textsf{ in} \\
& \qquad\qquad \textsf{let } \overline{d} = \overline{\bigoplus} \{\overline{d^{e_0}}, \overline{d^{e}}\} \textsf{ in} \\
& \qquad\qquad \overline{d} \\
& \quad \ldots \\
& \quad \textsf{otherwise} \to \bot \\
& \quad \textsf{end} \\
\textsf{end} &
\end{array}$$

Figure 3.21: Exact demand propagation semantics for expressions #2 (data structures)

As for $\overline{\mathcal{E}}_k^+$ (Fig. 3.21), when a constructor is encountered $[\![K_{ij}\, e_1 \cdots e_{a_{ij}}]\!]$, we first check whether the demand d matches, i.e., uses the same constructor ($cons = K_{ij}$). If so, similar to a function call, we propagate the appropriate demand to each of the argument ($\overline{d_l} = \overline{\mathcal{E}}_k[\![e_l]\!]\, d_l \ldots$) and then merge the resulting set of vectors. If the constructor does not match, then the program is sure not to satisfy the demand, so $\perp$ is returned. Note that, in order to handle nilary constructors — i.e., those with 0 arguments — we assume (within $\overline{\mathcal{E}}_k$) that $\overline{\bigoplus}\,\{\,\}$ (i.e., the $\overline{\bigoplus}$ of an empty set) returns $[@ \mid 1 \leq j \leq k]$, so the proper demand will be obtained.

When a selector is encountered $[\![K_{ij}\_l(e)]\!]$, the demand for the result of the selector (d) is transformed into a demand of the proper variant (K_{ij}) with the d at the selected position (l), all other components being not needed ($d' = mkComponent_{ED}\, K_{ij}\, l\, d$) and that demand is then propagated further to e.

Finally, for a case-expression $[\![\{\texttt{case}\, e_0\, \texttt{of}\, K_{ij} \Rightarrow e_j; \ldots\}]\!]$, we refer to the standard semantics to determine which branch will effectively be executed. We then combine the demand made within the branch to be executed ($\overline{d^e}$) with the demand made by testing the guard and knowing it effectively holds ($\overline{d^{e_0}}$); this latter demand is obtained by creating a demand of the proper variant but with all components considered as not being needed ($mkVariant_{ED}\ K_{ij}$).

3.3.4 Soundness of the exact demand propagation semantics

Demands as sets of values

We now define a concretization function for exact demands, function which will relate the standard semantics and the exact demand propagation semantics. As before, @ represents the singleton set $\{\perp\}$, whereas \$ is associated with the whole set of integers; as for a data structure demand, it represents all the data structures having the indicated constructor and with components being in the concretization sets of the related exact demand components.

Definition 3.37 *(Concretization of exact demands)*

$$conc : EDmd \rightarrow \mathcal{P}(Value)$$

$$\begin{array}{l} conc(ed) \quad \triangleq \\ \quad \textsf{if } ed = @ \\ \quad \textsf{then } \{\bot\} \\ \quad \textsf{else if } ed = mk\text{-}EPDmd(\$, [\,]) \\ \qquad \textsf{then } \mathbb{Z} \\ \qquad \textsf{else let } mk\text{-}EPDmd(cons, [ed_1 \ \cdots \ ed_k]) = ed \ \textsf{ in} \\ \qquad\quad \{mk\text{-}Struct(cons, [v_1, \ldots, v_k]) \mid v_1 \in conc(ed_1), \ldots, v_k \in conc(ed_k)\} \end{array}$$

The notion of a value satisfying a demand is unchanged from the one given for scalars. We present it below as a reminder of its definition.

Definition 3.38 *(Demand satisfaction by a value)*

$$satisfy : Value \rightarrow EDmd \rightarrow \mathbb{B}$$

$$satisfy(v, d) \quad \triangleq \quad \exists u \in conc(d) \cdot v \sqsupseteq u$$

Important properties of $\oplus$

As for scalars, the following property, which is the crux of the soundness proof, still holds. The property states that the concretization set of the merge of d_1 and d_2 is composed of the least-upper bound of all compatible values which are respectively in the concretization sets of d_1 and d_2:

Fact 3.14

$$\begin{array}{l} \forall ed_1, ed_2 \cdot ed_1 \sim ed_2 \cdot \\ \quad conc(ed_1 \oplus ed_2) = \{v_1 \sqcup v_2 \mid v_1 \in conc(ed_1) \wedge v_2 \in conc(ed_2) \wedge v_1 \sim v_2\} \end{array}$$

Proof: Cf. Appendix H.2.2, p. 294. □

Given that this property holds, then the following fact also still holds, i.e., a value satisfies the merge of two demands if and only if it satisfies both demands:

Fact 3.15

$$\forall v, d_1, d_2 \cdot satisfy(v, d_1) \wedge satisfy(v, d_2) \ \Leftrightarrow \ satisfy(v, d_1 \oplus d_2)$$

The exact demand propagation semantics as an inverse of the standard semantics

The notions of inverse for programs involving data structures are given by exactly the same predicates as those for scalars, thus, they will not be repeated here (these predicates were defined in Section 3.2.4: for expressions, the definition was given by Defn. 3.15 p. 50, whereas for function environments it was given by Defn. 3.16 p. 51).

The claims about the relationships between the standard and exact non-standard semantics are also expressed by the same theorem and lemma, which are repeated below. The soundness theorem states that the function environment produced by the exact demand propagation semantics for a program p is an appropriate inverse of the function environment of the standard semantics.

Theorem 3.16 (Soundness of exact demand propagation semantics)

For any program p:

$$is\text{-}inverse(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$$

Proof: The proof is exactly the same as for scalars, i.e., fixed point induction using the lemma stating the soundness of expression semantics, in this case, Lemma 3.17 (cf. the proof for scalars, App. H.2.1, p. 275). □

Lemma 3.17 states that a similar relationship (being an appropriate inverse) holds for expressions, i.e., between the meaning obtained from the exact demand propagation semantics and the standard meaning, assuming that appropriate inverse function environments are provided:

Lemma 3.17 *(Soundness of expression semantics)*

$$\begin{array}{l} \forall e, d, bve, fve, \overline{fve} \cdot \\ \quad is\text{-}inverse(fve, \overline{fve}) \\ \quad\quad \Rightarrow \\ \quad is\text{-}inverse_{bve,fve}(e, d, \overline{\mathcal{E}}_k[\![e]\!]\ d\ bve\ fve\ \overline{fve}) \\ \quad\quad \text{where} \\ \quad\quad\quad k = \mathsf{len}\ bve \end{array}$$

Proof: Again, as for scalars, the proof is by structural induction. App. H.2.2, p. 295 presents the parts which differ from the proof for scalars. □

At this point, much like for scalars, we now have a notion of demand propagation for data structures that can be interpreted as a form of inverse computation. And, again, in order to incorporate this model of demand propagation into a compiler, so that definite termination is ensured, an abstraction of the demand propagation semantics has to be defined ... which leads us to our next section.

3.3.5 An abstraction of the exact demand propagation semantics

The next step in our process is to define an abstraction of the exact (non-standard) demand propagation semantics presented in Section 3.3.3 and then to show that this abstraction is safe. In order to do this, an abstract domain of demands for data structures has to be defined and then, using that domain of demands, an appropriate abstract semantics can be formalized. In the present section, the basic ideas and intuitions concerning the abstract version of the demand propagation semantics will be presented; the formal definitions and detailed explanations will be presented in Appendix B. Regarding safety, it will, like for scalars, be formally defined and the appropriate theorem stated (a detailed, although somewhat informal proof, being found in App. H.2.2).

Abstract demands, demand references and demand environments

The abstract domain for data structures is clearly more complex than the one used for scalars. As we saw in Section 3.2.5, abstract scalar demands corresponded to sets of exact scalar demands; this correspondence was straightforward to establish because the sets contained only a (small) finite number of elements. The situation for data structures, on the other hand, is different because data structures can be (potentially) infinite.

As explained earlier when describing the interpretation of the exact demand propagation semantics (Section 3.3.2, p. 72), an exact demand always has to be finite (although it can represent a demand for a non-finite result, e.g., @ is a finite demand, which can be satisfied by $\perp$). Now, when the result is potentially infinite, it should be acceptable to demand any finite prefix of the result. In other words, it should be possible for an abstract demand to represent an *infinite set* of exact finite demands. Furthermore, as we are dealing with a compile-time analysis, this representation should be finite. In the abstract demand semantics, such demands will be represented by *cyclic demands*.

Cyclic demands will be made possible by the use of abstract demand environments (σ, σ'), i.e., through the use of a global state. A demand environment σ can map an abstract demand reference (δ) to an abstract demand value, value which is a *set of basic* abstract *demands* (bd). Any such basic demand is either a negative demand or a positive one, i.e., a data constructor name together with appropriate demands for the components.

Let's illustrate the notion of cyclic demands with an example. A well-formed demand environment σ is presented in Fig. 3.22 (using a simplified notation), where each demand used as component of a data structure constructor represents a demand reference, i.e., which must be interpreted relative to environment σ. Graphically, this environment could be pictured as presented in Fig. 3.23.

$$\begin{array}{lll} \sigma = & \{ & \begin{array}{ll} \phi & \mapsto \{\,\} \\ @ & \mapsto \{@\} \\ \$ & \mapsto \{\$\} \\ HTS & \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ HTS\} \\ HS & \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ HS'\} \\ HS' & \mapsto \{@, \mathsf{Nil}, \mathsf{Cons}\ \$\ HS'\} \end{array} \\ & \} & \end{array}$$

Figure 3.22: An example of a demand environment

In this environment, *HTS* represents a head-and-tail strict demand, i.e., a demand which requires all heads and all tails of a list to be fully evaluated. As will be seen in Section B.6 (Defn. B.8, p. 225), *HTS* would then correspond to the following set of exact demands: [24]

$$HTS = \{Nil, Cons\,\$\,Nil, Cons\,\$\,(Cons\,\$\,Nil), Cons\,\$\,(Cons\,\$\,(Cons\,\$\,Nil)), \ldots\}$$

As for *HS*, it denotes a head-strict demand, i.e., a demand which, whenever a cons-cell is needed, also requires the evaluation of the head of the cell; it would correspond to the following set of exact demands: [25]

[24]Alternatively, *HTS* could be described, using the notation of Section 2, by the following set: $\{\$^k . Nil \mid k \geq 0\}$.

[25]That is, $HS = \{\$^k . Nil \mid k \geq 0\} \cup \{\$^k . @ \mid k \geq 1\}$.

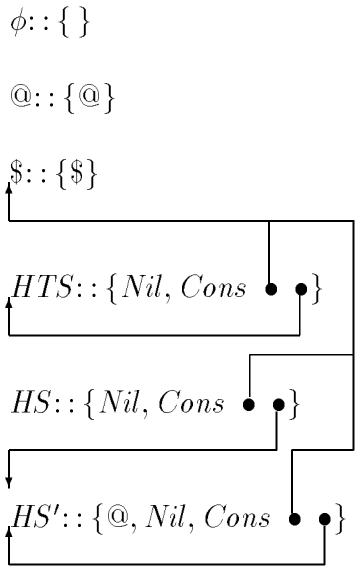


Figure 3.23: Graphical representation of demand environment σ

$$
\begin{aligned}
HS = \{&Nil, \\
&Cons \ \$ \ Nil, Cons \ \$ \ @, \\
&Cons \ \$ \ (Cons \ \$ \ Nil), Cons \ \$ \ (Cons \ \$ \ @), \\
&Cons \ \$ \ (Cons \ \$ \ (Cons \ \$ \ Nil)), Cons \ \$ \ (Cons \ \$ \ (Cons \ \$ \ @)), \ldots\}
\end{aligned}
$$

Ordering between abstract demands and between demand environments

Given a demand environment, an ordering between any two demand references can be defined as presented in App. B.1.2 (Definitions B.2, B.3 and B.4, p. 208). Informally, the ordering says that demand reference δ_1 is less defined or equal to demand reference δ_2 relative to environment σ ($\delta_1 \sqsubseteq_\sigma \delta_2$) if for any basic demand bd_1 in δ_1 (in σ) there is a basic demand at least as defined in δ_2, i.e., either a negative one if bd_1 is negative, or one with a similar constructor but with components at least as defined if bd_1 is a positive demand.

As for the ordering between demand environments, because of the way in which they will be built, i.e., incremental construction, we will simply have them ordered by the subset ordering on their domain, i.e., $\sigma_1 \sqsubseteq \sigma_2$ implies $\mathsf{dom}\,\sigma_1 \subseteq \mathsf{dom}\,\sigma_2$ (cf. Defn. B.5, p. 209).

Operations on abstract demands: Demand environment transformers

Using a demand environment to represent demands clearly brings some additional complexities:

- Any operation that examines an abstract demand, e.g., a predicate, must always be passed an additional argument representing the demand environment relative to which the demand to be examined is defined.
- Any operation that constructs a demand must *i)* receive a source demand environment; *ii)* produce a new demand environment which subsumes the initial one together with information appropriate for the newly constructed demand.

To facilitate the presentation and use of operations that refer to and modify the demand environment, the approach and syntax of *monads for state transformations* presented in [Wad92, Lau93] will be used. A short presentation of this approach, adapted for demand environments, is presented in Appendix C (p. 227).

Any operation using a demand environment and creating a new environment (i.e., updating it) will have its result type be an instance of the type of *Demand environment Transformers*, defined as follows:

Definition 3.39 *(Demand environment transformer)*

$DT\ t = ADmdEnv \rightarrow t \times ADmdEnv$

A demand environment transformer of type $DT\ t$ is thus a function which, given a demand environment, produces a value of type t paired with a (possibly) updated demand environment. [26]

The merge ($\oplus$) operator The exact definition of the merge ($\oplus$) operator for abstract demands is presented in Appendix B.2.2, p. 213. Simply note that, as $\oplus$ may produce a new demand and thus create an updated environment, it must be defined as a demand environment transformer. Thus, its heading is given by the following type:

$$\oplus : ADmdRef \rightarrow ADmdRef \rightarrow DT\ ADmdRef$$

[26] Cf. Defn B.6, p. 209 for a more detailed explanation, together with a specification of appropriate constraints on such environments.

Note that an important characteristic of this abstract version of the $\oplus$ operator is that it is safe, i.e., the abstract result is always a safe approximation of the exact one (cf. Section 3.3.6, p. 89 for the exact notion of safe approximation).

The union ($\cup$) operator The union ($\cup$) operator is used to combine demands coming from different branches of conditionals, i.e., case-expressions. Its exact definition is presented in Appendix B.2.2, p. 211. Its type is also that of a demand environment transformer:

$$\cup : ADmdRef \to ADmdRef \to DT\ ADmdRef$$

Also, like for the $\oplus$ operator, $\cup$ will effectively produce a safe result, in this case, an upper-bound of its arguments (cf. Fact H.31, p. 302).

Note that $\oplus$ and $\cup$ are really distinct operators: $\oplus$ preserves as much certainty as possible whereas $\cup$, being an upper-bound, always gives precedence to uncertainty. As an example, suppose we are given the following demands (TS = tail-strict; HS = head-strict):

$$\begin{array}{lll} TS & \mapsto & \{\mathsf{Nil}, \mathsf{Cons}\ @\ TS\} \\ HS & \mapsto & \{\mathsf{Nil}, \mathsf{Cons}\ \$\ HS'\} \\ HS' & \mapsto & \{@, \mathsf{Nil}, \mathsf{Cons}\ \$\ HS'\} \end{array}$$

Then the following demands would be obtained using the above operations, i.e., none of the tails of $HS \oplus TS$ are delayed, whereas all tails of $HS \cup TS$ get delayed (and the first head of $HS \oplus TS$ need not be delayed either):

$$\begin{array}{lllll} HS \oplus TS & = & TS' & \mapsto & \{\mathsf{Nil}, \mathsf{Cons}\ \$\ TS''\} \\ & & TS'' & \mapsto & \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ TS''\} \\ & & \hat{}\$ & \mapsto & \{@, \$\} \\ HS \cup TS & = & S & \mapsto & \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ S'\} \\ & & S' & \mapsto & \{@, \mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ S'\} \end{array}$$

Auxiliary operations A number of auxiliary operations are needed in order to define the abstract semantics. These are presented in more detail in Section B.2.3 (p. 214). Most of these operations are simply analog of operations found in the exact demand propagation semantics, but adapted to deal with abstract values, thus, with demand environments. Also, as for the $\oplus$ and $\cup$ operators, these all produce results which are effectively safe (cf. App. H.3.1, Facts H.37, H.38, H.39).

Extending the domain to vectors of demands

As in the exact demand semantics, we will be computing vectors, now composed of abstract demands. The domain for demand vectors is defined in Fig. B.2 (App. B.3, p. 217). The ordering between vectors of demands is simply the pointwise extension of the basic abstract demands ordering.

The merge and union ($\oplus$ and $\cup$) operators also have to be extended to deal with vectors of abstract demands ($\overline{\oplus}$ and $\tilde{\cup}$): these will simply be defined as the element-wise extension of $\oplus$ and $\cup$ (taking into account the need to manipulate the demand environment). The definitions are presented in Appendix B.3, p. 217.

The abstract demand propagation semantics

The definitions for the abstract semantics are presented in Fig. B.3, B.4 and B.5 in App. B.4. These semantic functions are explained in more detail in App. B.4, although we briefly present them here.

The meaning of a program p — $\widetilde{\mathcal{M}}[\![p]\!]$, denoted by $\widetilde{fve}_{[\![p]\!]}$ — (Fig. B.3, p. 218) is still a function environment; however, such an environment now associates to function name f_i defined in p a function which, given an abstract demand, produces a demand environment transformer of type $DTADmdVec$, i.e., encapsulating a result which is a demand vector. In other words, the meaning of a function is a function which, given an abstract demand reference and a demand environment, produces a vector of abstract demands paired with an updated demand environment.

Much like the exact non-standard semantics for data structures (cf. Figs. 3.20 and 3.21), the meaning of expressions is given by two functions: $\widetilde{\mathcal{E}}_k$ (Fig. B.4, p. 219), which is the top-level function that deals with any demand, and $\widetilde{\mathcal{E}}_k^+$ (Fig. B.5, p. 220), which only deals with positive demands. The latter function is quite similar to the one for the exact demand semantics, although demand environment transformers operations are now used. The major difference is the semantics of a case-expression: as for scalars in Section 3.2.5 (Fig. 3.12, p. 60), the demands from all branches are combined together in order to account for all possibilities, which is where the loss of precision associated with the abstraction comes from.

3.3.6 Safety of the abstract demand propagation semantics

Definition of safety

The goal of our safety theorem is still to relate the exact demand propagation semantics with the abstract one. In order to do this we first define an explicit safety relationship between an exact demand d and an abstract demand δ. [27] This definition (Defn. 3.40) says that abstract demand δ is safe for exact demand d (relative to the demand environment σ of which it is part: cf. the pre-condition) if either d is a negative demand, in which case δ must be associated with a set containing a negative demand, or else d is a positive demand with constructor K, in which case δ must contain some positive demand with this same constructor and such that the components of the abstract positive demand are safe for the components of the exact one.

Definition 3.40 *(Safe abstract demands)*

$$\lhd : EDmd \to ADmdRef \to ADmdEnv \to \mathbb{B}$$

$$
\begin{aligned}
d \lhd_\sigma \delta \quad \triangleq \quad & \\
& d = @ \wedge @ \in \sigma(\delta) \\
& \vee \\
& d = \textit{mk-EPDmd}(K, [d_1 \ \cdots \ d_k]) \wedge \\
& \quad \exists \textit{mk-APDmd}(K, [\delta_1 \ \cdots \ \delta_k]) \in \sigma(\delta) \cdot d_j \lhd_\sigma \delta_j
\end{aligned}
$$

$\textsf{pre}\ \delta \in \textsf{dom}\, \sigma$

Next, the definition is extended (component-wise) to vectors of demands. Given $\overline{d} \in EDmdVec_k$ and $\widetilde{\delta} \in ADmdVec_k$, we then say that $\overline{d}$ is safely approximated by $\widetilde{\delta}$ in demand environment σ ($\overline{d} \lhd_\sigma \widetilde{\delta}$) whenever the following holds, i.e., each component of the vector of exact demands is safely approximated by the related component of the vector of abstract demands:

[27] Section B.6 (p. 224) examines further the relationships between exact and abstract demands in terms of concretization sets.

Definition 3.41 *(Safe abstract demand vectors)*

$$\lhd : EDmdVec_k \rightarrow ADmdVec_k \rightarrow ADmdEnv \rightarrow \mathbb{B}$$

$$\overline{d} \lhd_\sigma \widetilde{\delta} \quad \triangleq$$
$$\qquad \overline{d} \neq \bot \;\Rightarrow\; \forall j \in \{1, \ldots, k\} \cdot \overline{d}(j) \lhd_\sigma \widetilde{\delta}(j)$$

$$\mathsf{pre}\ \forall j \in \{1, \ldots, k\} \cdot \widetilde{\delta}(j) \in \mathsf{dom}\,\sigma$$

Note that this definition reflects the property that the abstract answer needs to contain the exact answer *only if* the latter can effectively be obtained, i.e., it is $\neq \bot$.

Finally, the definition of $\lhd$ is extended to function environments. We say that $\overline{fve}$ is safely approximated by $\widetilde{fve}$ in demand environment σ ($\overline{fve} \lhd_\sigma \widetilde{fve}$) if the following (Defn. 3.42) holds, i.e., given any safe demands δ for d ($d \lhd_\sigma \delta$), the result produced for f_i under the exact function environment for any arguments u_j ($\overline{d} = \overline{fve}[\![f_i]\!]\ d\ (u_1,\ \ldots,\ u_{n_i})$) is safely approximated by the result produced by the abstract environment ($(\widetilde{\delta}, \sigma') = \widetilde{fve}[\![f_i]\!]\ \delta\ \sigma$), this relationship holding in the newly created abstract environment σ' ($\overline{d} \lhd_{\sigma'} \widetilde{\delta}$):

Definition 3.42 *(Safe abstract function environments)*

$$\lhd : \overline{Fve} \rightarrow \widetilde{Fve} \rightarrow ADmdEnv \rightarrow \mathbb{B}$$

$$\overline{fve} \lhd_\sigma \widetilde{fve} \quad \triangleq$$
$$\qquad \forall f_i \cdot \forall u_1,\ \ldots,\ u_{n_i} \cdot$$
$$\qquad\qquad \forall d \cdot \forall \delta \in \mathsf{dom}\,\sigma \cdot d \lhd_\sigma \delta \cdot$$
$$\qquad\qquad\qquad \mathsf{let}\ \overline{d} = \overline{fve}[\![f_i]\!]\ d\ (u_1,\ \ldots,\ u_{n_i})\ \ \mathsf{in}$$
$$\qquad\qquad\qquad \mathsf{let}\ (\widetilde{\delta}, \sigma') = \widetilde{fve}[\![f_i]\!]\ \delta\ \sigma\ \ \mathsf{in}$$
$$\qquad\qquad\qquad \overline{d} \lhd_{\sigma'} \widetilde{\delta}$$

Safety theorem

We now state the major theorem concerning the safety of the abstract demand propagation semantics, i.e., the result obtained from the abstract function environment safely approximates the one obtained from the exact function environment. Note that the statement is the same as for scalars (cf. Theorem 3.7) except for the need to deal with a proper demand environment:

Theorem 3.18 (Safety of abstract demand propagation semantics)

For any program p:

$$\forall \sigma \cdot \qquad \overline{fve}_{[\![p]\!]} \lhd_\sigma \widetilde{fve}_{[\![p]\!]}$$

Proof: The proof is by fixed point induction using lemmas relating the exact and abstract non-standard semantic functions for expressions. The statement of these lemmas is presented below. The details of the proof are presented in App. H.2.2, p. 298. □

Lemma 3.19 states that a safe relationship exists between the top-level semantic functions for expressions, i.e., between $\overline{\mathcal{E}}_k$ and $\widetilde{\mathcal{E}}_k$. As for Lemma 3.20, it states that a similar relationship exists between the semantic functions for expressions under positive demands, i.e., between $\overline{\mathcal{E}}_k^+$ and $\widetilde{\mathcal{E}}_k^+$.

Lemma 3.19 *(Safety of expression semantics: arbitrary demand)*

$$\begin{array}{l} \forall \sigma \cdot \\ \quad \forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd_\sigma \widetilde{fve} \cdot \\ \qquad \forall d \cdot \forall \delta \in \mathsf{dom}\,\sigma \cdot d \lhd_\sigma \delta \cdot \\ \qquad\quad \forall k, bve \cdot \mathsf{len}\, bve = k \cdot \\ \qquad\qquad \mathsf{let}\ \overline{d} = \overline{\mathcal{E}}_k [\![e]\!]\ d\ bve\ fve\ \overline{fve}\ \ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (\widetilde{\delta}, \sigma') = \widetilde{\mathcal{E}}_k [\![e]\!]\ \delta\ \widetilde{fve}\ \sigma\ \ \mathsf{in} \\ \qquad\qquad \overline{d} \lhd_{\sigma'} \widetilde{\delta} \end{array}$$

Proof: Straightforward case analysis comparing the two semantic functions. Cf. Appendix H.2.2, p. 299. □

Lemma 3.20 *(Safety of expression semantics: positive demand)*

$$\begin{array}{l} \forall \sigma \cdot \\ \quad \forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd_\sigma \widetilde{fve} \cdot \\ \qquad \forall d \cdot \forall \delta \in \mathsf{dom}\,\sigma \cdot d \lhd_\sigma \delta \cdot \\ \qquad\quad \forall k, bve \cdot \mathsf{len}\, bve = k \cdot \\ \qquad\qquad \mathsf{let}\ \overline{d} = \overline{\mathcal{E}}_k^+ [\![e]\!]\ d\ bve\ fve\ \overline{fve}\ \ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (\widetilde{\delta}, \sigma') = \widetilde{\mathcal{E}}_k^+ [\![e]\!]\ \delta\ \widetilde{fve}\ \sigma\ \ \mathsf{in} \\ \qquad\qquad \overline{d} \lhd_{\sigma'} \widetilde{\delta} \end{array}$$

Proof: The proof is by structural induction using various properties relating the operations on exact demands vs. those dealing with abstract demands. Cf. Appendix H.2.2, p. 300. □

3.3.7 Putting it all together

We now tie everything together, i.e., soundness and safety, using the approach presented in Section 3.2.8; in other words, we formalize the idea that the abstract demand propagation semantics for data structures effectively computes an inverse of the standard semantics.

The first step is to define a concretization function which maps an abstract demand reference into a set of standard values. The definition is relatively straightforward: each exact demand d approximated by δ is mapped into its associated set of standard values using the concretization function defined for exact demand, i.e., $conc(d)$ (cf. Defn. 3.37, p. 80). [28]

Definition 3.43 *(Concretization of an abstract demand reference)*

$$conc : ADmdRef \to ADmdEnv \to \mathcal{P}(Value)$$

$$conc\ \delta\ \sigma \quad \triangleq \quad \bigcup_{d \triangleleft_\sigma \delta} conc(d)$$

$\mathsf{pre}\ \delta \in \mathsf{dom}\, \sigma$

Next, revised versions of the *is-inverse* predicates appropriate to relate standard and abstract non-standard values are defined. Here are the definitions, which differ from earlier ones only in the need to deal with demand environments:

Definition 3.44 *(Inverse of a function environment)*

[28]Note that, because of Lemma B.1 (p. 226), this is really equivalent to $\bigcup_{d \in conc\ \delta\ \sigma} conc(d)$. That is, when an abstract demand is viewed as a set of exact demands (cf. App. B.6), then the concretization set of an abstract demand is simply the union over all these exact demands of their associated concretization set.

$$is\text{-}inverse : Fve \rightarrow \widetilde{Fve} \rightarrow \mathbb{B}$$

$$\begin{array}{l}
is\text{-}inverse(fve, \widetilde{fve}) \quad \triangleq \\
\quad \forall f_i \in \mathsf{dom}\, fve \cdot \\
\quad\quad \forall d, v_1, \ldots, v_{n_i}, \delta, \sigma \cdot \\
\quad\quad\quad d \lhd_\sigma \delta \wedge (\widetilde{\delta}, \sigma') = \widetilde{fve}[\![f_i]\!]\, \delta\, \sigma \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad is\text{-}inverse_{bve,fve,\sigma'}([\![f_i(x_1, \ldots, x_{n_i})]\!], d, \widetilde{\delta}) \\
\quad\quad\quad\quad \text{where } bve = [v_j \mid 1 \le j \le n_i]
\end{array}$$

Definition 3.45 *(Inverse of an expression)*

$$\begin{array}{l}
is\text{-}inverse_{bve,fve,\sigma} : Expr \rightarrow Dmd \rightarrow ADmdVec \rightarrow Bve \rightarrow Fve \rightarrow ADmdEnv \\
\quad\quad\quad\quad\quad\quad \rightarrow \mathbb{B}
\end{array}$$

$$\begin{array}{l}
is\text{-}inverse_{bve,fve,\sigma}(e, d, [\delta_1, \ldots, \delta_k]) \quad \triangleq \\
\quad bve = [v_j \mid 1 \le j \le k] \\
\quad \Rightarrow \\
\quad \forall v \in conc(d) \cdot \\
\quad\quad \mathcal{E}[\![e]\!]\, bve\, fve \sqsupseteq v \\
\quad\quad \Rightarrow \\
\quad\quad \exists u_1 \in conc\, \delta_1\, \sigma, \ldots, u_k \in conc\, \delta_k\, \sigma \cdot \\
\quad\quad\quad \mathcal{E}[\![e]\!][u_j \mid 1 \le j \le k]\, fve \sqsupseteq v \\
\quad\quad\quad \wedge \\
\quad\quad\quad u_j \sqsubseteq v_j \;\; (j = 1, \ldots, k) \\
\mathsf{pre}\ \delta_j \in \mathsf{dom}\, \sigma \;\; (j = 1, \ldots, k)
\end{array}$$

Based on these modified definitions, the following theorem can then be stated:

Theorem 3.21 *(The abstract semantics is an inverse of the standard semantics)*

For any program p:

$$is\text{-}inverse(fve_{[\![p]\!]}, \widetilde{fve}_{[\![p]\!]})$$

Proof: The proof would follow along the lines of the proof of the similar theorem for scalars (Theorem 3.13), i.e., would combine the property of soundness and safety, with the additional requirement of having to deal with demand environments. The details are left as an exercise (sic). □

3.4 Related work

This section presents work related to the problem of strictness analysis; it also mentions some related work that uses demand-driven analysis for purpose other than strictness analysis, e.g., multi-thread partitioning.

3.4.1 Forward abstract interpretation

The initial research effort on strictness analysis focused on the use of *forward abstract interpretation* [Myc80, BHA85, HY86, CP85, Wad87]. Although these approaches were sufficiently powerful to deal with both first-order and higher-order functional languages, they have had very little success with the analysis of data structures.

Using a forward abstract interpretation approach based on a four point domain, Wadler [Wad87] was able to detect *tail-strictness* and *head-strictness combined with tail-strictness*. However, he was not able to detect *head-strictness* by itself. As was mentioned in Section 2.3 (cf. also Section F.6), this is an important limitation, as head-strictness, in the case of linear lists for example, makes it possible to combine the evaluation of the head of a cell with the allocation of the cell, thus saving one thunk per cell.

The approach proposed by G. Burn [Bur87, Bur88] has been concerned from the start about parallelism. Burn introduced the idea of *evaluators*. An evaluator describes how much of a structure can be safely evaluated without changing the semantics of the program. Four evaluators for linear lists are defined: ξ_0, ξ_1, ξ_2, ξ_3. Their intended meaning are respectively: perform no evaluation, evaluate the list to WHNF, evaluate the structure of the list, evaluate all the list. These four evaluators are directly inspired from Wadler's domain, as described in the previous paragraph.

At compilation, each function is associated with an *evaluation transformer*, transformer which maps an evaluator for the output of the function into the evaluators required for the inputs. These evaluation transformers can then be implemented dynamically [Bur87] or statically through the creation of multi-version functions [Bur88].

The notion of evaluator seems interesting as it can be used, at run-time, to generate parallelism. For example, let's define `f` as follows:

$$f\ l\ x\ =\ \textsf{if}\ \ x = 0\ \ \textsf{then}\ \ l\ \ \textsf{else}\ \ [\,]$$

In this case, no evaluation of l can be done before the call. However, as soon as the condition is known to be true, then parallel evaluation of l can be started. Such concurrent evaluation can be represented by the use of evaluators.

The major weakness of Burn's approach is his compile-time analysis. Although the use of a small set of evaluators is appropriate at run-time, it may lose many opportunities to remove unneeded laziness and obtain more parallelism when used at compile-time (see the discussion of Hughes and Wadler's work below in Section 3.4.2). Furthermore, although Burn's approach deals with some form of contextual information by using evaluation transformers, he can not represent head strictness or absence, as his compilation approach is essentially based on forward abstract interpretation; in fact, his analysis simply uses the domain proposed by Wadler [Wad87].

Below is a short example illustrating the weakness of Burn's approach in regards to compile-time analysis. Let f and g be the functions defined in Figure 3.24.

$$\begin{aligned} f\ l_1\ l_2 &= \textsf{if}\ (hd\ l_1) > (hd\ l_2)\ \textsf{then}\ (g\ l_1)\ \textsf{else}\ (g\ l_2) \\ g\ l &= (tl\ l) \end{aligned}$$

Figure 3.24: Limitation of using a small domain

Burn's analysis would conclude that both l_1 and l_2 can, *a priori*, be evaluated only to WHNF. However, the first head of both lists could also be evaluated, thus saving one thunk for each list. In addition, suppose that f is called in a context requiring all heads and tails of the result, e.g., the `sum` function. The evaluator inferred for g would be: *evaluate all tails*. Thus, when the test condition becomes known in f, only the tails of l will be evaluated concurrently. But again, much more could be inferred: *i*) from the test condition, we know that the first head of each list can be evaluated; *ii*) from g's definition, we know that all tails and all heads except the first one can be evaluated; *iii*) from *i*) and *ii*), we can then infer that, when the test condition becomes known, *all* heads and *all* tails of the selected alternative can safely be evaluated.

A major weakness of the main forward approaches to strictness analysis is that they can not express and analyze head-strictness, as was discussed in Chapter 2. However, there does exist an alternative form of forward strictness analysis which can *express* head-strictness [Hun90]. This newer approach uses an abstract domain consisting of values which are associated with partial equivalence relations instead of being associated with Scott-closed subset of values (cf. Section 2.1). The major drawback of this approach is that it has only been extended to deal with lists in an *ad hoc* manner, i.e., arbitrary user-defined data types are not handled. Furthermore, although it can indeed express the notion of head-strictness of lists, so far, no implementation has been constructed showing that head-strictness can effectively be analyzed; in fact, [HL91] even conjecture that "... although Hunt's abstract domain can capture head-strictness, his analyzer will be unable to discover it".

Finally, an additional important disadvantage of forward abstract interpretation approaches, when compared with backward approaches such as those to be described in the following section, is well expressed by Burn [Bur90b, p. 227]:

> [Backward approaches] are able to say something about some functions that do not necessarily have to evaluate their arguments, whereas [forward] abstract interpretation does not appear to be able to give any information when it cannot determine that an argument expression need to be evaluated.

3.4.2 Backward strictness analysis

The general idea of classical abstract interpretation approaches to strictness analysis consists of computing a (abstract) description of the output of a function from a (abstract) description of its inputs, thus propagating the information in a *forward* manner. In contrast, the approaches based on *backward analysis* infer an abstract description of the inputs of a function from an abstract description of its output.

The initial work on backward analysis was performed by Hughes [Hug85, Hug87] and was later refined into *projection analysis* by Hughes and Wadler [HW87]. Further work by Hughes [Hug88] also discussed the analysis of (non-cyclic) data structures. Let us first discuss the projection analysis approach of [HW87].

The goal of projection analysis is to associate a *projection transformation* with every function; this transformation describes the contexts which are required/acceptable for each of the arguments given a required/acceptable context for the function's result.

An advantage of the approach presented in [HW87] — which is also one of its weakness — is the simplicity of the domain of contexts being used. [29] A simple

[29] Note that this is not an *a priori* limitation of the notion of projection: it would certainly be

domain means computing fixed points over it can be done relatively rapidly. On the other hand, this leads to a lack of expressiveness, which may result in weak approximations. For example:

- They can not express the fact that the elements of a list are sure not to be needed but the structure of the list itself is required;
- They can not express contexts which treat the elements of a list in a non-uniform manner — e.g., "*evaluate all the tails but only the first head*" — thus losing opportunities for removing thunks.

Another weakness is that no specific concern is given to the generation of parallelism or to the generation of multiple function versions, which is often required in order to make full use of the results of strictness analysis. Although more powerful than Burn's approach in that it can handle absence and head strictness, projection analysis would suffer from the same drawback as Burn's for the example presented in Fig. 3.24. Again, this is because projection analysis only uses a fixed small set of projections, in order to simplify the computation performed by the analysis.

Hughes' initial work on strictness analysis by abstract interpretation of continuations [Hug87] is, in certain ways, more similar to our own approach than the later approach of projection analysis. For example, the characterization of the merge operator (the & operator similar to our own $\oplus$ operator) as the set of the least-upper bound of all compatible values (cf. Fact 3.3 in Section 3.2.4, p. 49) is also presented in [Hug87, p. 78], although it has been discovered independently within our own approach. The interpretation of strictness analysis in terms of demands and inverse computation, as described in Chap. 3, seems simpler to motivate and understand than the continuation-based approach, although there is clearly some relationships between the two, as a demand could possibly be interpreted as a representation of some form of continuation. Furthermore, the demand approach, contrary to that of continuations or projections, does not require the use of a lifted domain, i.e., the introduction of a new bottom element (called ?) in addition to $\perp$, element whose is role is not intuitively clear.

This is not, however, the major weakness of the approach presented in [Hug87]; rather, it is the fact that the domain of *contexts* — a context represents a set of

possible to extend the domain of projections in a way similar to our domain of demands. However, doing so would clearly make the manipulation of projections, e.g., fixed point computation, much more complex.

abstract continuations — over which abstract interpretation is performed requires solving algebraic equations involving terms which are difficult to manipulate, that is, according to the words of Hughes "the terms we get are quite complicated in general". Simplifying rules are thus introduced, in order to arrive at the appropriate simplified solution; these rules are themselves quite complex, as they may need to deal with infinite expressions and, furthermore, "the order in which the rules are applied is significant — they are not Church-Rosser" [Hug87, p. 93], thus "context analysis yielded equations that were difficult to solve" [HW87, p. 385]. [30]

C. Hall [Hal87, HW89] introduced the concept of *rational strictness patterns* in order to deal with lazy data structures, more precisely, lazy streams. Strictness patterns can be interpreted, much like evaluators, as descriptions of *demand* for values. The major characteristic of *rational* strictness patterns is that they can represent, in a finite manner, non-uniform demands for potentially infinite data structures. This finite representation, which is obtained by the use of *cyclic* data structures, is analogous to the periodic representation of rational numbers, e.g., $1/11 = 0.09090909... = 0.\overline{09}$, thus the name. The use of rational strictness patterns allows a more precise compile-time analysis to be performed, because it leads to a more expressive domain of demands; thus, it is possible to remove more of the unnecessary laziness, i.e., to reduce the number of required thunks.

One limitation of Hall's work is that it is only concerned with compile-time improvement and really gives no concern to exploiting the maximum parallelism. If no useful information can be inferred at compile-time, then nothing is done at run-time either. For example, suppose `f` is defined by the following function:

$$f\ l_1\ l_2\ =\ \textsf{if}\ (hd\ l_1) > (hd\ l_2)\ \textsf{then}\ l_1\ \textsf{else}\ l_2$$

Moreover, suppose an invocation of `f` is to be evaluated in a context requiring all heads and all tails of its result. Hall's analysis would be able to conclude that the first head of both l_1 and l_2 need not be delayed, thus saving two thunks. However, nothing would be done about the rest of l_1 or l_2, even *after* the test condition has been evaluated, i.e., at run-time. Unnecessary sequentialization would occur because all tails and all remaining heads of both l_1 and l_2 would be delayed. Clearly, it would be more interesting if the evaluation of the result list was triggered as soon as the

[30] A main reason for the complexity of the simplifying rules is because of the way in which the merge (&) operator is dealt with: although its exact characterization is presented (cf. above), there is no explicit definition of its abstract counterpart, contrary to our own approach where an explicit definition can be given: cf. App. B.2.2, p. 213.

condition becomes known.

There are other weaknesses to C. Hall's approach:

- Absence is not handled: Hall is not able to obtain any information about sub-expressions which are sure *not* to be needed, and thus can be removed. In addition, ignoring absence leads to poor results in the presence of conditionals. Here is a small example using the function:

$$f\ x\ y\ c\ =\ \textsf{if}\ \ c\ \ \textsf{then}\ \ x\ \ \textsf{else}\ \ y$$

  Suppose all heads and tails of `f` are demanded. Now, because `x` appears in only one branch of the conditional, then C. Hall's approach would not be able to infer anything interesting about `x`, i.e., it would infer "$\perp$". However, in the approach we propose, we will be able to infer the demand "$\hat{}HTS$", i.e., `x` is not sure to be needed, *but if it is* then all its heads and tails can be evaluated.

- Use of a strict cons: Hall's approach is based on the utilization of a *strict cons*. This has the effect that she must use, as her maximum strictness pattern, the *head-strict* pattern. This pattern is weak in a context of parallel evaluation, as it does not allow demands of the form "*evaluate all tails*", or "*evaluate all head and all tails*". [31]

Another method for strictness analysis on non-flat domains is also presented in [JL90]. This method, called *necessity analysis*, is effectively a backward analysis method using a form of demand propagation. Their abstract domain of demands is defined by the equation below, with 0 and 1 indicating, respectively, a part which *may not be needed* and a part *which will certainly be needed*, and pairs representing demands for data structures:

$$P\ =\ \{0,1\} + (P \times P)$$

The main thing to note is that their domain can not represent head-strictness, as there is no way to indicate that a tail may not be needed but if it ever becomes needed, then it will be demanded according to some demand (a tail is either 0,

[31] Another impact of the use of a strict cons is that recursively defined data structures end up being represented by potentially infinite data structures, i.e., by their unfolding. In our own compiler — where recursively defined data structures are effectively allowed, although this is not described in Chap. 4 — such data structures can be represented using *cyclic* data structures, i.e., using a finite representation.

meaning not needed, or a pair (x, y), meaning it *is* needed, and its inside is needed as indicated by x and y).

Another interesting line of work based on demand propagation is presented in [Lin86]. Like our work, it is based on propagation of demands in dataflow graphs using the arcs in a backward manner. Also like is done in our compiler, [Lin86] uses, for analyzing conditionals, a form of abstract conditions which are called *path contexts* (such a context being a list of *conditional branch records*) — the usefulness of these abstract conditions comes from the more precise results that can be obtained when demands are merged. For example, when demands with identical conditions are merged, this can be done without any loss of information (using a l.u.b. operator for merging the demands); loss of information (approximation) may occur only when demands with complementary conditions are merged, using a g.l.b. operator: in that case, only the part common to both branches will be used (no information is lost if the same demand is made in both branches, as g.l.b.$\{x, x\} = x$!).

Although there are a number of similarities between the approach of [Lin86] and our approach, there is also a number of important differences. First of all, the underlying domain of demands is different, e.g., the domain of [Lin86] has no notion of *absence/not-needed*. This means that they can not infer, for example, that a given argument is sure not to be needed. More important is the fact that the notion of rational or cyclic demand is not present. These two aspects imply that notions such as *tail-strictness* or *head-strictness* can not be expressed in their domain; however, note that hyper-strictness, i.e., head-and-tail-strictness, can be expressed using a special demand value: d^e.

As for abstract conditions, their use is limited in the sense that they are used only within procedures boundaries, whereas in our compiler abstract conditions may be lifted to the inter-procedural level; thus, there are cases where more precise results can be obtained with our approach.

Another important weakness of the approach presented in [Lin86] is the treatment of recursive functions. The result they obtain using the least fixed point solution is, in certain cases, weaker than the one that can be obtained by our approach. For example, their analysis of the *append* function under a head-and-tail strict demand can only infer that append is simply strict in its first argument, and non-strict on its second; on the other hand, our analysis, under the same demand, can effectively conclude that both arguments can be evaluated under the head-and-tail-strict demand. In their approach, as they say in [Lin86], "[t]here are situations, however, in which the least fixpoint solution is *not* the optimal solution". To obtain the more precise result, they have to use a fixed point that is not the least one, which "is done by assuming d^e [i.e., hyper-strictness,] rather than $\perp$ as the base of [their] solution chains." But, as they also say, this "means that the method is no longer safe" (sic).

The way they deal with fixed point in their approach, i.e., not using the least fixed point solution to get the best result, seems to imply that their domain of demands might not be properly defined. Our conjecture is effectively that their domain is *upside-down* and results from a lack of a formal understanding and interpretation of the meaning of demands. For example, in their approach, the least element of the domain means everything has to be delayed. On the other hand, in our domain, the element saying that everything must be delayed is the top one, the least one indicating instead that the program *is sure not to terminate*, i.e., that no set of arguments will allow the program to produce the desired result. These ideas have been formalized in Sections 3.2.4 and 3.3.4, where demands were viewed as abstracting sets of acceptable values and demand propagation was seen as an abstract form of *inverse computation*. When viewed in this manner, the resulting domain of demands effectively has to be flipped up and, then, it is effectively the least fixed point solution which produces the best result, i.e., which leads to the most precise solution and allows more (safe) optimizations to be performed.

In subsequent papers by Lindstrom and others [Lin88, LLY87, Yeh90], the notions of *conditional branch record* and *path context*, i.e., of abstract conditions, have been dropped and have been replaced by the notions of *blocks* and *block numbers*. The interesting aspect of these new notions is that they allow the expression of a limited form of uncertainty. In other words, it is possible to infer demands about components of data structures that will not be strictly needed, so that useful information can be gathered even within non-strict components of data structures. However, because their new domain can still not express cyclic pattern of demands, thus can not express head-strictness or tail-strictness, it is doubtful that precise results can be inferred.

Another backward approach is presented in [SPR90]. Their approach is based on a form of demand propagation, but restricted to only *two* (2) demands: "evaluate to WHNF" (called *dd-analysis*) and "evaluate all heads and tails" (*ee-analysis*). Although it can handle arbitrary data types, clearly, head-strictness can not be expressed.

An interesting aspect of [SPR90] is that it examines, for various test programs, the parallelism that can be identified and exploited by their analysis. From the analysis of these test programs, they conclude that their approach can identify all of the parallelism. However, this conclusion should be taken *with a grain of salt*, because there is a major weakness in their experiments: *none of their program really require lazy evaluation*, i.e., none of them really exploits the major characteristic of lazy evaluation, which is the possibility of dealing with potentially infinite objects (cf. Section 1.3.3, p. 10 for an example). In that respect, it is no surprise that their analysis (ee-analysis) is effective: there is no danger with demanding everything when the object being demanded is *finite* (Cf. Section 5.4.5). On the other hand, if the object is (potentially) *infinite*, then demanding everything is sure to lead to *non-termination*; only head-strictness (or *bounded* speculative form of head-strictness, cf. Section 5.4.3) can deal correctly with potentially infinite objects.

3.4.3 Inverse image analysis

The idea of viewing strictness analysis as computing inverse of functions, as was done in Section 3.1, was initially introduced in [Dyb87, Dyb88]. In these papers, Dybjer introduced the idea of analyzing the strictness of a function f, not by determining whether $f\ \bot = \bot$, but instead by finding out whether f, in order to produce a *total* element (i.e., an element different from $\bot$) requires its argument to also be a total element.[32]

Dybjer presents a categorical programming language, together with a denotational semantics defined in terms of *neighborhood systems*; a *calculus of open sets* is then introduced, which is used to compute inverse image of functions. Although his domain of *open set expressions* is powerful enough to express cyclic demands — e.g., tail-strictness, or head-and-tail-strictness — it is still not possible to express head-strictness, as the notion of absence (not needed) can not be expressed.

[32]That is, using the alternative definition of strictness presented in Section 3.1.1: f is strict if and only if the following holds:

$$f\ x \neq \bot \ \Rightarrow\ x \neq \bot$$

Two additional weaknesses of Dybjer's presentation are the following:

- It is far from clear how this language and its semantics relates to standard denotational semantics. The presentation in terms of neighborhood systems and open sets does not clearly show what demands really mean.
- More importantly, it is clear that exact inverse computation can not be done at compile-time, as analysis of conditionals must suffer some form of approximations. In other words, an *abstraction* of the semantics must be used at compile-time. How to obtain such an abstraction is far from clear given the semantics presented in the paper, and the presentation in terms of open set expressions does not seem to lead to an easy approach to approximation.

3.4.4 Type inference

An alternative approach to strictness analysis not based on abstract interpretation has been proposed in [KM87, KM89, LM91]; instead, their approach is based on type inference principles, as used in polymorphic functional programming languages [Mil78]. The underlying idea is that strictness properties can be characterized as types.

It is claimed that the interesting aspect of this approach to strictness analysis is that no fixed point iteration is required to deal with recursive definitions. However, there is an admitted drawback: there are some recursive functions which can be correctly analyzed using abstract interpretation but for which the associated strictness properties can not be deduced using the type inference approach [KM89].

Let simply mention that, from our point of view, another weakness of the type inference approach of [KM89] [33] is that, so far, it has not been extended to deal with the notion of head-strictness. [34]

3.4.5 Multi-thread partitioning

A different line of work that has some relationships with our work is the *compilation as partitioning* approach of [Tra88, SCvE91, Tra91, TCS92]. Although this approach

[33] And, also, of the refined type inference approach presented in [Wri91].

[34] Our conjecture is that head-strictness can not be recognized by such an approach. Types can be seen as representing sets of terms. However, as discussed in Section 2, there is not finite representation for a set of terms/values that would represent the general notion of head-strictness. Thus, it is doubtful that the type inference approach could be extended to express head-strictness.

deals mainly with the implementation of non-strict functional languages, it can also be applied to lazy languages.

When dealing with lazy languages, the strategy used for partitioning would be based on *demand-set partitioning*, which can be seen as a form of demand propagation. Grosso modo, this would lead to distinct thunks being turned into independent threads.

The work described in [TCS92] is essentially an extension, at the global level, i.e., between basic blocks, of the work of [SCvE91, Tra91]. However, as they admit, there is a fundamental limitation to their algorithm: "[its] ability to deal with recursion" [SCvE91, p. 332], i.e., it can not deal with recursive function definitions. Furthermore, no discussion is presented on how programs involving complex data structures would be analyzed. [35] Thus, it is difficult to see how this approach, as it currently stands, would produce results as good as those of strictness analysis approaches.

[35] Although this is discussed in [Tra88], the 2-points domain which is used would probably not lead to a very precise analysis. Furthermore, it seems that obtaining the properties required for the dependence analysis itself depends on the results that would be obtained from a strictness analyzer: "The author [i.e., K. Traub] has successfully used the Wadler and Hughes context analysis technique [HW87] to infer these five properties from functional quads programs." [Tra88, p. 119]

Chapter 4

A prototype compiler based on abstract demand propagation

This chapter presents the compiler which has been implemented as part of this thesis. As will be seen in the next chapter (Chap. 5), our approach leads to interesting improvement of the performance of the resulting programs. For example, in most cases, using our compiler reduces the total number of instructions executed somewhere between 15–25%, whereas the critical path length is reduced somewhere between 20–30%. However, as we will also see, the optimizations performed by the compiler have a weaker impact on the average parallelism, result which will be explained in Section 5.3.3.

The prototype compiler implementation to be described in the current chapter is an optimizing compiler which uses a strictness analysis strategy based on the abstract demand propagation model presented in Chapter 3. In addition, it includes a number of other features: abstract conditions, provision for multiple function versions, run-time evaluators, various forms of initial demands and a notion of bounded speculation appropriate for laziness. All these features, except for speculation which will be described in Chapter 5, will be introduced and presented in the present chapter.

The overall structure of the chapter is the following: First, Section 4.1 presents an overview of the existing Id compiler — compiler which was used as the basis of our prototype lazy compiler. This is then followed, in Section 4.2, by an overview of how the lazy compiler was made to fit within the existing Id compiler. In Section 4.3, the domain of demands used by the compiler, which forms the heart of its strictness analysis approach, is presented. Finally, the major phases of the compilation process within the lazy compiler are described in Sections 4.4, 4.5, 4.6 and 4.7.

4.1 Overview of the existing Id compiler

The lazy compiler was built on top of the existing Id compiler implementation. [1] The compiler that was used is an old version (circa 1989) defined for Id Version 88.0 [Nik88]. The reason this version of the language and compiler was used, instead of a more recent version [Nik91], is because of its availability on the Lisp machines that were available to us. Given that the existing Id compiler is written in Common Lisp [Ste84], those Lisp machines (TI Explorers) provided a nice and easy to use programming environment.

General structure of the compiler

An Id compiler is composed of a collection of modules, together with an implicit top-level procedure that takes care of the coordination of the various modules. Thus, it is relatively easy to define a new compiler by adding new modules or modifying existing ones.

The logical top-level decomposition of a compiler is presented in Fig. 4.1. Each major part is composed of a number of modules, grouped together based on the internal representation being manipulated: [2]

- Front-end (parse trees): This takes as input the source program, performs lexical and syntactical analysis, together with type-checking and various transformations of the source programs, e.g., lambda-lifting.
- Middle-end (program graphs): This part transforms the parse trees into programs graphs and performs various machine independent optimizations, e.g., constant propagation, common sub-expression identification, dead-code elimination, fetch elimination, etc.
- Back-end (machine graphs): This part of the compiler transforms the (machine-independent) program graphs into the appropriate machine graphs, performs

[1]This is why all program examples presented in the present chapter will use the Id concrete syntax instead of the abstract syntax of the previous chapter.

[2]For a more detailed presentation of the internal working of the Id compiler, see [ACGS91, Tra86].

various peephole optimizations and then generates machine code (in our case, TTDA machine code for the GITA emulator [AN87]).

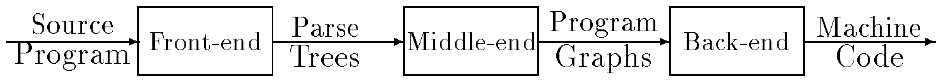


Figure 4.1: The top-level decomposition of an Id compiler

Program graphs

Our lazy variant of the Id language was defined by adding a number of additional modules to the *middle-end* of the existing Id compiler. No modifications had to be done to either the front-end or the back-end. The analysis and transformations required for the implementation of the lazy variant were all done by manipulating program graphs; because of this, we will give a brief description of the programs graphs used in the Id compiler, but will leave aside the details of parse trees and machine graphs.

The program graphs used in the Id compiler are a form of structured dataflow graphs [Tra86]. A program graph is composed of a collection of instructions, together with a set of directed arcs connecting instruction output ports to instruction input ports. There are two major classes of program graph instructions:

1. Simple instructions: These instructions represent primitive operators, e.g., constants values, basic arithmetic and boolean operators, data structure constructors or selectors, apply nodes, etc.

2. Encapsulating instructions (also called *encapsulators*): These are used to represent more complex (structured) expressions from the source programs, e.g., if, case, function/constant definitions.

Figure 4.2 presents an example of a program graph for the following simple function:

```
Def f x = If x > 0 Then x-1 Else 0;
```

The major advantage of these structured program graphs (i.e., with encapsulators) as an intermediate form is that they clearly identify the various basic blocks

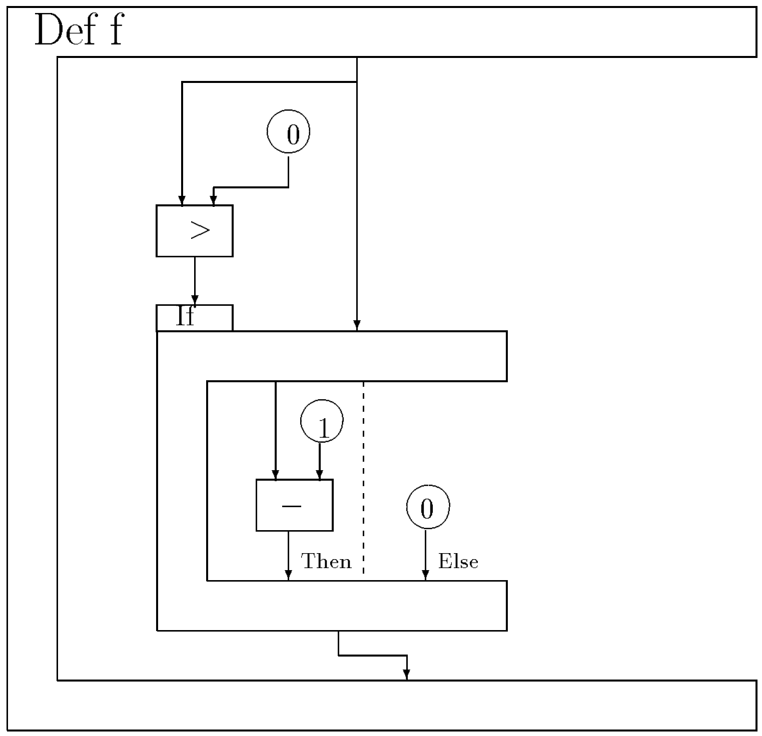


Figure 4.2: The program graph for `Def f x = If x > 0 Then x-1 Else 0;`

of the program [Tra86]. This provided a useful framework for performing the transformations and optimizations required by the lazy compiler.

Support for lazy data structures

The Id language is well-known for its support of non-strictness both for function calls and data structures which is achieved through the use, at the implementation level, of *I-structures* [ANP89, AN90]. Essentially, an I-structure is a form of data structure which can be defined incrementally and can provide support for low-level producer/consumer synchronization. When a slot of an I-structure is fetched by a consumer and the slot's content is available, then nothing special happens; on the other hand, when that slot has not yet been filled by the producer, the consumer is delayed until the content becomes available.

Although this form of data structure has nothing to do with laziness, it can easily be extended/modified to support a form of *lazy data structures*. This was done by S. Heller, who extended I-structures into *L-structures* [Hel89]. [3]

[3]Note that this does not provide full support for lazy evaluation, as functions are not yet made to behave lazily. Cf. Section 1.3.2, p. 9.

The first major difference between an I-structure and a L-structure is that the latter can contain a *thunk* (i.e., a data structure representing a delayed expression with its appropriate environment) in addition to containing a regular value or being empty. The other major difference occurs when a consumer tries to obtain the content of a slot which contains a thunk instead of a regular value: in this case, the consumer is delayed while evaluation of the thunk is started; when the thunk's value finally becomes available, the consumer is reactivated. This behavior, which we will call *implicit force on fetch*, requires a rather simple extension of the original I-structure implementation, since all the machinery required to suspend consumers and reactivate them is already available.

Our implementation will make essential use of L-structures. These will be used to deal both with lazy data structures and lazy function calls, the latter being implemented using single slot L-structure. Of course, the trick to obtain an efficient implementation will be to avoid using delayed slots as much as possible; this will be done through a form of strictness analysis, as described earlier in Chapter 3.

4.2 Overview of the lazy Id compiler

Before going into the detail of the compiler implementation, it should be mentioned that the compiler which has been implemented only handles a subset of the Id 88.0 language, more precisely, the first-order purely functional subset without arrays and loops. Thus, the following features are not handled by the lazy compiler: arrays, list and array comprehensions, loops, programmer level I-structures, higher-order functions.

The major steps of the lazy compilation process

As mentioned earlier, our prototype compiler performs all its work by manipulating program graphs. [4] Thus, all the modifications that had to be done in order to implement our lazy compiler were made within the middle-end part. Figure 4.3

[4]Note that this is in contrast with the abstract model presented in Chapter 3, where the compiler was modelized using a syntax-directed approach. This was done to simplify the presentation in terms of denotational semantics. An interesting line of further research, beyond the scope of the present thesis, would be to establish some formal relationships between the two presentations, i.e., syntax-directed vs. program graphs based. This would probably be similar to a congruence proof between an operational and a denotational semantics [Sto81, Les87], as programs graphs and demand propagation through them clearly have more of an operational nature.

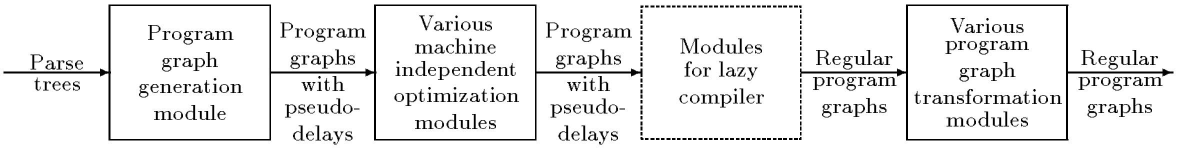


Figure 4.3: Where the lazy compiler's modules fit within the middle-end of the Id compiler

shows where the modules of the lazy compiler (indicated by the dashed box in Fig. 4.3) fit within the middle-end. Except for some modifications to the program graphs generation module, to be described soon, all other existing modules were left unchanged. [5]

Our compiler is a *force-and-delay* style of compiler [Hen80]: all expressions occurring in (possibly) non-strict contexts, i.e., as actual arguments to function calls or to data constructors, or as expressions bound to identifiers introduced by global constant definitions or let-expressions, are first encapsulated/embedded within a *delay*, which is done by a modified version of the program graph generation module; then, in a subsequent phase really part of the lazy compiler code (i.e., dashed box), appropriate force instructions are inserted, in order to trigger the evaluation of the (possibly) delayed subexpressions, i.e., those which are still present after the unnecessary ones have been removed using the results of strictness analysis.

The major steps of the lazy compilation process, i.e., those enclosed within the dashed box of Fig. 4.3, are presented in Fig. 4.4. [6] Step 0 (not shown on the figure) is the modified generation of program graphs (part of the regular compiler) which encapsulates the expressions in non-strict contexts within *pseudo*-delays. [7]

[5]This is not totally true because two new encapsulating instructions had to be defined. Thus, this also required adding appropriate code to the following modules, in order that they recognize and correctly handle the new encapsulators: dead-code elimination, constant propagation. These modifications were straightforward, requiring only a few additional lines of code.

[6]Note that only the major logical steps are being presented. Within the prototype compiler, some additional modules are required to deal with various idiosyncrasies of the regular Id compiler or less important features of the lazy compiler. For example, the expansion of pattern-matching expressions, which is performed within the program graph generation module, is done in such a way that laziness is not obtained; in order to obtain a correct lazy semantics for the pattern-matching operations, some data structure selectors have to be moved inside encapsulators, instead of being left at the outermost level of the defining encapsulator. Two additional modules were also needed: one module to deal with the various ways of implementing thunks, i.e., direct or indirect (cf. Section F.7); another to implement the run-time forcing/printing process (cf. Section 5.4.1).

[7]These are called *pseudo*-delays to distinguish them from *lazy*-delays and from regular delays. Lazy-delays are obtained after the program graphs have been annotated and the delays which are effectively needed in the final program have been identified; the regular delays are defined and used in the regular Id compiler. It is these latter delays which are then mapped, within the back-end,

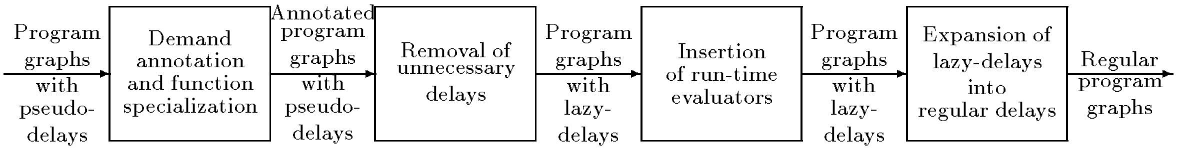


Figure 4.4: The major steps of the lazy compilation process

Once the program graphs with (pseudo-) delays have been generated, the following transformations are performed:

- The program graphs are annotated with demands (first box of Fig. 4.4). This is done by propagating through the programs graphs, *in a backward manner*, an initial demand representing the minimal demand that is sure to be made at run-time (cf. Figures 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11 for an illustration of demand propagation and annotation). Conceptually, this can be seen as a form of *partial evaluation* leading to a process of *function specialization* [BEJ88, Lau88]: when a demand is propagated to an instruction representing a function application, this application is specialized for that particular demand, giving rise to a new function version. Thus, each such new version is specialized to deal efficiently with a particular demand, i.e., a particular context of execution.

  After this step is complete, all subexpressions within the programs graphs are annotated with a *safe* demand, i.e., a demand that faithfully represents what is sure to happen at run-time.

- Based on the resulting demand annotations, some of the subexpressions which were initially embedded in (pseudo-) delays can now be identified as sure to be needed at run-time (second box of Fig. 4.4); in that case, the associated (pseudo-) delays can be removed (cf. Fig. 4.15). As for the other delayed expressions, they are transformed into *lazy-delays*, indicating that they must remain delayed until explicitly forced at run-time.

- The next phase is the insertion of run-time evaluators, including the insertion of appropriate forces (third box of Fig. 4.4). Suppose a function f has some argument x for which a demand d has been inferred. Now, it is quite possible that, within some subexpression contained in f, the demand for x can be inferred to be stronger than d, e.g., d'. Thus, as soon as that subexpression becomes needed, i.e., starts executing, the level of demand needed for sure for

into the L-structures described in Section 4.1.

x can be raised from d to d'. In the special case where d is simply a delayed version of d', then raising the level of demand essentially means forcing the expression. However, there are other cases where this can be more complex; such cases are described in Section 4.6 (cf. Fig. 4.16 for an illustration).

- The final phase of the lazy compiler (fourth box of Fig. 4.4) is concerned with the expansion of the lazy-delay program graph instructions — instructions which are specific to the lazy compiler — into the regular delays of the standard Id compiler (cf. Fig. 4.18 for an illustration). After this phase, the program graphs are composed only of regular instructions, so that the standard back-end (machine code generator) can be used.

Compiler options

There are a number of options that can affect the behavior of the compiler. They are briefly described below and, in most cases, are explained in more detail in later sections.

- `finite-demands-domain-size`: This option indicates the size of the domain of demands which is to be used to perform strictness analysis through demand propagation (cf. Section 4.3).
- `type-of-abstract-conditions`: This option indicates the type of abstract conditions to be used by the compiler. Abstract conditions are used to represent *conditional demands*, i.e, they can indicate the conditions under which a demand can be assumed to hold. There are three possible values:

  1. `None`: No abstract conditions allowed, so that a demand is either certain or uncertain.
  2. `Intra-procedural`: Abstract conditions are defined only within the scope of the function definition in which they appear, i.e., within the function in which their associate predicate is defined.
  3. `Inter-procedural`: Abstract conditions are allowed to cross functions' boundaries, thus allowing a form of inter-procedural analysis leading (in some cases) to more precise results.

  Further detail about abstract conditions are presented in Section 4.3.

- `maximum-nb-of-versions`: This option indicates the maximum number of versions that can be created for any particular function definition. In theory, the maximum number of versions required for a function can grow as large as the size of the domain of demands, which is why it can sometimes be important to impose a smaller bound. In practice, however, it turns out that in most cases only a small number of versions is effectively required (cf. Section F.3).

- `type-of-initial-demand`: The whole process of demand propagation and function specialization is triggered by specifying an initial demand which can be assumed to be made for the result of the program. Using a stronger (less lazy) initial demand should lead to more precise analysis results (by monotonicity) and, thus, to more efficient programs.

  Four types of initial demands can be indicated:

  - `Simple-strict`: This is the weakest assumption, as it implies, in the case of data structures, that all sub-components have to be delayed. Note that such a demand is the strongest that can be assumed in a compiler that does not express head-strictness.
  - `Sequential-head-strict`: This demand may be used when the compiler can express the notion of head-strictness. It corresponds to a sequential view of laziness, as it allows the first component of any data structure to be demanded, but delays all subsequent components.
  - `Parallel-head-strict`: This is a novel notion of head-strictness, better suited for parallel machines, as it allows for the concurrent evaluation of all the components of data structures. This is described in more detail in Section 5.4.2 and Appendix G.
  - `Hyper-strict`: This is the strongest assumption, as it indicates that nothing needs to be delayed, i.e., the result will be fully evaluated. Clearly, such a demand can only be assumed when the program's result is sure to be finite, or, as is the case in the experiments presented in Section 5.3.2, when the lazy compiler is made to emulate a lenient compiler.

- `head-strict-speculation-level`: This parameter can be used when the previous option (`type-of-initial-demand`) is either `Sequential-` or `Parallel-`

`head-strict`. It allows a stronger initial compile-time demand to be assumed by specifying a *bounded speculation level* (cf. Section 5.4.3 and Appendix G).

- `use-run-time-evaluators?`: This option indicates whether complex run-time evaluators should be introduced in the resulting program. This is described in more detail in Section 4.6.
- `use-indirect-pointers`: This indicates whether, in lazy data structures, sub-components (i.e., thunks) should be represented using direct or indirect thunks. This is discussed in more detail in Section F.7.

4.3 The domain of demands

The main characteristic of the lazy compilation process is the strictness analysis it performs using the approach of abstract demand propagation presented in Chapter 3. The key element of the abstract demand propagation approach is the domain of demands which it uses: it is really the domain of demands which determines the expressive power of the analysis, which allows it to infer strictness results which are more or less strong. Of course, there is some trade-off here: a more complex domain will allow better results to be obtained but will also imply greater computation costs.

A first domain of abstract demands was presented in Chapter 3. In the present section, we introduce some modifications to this basic domain of abstract demands; the resulting domain of demands — domain which is effectively used within the prototype compiler — will make it possible to obtain, in a number of cases, more precise analysis results. As we will shortly see, the major addition will be the notion of *abstract condition*, which will lead to the notion of *conditional demand*.

Strict and delayed demands

The abstract demands used in the model presented in Chapter 3 (cf. Fig. B.1, Section 3.3.5) correspond to what are called, in the present section, *strict* and *delayed* demands. [8] Strict demands represent demands which are strict in the sense

[8]The difference in terminology is due the choice of a different representation of the abstract demands. The representation used in the present chapter is slightly more efficient: in many cases, one less demand reference is required to represent a given abstract demand. Furthermore, negative demands are handled differently from positive demands, contrary to Chap. 3 where they were part

that they lead at least to evaluation to WHNF (i.e., in the terminology of Chap. 3, they correspond to demand references associated with a set containing *only* positive demands). Strict demands, when combined with *certainty-indicators*, become *delayed demands* and can then be used to represent demands for strict (Certain, i.e., only positive demands) as well as non-strict (Uncertain, i.e., associated, among other demands, with a negative one) arguments.

Note that it is also the strict and delayed demands which are used to implement cyclic demands — much like the domain of demands used in Chap. 3 — using an appropriate map associating demand references with their effective value. This will be described later, when presenting the specification of the domain of demands (cf. Fig. D.1 in App. D, p. 233).

Abstract conditions and conditional demands

Strict and delayed demands, i.e., the demands which are analogous to those used within the framework of Chapter 3, are sufficiently powerful to properly analyze simple conditional expressions (e.g., if and case). For example, the abstract model of Chap. 3 can effectively identify both arguments of the following function as being strict:

```
Def f x y = If x==0 Then y Else 2*y;
```

Within the prototype compiler, however, an additional goal was to examine a number of more powerful notions of demands. For example, it was hoped that, given the following function definition, both arguments could be identified as being strict:

```
Def f x y = (If x==0 Then 0 Else y) + (If x==0 Then y Else x);
```

Identifying `y` as strict requires recognizing that although `y` appears in two different conditional expressions, the two conditions under which `y` may be needed are effectively *complementary*, i.e., when combined together, they are certain to be true. This can be done in the following manner, assuming common sub-expression identification has been performed:

---

of set of demands just like positive demands; this makes it possible to obtain a slightly more efficient implementation of the merge and union operations — in fact, a single operation can be used to deal with both operations using appropriate parameters, i.e., certainty indicators.

- First, it must be noted that the predicate "`x==0`" can effectively be fed, through common sub-expression identification, as the predicate to both conditionals of `f`. Let α represent the condition associated with that predicate — which we call an *abstract condition*.
- The demand made for `y` within the left conditional, assuming `f` is demanded with \$ (i.e., WHNF) will be \$ when "`x==0`" is false. Thus, from the first conditional expression, we can infer that the demand for `y` will be the $\alpha_F \rightarrow \$$, i.e., `y` will be required to WHNF whenever the abstract condition α, i.e., the predicate "`x==0`", is false.
- Similarly, for the second conditional expression, the following demand can be inferred: $\alpha_T \rightarrow \$$, i.e., `y` is needed to WHNF when the condition is true.
- From the previous steps, we would then conclude that, *within* `f`, two demands would be made for `y`: $\alpha_T \rightarrow \$$ and $\alpha_F \rightarrow \$$. But then, it could also be realized that these two demands are *complementary*, i.e., the disjunction of the abstract conditions are effectively equivalent to `true` ($= \alpha_T \vee \alpha_F$).

In the prototype compiler, the notions of abstract conditions and complementary conditions have been defined not only for boolean-driven conditionals but for arbitrary user-defined algebraic types and related case-expressions. Here is a simple example:

```
Type tree12 = Leaf N | Node1 tree12 | Node2 tree tree12;
Def sum t = { case t of
                Leaf n = n
              | Node1 t1 = sum t1
              | Node2 t1 t2 = (sum t1) + (sum t2) };
```

Letting α represent the predicate used for selecting the branch of the case-expression to be executed, the following are 3 complementary demands which, when merged together, are equivalent to \$:

$$\alpha_{Leaf} \rightarrow \$, \quad \alpha_{Node1} \rightarrow \$, \quad \alpha_{Node2} \rightarrow \$$$

The notions of abstract conditions and conditional demands can also be extended to the inter-procedural level. This extension can allow a backward approach to analyze more precisely cases involving correlation between arguments. For example, suppose we have the following function definitions:

```
Def g x y = h x y y;
Def h x y z = If x==0 Then y Else z;
```

A forward approach to strictness analysis would be able to recognize that both `x` and `y` are strict arguments of `g` [Bur90b]. However, a simple backward approach would not be able to do so. This would not be possible because the analysis of function `h`, which encapsulates a conditional expression, would have to infer that `y` and `z` are non-strict because they are distinct parameters used in different branches of the `If` expression.

On the other hand, using abstract conditions, the following inferences could be made for the arguments of `h`: `x`: $T \rightarrow \$$; `y`: $\alpha_T \rightarrow \$$; `z`: $\alpha_F \rightarrow \$$. Now, when analyzing `g`, the two demands for `y` ($\alpha_T \rightarrow \$$; and $\alpha_F \rightarrow \$$) could then be seen as being complementary; they could then be combined to infer that `y`, within `g`, is demanded with $T \rightarrow \$$, i.e., it is effectively strict. [9]

Another important impact of our underlying goal was the requirement to deal with *sets of (conditional) demands*, as was shown in the previous examples. [10] When dealing with abstract conditions, each use of an identifier, within a function, gives rise to a specific conditional demand, where the abstract conditions indicate the (conjunction of) conditions under which that particular use of the identifier is effectively required. [11] These conditions effectively correspond to the triggering conditions of the basic block in which the identifier is used; as basic blocks may be

[9] Note that such form of inter-procedural abstract conditions can be useful not only for if-expressions but also for arbitrary case-expressions involving data structures. Here is a small example:

```
Def f l x y =  Case l Of nil = x | cons y ys = y ;
Def g l a = f l a a;
```

Using inter-procedural abstract conditions, it can be found that `g` is indeed strict in both arguments.

[10] Some initial relationship between the expressive power of forward and backward analysis, for flat domains, has been presented in [NM92]. Effectively, they present a set-based backward approach that has the same deductive power as an abstract interpretation based forward approach. Although we do not prove any formal equivalence results, our conjecture is that some similar relationships could be established in the case of non-flat domains. In fact, it could probably be proved that a set-based backward approach would be strictly more powerful, because it could deal, as the forward approach, with correlations between arguments, but it would also obtain better results when analyzing data structures, e.g., by expressing notions such as head-strictness.

[11] A similar notion can be found in [Tra88], where a functional quad is associated with a *control string*. Such a string "indicates the control conditions governing the execution of each statement[; ...] the control string of a variable x can be interpreted as a boolean formula which must be true in order for the binding defining x to be executable;" [Tra88, pp. 147–148].

embedded within one another, the triggering conditions are effectively compound (conjunction of) conditions. Here is a small example:

```
Def f x y = If x==0 Then
               y
            Else If x==1 Then
               0
            Else
               y;
```

Let α, resp. β, be the abstract condition representing the predicate "`x==0`", resp. "`x==1`". The conditional demand associated with the first use of `y`, assuming a demand of \$ for the result of `f`, is $\alpha_T \rightarrow \$$, as it simply requires the first predicate to be true; as for the second use of `y`, it requires both predicates to be false, so the associated demand for `y` would be $\alpha_F \& \beta_F \rightarrow \$$. Thus, we would then infer the set of demands for `y` to be $\{\alpha_T \rightarrow \$, \alpha_F \& \beta_F \rightarrow \$\}$.

Note that it is clear dealing with sets of abstract conditions in the manner just described incurs some important costs. That is, finding complementary subsets of abstract conditions can be quite costly, even when done in a non-optimal manner using some approximate but safe heuristics, as done in the prototype compiler. Whether it is worth paying this price will be discussed in the next chapter (Section 5.5) when the benefits brought by using these abstract conditions will be examined.

Tagging of inter-procedural abstract conditions

When dealing with inter-procedural abstract conditions, i.e., when allowing abstract conditions to escape the function bodies in which they are defined, it is necessary to distinguish the invocation contexts involving similar abstract conditions. For example, suppose we have the following function definitions:

```
Def f x y z = If x==0 Then y Else z;
Def g x y = f x y y;
Def h x y = (f 0 x y) + (f 1 y x);
```

Letting α represent the predicate "`x==0`" of `f`, we could infer, as we saw earlier, that both `x` and `y` are strict arguments of `g`. However, the same is not true for function `h`, because the actual value bound to the first argument of `f` is not the same in the two cases, so the actual value of the predicate associated with abstract condition α would not be the same. This means that the interpretation of an abstract condition, when it escapes its defining context, must be related to the appropriate invocation context. To express this within our prototype compiler, inter-procedural abstract conditions get tagged with a unique identifier associated with the function application node realizing the function call. The combining of complementary abstract conditions can then only be done when both abstract conditions are tagged with the same application node identifier, as is the case with the single call to `f` in `g`, but not with the two distinct calls of `f` in `h`.

Unknown conditions and the analysis of recursive function definitions

When dealing with recursive function definitions, the use of tagged inter-procedural abstract conditions is not sufficient. As mentioned above, the tag really describes the invocation context leading to the indicated condition being true. For example, given a call to a function `f` which calls a function `g` and which in turn calls a function `h`, an abstract condition coming out of `h` would be tagged, at the outermost level, with the apply node identifier for `h`, `g` and `f`. In the case of a recursive function `f0`, the context of an abstract condition from `f0` would then have to encode the whole call stack; clearly, this is not possible, as there is no way to know, at compile-time, when the recursion is going to terminate.

So, when analyzing recursive function definitions, inter-procedural abstract conditions are not allowed. Instead, the demands for the arguments of recursive definitions are always described by *approximate demands*. An approximate demand is a special case of a set of conditional demands: it is a *singleton* set containing a conditional demand for which the abstract condition is either T (= true, i.e., a strict/certain demand) or an *unknown* condition, denoted by ?. [12]

When computing the demands made for the arguments of a recursive function, the demand propagation and annotation process is first performed as usual on the

[12]An unbounded supply of distinct unknown conditions is required, as each distinct non-strict invocation context must be distinguished by a different (approximate) condition. This is necessary because each abstract condition is supposed to represent the exact condition under which a block is to be executed, which is required so that subsequent modules, e.g., insertion of run-time evaluators, can work properly by recognizing the current basic block triggering conditions and the conditions of the demands associated with the various inputs of the basic block.

body of the function. The main difference occurs before returning the demands computed for the arguments by the demand propagation process through the body: the computed demands are not sent as they are but are effectively *approximated*, i.e., for each argument, the set of demands associated with an argument is simplified into a singleton set containing an approximate demand which is the l.u.b. of the set of elements.

For example, suppose we have the following function definition: [13]

```
Def f x y z = If x==0 Then y Else If x==1 Then y + z Else f (x-2) y z;
```

The following demands would then be inferred for the arguments of `f`: `x`: $T \rightarrow \$$; `y`: $T \rightarrow \$$; `z`: $? \rightarrow \$$, where the demand for `z` would be obtained as an approximation of the following (singleton) set of conditional demands: $\{\alpha_F \& \beta_T \rightarrow \$\}$. In this case, the condition $\alpha_F \& \beta_T$, which is clearly not true, is transformed into the unknown condition ?.

Within the prototype compiler, unknown conditions are used not only when dealing with recursive function definitions but also when inter-procedural abstract conditions are not allowed, as determined by the appropriate compile-time option — `type-of-abstract-conditions` — which is set by the user. For example, suppose we are given (again) the following definitions:

```
Def f x y z = If x==0 Then y Else z;
Def g x y = f x y y;
```

When inter-procedural abstract conditions are allowed, the demands inferred for the arguments of `f` are, as we saw earlier, the following (assuming α represents "`x==0`"): `x`: $T \rightarrow \$$; `y`: $\alpha_T \rightarrow \$$; `z`: $\alpha_F \rightarrow \$$. On the other hand, when such conditions are disallowed, then the demands associated with an argument, before being sent out of the definition's scope, is approximated. In this case, approximation leads to the following demands being inferred: `x`: $T \rightarrow \$$; `y`: $?_1 \rightarrow \$$; `z`: $?_2 \rightarrow \$$. Thus, when inter-procedural conditions are not allowed, only the first argument can be inferred as definitively strict.

Computing the l.u.b. of strict and delayed demands corresponds to the union operator which is described in App. B.2.2. But what does it mean to compute l.u.b.

[13] Further examples will be presented in Section 4.4.

of demands involving abstract conditions? In other words, when can we say that an abstract condition γ_1 is more precise (stronger) than γ_2? When neither γ_1 and γ_2 involves unknown conditions, it can easily be seen that γ_1 is more precise (stronger) than γ_2 when $\gamma_2 \Rightarrow \gamma_1$. For example, T is clearly stronger than any condition γ, as $\gamma \Rightarrow T$ always hold. As for unknown conditions, they are never stronger than any other conditions, except themselves.

As another example, suppose we have the following function:

```
Def f l x = If x==0 Then 0 Else If x==1 Then (len l) Else (sum l);
```

Before approximation is performed, the demands for `l` would be the following: $\{\alpha_F \& \beta_T \rightarrow TS,\ \alpha_F \& \beta_F \rightarrow HTS\}$. The approximate result would then be $\{? \rightarrow TS\}$, i.e., it is not sure that `l` is needed (the associated condition is unknown, so it is possible that it never becomes true), but if it ever becomes needed, then all its tails can safely be evaluated.

Rounding-up of demands

The domain of demands used in the prototype compiler is a finite one, the size of which can be specified by a compile-time option (cf. Section 4.2). Setting the size of the domain to k_{max} ensures that all demand objects can be represented using data structures whose *size* is smaller or equal to k_{max}. [14]

As will be seen in Section 4.4, there are cases — viz., data structure constructors — where, given a demand d of size k, a new demand d' of size k' can be created, with $k' > k$. As long as $k' \leq k_{max}$, there is no problem; however, when $k' > k_{max}$, this means d' is not a proper demand relative to the selected domain of demands. In order to obtain a proper demand from d', the process of *rounding up* can be applied. Rounding-up is a form of *closure* operation [PJP89]: [15] given d', rounding-up produces a new demand d'' such that the following hold: i) d'' is a proper element of the domain of demands; ii) d'' is safe for d', i.e., $d' \sqsubseteq d''$.

The strategy used within the prototype compiler to round-up a demand is relatively straightforward and, although it does not necessarily find the optimal solution,

[14] The exact definition of *size* will not be given here. All that really matters is that this size is related to the *representation* of the object, not to the abstract value it represents. Thus, a cyclic demand structure of size 1 can effectively be used to represent a potentially infinite demand.

[15] A function f is a closure operator if $x \sqsubseteq f\ x = f\ (f\ x)$.

$a)\ k_{max} = 1$:
$\{\mathsf{Cons}\ \$\ \{\mathsf{Cons}\ \$\ \{\mathsf{Nil}\}\}\}$
$\Rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$
$b)\ k_{max} = 1$:
$(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Cons}\ @\ x\}\})$
$\Rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})$
$c)\ k_{max} = 1$:
$(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ x\}\}\}\})$
$\Rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})$
$d)\ k_{max} = 2$:
$(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ x\}\}\}\})$
$\Rightarrow \{\mathsf{Nil}, \mathsf{Cons}\ \$\ (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})\}$
$e)\ k_{max} = 3$:
$(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ x\}\}\}\})$
$\Rightarrow \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})\}\}$

Figure 4.5: Examples illustrating the rounding up of demands

its results have been quite acceptable. [16] Essentially, it consists in choosing d'' such that its final tail is always a simple cyclic demand of size one (1) with a prefix matching, for as long as possible, the prefix of the demand to be approximated.

Figure 4.5 presents a number of rounding examples. Cases c, d and e illustrate how improving the size of the domain (k_{max}) effectively allows a better approximation to be obtained, through the use of a longer matching prefix. These examples also show, however, that the rounding-up strategy does not always choose the best solution: in cases d and e, i.e., $k_{max} \geq 2$, the following solution would have been much better (solution for which the cycle length is 2 instead of 1):

$$(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ x\}\})$$

Note that the rounding-up process is not always performed immediately as soon as a *large* demand is encountered. Rounding up at every possible propagation step could lead to much less precise results, as shown in the following example, where we assume that a domain of demands of size 1 is used:

[16] Initially, a different strategy leading, in most cases, to more precise results had been used. However, it turned out that this initial strategy was not monotonic: increasing the size of the domain of demands did not always lead to an improvement in the precision of the results of the analysis. Of course, a more precise strategy that would have tried to find the best/optimal solution could have been implemented; such a strategy, however, would have been quite costly to implement, as it would have required a more thorough search of the domain of demands.

```
Def id l = { Case l Of nil = nil | cons x xs = cons x xs };
```

Suppose `id` is demanded with HTS, i.e.:

$$(\mathsf{Rec}\ x\!::\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$$

Clearly, the demand inferred for `l` should also be HTS. However, if rounding-up was performed at every possible propagation step, then only TS' would be inferred, i.e., all heads would have to be delayed:

$$TS' \ = \ (\mathsf{Rec}\ x\!::\ \{\mathsf{Nil}, \mathsf{Cons}\ \ \hat{}\$\ x\})$$

This would happen because demand propagation of HTS through the (implicit) tail selector (i.e., `xs`) would produce a demand which would be too large (Cons @ HTS) and would have to be rounded-up (to TS'), thus losing precision. [17]

In the prototype compiler, rounding-up is not performed at every propagation step but only in the following two situations: *i*) when a demand is going to be propagated outside of a function body, i.e., before the demand computed for the argument of a function is returned to the caller; *ii*) when a demand is propagated to an apply instruction. [18] Note that these two possible situations leading to rounding-up are intimately related to the reason why a finite domain of demands was used in the first place: *i*) to ensure termination of the fixed point computation process; *ii*) to bound the number of functions versions when no *a priori* limit has been set (option `maximum-nb-of-versions`: cf. Section 4.2).

Formal specification of the domain of demands and examples of strictness analysis results

A formal specification of the various domains involved in defining demands is presented in Appendix D (pp. 232–235). As for Appendix E (pp. 236–243), it presents a number of examples showing results that can be obtained by the strictness analysis process of the compiler using the above domain of demands.

[17] This situation is similar to rounding errors that occur in numerical computations using a limited number of digits. For example, suppose we allow only a single digit of precision and we need to compute the following: $0.4 * 0.4 + 0.8 * 0.2$. Using rounding/truncating after every step of computation would produce the following answer: $0.16 + 0.16 \Rightarrow 0.2 + 0.2 = 0.4$. On the other hand, rounding/truncating only in the final step would produce the following: $0.16 + 0.16 = 0.32 \Rightarrow 0.3$. The latter is clearly a better answer, the real answer being 0.32.

[18] Note that these really correspond to the two uses of *round-up* in the abstract semantics presented in Fig. B.3, p. 218.

Figure 4.6: Demand propagation for a constant value

4.4 Demand annotation and function specialization

Demand propagation and annotation

The main module of the lazy compiler, i.e., where most of the work is performed, is the one that performs the annotations of the program graphs (through the backward demand propagation process) and creates the various function versions.

The annotation of the program graphs is performed by propagating the initial demand assumed for the result of a program, initial demand whose exact value depends on the type of the result and on the value of the option `type-of-initial-demand`. Given a demand for the result of the program expression, it is then possible, based on the structure of that expression, to compute the demands this will generate for all its components subexpressions. This is done in a backward manner, similar to the *demand-driven* execution model of dataflow graphs presented in [DK82], the major difference being it is performed at compile-time using an abstract domain of demands and that there is only a backward propagation process. As for the forward (i.e., dataflow) propagation process, it is only performed at run-time; furthermore, it is done in a purely data-driven fashion, i.e., no backward demand propagation is needed at run-time.

Figures 4.6–4.11 illustrate a number of demand propagation cases. When encountering a constant value (Fig. 4.6), no further propagation is required, because such a value makes no reference to any other subexpressions. In the case of a binary operator (Fig. 4.7), the demand made for the result of the expression is transmitted directly to each of the operands. [19]

[19] In the various figures illustrating demand propagation, only a single demand has been indicated, to simplify the presentation. However, it is really sets of demands which are used and propagated through the various operators. The propagation of such sets is simply performed element-wise.

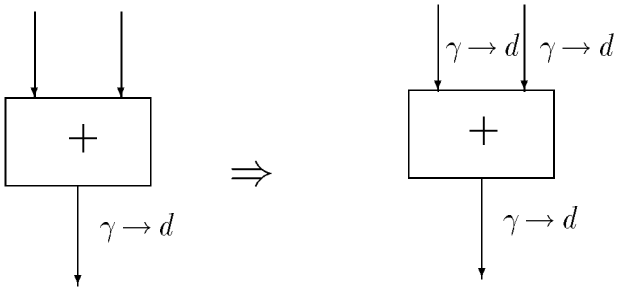


Figure 4.7: Demand propagation for a binary operator

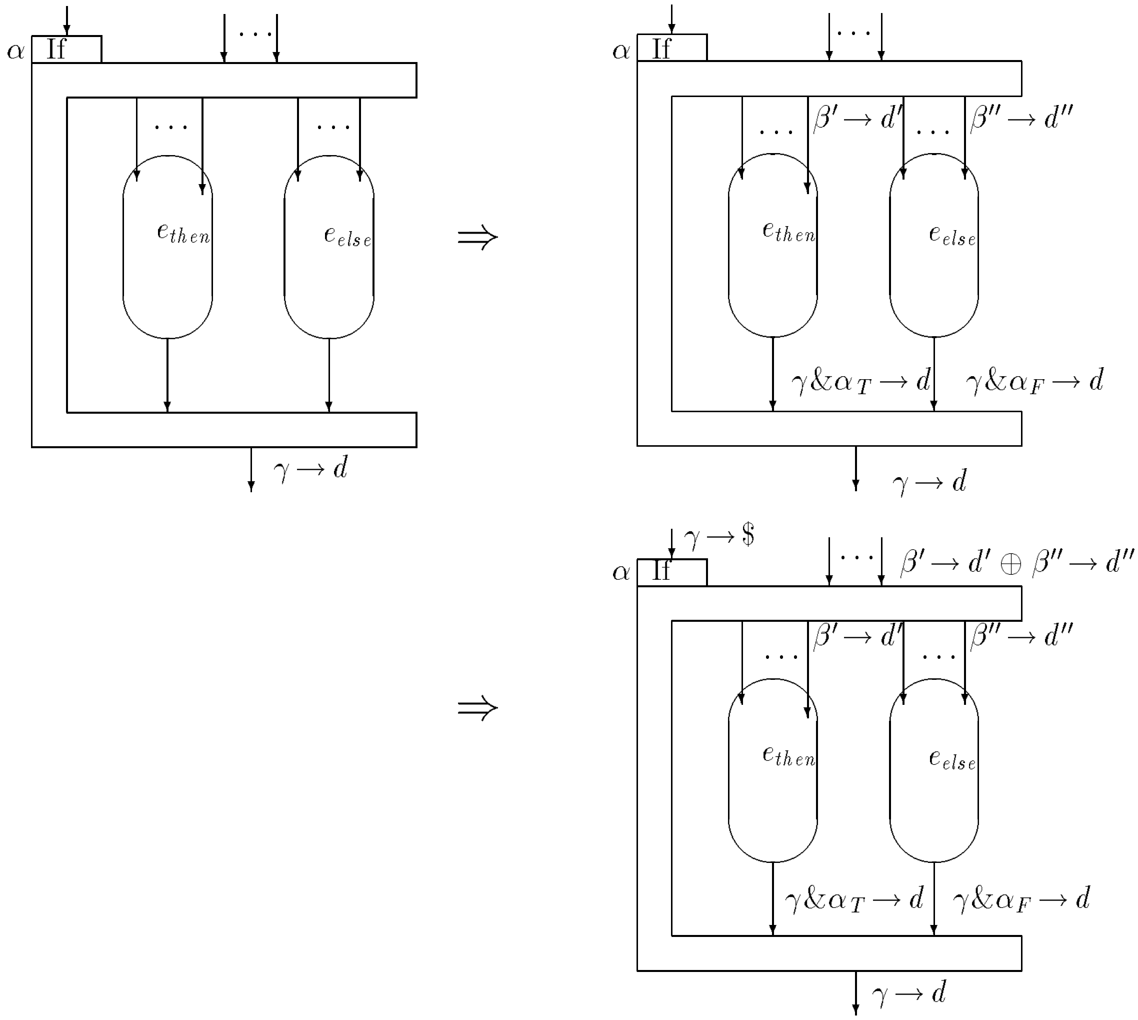


Figure 4.8: Demand propagation for a conditional expression

The conditional (Fig. 4.8) is more interesting, as it illustrates the use of conditional demands and the propagation of demands through encapsulators. In order to determine the demands made for the various inputs of the conditional expression, both arms of the conditional must be analyzed. This is illustrated in the top part at the right of the Fig. 4.8, where we assume that a demand for each of the branch has been made and propagation through the subexpressions has been performed. In this case, note that the conditions associated with the demands made for the two branches are distinguished by the rightmost abstract condition: α_T vs. α_F. Once the demands associated with the inner subexpressions have been computed, [20] the results are then combined and propagated back *outside* the `if` encapsulator, as illustrated in the lower right part of Fig. 4.8; [21] of course, a demand for the predicate also has to be propagated, which, assuming a boolean selector, is simply $\gamma \rightarrow \$$. [22]

Figures 4.9 and 4.10 illustrate the handling of data structures, in this case, lists. When encountering a data structure selector, e.g., access to the head of a component (Fig. 4.9), the demand made for the input of the selector is transformed into a demand for a data structure where the head is demanded as indicated by the demand for the selector and where the other component — the tail — is indicated as being not needed; a similar demand propagation would occur in the case of a tail selector but with the head being indicated as not needed.

The handling of data structure constructors is illustrated in Fig. 4.10. We assume that the demand made for the output is appropriate, i.e., it is a constructor demand which matches the indicated data constructor. In this case, the components of the demands are fetched and transmitted to the appropriate input. Note that the components of a constructor demand are delayed-demands; such demands are composed of a certainty-indicator (ci_h and ci_t) which denotes the strictness of the data structure component and of a strict-demand (d_h and d_t). Propagation through

[20] In Fig. 4.8, only the demand for the last input has been indicated.

[21] Note that, in Fig. 4.8, a $\oplus$ operator has been used to combine the results from the two branches. However, as we are really dealing with sets of demands, the demands from both branches would simply be combined into the same set.

[22] In this case, what would really be propagated to the predicate is the following demand: $\gamma\&\alpha_T \rightarrow \$ \oplus \gamma\&\alpha_F \rightarrow \$$. This then simplifies to $\gamma \rightarrow \$$. The more general case, defined to deal with arbitrary case expressions for which the predicate/selector need not necessarily be a boolean value, can be illustrated with the following example:

```
Def isCons? l = { Case l Of Nil = False | Cons x xs = True };
```

In this case, the demands for `l` would be due only to the case selector and would be the following: $\alpha_{Nil} \rightarrow \textsf{Nil}$ and $\alpha_{Cons} \rightarrow \textsf{Cons}\ @\ @$. It would thus indicate that `l` can be evaluated to WHNF only.

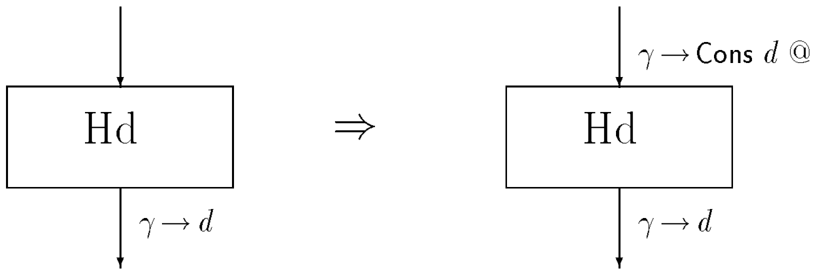


Figure 4.9: Demand propagation for a data structure selector

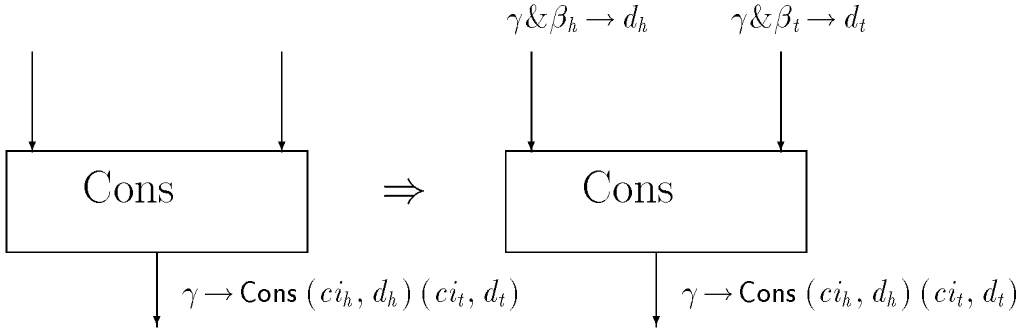


Figure 4.10: Demand propagation for a data structure constructor

the constructor instruction transforms these certainty-indicators into new abstract conditions, used to annotate the input arcs. When ci_h (resp. ci_t) is Certain, this generates a new abstract condition β_h (resp. β_t) which is true, indicating the relative strictness of the head (resp. tail); on the other hand, when ci_h (resp. ci_t) is Uncertain, then β_h (resp. β_t) is a new unknown-condition, thus indicating the non-strictness of the head (resp. tail).

Function application and function specialization

An interesting case occurs when a function application is encountered. Clearly, functions only need to be called for demands which are certain, otherwise, there is no need to perform the call; in other words, we can assume that functions only need to be analyzed under *Strict-demand*s. Suppose we can infer that f, when called with strict demand d, produces the demand $\beta_i \rightarrow d_i$ for the ith argument. In this case, the demands that will effectively be propagated to the ith argument, when f is demanded with $\gamma \rightarrow d$, will be $\gamma \& \beta_i \rightarrow d_i$, i.e., the condition to be associated with the demand for the argument depends on the context of the call (γ) and the condition under which the argument is required by the function being called (β_i).

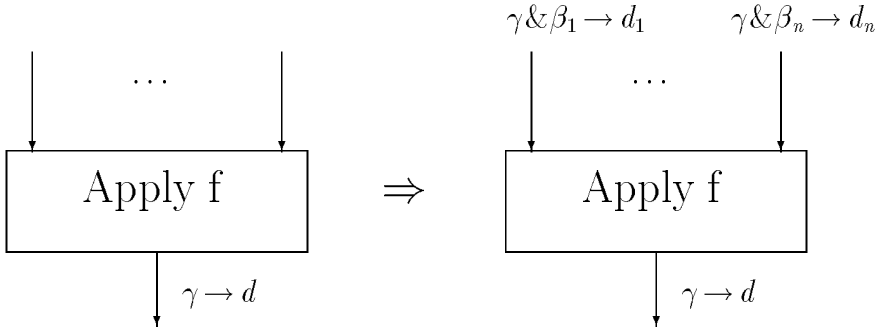


Figure 4.11: Demand propagation for a function application

This is illustrated in Fig. 4.11 for a function f with n arguments.

When encountering a function application, in order to determine the demands to be propagated to the actual argument expressions, the body of the function being applied must be analyzed (unless it has already been encountered before). Informal pseudocode illustrating the analysis of a function definition under a specific demand is presented in Fig. 4.12. Among other things, the pseudocode shows the process by which a set of demands gets approximated by a singleton approximate demands when this is required: each set of (conditional) demands effectively computed for an argument (`argument-demands(def,i)`, where `def` represents a copy of the program graph associated with function `fn` created for the purpose of demand annotation and specialization) gets approximated using the `approximate` function. Such approximations are required either when `fn` is a recursive function definition or when inter-procedural abstract conditions are disallowed. [23]

What Figure 4.12 does not show, however, is the strategy used to deal with functions which might be recursive. To deal correctly with such functions, a least fixed point computation strategy is required. Pseudo-code representing the strategy

[23] In fact, further conditions for "approximating and rounding" are possible, viz., when the lazy compiler is made to *emulate* a different compiler, e.g., a non-optimizing/purely lazy compiler, a lenient compiler, or a compiler based on the evaluation transformer model. In such cases, the approximation being applied depends on the emulated compiler:

- Purely lazy compiler: The demands computed for the arguments are ignored and the demands returned for the various arguments are all fully delayed.
- Lenient compiler: The demands computed for the arguments are ignored and the demands returned for the various arguments are all hyperstrict.
- Evaluation transformer compiler: Any head-strictness present in the computed demands is removed and made lazy.

```
Function analyze-def( fn, sd )
   def ← make a fresh copy of the program graph of fn (for specialization)
   propagate the demand {T → sd} within the body of def
   (* At this point, propagation within the body is over *)
   If an approximation is required for (fn, sd) Then
      final-demands-for-arguments ←
            [approximate(argument-demands(def, i)) | 0≤i<arity(fn)]
   Else
      final-demands-for-arguments ←
            [argument-demands(def, i) | 0≤i<arity(fn)]
   End
   Return final-demands-for-arguments
End analyze-def
```

Figure 4.12: Pseudocode for analysis of function body of `fn` under demand `sd`

used in the prototype compiler to compute the inverse of a function is presented in Fig. 4.13. It is this function — `inverse-function-version` — which gets executed when an apply instruction is encountered (i.e., a function call: cf. Fig. 4.11) and which determines the demands to be inferred for the arguments by analyzing the body of the function (line 6) and performing the appropriate fixed point computation process when necessary to do so (lines 9–14). More precisely, given a function `fn` and a strict-demand `sd` for its result, `inverse-function-version` computes the demands that would be made for the arguments of `fn`, taking into account the possibly recursive nature of `fn` (line 4 and lines 9–14), but also accounting for the fact that `fn` demanded with `sd` might have been encountered before and thus may have already been computed (line 2), or may have had some approximation associated with it due to some on-going fixed point computation process (line 3). (The next subsection (p. 131) explains in more detail the least fixed point computation strategy and the pseudocode for `inverse-function-version`. A step-by-step example will also be presented on p. 133.)

An important point to mention is that in our prototype compiler, in order to obtain better analysis results and more efficient resulting programs, multiple versions of a function definition are allowed to be created. A major assumption of the pseudocode presented in Fig. 4.13 is that an arbitrary number of function versions can be created; this is why `analyze-def` and `inverse-function-version` are each parameterized by a specific strict-demand `sd`. Note that a strategy will be presented shortly (p. 135) to limit the maximum number of versions which can be created for

```
Function inverse-function-version( fn, sd )
 Cond
   sd = ⊥ ⇒                                                              (1)
      Return [{T → ⊥} | 0 ≤ i < function-arity(fn)]
   A final result has already been computed for (fn, sd) ⇒              (2)
      Return final-approximation(fn, sd)
   A temporary approximation exists for (fn, sd) ⇒                       (3)
      Return current-approximation(fn, sd)
   A pending call for (fn, sd) exists (but with no approximation yet) ⇒  (4)
      Create an initial (temporary) approximation for (fn, sd)           (5)
      Return current-approximation(fn, sd)
   Otherwise
      (current-approx, dependencies) ← analyze-def(fn, sd)               (6)
      Cond
         dependencies = { } ⇒                                            (7)
            Set current-approx as the final result for (fn, sd)
            Return current-approx
         (fn, sd) ∉ dependencies ⇒                                       (8)
            Return current-approx
         Otherwise (* Fixed point iteration process has to be performed *) (9)
            modified? ← Update the current approximation to (fn, sd)     (10)
            While modified? Do                                           (11)
               (current-approx, dependencies) ← analyze-def(fn, sd)
               modified? ← Update the current approximation to (fn, sd)
            End
            If dependencies = {(fn, sd)} Then                            (12)
               Set current-approx as the final result for (fn, sd)       (13)
               Return current-approx
            Else                                                         (14)
               Remove the current approximation to (fn, sd)
               Return current-approx
            End
      End
 End
End inverse-function-version
```

Figure 4.13: Pseudocode for computing the inverse of a function version

any given function.

Least fixed point computation

The major strategy used to compute least fixed point is based on the use of the Kleene ascending chain. As is well-known, the least fixed point of a continuous function $g: D \to D$ on a domain D is given by:

$$\mathbf{fix}\ g = \bigsqcup_{i \geq 0} g^i(\perp)$$

When the domain of analysis is finite, using this strategy makes it possible to compute the least fixed point in a finite number of steps. Using the domain of demands introduced in Section 4.3 together with the rounding strategy presented there, the computation of the fixed point can then be seen as being performed on a subdomain of finite size (determined by the value of the parameter `finite-demands-domain-size`). Thus, fixed point computation using Kleene ascending chains can effectively be used.

However, given that we need to represent potentially infinite demands using finite cyclic demands, there are cases where a better approach is possible (and required). For example, suppose we have the following function definition, which computes the sum of a list:

```
Def sum l =
    { Case l of
      Nil = 0
    | Cons x xs = x + sum xs };
```

When the result of that function is strictly demanded (i.e., demanded with \$), then the demand inferred for the argument `l` should be HTS, that is, the following cyclic demand should be obtained: $HTS = (\textsf{Rec}\ x{::}\ \{\textsf{Nil}, \textsf{Cons}\ \$\ x\})$. Using the ascending chain process to compute this least fixed point would lead, however, to the following sequence of approximations, where step k indicates the demands that would be made assuming that at most $k-1$ recursive calls were performed: [24] [25]

[24]The notation $\$^j.\,\textsf{Nil}$, which has been used earlier in Section 2 to denote list values using regular style of expressions, is used here to denote demand values: in this case, it indicates a demand for a finite list (i.e., terminated by Nil) of length j where each component is demanded to WHNF (\$).

[25]Note that because we are dealing with sets of demands, the initial bottom approximation is simply the empty set, i.e., there are no value of l which can lead to *sum* producing a well-defined

0: $\{\}$ (* Initial bottom approximation *)

1: $\{\mathsf{Nil}\}$ (* No recursive call *)

2: $\{\mathsf{Nil}, \mathsf{Cons}\ \$\ \mathsf{Nil}\} = \{\mathsf{Nil}, \$^1.\ \mathsf{Nil}\}$ (* At most 1 recursive call *)

3: $\{\mathsf{Nil}, \$^1.\ \mathsf{Nil}, \$^2.\ \mathsf{Nil}\}$ (* At most 2 recursive calls *)

...

k: $\{\$^j.\ \mathsf{Nil} \mid 0 \leq j \leq k-1\}$ (* At most $k-1$ recursive calls *)

...

The limit of this sequence would represent an infinite set of demands which could effectively be represented by the cyclic demand *HTS*. However, it is not clear, a priori, how the cyclic representation can be obtained from the infinite sequence. One possible strategy, used in the prototype compiler, is to use the rounding-up process mentioned earlier. When the demands for the argument of a function have been computed, an approximation of these demands is computed before the demands are effectively returned as the result of that function. In the case of the `sum` function, this approximation would occur because, at some step in the iteration process, the demand which has been computed would get *too large* relative to the domain of demands being used. [26] In such a case, rounding-up would be done, i.e., a safe, but possibly less precise, demand would be used in place of the larger demand.

Although, in the case of sum, rounding up would effectively produce the exact answer, this is not always the case because of the approximation, i.e., loss of precision, involved in rounding-up. Furthermore, note that the number of steps required to obtain the least fixed point for *sum* using rounding-up would depend on the size of the domain of demands; thus, the number of steps required before the appropriate cyclic demand can be created would increase whenever the size of the domain would be increased. Clearly, when the solution can be represented by a finite cyclic demand, it would be preferable, if possible, to obtain the cyclic representation directly, independently of the size of the domain and, whenever possible, without using rounding-up. There are many cases where this is indeed possible and

integer result if no call is made to *sum*. The next approximation, at step 1, then indicates the possible values which can ensure that a call to *sum* will produce a well-defined result without making any recursive call: this can happen only if the list is empty, i.e., *Nil*. Similarly, step k indicates the possible values which will ensure that *sum* will produce a well-defined result ($) by doing at most $k-1$ recursive calls: this can happen only if the list l is a fully defined finite list of length at most $k-1$, i.e., described by a demand of the form $\$^j.\ \mathsf{Nil}$ with $0 \leq j \leq k-1$.

[26] Here, the notion of *largeness* is really an issue related to the implementation/representation of demands and has nothing to do with the partial ordering defined for demands.

the example above is one of them.

A strategy for obtaining directly cyclic demands, in the cases where this is possible, has been implemented in the prototype compiler. This strategy, called *lazy fixed point computation* and which can be seen as a form of *pending analysis* [YO88, You89], can also be viewed as a form of fixed point computation where logic style of variables are used, i.e., where it is possible to give and use a name for some expression to be computed without having the expression's value be available yet.

Let's illustrate how the above example (*sum*) would be handled by our lazy fixed point strategy. The major steps that would occur when analyzing *the first call* to the `sum` function are presented below, which will also explain the major steps of function *inverse-function-version*. [27] Note that we assume the existence of a global data structure (*current-approximations*) that indicates, given a function and a demand for the result of that function, the demands that are to be inferred for the arguments (i.e., the current, or possibly final, approximation to be used for that particular `fn` and `sd`). [28]

- First, it would be realized that `sum` when demanded with \$ has not yet been encountered (we assume it is the first such call), so no approximation (2, 3) or pending call (4) exists yet. Thus, a call to `analyze-def` would be made (6).
- While analyzing the body of `sum`, a second call to `sum` with the same demand (\$), i.e., a pending call, would be encountered (4). This means a recursive dependency exists. An approximation of the result to be computed would then be created (5); this approximation would be set to $T \rightarrow d$; here, d would be a newly created demand identifier whose value would not yet be defined, i.e., d would be *unbound*.
- Propagation of $T \rightarrow d$ through the body of `sum` would then continue (within `analyze-def`) even though d is still unbound. When encountering the tail selector — due to the use of `xs` within `sum` — this d would then be used to construct a new demand (Cons @ d), which would be possible because constructing this demand would not require that the effective value of d be known

[27]The case where the function with this particular demand has already been analyzed is handled, in Fig. 4.13, by checking whether a final result has already been computed for that particular function and demand, in which case the already computed result is used directly.

[28]In the following paragraphs, the numbers between parentheses refer to the line numbers appearing in Fig. 4.13.

or available.

- Once the analysis of the body of `sum` is over, the result computed by `analyze-def` would be returned to `inverse-function-version` (6); this would be done after an approximation has been performed, as `sum` is recursive (cf. Fig. 4.12). The result returned to `inverse-function-version` would then indicate the demand for `l` to be $T \rightarrow d'$, where d' would be bound to $\{\mathsf{Nil}, \mathsf{Cons}\ \$\ d\}$; it would also indicate that this result was obtained using an approximation to (`sum`, \$), i.e., `dependencies` = {(`sum`, \$)} — which means the predicates at lines (7) and (8) would be false. [29]

  Within `inverse-function-version`, the previous approximation for `sum` demanded with \$ (i.e., $T \rightarrow d$, stored in the global data structure mentioned above) would then be updated (10). Now, it is here that the lazy fixed point computation strategy would occur: because d is still unbound, updating the value of the old approximation (d) with the new one (d') would entail no modification of the approximation; d would simply become bound to $\{\mathsf{Nil}, \mathsf{Cons}\,\$\,d\}$ — i.e., a cyclic reference would be created! — and `modified?` would be set to false (10). Now, because `modified?` is false, the loop used to compute the Kleene ascending chain would not be entered (11). Finally, as the result for `sum` demanded with \$ only depends on (`sum`, \$) (12), then the result d (bound to $\{\mathsf{Nil}, \mathsf{Cons}\ \$\ d\}$) would be set as *the final* result/approximation for `fn` under demand \$ (13).

Sadly, the lazy fixed point strategy can not always be applied. For example,

[29] Let's briefly explain the notion of *dependencies*. Although this was not explicitly indicated in Fig. 4.12, function `analyze-def` returns, in addition to a sequence giving the demands made for the arguments, a set of *dependencies*. A dependency is a pair made up of a function name `fn` and a strict-demand `sd`; any such dependency results from having encountered, within the body having been analyzed, a call to function `fn` under demand `sd` for which an approximation was used. Thus, dependencies indicate all the approximations on which a result depends. When there are no such dependencies (7), either because there was no function call or all such calls had final results computed for them, then we can be sure that the result which has just been computed can not change further, so it really represents a final result. If there are dependencies, but not on `fn` demanded with `sd` (8), then we are not handling a direct recursive call for `fn`, so there is no need to perform fixed-point iteration. On the other hand, when dealing with a recursive call (9), once a fixed point has been reached, i.e., after the loop at line (11), then there are two possibilities: (12): the result just computed only depends on itself, so, because a fixed point for it has been reached, we effectively have a final result; (14): the result also depends on other approximations, i.e., we just reached a fixed point within a *more global/enclosing* fixed point iteration process, in which case we return the current result so it can be used by the enclosing fixed point process, but without indicating that it is a result which is final.

suppose we are given the following function definitions:

```
Def inc l =
    { Case l of
      Nil = Nil
    | Cons x xs = Cons (x+1) (inc xs) };
Def sumInc l =
    { Case l of
      Nil = 0
    | Cons x xs = x + sumInc (inc xs) };
```

When analyzing the body of `sumInc`, a recursive call will be encountered, leading to the creation of an initial approximation. Let's denote this approximation by $T \rightarrow d$. That demand will then be propagated to the apply instruction for the call to function `inc`; in this case, the value of the strict demand d is effectively required, so that a proper version for `inc` can be created and analyzed. In order for a correct Kleene ascending chain process to occur, the effective value returned for d should then be $\perp$. Thus, our logic variable style of demands are defined so that accessing the value of an unbound demand effectively has the *side-effect* of binding that demand to $\perp$, ensuring the proper behavior for the least fixed-point computation in which it is involved.

Limiting the number of versions/specializations for a given function

In some cases, it might be decided to limit the number of versions created for any specific function definition. Such a limit can be set using the option `maximum-nb-of-versions`. The strategy used for the creation of new versions, when such a limit on the number of versions has been set, is presented in Fig. 4.14.

Below are some comments explaining the pseudocode in Figure 4.14:

(1) An already existing version can be used when its associated demand is weaker than the demand for the newly required version (d). Using this existing version will lead to less precise, although safe, results. Note that such a version may not be unique, i.e., there may be many versions, all weaker than d, which are incomparable with each other.

(2) Again, this version may not be unique. In this case, an arbitrary one is selected among all the weaker ones.

```
If the maximum number of versions for f has not yet been reached Then
   Create a new version of f specialized for demand d
Else if one of the existing version for f can be used Then                  (1)
   Use a version of f, appropriate for d, with the strongest demand
Else
   Select a version of f with the weakest demand (assume it is d')         (2)
   Suppress/invalidate the selected version (and all its dependents)       (3)
   Create a new version specialized for demand d'',
      where d'' is the strongest demand which is safe for both d and d'
End
```

Figure 4.14: Pseudocode for the creation of new function versions for f

(3) This invalidation process represents the costly part of the merging of versions. Invalidating a version may lead to invalidation of additional existing versions, and so on, transitively. Furthermore, whenever a version is invalidated and a new version is created, the fixed point computation process has to be performed again. Thus, using a small number of versions instead of allowing as many as required may sometimes be very costly.

4.5 Removal of unnecessary delays

After demand propagation and annotation has been performed, each program graph instruction is annotated with a safe demand. Among these instructions are *pseudo-delay* encapsulators, used to group together the program graph instructions representing a source program subexpression appearing in a possibly non-strict context. As mentioned earlier, these possible non-strict contexts are the following:

- Expressions used as actual arguments to function calls, e.g., $e_1, \ldots, e_k$ in `f` $e_1 \cdots e_k$;
- Expressions used as actual arguments to data structure constructors, e.g., e_1 and e_2 in `Cons` e_1 e_2;
- Expressions used as definition of global constants, e.g., e in `Def c =` e;
- Expressions used as right-hand side of let-bound identifier, e.g., `e1` in {`x1 = e1 In e`}.

In many cases, it will be possible to determine that those contexts can effectively be made into strict contexts. In such cases, the associated subexpression will not need to be delayed and the encapsulating pseudo-delay can then be removed. This is

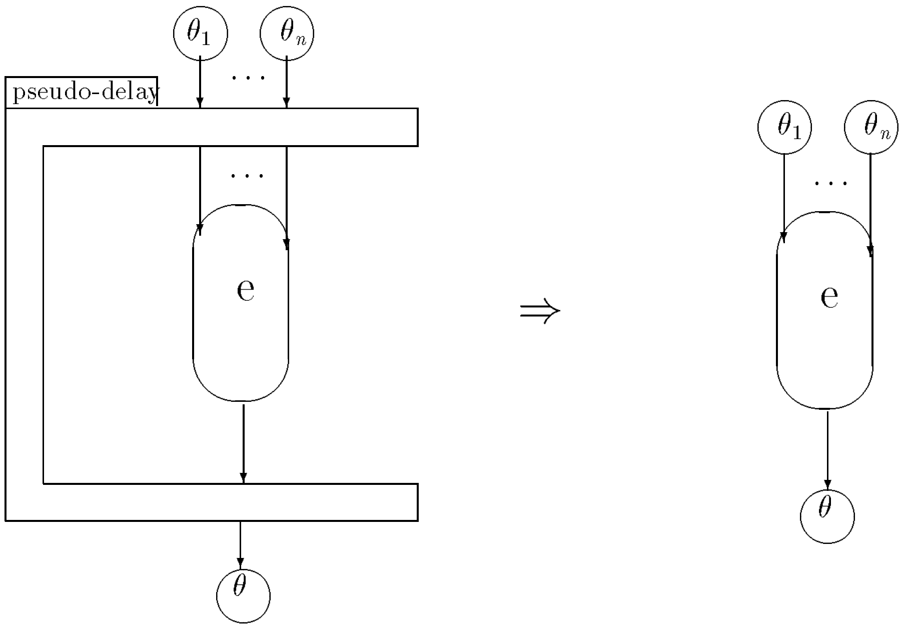


Figure 4.15: Removal of unneeded pseudo-delay

illustrated in Fig. 4.15, where the removal of a pseudo-delay simply requires splicing in the delayed subexpression in place of the pseudo-delay.

As mentioned earlier, our domain of demands allows us to recognize more than simple strictness. For example, it is possible to recognize cases where an expression is sure not to be needed, or where an expression is sure to get into an infinite loop. In both cases, the associated program graph can be simplified: whenever a pseudo-delay's demand indicates that the expression is sure not to be needed or is sure not to terminate, then the associated subexpression can simply be eliminated.

The process of pseudo-delay removal is performed using a straightforward inside-out traversal of the program graphs, searching for pseudo-delay program graph instructions. Of course, when a pseudo-delay annotated with a non-strict demand is encountered, then the delay must be preserved. In such a case, the pseudo-delay instruction is transformed into a *lazy-delay* instruction, in order to indicate to the subsequent modules that it effectively represents a delayed expression. This transformation is trivial: it only requires changing the opcode of the program graph instruction, as both pseudo- and lazy-delays have the same structure.

4.6 Insertion of forces and run-time evaluators

The run-time evaluators insertion process

Suppose we have the following function definition, where `l` is a list:

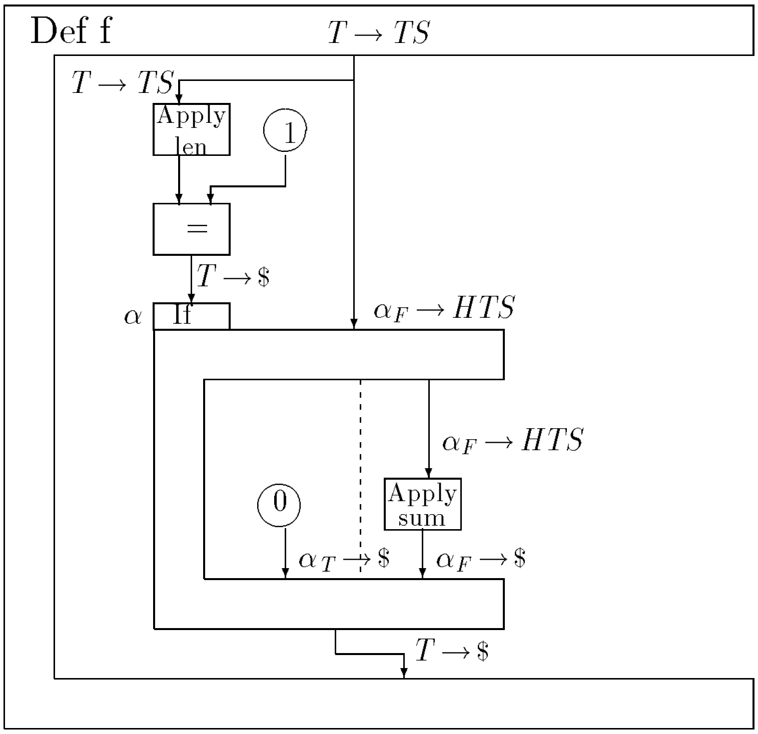


Figure 4.16: Example illustrating the insertion of run-time evaluators

```
Def f l = If (len l)==1 Then 0 Else (sum l);
```

Figure 4.16 shows the program graph for function `f` after demand propagation has been performed, assuming `f` has been demanded with \$, i.e., a scalar WHNF demand.

All we can infer, at compile-time, is that `l` will be demanded under a tail-strict demand, caused by the call to the `len` function; at the outermost level of `f`, the demand due to the call to `sum` can not be assumed to have any effect, as it is not even certain to be made. However, as soon the else-branch is entered, the situation changes: when entering the else-branch, we know for sure that all heads and tails of `l` will be needed, as `sum l` is going to be computed within that basic block. Thus, as soon as the else-branch is entered, it is safe to trigger the evaluation of all the heads and tails of `l`, which can be done in parallel with the call to `sum`. Such a concurrent call — to a run-time *evaluator* — is illustrated for the `sum` of Fig. 4.16, in Fig. 4.17.

The introduction of such run-time evaluators involves two aspects. First, the demand that can be assumed to hold within a basic block must be determined. This can be computed in the following manner: given the demand computed for the input within the basic block, we can remove the conditions which are sure to

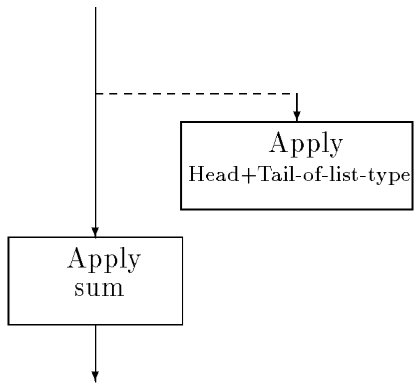


Figure 4.17: An example of a call to a run-time evaluator

hold within that basic block, i.e., the conditions which will trigger the execution of the basic block. In the example of Fig. 4.16, the demand for the input within the else-branch is $\alpha_F \rightarrow HTS$, whereas the condition holding within the block is α_F; relative to that block's triggering conditions, the demand for `l` is then $T \rightarrow HTS$, i.e., all heads and tails of the input are sure to be needed.

Once this is known, the effective level of demand of the input must be determined: in this case, the most that can be assumed is $T \rightarrow TS$, which can be obtained by propagating, in a forward direction, the demand that has been assumed for the input of the function.

Given these two demands, they can then be compared: if the demand holding within the basic block is stronger (less lazy) then what is received/propagated from outside the basic block, then the level of evaluation of the input can be *raised* using a run-time evaluator, if an appropriate one exists.

The insertion of simple forces required to trigger the evaluation of delayed expressions are handled in the same manner. For example, suppose we have the following function:

```
Def g x y = If x==0 Then 0 Else y;
```

Within the else-branch, the demand that is sure to hold for `y` is $T \rightarrow \$$; on the other hand, the received evaluation status can only be assumed to be at most $\alpha_F \rightarrow \$$. The evaluation level must then be raised from a non-strict level to a strict one, i.e., the input must be forced.

The domain of run-time evaluators

In our prototype compiler, only a small domain of run-time evaluators has been defined. This domain can express, for any algebraic type, simple-, tail- and head-and-tail-strict evaluation. The latter type of evaluation triggers the complete evaluation of data structures, i.e., it corresponds to a *hyperstrict* demand. As for tail-strict evaluation, it triggers the evaluation of the *spine* of data structures: given a recursive data structure of type T, all recursive subcomponents of type T are (recursively) evaluated, and only them, i.e., none of the other components.

As will shortly be discussed in Section F.4, there seems to be very few cases where run-time evaluators are effectively needed. Using a more complex domain of run-time evaluators might have allowed more situations where their introduction would have been possible and useful. However, as will also be discussed, implementing run-time evaluators can lead to important overhead; given the experience presented in Section F.4, it is doubtful whether this would have brought much benefits.

The implementation of run-time evaluators

Only a simple-minded implementation of run-time evaluators has been done, for reasons described in Section F.4. Essentially, a run-time evaluator is implemented as a regular function; triggering a run-time evaluator is thus realized as a regular function call.

The task of a function implementing an evaluator — each type has its own specific evaluators, a tail-strict one and a head-and-tail-strict one, as the exact code for the evaluator depends on the type definition itself — involves a straightforward traversal of the data structure in order to force the evaluation of the appropriate components. Note that, in the current implementation, no information is kept about the effective evaluation status of data structures, so that unnecessary traversals may effectively be performed (again, cf. Section F.4 for an explanation of this limitation).

4.7 Expansion of lazy-delays into regular delays

The (pseudo- and lazy-) delays used within the lazy compiler have a slightly different structure from those used within the regular Id compiler. In the lazy case, delays are used essentially to encapsulate expressions while performing the demand propagation process. Such encapsulating instructions have the same inputs as the free variables of the expression being (potentially) delayed and also have an output representing the expression's result. On the other hand, the regular delays have a different structure: *i*) they require an additional argument indicating the address of the data structure slot into which the delay is going to be stored; *ii*) they have no output port, as it is implicitly indicated by the address into which the delay is to be stored.

There were two main reasons why lazy-delays were introduced:

1. The interface to pseudo- and lazy-delays allows a simpler traversal of the program graphs when performing demand propagation, as the result arcs are explicitly present.
2. In the regular Id compiler, only slots of data structures can be delayed. In the lazy compiler, in order to allow for delayed function arguments, single-slot data structure (called *indirections*) have been introduced. These indirections are not initially present in the program graphs, as they might not even be required if an argument is strict.

Indirections are useful to deal with arguments which are passed directly to another function, as is the case with `y` in the following example: [30]

```
Def f x y z = If x==0 Then g y (z + 1) Else h (z - 1) y;
```

Assuming that, in both the calls to `g` and `h`, `y` occurs in a non-strict argument position, then the thunk associated with `y` can also be used for the calls to `g` and `h`, i.e., delayed actual argument expressions can be shared. There are two main advantages thus obtained: this precludes the (costly) creation of a new thunk; the result of the actual argument expression evaluation can effectively be shared. Of course, there is a small price to pay: every access to an argument effectively requires going through an indirection, i.e., a *fetch*, in order to obtain the effective value.

[30] Cases like these often occur when lambda-lifting is performed [Joh85], as happens in the Id compiler. For example, suppose we have the following definition:

```
Def f x y z = { g a = x * (y + a); In (g 1) + (g (z + 1)) };
```

Lambda-lifting would transform this definition into the following ones:

```
Def f x y z = (g_of_f 1 x y) + (g_of_f (z + 1) x y) ;
Def g_of_f a x y = x * (y + a);
```

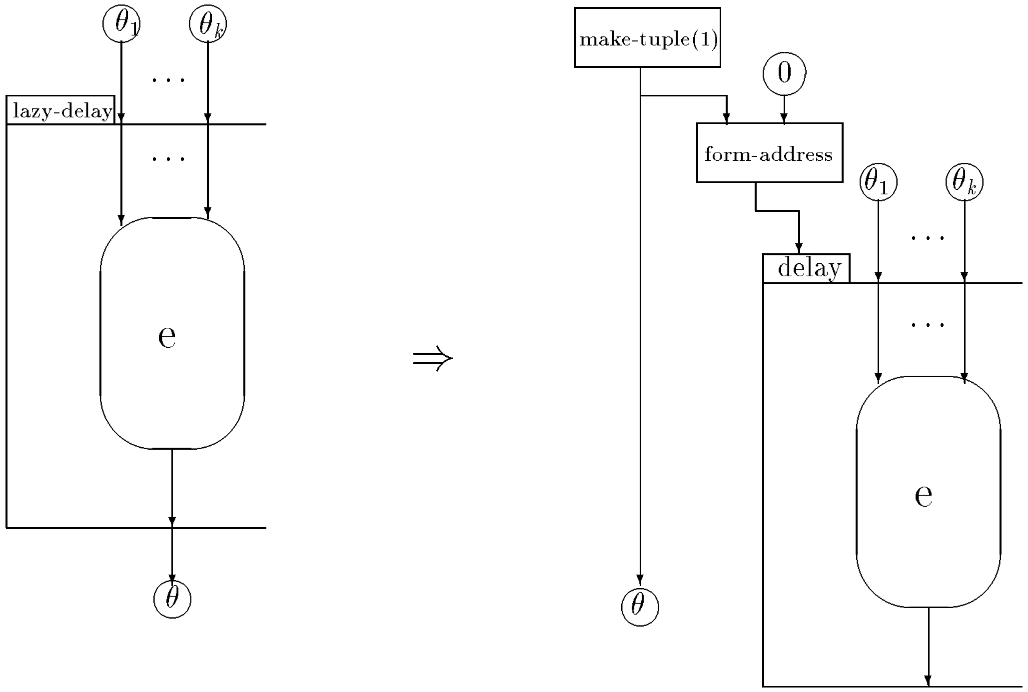


Figure 4.18: Expansion of a non-strict argument using a delayed indirection

The purpose of the last phase of the lazy compiler is then to introduce the appropriate indirections for delayed function arguments and to transform the remaining lazy-delays into regular delays — i.e., into delays where the result, when evaluated, will feed a specific data structure slot (indirections, in the case of function arguments) specified by the appropriate input.

Figure 4.18 illustrates the transformation that would occur for a non-strict argument to a function call. The *make-tuple*(1) instruction is used to create the indirection, i.e., the 1-slot L-structure used to contain the delayed argument; as for the *form-address*, it computes the address of that slot into which the result of the delay is to be stored.

In addition to the expansion of lazy-delays into regular delays, a number of simple optimizations are also performed by the last phase of the lazy compiler. Essentially, these optimizations consist in removing certain special cases of forcing, more precisely, delays whose only task is to embed the forcing of a delayed object. Two such cases are presented in Figures 4.19 and 4.20. In Fig. 4.19, c represents a global constant which is not sure to be needed, thus has to be forced before being used. Similarly, in Fig. 4.20, the only task of the delay is to embed the forcing of some non-strict variable; in this case, the delay produced for that variable can also

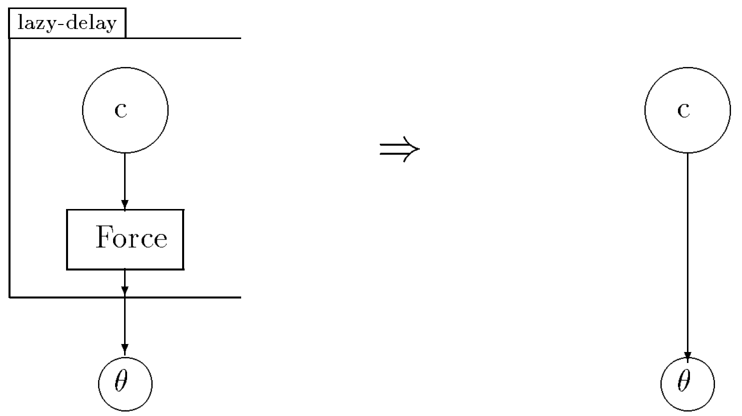


Figure 4.19: Special case for global non-strict constant

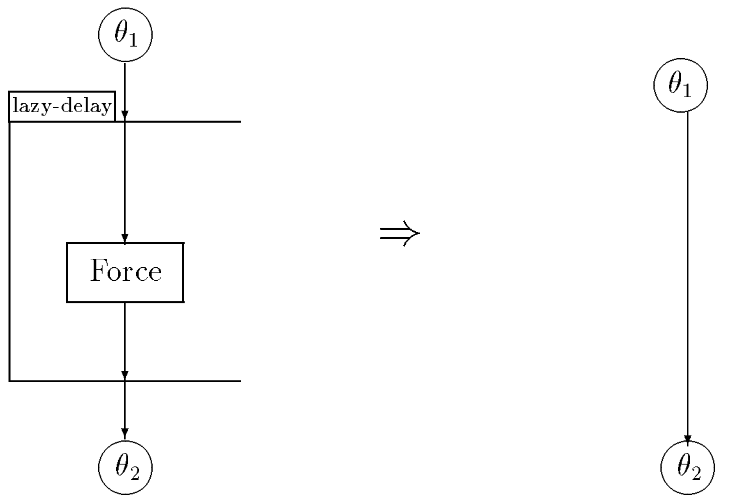


Figure 4.20: Special case for non-strict identifier

be used directly. [31]

[31] There are other similar special cases. For example, a variation of Fig. 4.20 is when the force is not directly included in the delay but, instead, is received as a free variable of the delay. In this case, the delay can also be removed and the non-forced value used directly.

Chapter 5

Experimental results and analysis

The initial motivating hypothesis/conjecture that lead to the development of this thesis was that lazy languages, if implemented within an appropriate framework (inherent parallelism, complex/precise compile-time strictness analysis, support for run-time parallel demands, etc.), might be able to provide sufficient parallelism to be exploited by a fine-grain parallel machine. This initial assumption was motivated by the fact that, for lenient languages, *too much parallelism* can often be generated, so that it is necessary to *limit* the quantity of parallelism generated by programs using various forms of *throttling mechanisms* [AC86b, CA88].

However, as we will see in the coming sections, this initial conjecture was found to be invalid for the type of programs examined, i.e., programs dealing with various forms of dynamic data structures using recursively defined functions — which contrast with the programs for which too much parallelism were obtained in lenient languages, i.e., (scientific-style) code dealing with regular arrays through looping constructs. Furthermore, as long as we intend to emulate faithfully the work performed by a *sequential lazy* interpreter, then the amount of available parallelism that can be exploited is even more limited.

The first section of the present chapter, Section 5.1, presents the various program benchmarks which have been used to perform our experiments; it also describes the execution parameters that were measured for the experiments in order to compare the selected implementation strategies.

Section 5.2 then presents the major factors/parameters that influence the precision of the strictness analysis process — thus the efficiency of the resulting optimized programs — and indicates the values which were selected to set the various compilation options for the optimizing lazy compiler.

This optimizing compiler will then be used, in Section 5.3, to examine the two major questions that were the initial motivation of this thesis:

- Can an implementation of a lazy functional programming language provide sufficient parallelism to be exploited by a fine-grain data-driven parallel machine?
- Can laziness play the role of a *throttling* mechanism in order to avoid large quantity of useless parallelism?

Of course, these two questions are related: if there is not sufficient parallelism exposed by our optimizations, then clearly laziness is a *throttling* mechanism ... although one which is too extreme! Our goal is to obtain an appropriate balance: sufficient parallelism to efficiently utilize the underlying machine, but not too much that the machine is overloaded and the parallelism can not be exploited.

In order to do this, a number of benchmark programs will be run and various statistics will be collected and analyzed. All benchmark programs will be run using each of the following implementation strategies:

- Non-optimized lazy evaluation;
- Optimized lazy evaluation;
- Lenient (eager non-strict) evaluation.

As we will then see, in a certain way, *laziness and parallelism are incompatible*, that is, as long as we try to remain faithful to the usual view of laziness, then very little parallelism can be obtained. A first overview of the results to be obtained is presented in Figure 5.1, where the average parallelism obtained for various *real lazy* benchmarks [1] is presented for the non-optimized and the optimized lazy implementation. More precisely, the figure presents, for each of the indicated benchmark program and for two possible implementation strategies (pure lazy, i.e., unoptimized, vs. optimized lazy) the average number of dataflow instructions that were available for execution at each time step. It can be noted that for such lazy programs, the average parallelism is always around 5, except for one particular case,

[1] That is, programs that require a lazy evaluation strategy in order to terminate.

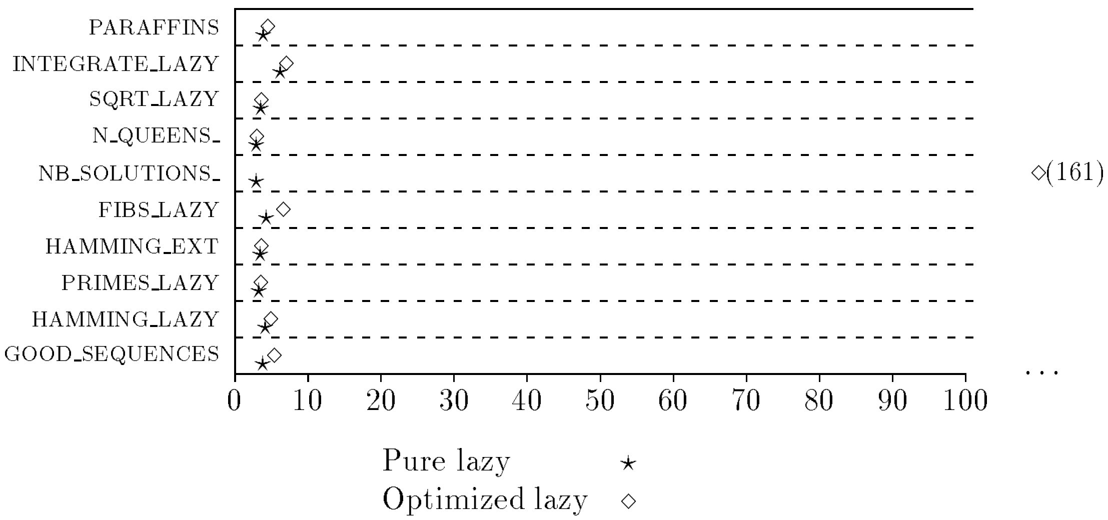


Figure 5.1: Average parallelism for various lazy benchmarks using non-optimized vs. optimized lazy implementation

viz., `NB_SOLUTIONS_`, for which, in the optimized implementation, the average parallelism was 161. Similar results are presented in Fig. 5.2, this time for benchmark programs that do not really require lazy evaluation: in this case, the results obtained with the lenient variant of the compiler, i.e., with a non-lazy implementation, are also presented. [2]

Similar summarizing figures are also presented in Figures 5.3 to 5.6. Figs. 5.3 and 5.4 present statistics concerning the total number of instructions executed. More precisely, Fig. 5.3 presents the ratio of the total number of instructions executed by the optimized lazy implementation relative to the purely lazy strategy, in the case of programs really requiring laziness. As for Fig. 5.4, it presents similar information, but for programs not requiring laziness, so that both the optimized lazy and lenient implementation can be compared. As for Figs. 5.5 and 5.6, they present similar information, but this time for the critical path length. Clearly, it can be seen that the optimizing compiler, when compared with the pure lazy implementation, can reduce significantly both the total number of instructions executed and the critical path length.

[2] Note that there are a few cases for which the average parallelism of the pure lazy implementation is slightly better than for the other two cases. Such anomalies are mainly due, strangely, to the much higher overhead associated with the implementation of lazy evaluation: in those cases, the length of the critical path length is only increased slightly, because the result to be produced is a scalar value, although the total number of instructions executed is largely increased, due to the lazy evaluation overhead, leading to an increased average parallelism.

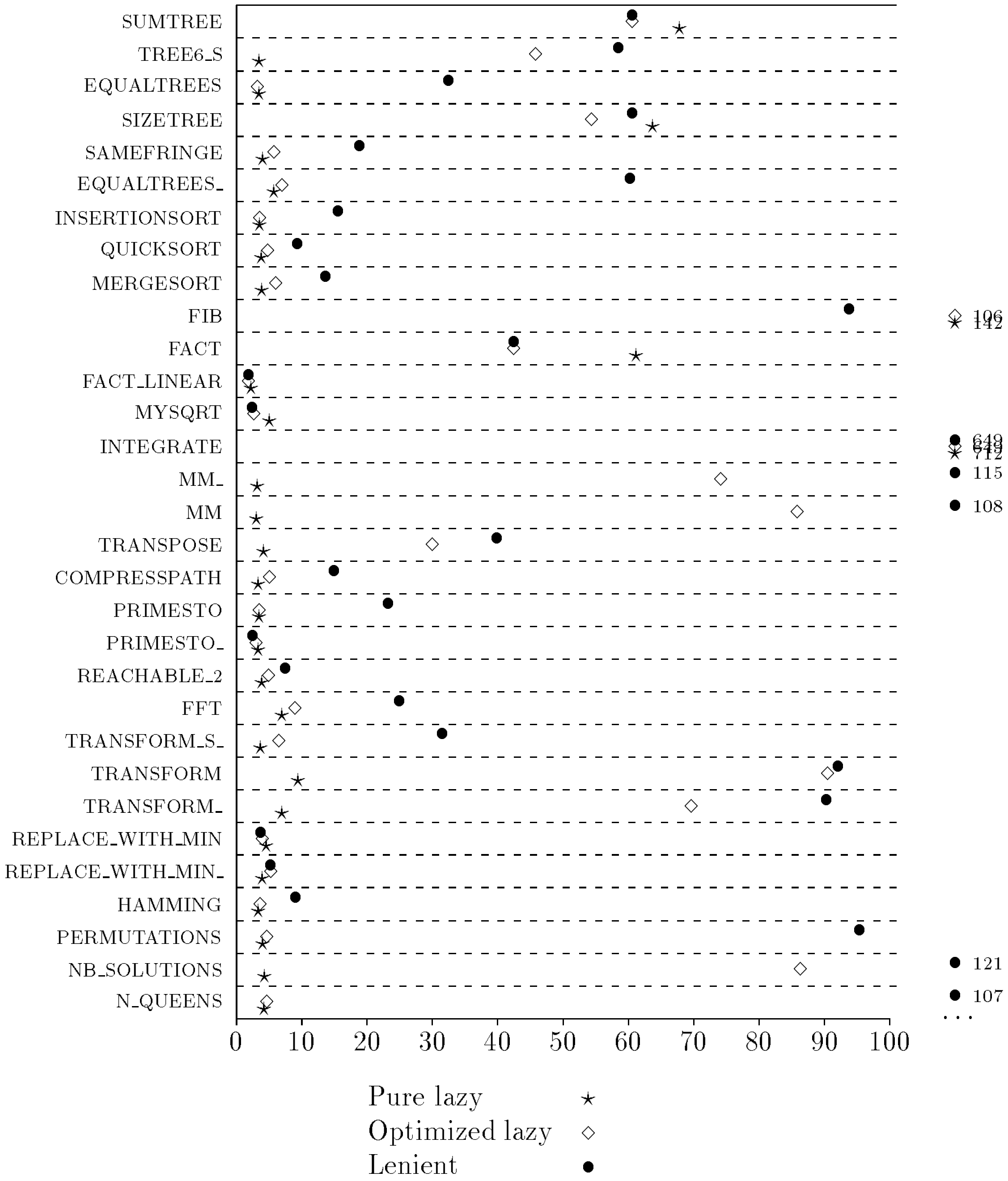


Figure 5.2: Average parallelism for various benchmarks under different evaluation strategies

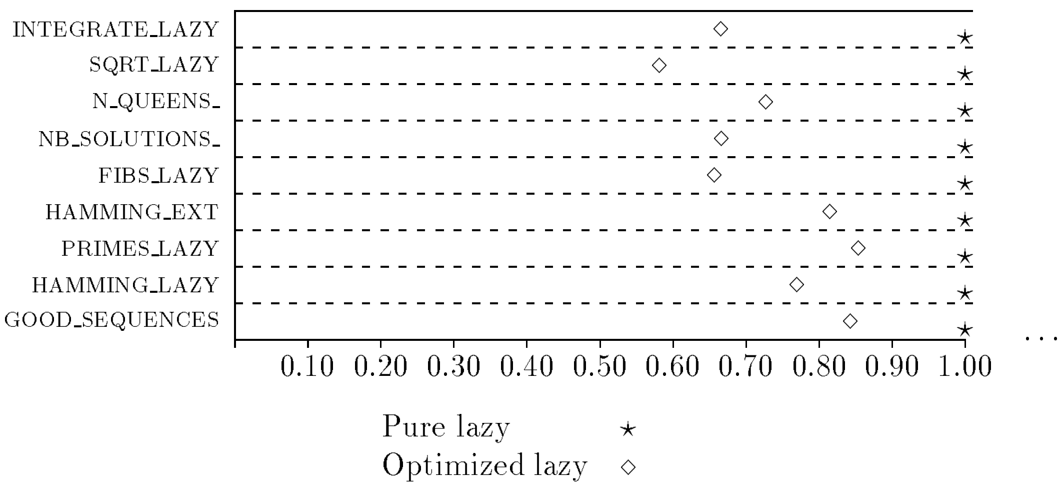


Figure 5.3: Ratio of total number of instructions executed for various benchmarks using non-optimized vs. optimized lazy implementation

The reason explaining the fact that little parallelism can be obtained for the lazy variants is because the standard view of laziness is based on a strictly sequential evaluation model, where even the printing of the final value is done sequentially, thus leading to a major bottleneck. Thus, in Section 5.4, we will examine various ways to go beyond the *pure lazy* approach but still preserves the major advantage of lazy evaluation, i.e., the possibility to deal with potentially infinite data structures. In order to do this, we will define an alternative printing process, more suitable for parallel machines. Then, we will introduce a new form of *head-strictness*. Finally, we will examine how various forms of *speculation* can be defined, in a tentative to augment the parallelism but still ensure that the amount of work performed remains *bounded* (*bounded speculative laziness*).

Although all the above approaches (strictness analysis, speculation, etc.) will have a positive effect on the efficiency of the resulting program, sadly, we will also see that laziness is effectively a drastic throttling mechanism, and that lazy programs, when executed on highly parallel machines, may not exploit them very efficiently, as will be discussed in Section 5.5.

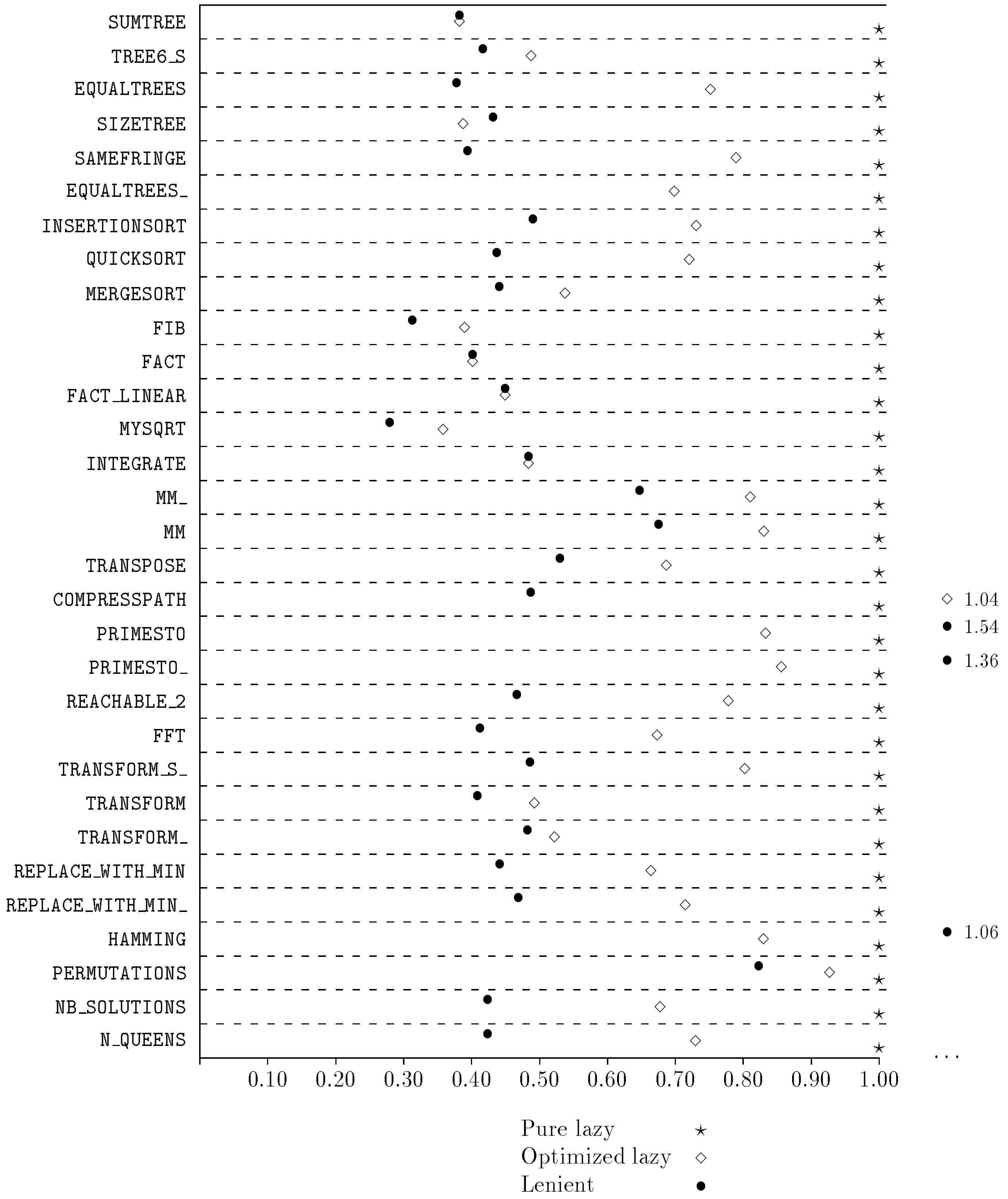


Figure 5.4: Ratio of total number of instructions executed for various benchmarks under different evaluation strategies

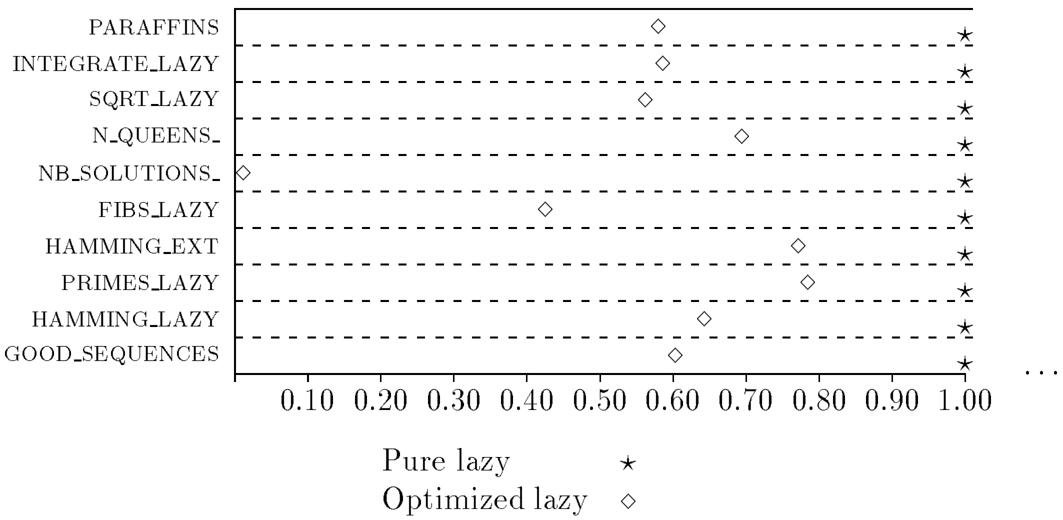


Figure 5.5: Ratio of critical path length for various benchmarks using pure (unoptimized) vs. optimized lazy implementation

5.1 The benchmark programs and the execution parameters being measured

The benchmark programs

The benchmark programs that have been used to perform the experiments described in the following sections have been taken from a wide variety of sources: [Hen80], [Tur81], [Hug84], [Tra88], [Hel89], [Bir84], [All93], [HS88], [NA90], [BHY88], [You89], [SPR90], [LH92].

The benchmark programs can be divided along the following major categories:

- Basic scalar computations not requiring lazy evaluation: numerical integration, factorial, square root computation, fibonacci [SPR90].
- Scalar computations using lazy evaluation: numerical integration and square root approximation [Hug84].
- Basic list-manipulating functions: length, append, reverse, last, take, drop, etc.
- Basic tree-manipulating functions: size, sum, equalTrees, sameFringe [Hel89].
- Various programs using lists to represent matrices: transpose, matrix multiplication [You89], Wang's algorithm for solving a system of linear equations

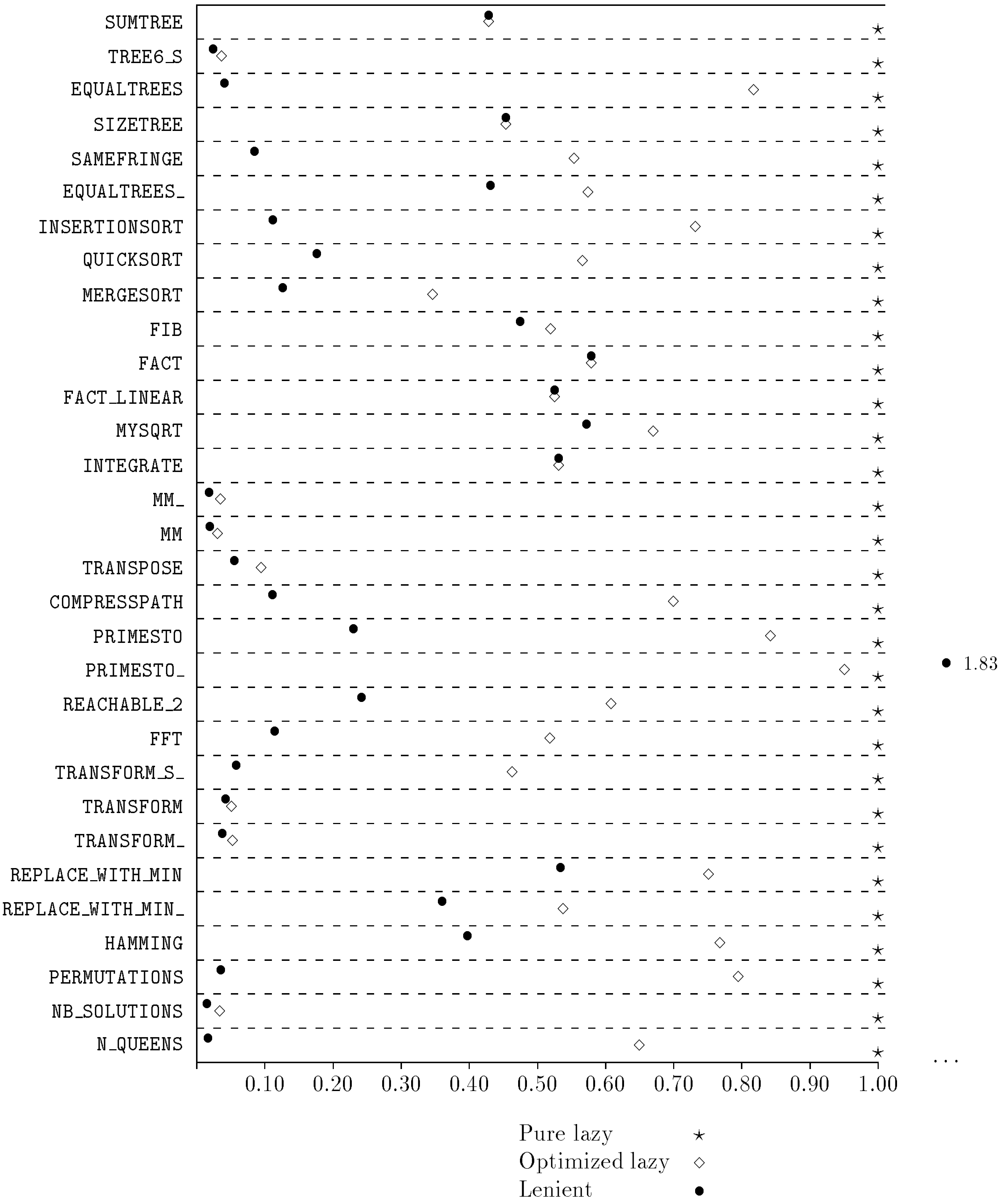


Figure 5.6: Ratio of critical path length for various benchmarks under different evaluation strategies

based on a tri-diagonal matrix [HL93].

- Fast Fourier Transform [SPR90].
- N-queens problem [Hel89].
- Paraffins problem [Tur81, HS88].
- Various programs dealing with lists or lists of lists: computing the reachable nodes in a graph [Maa92], finding primes numbers, path compression (also known as recursive doubling) [SPR90], extended hamming problem [NA90].
- Various problems solved using *circular* programs: computing prime numbers, building hamming sequences, solving the n-queens problem [All93], building a list/tree with the same structure but with all elements replaced by the minimum element [Bir84].
- Sorting algorithms: insertion sort [Hel89], mergesort, quicksort [SPR90].

There is an important point that should be stressed concerning the benchmarks used in our experiments: among the various programs included in the benchmarks, there are programs which do not really require the power of lazy evaluation to work correctly; on the other hand, there are indeed some programs which would not terminate if executed using a non-lazy evaluation strategy. In other words, among our programs, there are effectively some ($\approx 20\%$ of them) which manipulate potentially infinite data structures.

This point is very important and is in contrast with the benchmarks used in [SPR90]. In that paper, they claim that, using their approach to strictness analysis — approach which uses a very small domain of abstract values — they were able to eliminate all of the delays in the programs that were analyzed. However, when examining these programs, we realize that none of them really requires the power of lazy evaluation, i.e., all of them would execute and terminate correctly using a non-lazy evaluator. In fact, most of their programs do not even require the expressive power of non-strictness, i.e., they would also execute and terminate correctly using a *strict* evaluator.

Note that although some of our benchmark programs deal with arrays represented with lists, they do not deal with *true* arrays, as these have not been implemented in our compiler. Although implementing such arrays would not be a major problem, they were not considered a major element to be included in our prototype compiler because programs requiring laziness rarely involve the use of arrays: laziness is really important in order to deal with potentially infinite data structures, and arrays are not such structures!

Execution parameters measured

All our benchmarks are executed, after having been compiled with the setting of the compile-time options described in the following section (Section 5.2), under an *idealized execution model*, i.e., assuming an infinite number of processors with no communication latency. There are three key parameters that we measure and analyze under this model:

- the length of the critical path, i.e., the minimum number of time steps required to execute the program;
- the total number of instructions executed;
- the average parallelism, i.e., the average number of instructions executed per time step (= total number of instructions / critical path length).

We present our experimental data relative to the critical path length and the total number of instructions using histograms. Each bucket of an histogram cumulates the number of benchmark programs for which the use of a given option instead on another option lead to a certain reduction of the analyzed statistics. In most cases, we will simply present the data relative to the critical path length, as the data for the total number of instructions usually result in the same histogram. In a few cases where the resulting histograms are sufficiently different, both will be presented.

5.2 Parameters selected for the optimizing lazy compiler

This section examines the major factors/parameters that influence the precision of the strictness analysis process and, thus, the efficiency of the resulting programs. Each of the major factor is briefly introduced and the choice of an appropriate value to be used in setting the related parameter of the optimizing compiler is briefly motivated. Detailed explanation of these parameters, together with detailed statistics showing their effect, are presented in App. F.

The major factors/parameters influencing the precision of the strictness analysis process are the following:

- Size of the domain of demands: Using a larger domain of demands clearly increases the precision of the strictness analysis process. However, it also makes the fixed point computation process more costly. It can be seen, in App. F.1, that increasing the size of the domain of demands has an important effect when going from 1 to 2; however, further increase bring less important improvements. Because of this, a domain of demands of size 2 will be selected.
- Type of abstract conditions: The compiler defines three types of abstract conditions: `:None`, `:Intra-procedural` and `:Inter-procedural`. Going from no abstract conditions to intra-procedural ones has an important impact, whereas the effect is negligible when going from intra-procedural to inter-procedural, as can be seen in App. F.2. Because intra-procedural conditions are much simpler to handle, they will be selected for use within the optimizing compiler.
- Maximum number of versions: This parameter specifies the maximum number of distinct versions that can be created for any given function. A small number of distinct versions (2–3) seems to be sufficient in most cases (cf. F.3). However, because imposing a limit on the number of versions effectively increases the cost of compilation, no such limit will be imposed.
- Run-time evaluators: It turns out that allowing run-time evaluators has a negligible impact on the reduction of the critical path length although it increases the total number of instructions executed (App. F.4). Furthermore, only a very small number of benchmarks effectively use such evaluators. Thus, run-time evaluators will not be used within the optimizing compiler.
- Type of initial demand: Without speculation, there are two possible initial demands that can be made for the result of the program: simple strict, which represents evaluation to WHNF; (sequential) head-strict, which evaluates the first (leftmost) component and delays the remaining ones. [3] Using an initial head-strict demand instead of a simple strict one has an important impact on the efficiency of the resulting program, as it allows some strictness information to be propagated inside data structures, whereas simple strictness leads to

[3] Cf. Section 5.4.3 for a more thorough explanation of the concept of sequential head-strictness.

most components being delayed (cf. App. F.5). Thus, an initial head-strict demand will always be used to trigger the lazy compilation process.

- Head-strictness: Allowing head-strictness to be inferred for the arguments of functions effectively has, when combined with an initial head-strict demand, a positive impact of the efficiency of the resulting program, as it allows more thunks to be removed (cf. App. F.6). Thus, head-strictness will be allowed in the optimizing compiler.

- Implementation of data structures: Implementing data structures using the mechanism of L-structures (cf. Section 4.1, p. 4.1) can be done in two ways: using direct thunks or using indirect ones. It can be seen in App. F.7 that, when strictness analysis is performed, implementation of data structures using direct thunks is preferable over indirect ones, whereas the opposite is true for a purely lazy implementation. Thus, whereas the optimizing compiler will use direct thunks, the non-optimizing (purely) lazy variant will use indirect ones.

To summarize, for the optimized lazy implementation, the following setting of the parameters will be used:

- Size of the domain of demands = 2;
- Data structure components implemented using direct thunks; [4]
- Head-strict initial demand + head-strict demands allowed for function arguments;
- Intra-procedural abstract conditions;
- Unlimited number of function versions;
- No run-time evaluators.

5.3 Comparison of obtained parallelism

This section will compare the parallelism the can be obtained using the following three implementation strategies and using the setting of the parameters mentioned in the previous section (cf. above):

[4] For the pure lazy variant, indirect thunks will be used, not direct ones; the reason for this choice was mentioned above and is explained in more detail in App. F.7.

- Non-optimized lazy;
- Optimized lazy;
- Lenient. [5]

As it will soon be seen, using the optimized lazy implementation has an important advantage over the pure lazy one in terms of reducing both the total number of executions executed and the length of the critical path. It will also be seen, however, that the average parallelism remains rather low in both cases, whereas it can be much higher when laziness is dropped (viz., when a lenient, i.e., non-strict and non-lazy, implementation strategy is used) ; from this it will be concluded that an alternative view of laziness, better suited for parallel machines, has to be explored, leading to Section 5.4.

5.3.1 Comparing the pure lazy and the optimized lazy implementations

Figure 5.7 shows the effect, on the reduction of critical path length, of going from the non-optimized lazy implementation to the optimized one. Clearly, using the optimized lazy variant leads to important reductions of the critical path length.

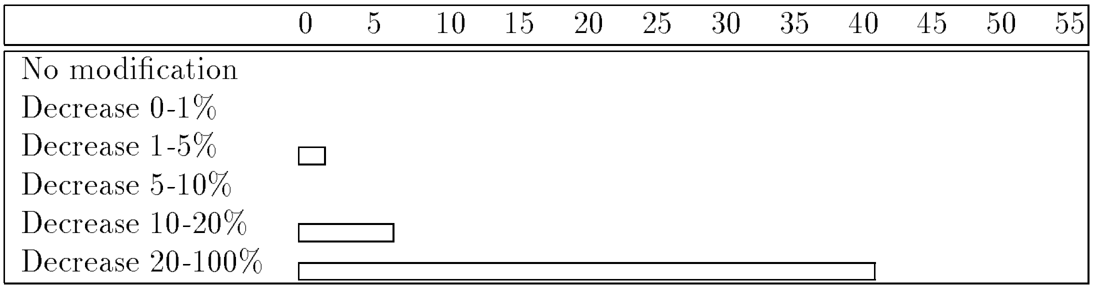


Figure 5.7: Pure vs. optimized lazy: Effect on critical path length

Note that we won't present the histogram for the total number of instructions, as it is identical to the one for the critical path length but for a single exception. This exception is the `COMPRESSPATH` benchmark program, for which the use of the optimizing lazy compiler, although it does not increase the critical path length, leads to an increase (4.6%) for the total number of instructions. This anomaly can easily be explained by noting the fact that the purely lazy variant uses indirect thunks,

[5] Note that the lenient strategy will be used only for programs which are sure to terminate even when a non-lazy evaluation strategy is used. Cf. Section 5.4.4 for more detail.

whereas the optimizing compiler uses direct ones. As noted in Section F.7, the `COMPRESSPATH` program is one of the cases for which the use of indirect thunks was preferable.

To really confirm the role of indirect thunks, Figure 5.8 presents a comparison between the purely lazy variant vs. the optimizing lazy compiler, but this time having both implementations use direct thunks. As can be seen, the advantage of the optimizing compiler over the purely lazy variant is even more clear.

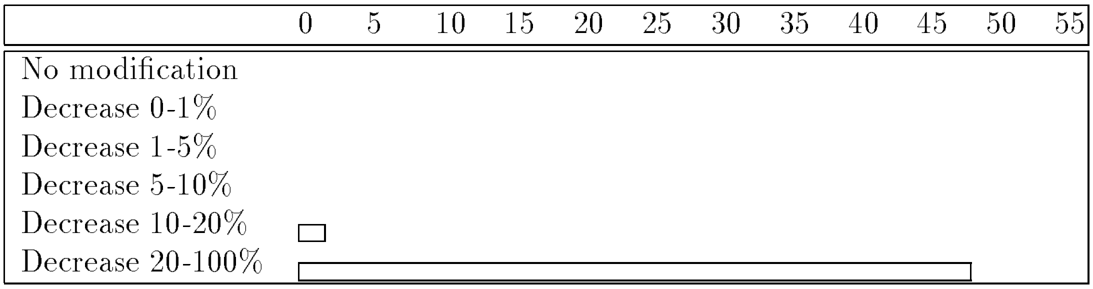


Figure 5.8: Pure (direct thunks) vs. optimized lazy: Effect on critical path length

Although the effect on both the critical path length and on the total number of instructions is clear, the effect on the average parallelism, as can be seen from Figure 5.9 is much less clear: there seems to be as many cases where the average parallelism increases as there are cases where it decreases.

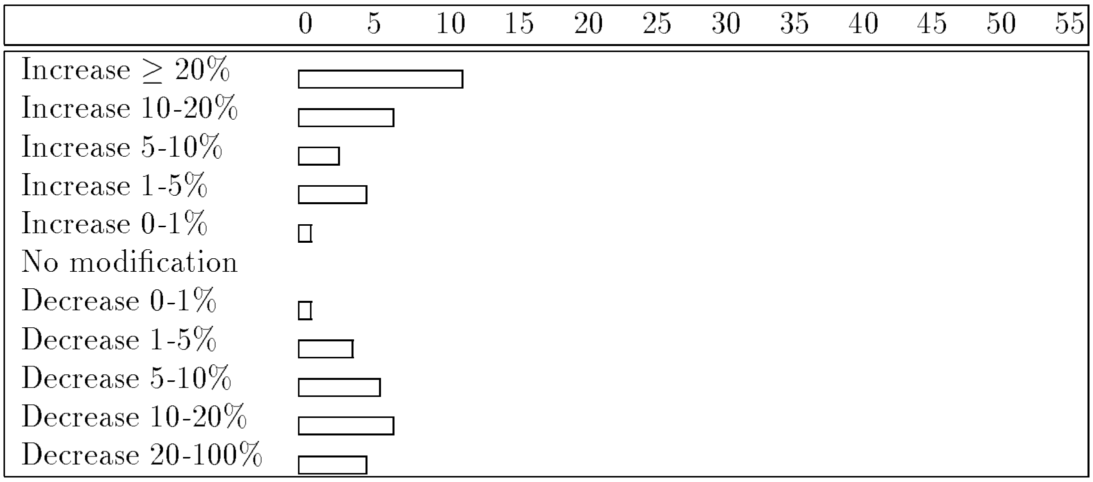


Figure 5.9: Pure vs. optimized lazy: Effect on average parallelism

The distribution of the average parallelism for both implementation is presented in Figures 5.10 and 5.11. The main thing to note is that, in both cases, most of the programs have a very low average parallelism, i.e., less than 5 instructions per time step. Clearly, this is quite insufficient parallelism to efficiently exploit a parallel machine.

A question that comes to mind, given these results, is whether this is due to laziness *per se* or, instead, is intrinsic to the benchmarks being analyzed. The next section will answer that question, i.e., whether the benchmark programs effectively contain much parallelism or not.

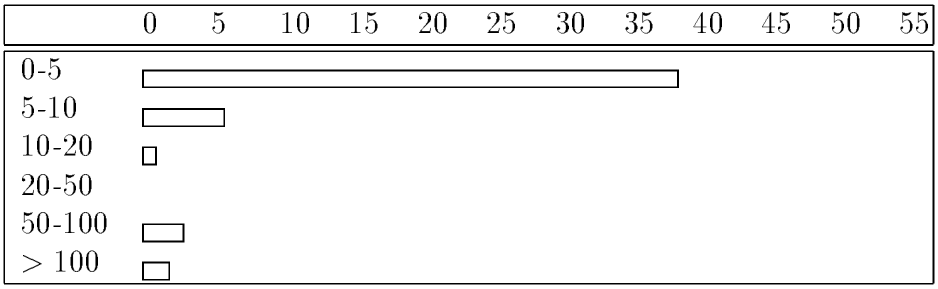


Figure 5.10: Average parallelism for pure lazy implementation

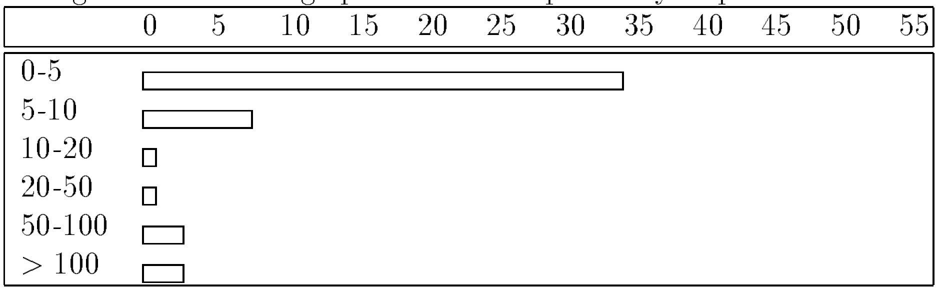


Figure 5.11: Average parallelism for optimized lazy implementation

5.3.2 Comparing the optimized lazy and the lenient implementations

Figure 5.12 shows the effect, on the reduction of critical path length, of going from the optimized lazy implementation to the lenient one. As for Figure 5.13, it shows the effect on the total number of instructions. [6]

In all cases but a few, the critical path length and the total number of instructions are reduced when going from the optimized lazy compiler to the lenient variant. The exceptions are easily explained: in all cases, the lenient variant performs work which is not really needed when compared with what is done by a lazy variant; thus, the lenient variant performs additional useless work. Let's have a look at some examples:

[6]Note that not all benchmark programs have been included: only those for which lenient evaluation was sure to terminate, i.e., those requiring a lazy interpreter were not included.

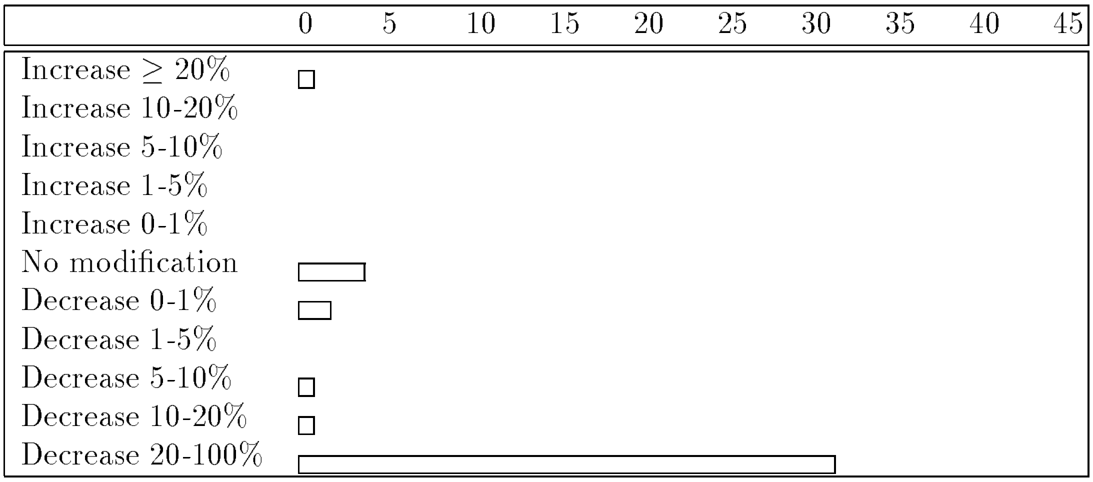


Figure 5.12: Optimized lazy vs. lenient: Effect on critical path length

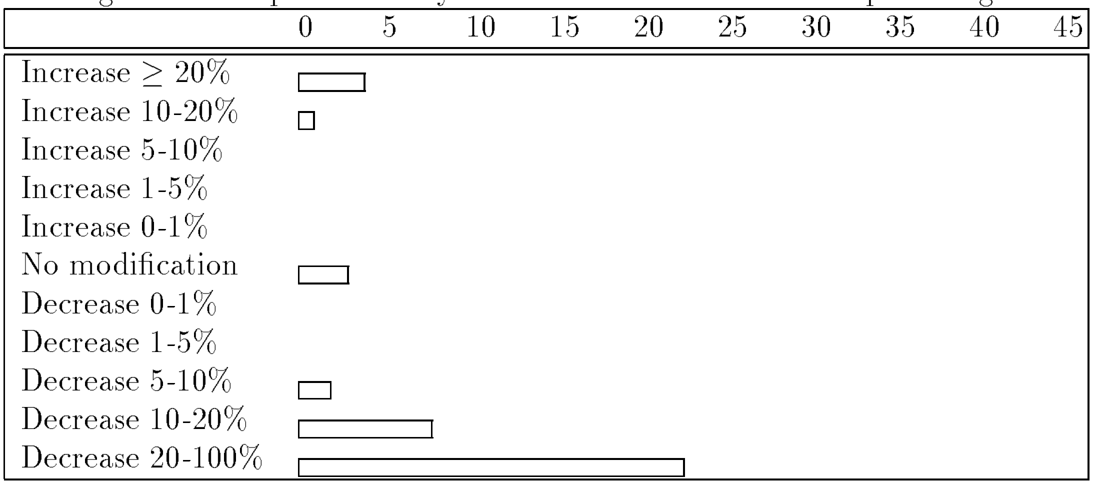


Figure 5.13: Optimized lazy vs. lenient: Effect on total number of instructions

- `SIZETREE`: When computing the size of a tree, the lazy compiler can infer that the elements of the tree are not needed, and so need not be evaluated at all. Thus, the argument to the function `sizeTree` only needs to have its spine built. The lenient compiler can not infer such information, and thus evaluates the elements of the trees even though they will never be needed/accessed.

- `EQUALTREES_`: When determining whether two trees are equal, the lazy interpreter will stop evaluating and traversing the trees as soon as some difference is found (lazy/short-cut conjunction). On the other hand, the lenient variant will *keep on* comparing the two trees, even after some difference has been found (strict conjunction).

- In the other cases, the resulting programs generate results which are quite large. In the lazy variant, only the parts specified by the selected printing-level get evaluated and printed (cf. Section 5.4.1). In the lenient case, although only the same portion gets (pseudo)forced, the rest of the result is evaluated anyhow, as nothing gets delayed. Again, additional work gets performed over what is done by the lazy variant.

The effect described in the latter case can be seen more clearly by setting the printing-level to 0. In this case, in the lazy variant, no work is done over what is initially required by the initial head-strict demand, i.e., for all programs the result is almost entirely delayed. This is presented in Figures 5.14 and 5.15, where the optimized lazy variant is compared with the lenient one, in both cases using a printing-level of 0: there are now many more cases where the lenient approach has a larger critical path length and total number of instructions.

What is more interesting is the distribution of the average parallelism for both implementation, as presented in Figures 5.16 and 5.17. Although there were a few cases (6) for which the average parallelism decreased (on the average, on the order of 10%), in most of the cases (35) the average parallelism increased. Thus, these figures answer the question asked in the preceding section: the small amount of parallelism obtained from the optimized lazy implementation is clearly an effect of the faithfulness to the lazy evaluation model, as the lenient model can generate much more parallelism.

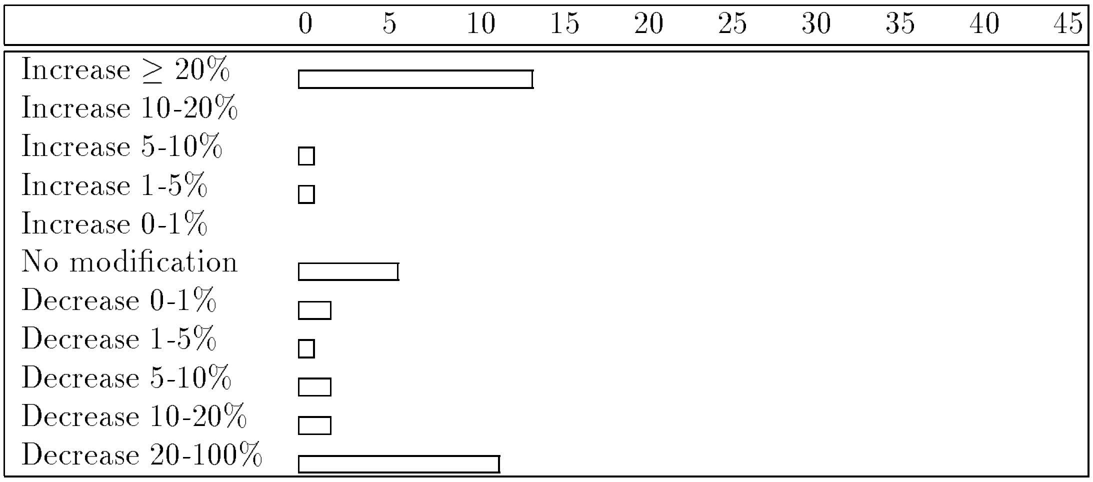


Figure 5.14: Optimized lazy vs. lenient (printing level = 0): Effect on critical path length

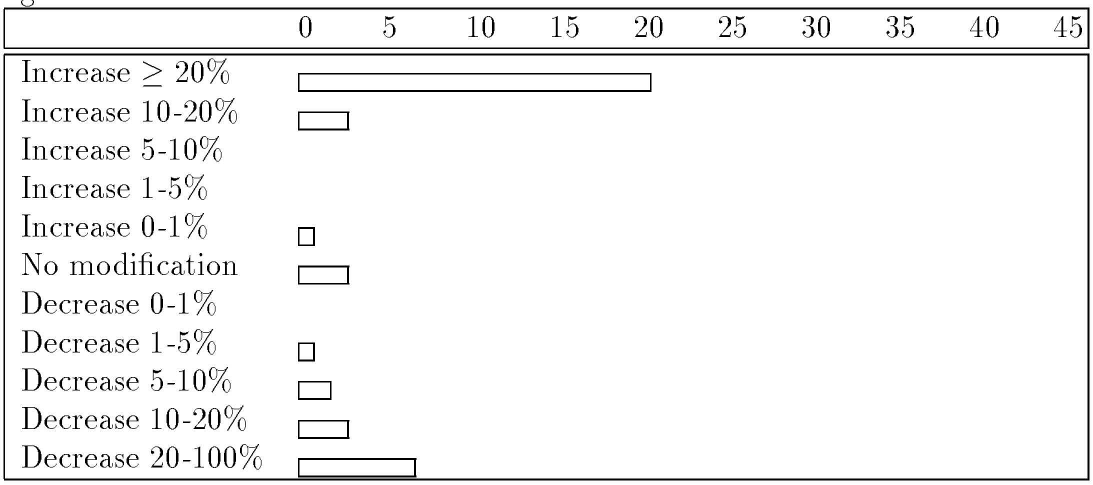


Figure 5.15: Optimized lazy vs. lenient (printing level = 0): Effect on total number of instructions

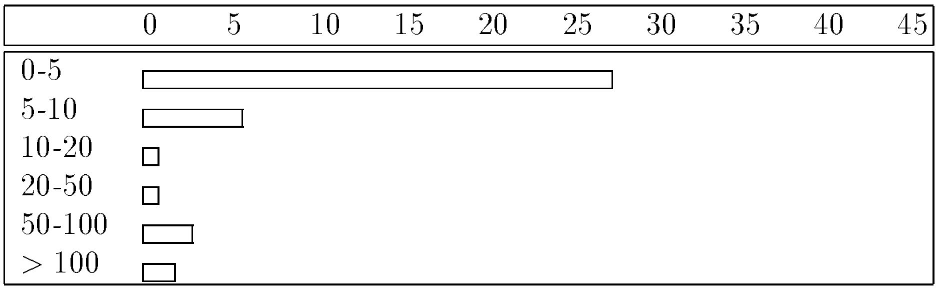


Figure 5.16: Average parallelism for optimized lazy implementation

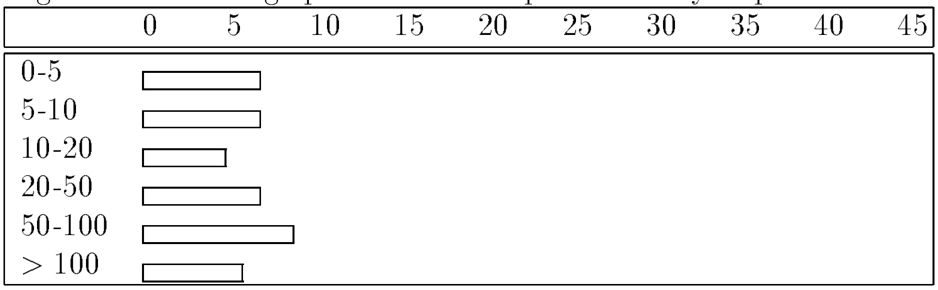


Figure 5.17: Average parallelism for the lenient implementation

5.3.3 Can a lazy implementation provide sufficient parallelism for a highly parallel machine?

A major question that lead to this thesis was whether it was possible, for a lazy implementation of a functional language, to provide sufficient parallelism to be exploited on a fine-grain parallel machine. The previous figures answer this question by the negative!

Of course, given that a lazy implementation of a functional language is required, it is clearly advantageous to use an appropriate optimizing compiler that performs compile-time strictness analysis, in order to remove delays as much as possible: as was seen in Section 5.3.1, this brings a major improvement to the efficiency of the resulting programs by reducing both the total number of instructions and the length of the critical path.

On the other hand, on average, the average parallelism is barely changed by going from a pure lazy implementation to an optimized one, and there is a quite simple explanation to this phenomena: the average parallelism is essentially defined as the total number of instructions divided by the critical path length; if both decreases (more or less) in the same proportion, as happens when going from the pure lazy to the optimized lazy variant, then the average parallelism essentially remains unchanged!

At this point, from the above data, we must thus conclude that being strictly faithful to the lazy semantics seems to be *incompatible* with obtaining parallelism. In a certain way, this should come as no surprise: the basic interpretation of what it means to be lazy has always been defined for a sequential interpreter/evaluation model. In the next section, we try to extend the notion of laziness in order that it be more appropriate for a parallel execution model and examine whether more parallelism can be obtained.

5.4 Beyond pure laziness

This section explores different strategies that attempt to improve the parallelism from lazy programs: an alternative parallel printing process, a new notion of head-strictness and two forms of speculation appropriate for lazy evaluation. It will be seen that the first strategy has a rather important impact on the parallelism; as for the others, their impact on parallelism *per se* will be smaller, although they will still have an important effect on the efficiency of the resulting programs by reducing both the critical path length and the total number of instructions.

5.4.1 Sequential vs. parallel printing

We first examine the major factor influencing the parallelism, factor which represents the main difference between the various lazy implementations and the lenient one: the *printing process* used to force and print the result of programs. [7]

In the lazy implementation, our goal was to remain faithful to the lazy semantics as defined by a classical sequential lazy interpreter. Because of this, the notion of head-strict initial demand was used (cf. Section F.5) to represent the demand made initially for the result of a program. Now, given such a demand, the result printed by most programs, using the GITA interpreter, is quite simple: it consists essentially of delayed thunks, as whenever the GITA printer encounters a delayed thunk, it simply prints the value "#`Thunk`" without forcing it any further. For example, suppose we have the following program:

```
Type list = Nil | Cons N list;
```

[7] In this section, the process of printing/forcing is explained informally through examples. For a more detailed presentation, including the relationships between compile-time speculation and run-time printing, see Appendix G.

```
Def from k = Cons k (from (k+1));
Def program1 = from 0;
```

Given an initial head-strict demand, the result printed for `program1` would simply be the following value: "*Cons* 0 #`Thunk`".

In order to have more elements printed, our compiler applies to each program result/answer an appropriate *forcing* function. The goal of this function is simply to (partially) traverse the result value, in order that when it gets printed, more of its content will effectively be displayed. The level to which the result effectively gets forced depends on a run-time parameter called the "`printing-level`".

When the printing-level is set to k, then this means that all the non-recursive components will be evaluated (with a level defined by the top-level `printing-level`) and that *at most* k additional unfoldings of recursive components will be forced. For example, given the program presented above and a printing-level of 5, then the following value would be obtained:

$$Cons\ 0\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Cons\ 4\ (Cons\ 5\ \#\mathtt{Thunk})))))$$

Here is another example, involving more complex data structures:

```
Type list = Nil | Cons N list;
Type list_of_list = Nil_lol | Cons_lol N list list_of_list;
Def from k = Cons k (from (k+1));
Def from_lol k = Cons_lol k (from k) (from_lol (k+1));
Def program2 = from_lol 0;
```

Given a printing-level of 2, the following value would be printed:

$Cons\_lol\ 0\ (Cons\ 0\ (Cons\ 1\ (Cons\ 2\ \#\mathtt{Thunk})))$
$\quad Cons\_lol\ 1\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ \#\mathtt{Thunk})))$
$\quad\quad Cons\_lol\ 2\ (Cons\ 2\ (Cons\ 3\ (Cons\ 4\ \#\mathtt{Thunk})))$
$\quad\quad\quad \#\mathtt{Thunk}$

In a way, the initial head-strict demand represents the demand which is assumed to be made, *at compile-time*, for the result of the program. As for the printing-level parameter, it represents the demand made for the result of the program *at run-time*. [8] Although the printing-level determines the shape of the result that will

[8] This is explained in more detail in Appendix G.4, p. 268.

get forced and be printed, it does not specify the exact way in which this forcing/traversal will be performed. In the lazy variants of the compiler, either the pure lazy or the optimizing variant, the printing process, given a specific printing-level, was made to emulate the one performed by a *sequential interpreter*. That is, it traversed and forced the resulting data structures using a *left-to-right depth-first* traversal. For example, given a linear list, the first head would be evaluated; only when the value has finally been obtained is the evaluation of the next tail started, and so on.

Clearly, this approach to forcing/printing the result represents a bottleneck in terms of parallelism. In contrast, the lenient variant of the compiler, also uses a forcing process; [9] but in the case of the lenient variant, there is an important difference: the printing/forcing process is not sequential but *parallel*, i.e., all components of a data structures get forced in parallel.

In order to see the effect of this parallel forcing process, the optimizing lazy variant of the compiler has been run using a parallel forcing process instead of a sequential one. Figure 5.18 shows the effect, on the critical path length, of using such a parallel printing process instead of a sequential one. Clearly, the effect on the critical path length is dramatic.

As for the effect on the total number of instructions, it is as expected: the strategy used for forcing the result to be printed should not have much effect on the total number of instructions, as forcing the result simply means traversing the data structure. This is effectively what happens, as seen in Figure 5.19.

Of course, as was also to be expected, the effect on the average parallelism is relatively important, as seen in Figures 5.20 and 5.21.

It is interesting to remark that when the printing level gets large enough, then increasing its value has no real effect on the average parallelism, whether a sequential or parallel printing process is used. Of course, using a parallel printing process ensures that the average parallelism is always higher. This is illustrated in Figures 5.22, 5.23 and 5.24 which show the average parallelism for various printing levels when using a sequential printing process, and in Figures 5.25, 5.26 and 5.27 when using a parallel printing process.

[9] Even though this is not really required, as nothing really needs to be forced because nothing is delayed: this *dummy* printing process has been added to the lenient variant in order that all variants being compared have similar overhead, so that the comparisons of the total number of instructions are made on a similar basis.

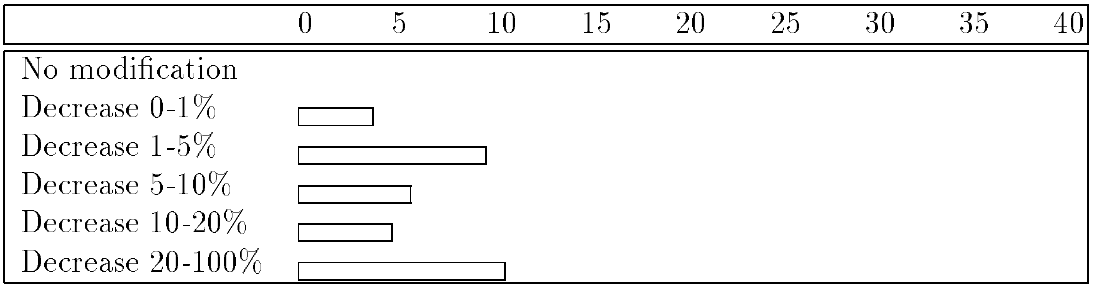


Figure 5.18: Sequential vs. parallel printing (sequential head-strictness): Effect on critical path length

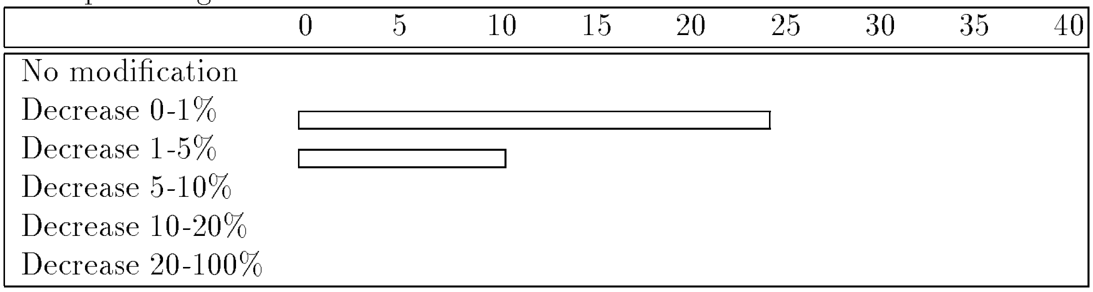


Figure 5.19: Sequential vs. parallel printing process (sequential head-strictness): Effect on total number of instructions

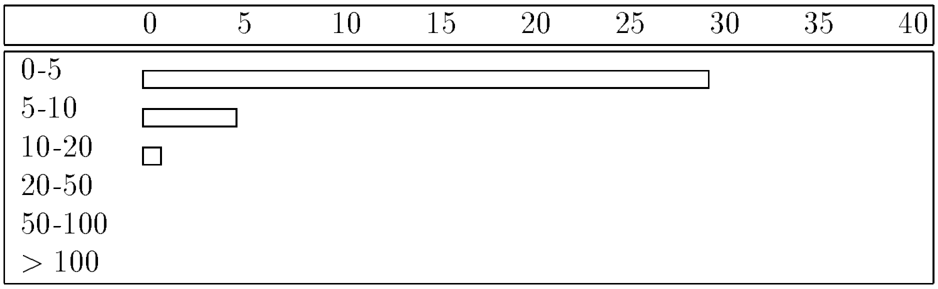


Figure 5.20: Sequential printing process (sequential head-strictness): Average parallelism

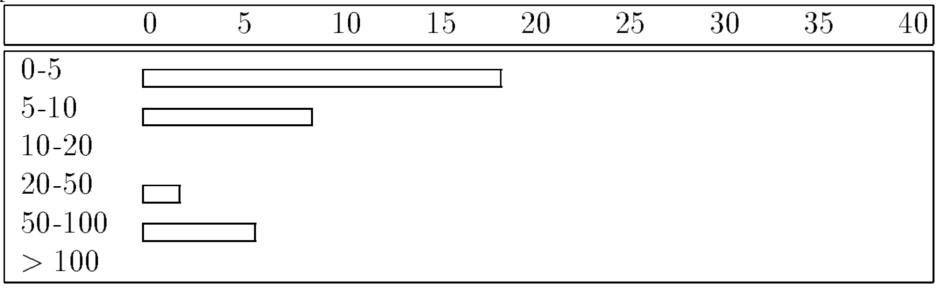


Figure 5.21: Parallel printing process (sequential head-strictness): Average parallelism

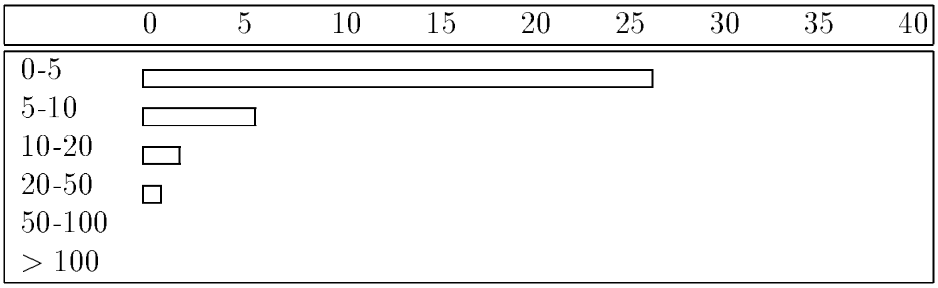


Figure 5.22: Average parallelism for (sequential) printing level = 10

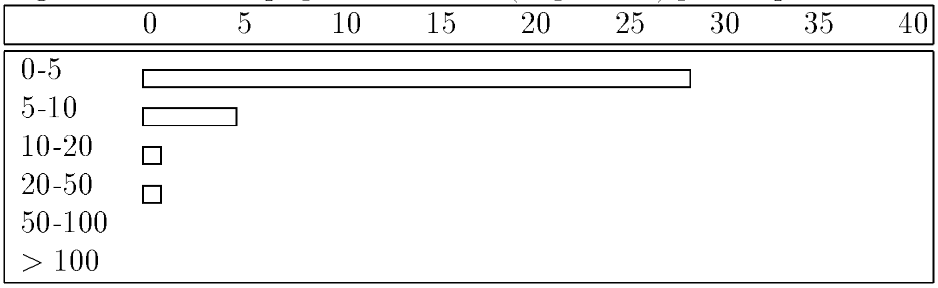


Figure 5.23: Average parallelism for (sequential) printing level = 30

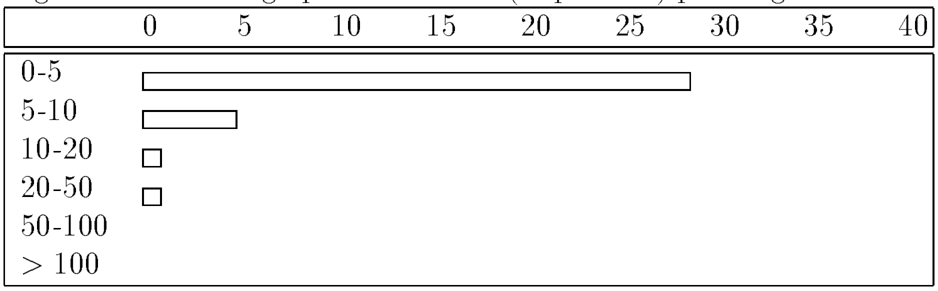


Figure 5.24: Average parallelism for (sequential) printing level = 50

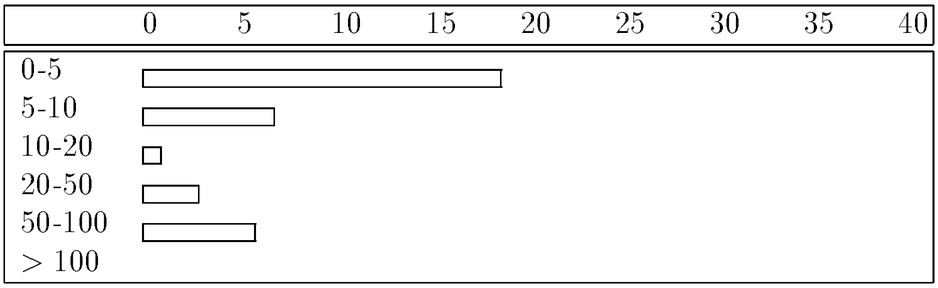


Figure 5.25: Average parallelism for (parallel) printing level = 10

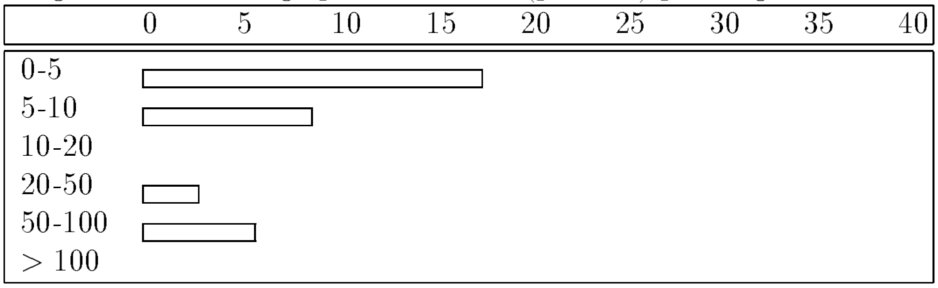


Figure 5.26: Average parallelism for (parallel) printing level = 30

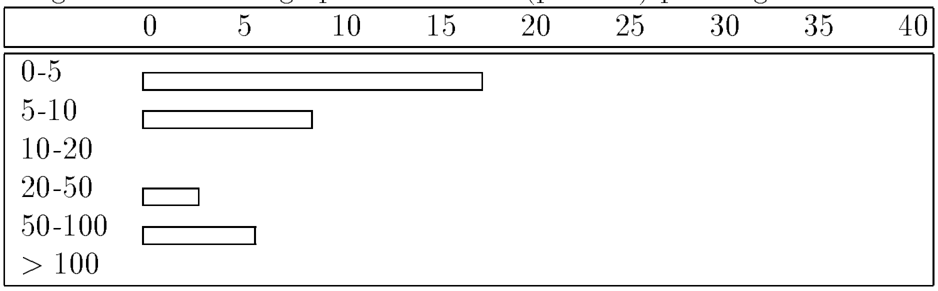


Figure 5.27: Average parallelism for (parallel) printing level = 50

From now on, because of the great advantage of the parallel printing process, all our benchmarks will be run using such a process.

5.4.2 Parallel head-strictness: A new notion of head-strictness

The basic idea of the classical notion of head-strictness is to ensure that, given a result data structure, none of its elements will get evaluated unless all the components that precede it when sequentially printing the structure have been fully evaluated and printed. Let's look at a simple example. Suppose we have the following binary tree data structure definition, where each leaf contains 3 integer values, and each node contains 2 integer values in addition to the pointers to the subtrees.

```
Type tree = Leaf N N N | Node tree N N tree;
```

The appropriate sequential head-strict demand for this data structure would be the following:

$$(Rec\ x :: \{Leaf\ \$\ \hat{}\$\ \hat{}\$,\ Node\ x\ \hat{}\$\ \hat{}\$\ \hat{}x\})$$

This demand implies that everything which is to the right of the first element to be printed effectively get delayed. This is required whenever the interpreter executing the program is a sequential one. It essentially represents a left-to-right depth-first traversal of the data structure.

However, if the interpreter/execution vehicle is a parallel one, then it could be quite acceptable to allow the sibling components to be evaluated *in parallel*. In the case of the trees presented above, we could then use the following demand instead:

$$(Rec\ x :: \{Leaf\ \$\ \$\ \$,\ Node\ \hat{}x\ \$\ \$\ \hat{}x\})$$

In this case, siblings are allowed to be evaluated concurrently. However, to prevent infinite unfolding of data structures, which must be done in order to deal/manipulate potentially infinite data structures, recursive tails (i.e., recursive subcomponents, e.g., subtrees) are effectively delayed. Essentially, this corresponds more to a *breadth-first* traversal of data structures.

One problem with the notion of sequential head-strictness is that it is very sensitive to the way a type definition is written. Suppose we have the following type definitions:

```
Def tree1 = Leaf N | Node tree1 N tree1;
Def tree2 = Leaf N | Node N tree2 tree2;
```

The associated head-strict demands would be the following:

- `tree1` : $(Rec\ x :: \{Leaf\ \$,\ Node\ x\ \hat{}\$\ \hat{}x\})$
- `tree2` : $(Rec\ x :: \{Leaf\ \$,\ Node\ \$\ \hat{}x\ \hat{}x\})$

In the case of `tree1`, because the first component of a `Node` is a recursive component, then no delay needs to be inserted for the left subtree, although a delay must be inserted for the scalar component; on the other hand, for `tree2`, both subtrees must be delayed, although the scalar component of the `Node` does not need to be delayed. Thus, slight changes in the way type definitions are written could have important impact on the precision of the strictness analysis process, which is a bit strange. [10]

Figures 5.28 (resp. 5.29) present the effect of using a parallel head-strict initial demand instead of a sequential one on the length of the critical path length (resp. the total number of instructions).

Clearly, simply using a parallel head-strict demand instead of a sequential one as the initial demand does not seem to bring much benefits. In fact, in many cases, this *increases* the length of the critical path and/or the total number of instructions. This phenomena is easily explained because, as indicated above, recursive *tails* always get delayed, whether they occur in the first position or not; on the other hand, when using sequential head-strictness, a recursive component appearing in the first position would not get delayed, as it would be sure to be printed by a sequential printer. Although this makes no difference for right-linear lists — in this case, the sequential and parallel head-strict demands are exactly the same — it does make a difference for binary trees such as those defined below: in the sequential head-strict case, the left subtree of a node would not get delayed, whereas it would be delayed for a parallel head-strict demand. In fact, in this case, the sequential head-strict demand would be strictly stronger than the parallel one, as illustrated below:

```
Type tree = Leaf N | Node tree tree;
```

- Sequential head-strict: $(Rec\ x :: \{Leaf\ \$,\ Node\ x\ \hat{}x\})$
- Parallel head-strict: $(Rec\ x :: \{Leaf\ \$,\ Node\ \hat{}x\ \hat{}x\})$

[10] An additional weakness of the notion of sequential head-strict demand, which will be examined in the next Section (5.4.3), is that its associated notion of speculation is less interesting than the parallel one.

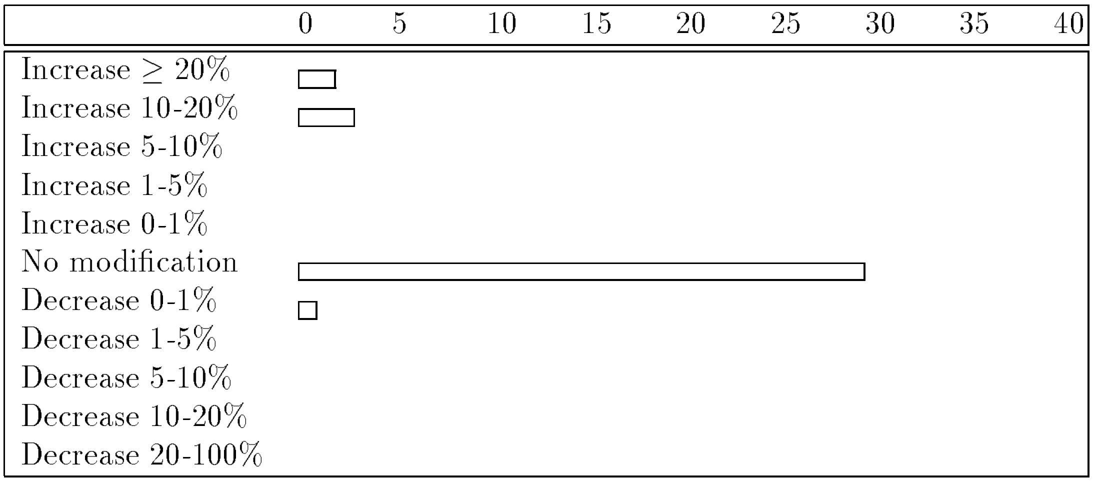


Figure 5.28: Sequential vs. parallel head-strictness (parallel printing): Effect on critical path length

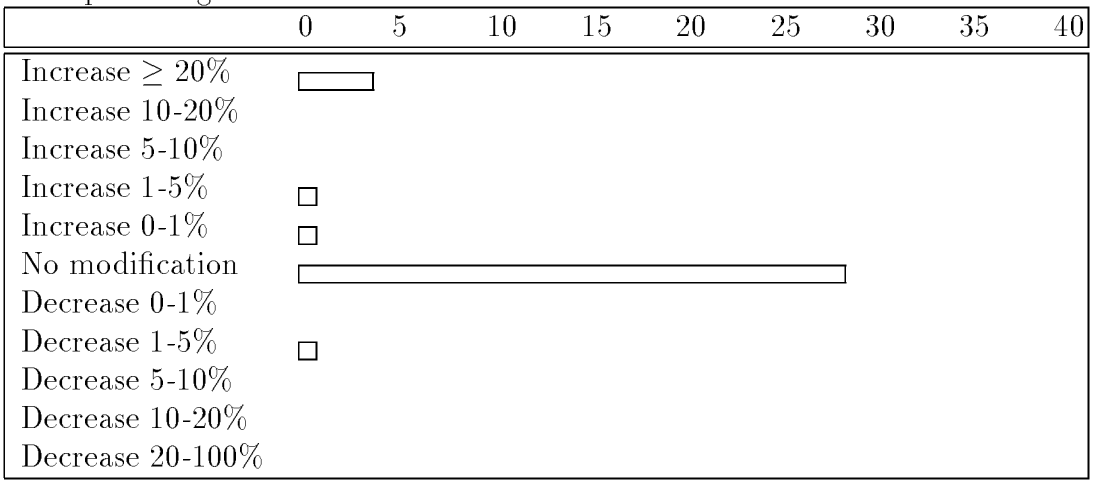


Figure 5.29: Sequential vs. parallel head-strictness (parallel printing): Effect on total number of instructions

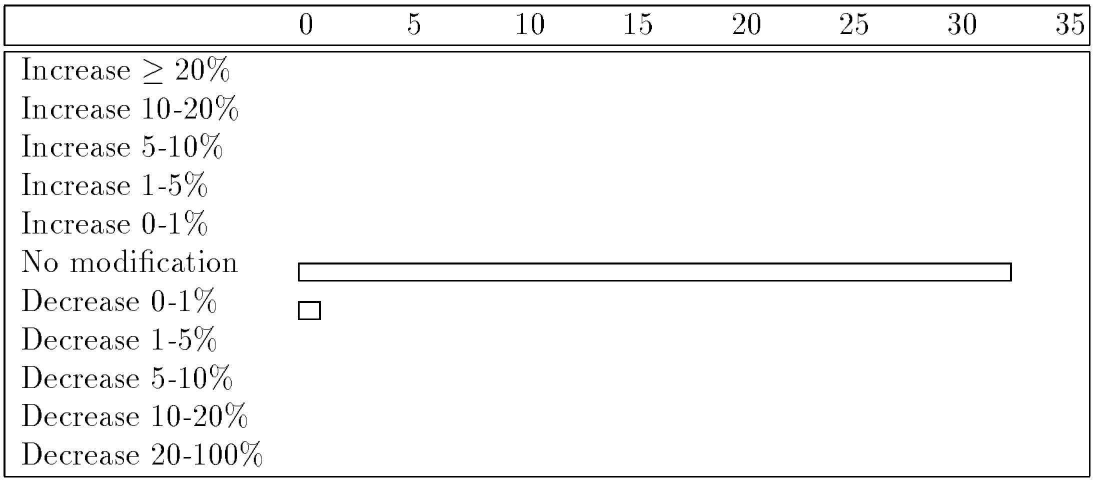


Figure 5.30: Sequential vs. parallel head-strictness for alternative type of tree (`tree2`): Effect on critical path length

Effectively, all cases for which there was an increase in the critical path length or total number of instructions in Figures 5.28 and 5.29 were programs producing binary trees of the indicated type as their result. Suppose, instead of using the previous type of trees, that we use the following, where each `Node` also include a scalar component, appearing in the first position:

```
Type tree2 = Leaf N | Node N tree2 tree2;
```

In this case, the effect, on the critical path length, of using parallel head-strictness instead of sequential head-strictness would lead to the results presented in Figure 5.30. There are now no cases where the critical path length increases.

Although the basic notion of parallel head-strictness, by itself, does not bring much additional parallelism, especially when the first component of a data structure is a recursive one, it will be seen, in the next section, to be interesting when introducing *speculation*. As we will see, the intuition behind the notion of speculation for parallel head-strictness will be much simpler to understand and visualize than the corresponding extension for sequential head-strictness. Furthermore, contrary to sequential head-strictness, parallel head-strictness (speculative or not) will not be sensitive to the order in which the components of a constructor are specified.

5.4.3 Bounded speculative laziness

In this section, we present an approach to increase the parallelism while still preserving the major property of laziness, which is the possibility to deal with potentially infinite data structures. The strategy we introduce is to allow some form of *speculation* over what would be done compared with the standard head-strict demand. Two such forms of speculation are introduced, corresponding to the two distinct forms of head-strictness introduced earlier: sequential head-strictness and parallel head-strictness. [11]

Sequential speculation

The basis for *sequential speculation* is the sequential head-strict demand, i.e., the regular head-strict demand corresponds to a speculation level of 0. The intuition behind the notion of sequential speculation is the following: a sequential speculation level of $k > 0$ means that whenever a thunk for a data structure component gets evaluated, then *at most k additional* scalar values might get evaluated when compared with what would have happened in the non-speculative case.

We illustrate this notion with a few examples. Suppose we are given the following type definitions:

```
Type triple = Triple N N N;
Type list = Nil | Cons N list;
Type triple_list = Nil_tl | Cons_tl triple triple_list;
Type tree = Leaf N | Node tree N tree;
Type triple_tree = Leaf_tt triple | Node_tt tree triple tree;
```

Examples of various speculative sequential head-strict demands would be the following:

- Speculation level = 0:

  `triple`: $\{\mathit{Triple}\ \$\ \hat{}\$\ \hat{}\$\}$

  `list`: $(\mathit{Rec}\ x :: \{\mathit{Nil},\ \mathit{Cons}\ \$\ \hat{}x\})$

  `triple_list`: $(\mathit{Rec}\ x :: \{\mathit{Nil\_tl},\ \mathit{Cons\_tl}\ \{\mathit{Triple}\ \$\ \hat{}\$\ \hat{}\$\}\ \hat{}x\})$

[11] Note that we are not aware of any work related to speculation in the context of lazy evaluation. Also note that our notion of speculation differs from the one used in modern computer architecture [TGH93] in the following way: while the latter tries to speculate over control information related to branches, our notion is more data-related, i.e., speculation is on the result which is to be demanded compared with what would be demanded by a lazy interpreter. The goal of both approaches, however, is the same, viz., they both attempt to increase the amount of exploitable parallelism.

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ x\ \hat{}\$\ \hat{}x\})$

`triple_tree`: $(Rec x :: \{Leaf\_tt\{Triple \$\ \hat{}\$\ \hat{}\$\}, Node\_tt\ x\ \hat{}\{Triple \$\ \hat{}\$\ \hat{}\$\}\ \hat{}x\})$

- Speculation level = 1:

`triple`: $\{Triple\ \$\ \$\ \hat{}\$\}$

`list`: $(Rec\ x :: \{Nil,\ Cons\ \$\ \{Nil,\ Cons\ \$\ \hat{}x\}\})$

`triple_list`: $(Rec\ x :: \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \hat{}\$\}\ x'\})$

Where $x' = \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \hat{}\$\}\ \hat{}x\}$

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ x\ \hat{}\$\ \hat{}x\})$

`triple_tree`: $(Rec x :: \{Leaf\_tt\ \{Triple \$\$\ \hat{}\$\}, Node\_tt\ x\ \hat{}\{Triple \$\$\ \hat{}\$\}\ \hat{}x\})$

- Speculation level = 2:

`triple`: $\{Triple\ \$\ \$\ \$\}$

`list`: $(Rec\ x :: \{Nil,\ Cons\ \$\ \{Nil,\ Cons\ \$\ \{Nil,\ Cons\ \$\ \hat{}x\}\}\})$

`triple_list`: $(Rec\ x :: \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \$\}\ x'\})$

Where $x' = \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \$\}\ x''\}$

Where $x'' = \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \$\}\ \hat{}x\}$

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ x\ \hat{}\$\ \hat{}x\})$

`triple_tree`: $(Rec\ x :: \{Leaf\_tt\ \{Triple\ \$\ \$\ \$\},\ Node\_tt\ x\ \hat{}\{Triple\ \$\ \$\ \$\}\ \hat{}x\})$

It is interesting to note that all demands for the `tree` data type are the same whatever the speculation level. The reason for this is because the initial head-strict demand does not delay the left subtree. Whatever the speculation level, if the scalar component of the `Node` was changed into a non-delayed component, then no *a priori* bound could be put on the number of scalars that would get evaluated, because we do not know the depth of the left subtree which has to be traversed in order to print the next element. Similarly, increasing the speculation level for the `triple_tree` datatype does not lead to any unfolding of the recursive components, although the `Triple` components do get improved.

Figures 5.31, 5.32, and 5.33 present histograms showing the effect, on the critical path length, of using various levels of sequential speculation. In most cases, this leads to a reduction of the critical path length. However, there are a few exceptions:

1) In one case (`CompressPath`), although there is a slight increase (less than 0.1%, i.e., 4 time units), the total number of instructions executed is effectively reduced (by 1.79%). The exact cause of this is not understood. 2) In all the other cases, this is because the result is a potentially infinite data structure, so increasing the speculation level effectively means producing more of the result.

Note that, for different speculation levels, domains of demands of different size have been used. This was done to prevent the occurrence of rounding errors that may be generated when the domain size gets too small, as described in Section F.1. In most cases but a few, the difference between the speculation level and the size of the domain was selected to be 2. In other words, given a speculation level of k (= 0, 1, 2, 3), a domain of size $k+2$ has been used in most cases, and a domain of size $k+3$ was used in the remaining (2) cases. [12] Note, again, that only programs whose result is a non-scalar have been used in the speculation experiments, as speculation is useless for scalar values. [13]

Parallel speculation

The interpretation of the speculation level, when dealing with parallel head-strict demands, is quite different from the sequential interpretation. As mentioned earlier, the intuitive idea behind parallel head-strictness is to emulate a breadth-first unfolding/traversal of data structures. In the basis case (speculation level = 0), none of the components get delayed, except for the recursive unfoldings. As for a speculation level of k, it means that, whenever a delayed component is evaluated, then this will trigger the evaluation of *at most* k additional unfolding of the recursive components. [14]

Here are some examples of various speculative parallel head-strict demands:

- Speculation level = 0:

  `triple:` $\{\mathit{Triple}\ \$\ \$\ \$\}$

  `list:` $(\mathit{Rec}\ x :: \{\mathit{Nil},\ \mathit{Cons}\ \$\ \hat{}x\})$

[12] The *smallness* of a domain is thus a *relative* concept: for a domain of a given size, when the speculation level increases, then the domain becomes relatively smaller, leading to a larger number of rounding errors.

[13] It must also be noted that in all experiments related to speculation, the printing level used was selected to be large enough (100) to ensure that all programs producing a finite result get fully printed.

[14] For a more detailed explanation, cf. Appendix G.4, p. 270.

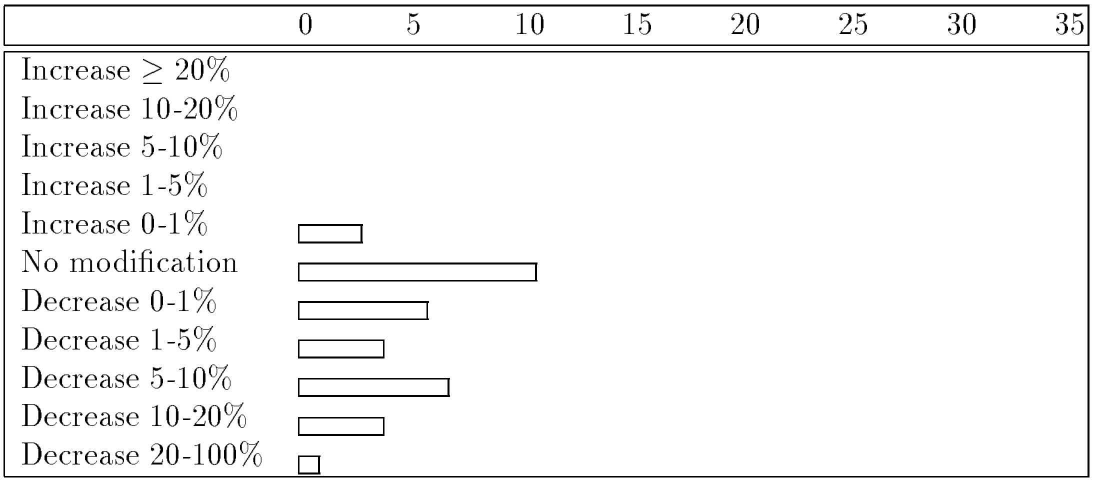


Figure 5.31: No speculation vs. speculation level = 1 (sequential speculation): Effect on critical path length

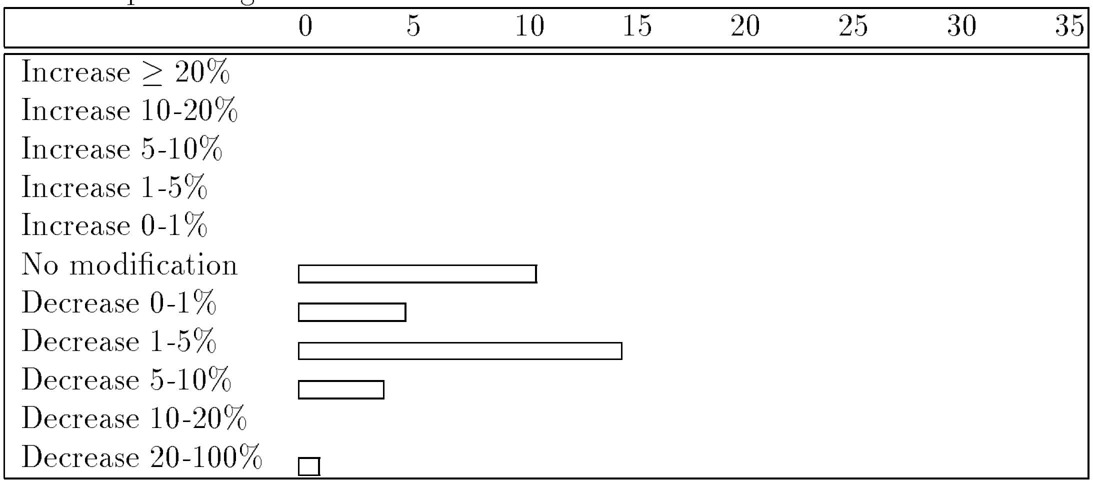


Figure 5.32: Speculation level = 1 vs. 2 (sequential speculation): Effect on critical path length

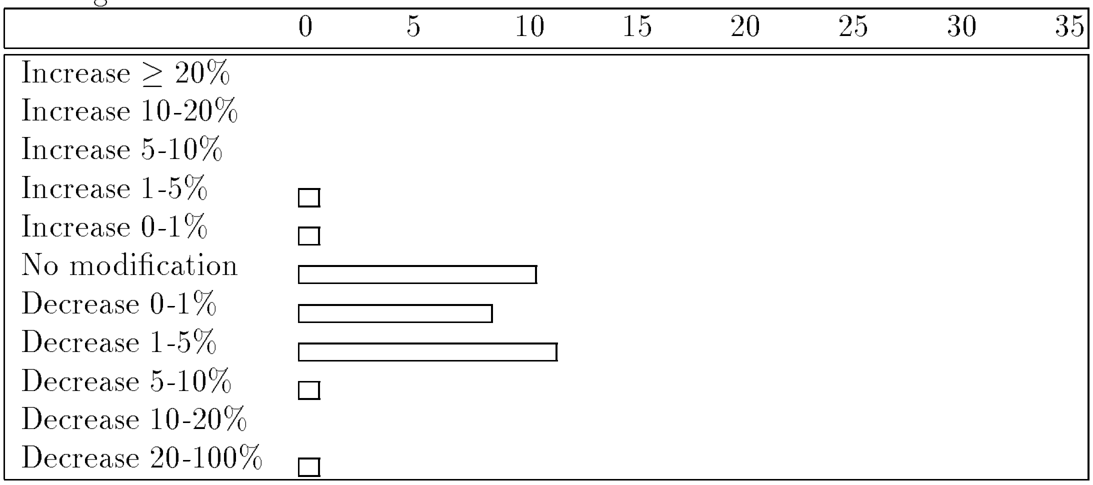


Figure 5.33: Speculation level = 2 vs. 3 (sequential speculation): Effect on critical path length

`triple_list`: $(Rec\ x :: \{Nil\_tl,\ Cons\_tl\ \{Triple\ \$\ \$\ \$\}\ \hat{\ }x\})$

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ \hat{\ }x\ \$\ \hat{\ }x\})$

`triple_tree`: $(Rec\ x :: \{Leaf\_tt\ \{Triple\ \$\ \$\ \$\},\ Node\_tt\ \hat{\ }x\ \{Triple\ \$\ \$\ \$\}\ \hat{\ }x\})$

- Speculation level = 1:

`triple`: $\{Triple\ \$\ \$\ \$\}$

`list`: $(Rec\ x :: \{Nil,\ Cons\ \$\ \{Nil,\ Cons\ \$\ \hat{\ }x\}\})$

`triple_list`: $(Rec x :: \{Nil\_tl, Cons\_tl\{Triple\$\$\$\}\{Nil\_tl, Cons\_tl\{Triple\$\$\$\}\ \hat{\ }x\}\})$

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ x'\ \$\ x'\})$

Where $x' = \{Leaf\ \$,\ Node\ \hat{\ }x\ \$\ \hat{\ }x\}$

`triple_tree`: $(Rec\ x :: \{Leaf\_tt\ \{Triple\ \$\ \$\ \$\},\ Node\_tt\ x''\ \{Triple\ \$\ \$\ \$\}\ x''\})$

Where $x'' = \{Leaf\_tt\ \{Triple\ \$\ \$\ \$\},\ Node\_tt\ \hat{\ }x\ \{Triple\ \$\ \$\ \$\}\ \hat{\ }x\}$

- Speculation level = 2:

`tree`: $(Rec\ x :: \{Leaf\ \$,\ Node\ x'\ \$\ x'\})$

Where $x' = \{Leaf\ \$,\ Node\ x''\ \$\ x''\}$

Where $x'' = \{Leaf\ \$,\ Node\ \hat{\ }x\ \$\ \hat{\ }x\}$

Figures 5.34, 5.35, and 5.36 present histograms showing the effect of using various levels of parallel speculation on the reduction of the critical path length. [15]

Sequential vs. parallel speculation

Figures 5.37, 5.38, 5.39 and 5.40 present histograms showing the effect of using parallel head-strict speculation instead of sequential speculation, for various level of speculation (0, 1, 2, 3), on the reduction of the critical path length. It can be seen that for non-zero speculation level, parallel head-strict speculation almost always lead to a reduction of the critical path length.

[15] Note that, as was the case for sequential speculation, there are a few exceptions: 1) Two cases — `CompressPath` and `FFT` — for which there is an increase of less than 0.01% but a decrease of the total number of instructions executed. 2) A number of cases for which the program produces a potentially infinite result. These can be explained in the same manner as the exceptions for sequential speculation.

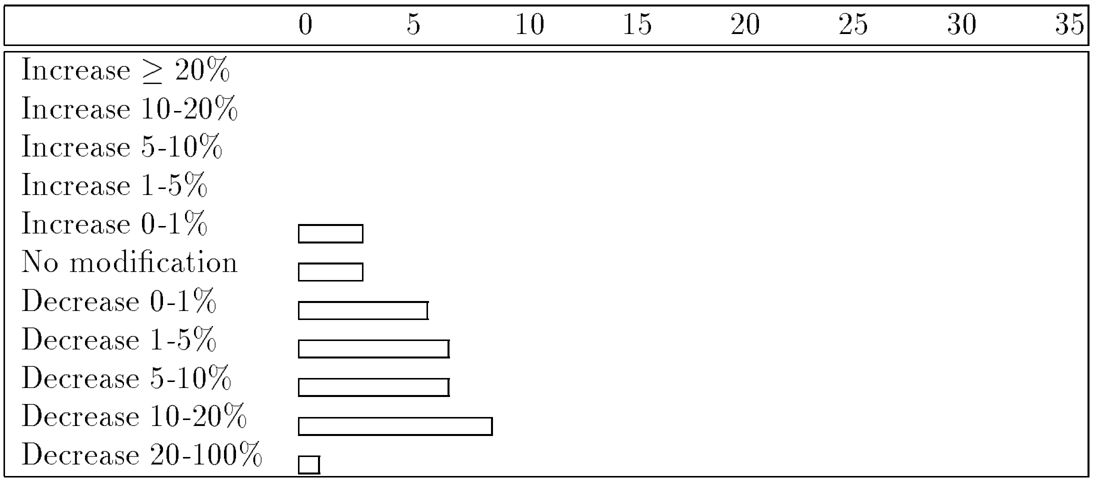


Figure 5.34: No speculation vs. speculation level = 1 (parallel speculation): Effect on critical path length

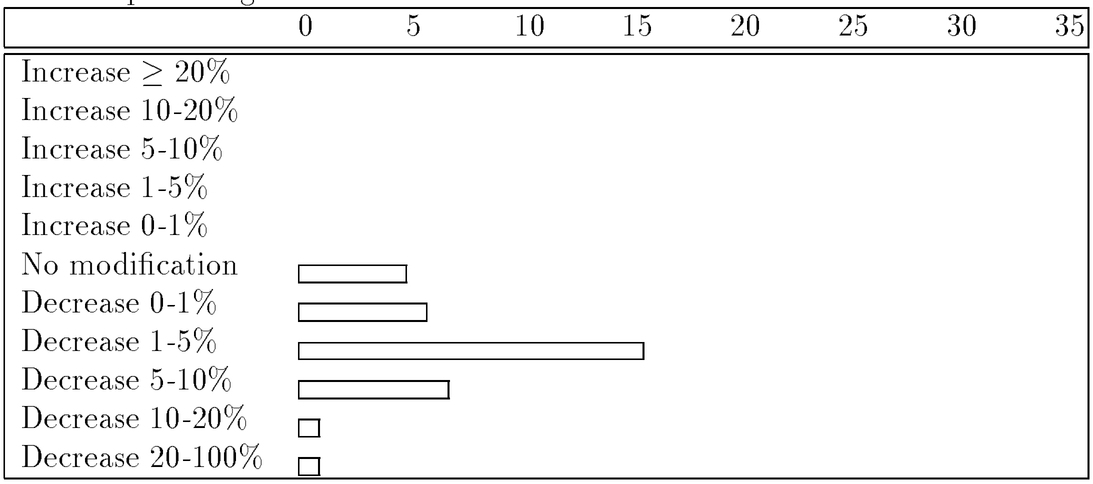


Figure 5.35: Speculation level = 1 vs. 2 (parallel speculation): Effect on critical path length

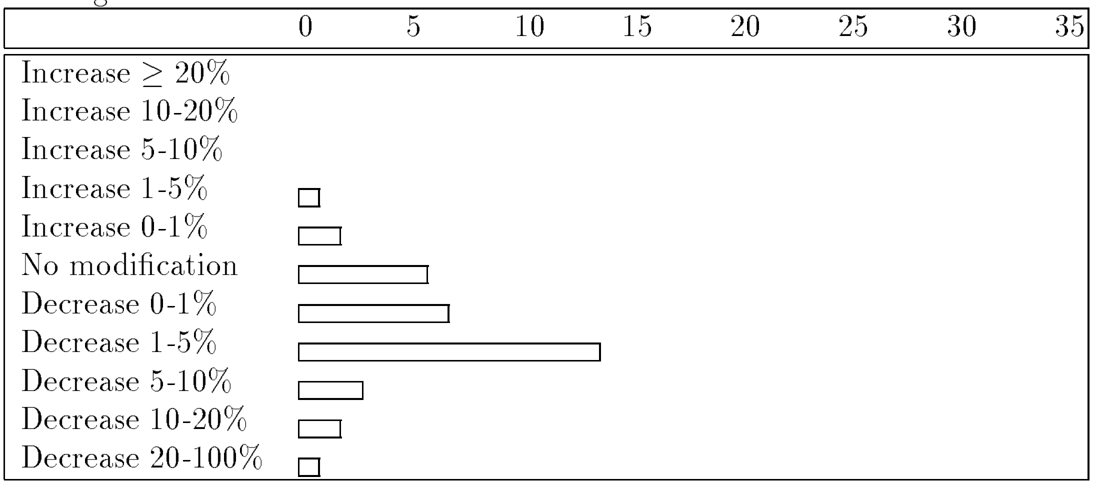


Figure 5.36: Speculation level = 2 vs. 3 (parallel speculation): Effect on critical path length

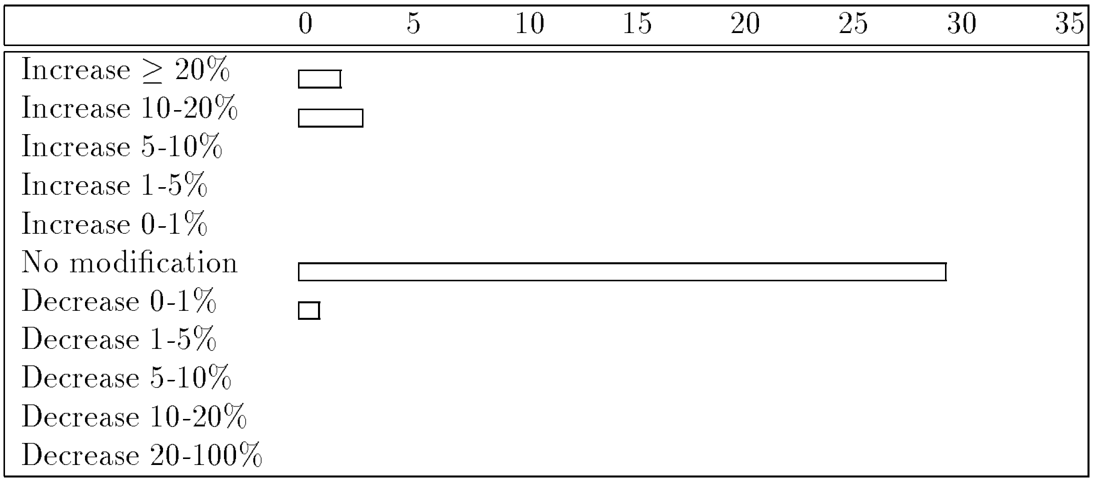


Figure 5.37: Sequential vs. parallel head-strictness with speculation level = 0: Effect on critical path length

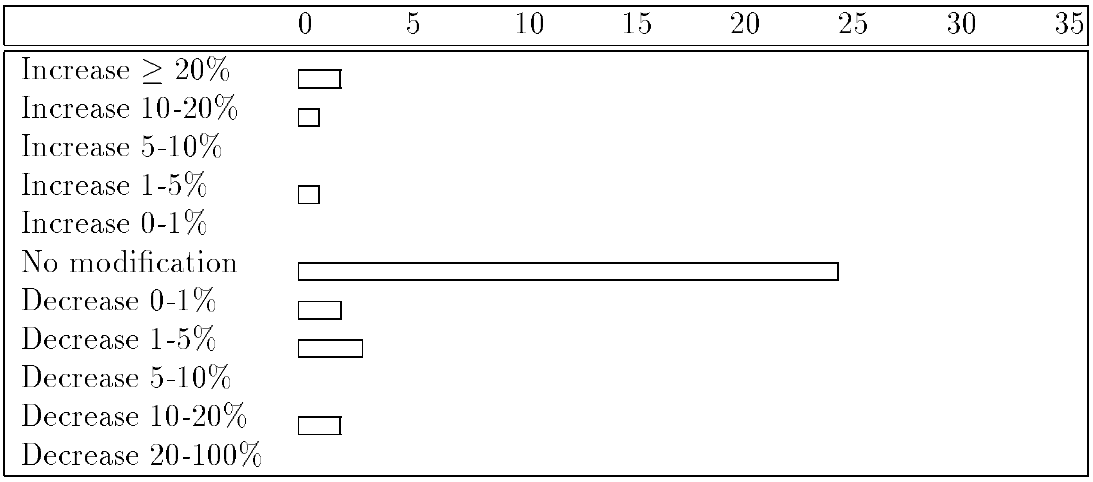


Figure 5.38: Sequential vs. parallel head-strictness with speculation level = 1: Effect on critical path length

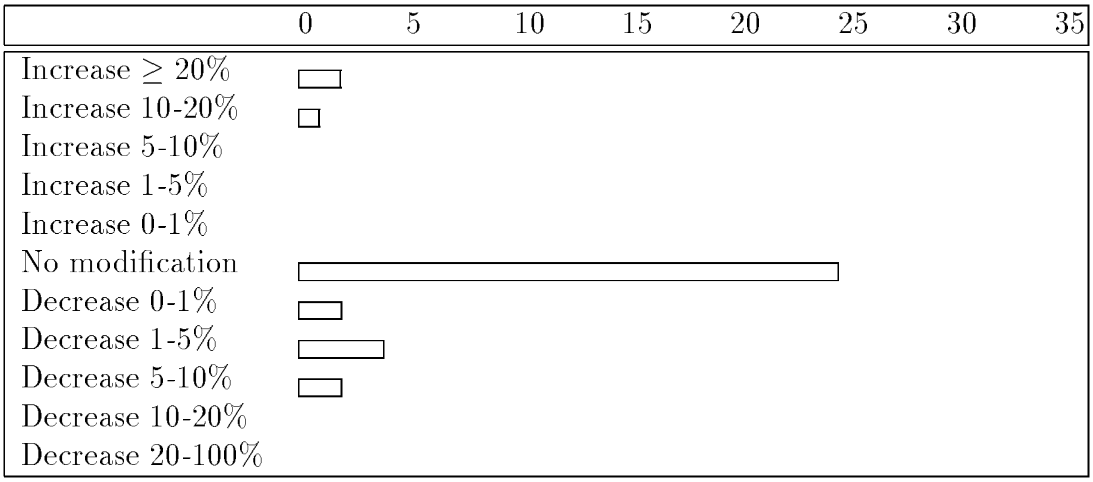


Figure 5.39: Sequential vs. parallel head-strictness with speculation level = 2: Effect on critical path length

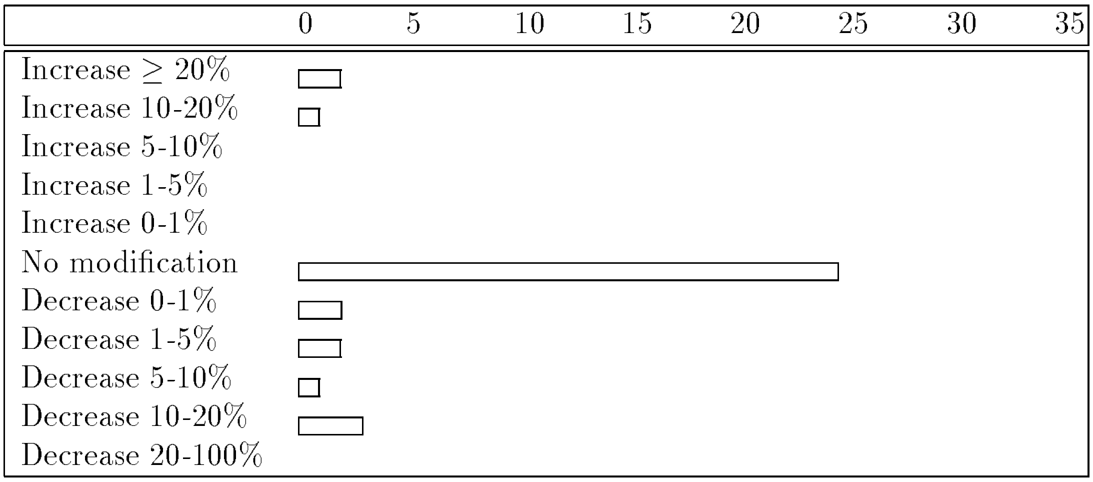


Figure 5.40: Sequential vs. parallel head-strictness with speculation level = 3: Effect on critical path length

Again, it is interesting to examine the cases where using speculative parallel head-strictness instead of speculative sequential head-strictness leads to an increase of the critical path length: in all cases, these are programs producing simple binary trees as their result, i.e., trees defined using the following type definition:

```
Type tree = Leaf N | Node tree tree;
```

The reason for this phenomena was explained earlier, in Section 5.4.2, where it was seen that slightly changing the type definition (i.e., making the tree not directly left-recursive) made parallel head-strictness more efficient.

Comparison of the average parallelism

We now present histograms comparing the average parallelism, for various speculation levels (0, 1, 2 and 3), between sequential and parallel speculative head-strictness.

- Speculation level = 0: Figures 5.41 and 5.42;
- Speculation level = 1: Figures 5.43 and 5.44;
- Speculation level = 2: Figures 5.45 and 5.46;
- Speculation level = 3: Figures 5.47 and 5.48;

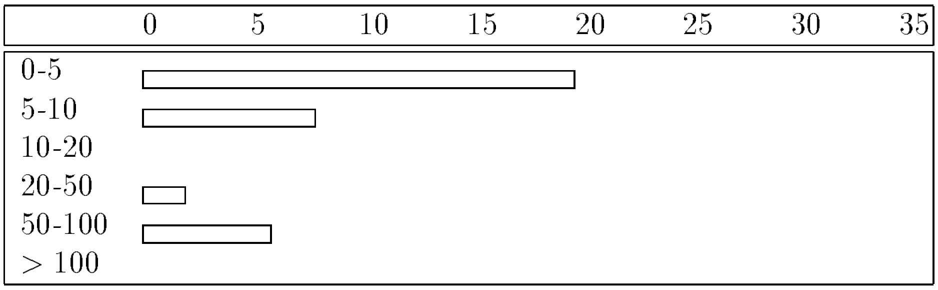


Figure 5.41: Average parallelism for sequential head-strictness: Speculation level = 0

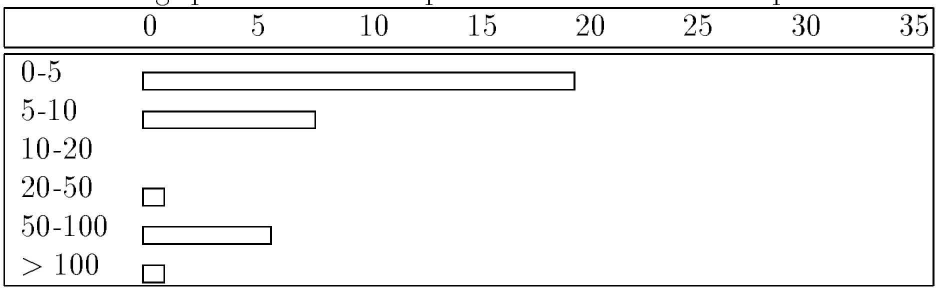


Figure 5.42: Average parallelism for parallel head-strictness: Speculation level = 0

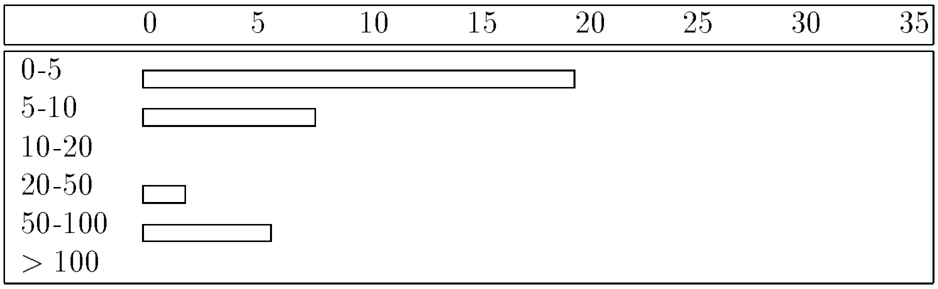


Figure 5.43: Average parallelism for sequential head-strictness: Speculation level = 1

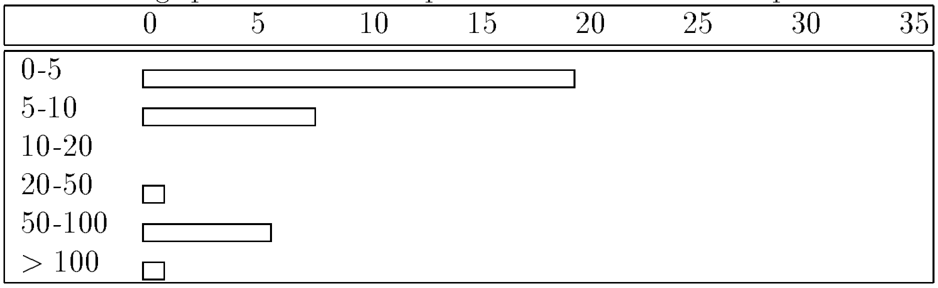


Figure 5.44: Average parallelism for parallel head-strictness: Speculation level = 1

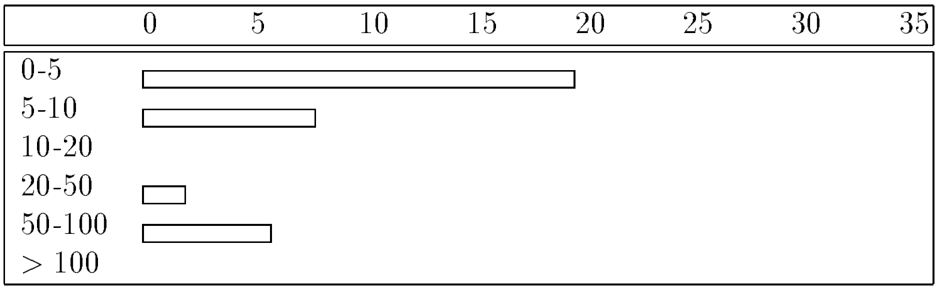


Figure 5.45: Average parallelism for sequential head-strictness: Speculation level = 2

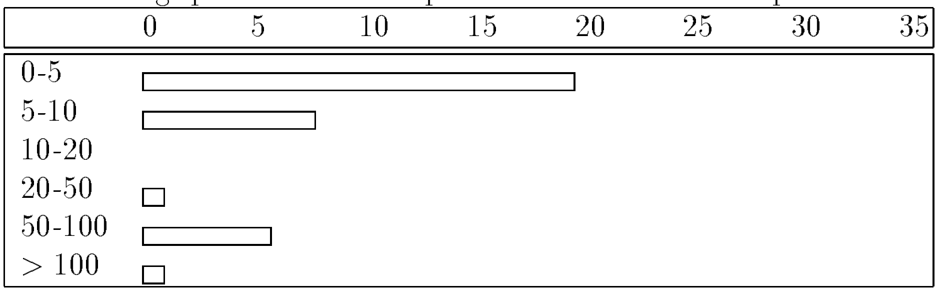


Figure 5.46: Average parallelism for parallel head-strictness: Speculation level = 2

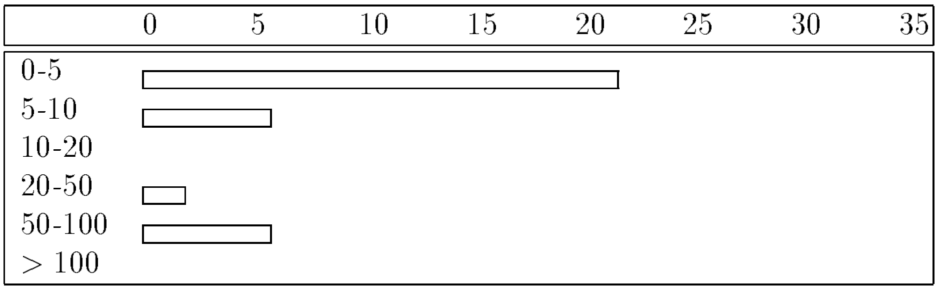


Figure 5.47: Average parallelism for sequential head-strictness: Speculation level = 3

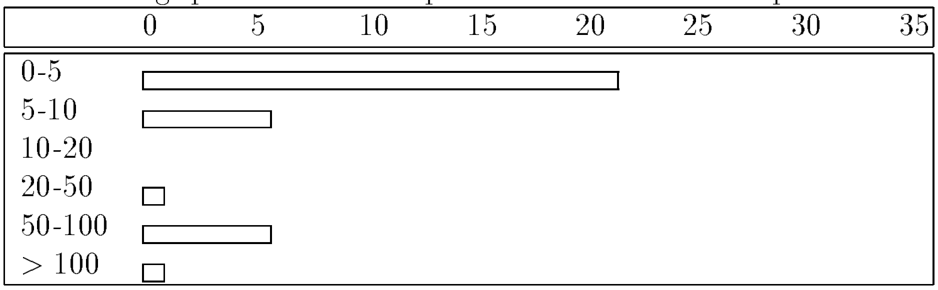


Figure 5.48: average parallelism for parallel head-strictness: speculation level = 3

As can be seen, for any given speculation level, there is not much difference between sequential and parallel speculative head-strictness in terms of average parallelism. Although this is not such a big surprise, the following fact is more interesting: in both cases, increasing the speculation level does not lead to much modifications of the *distribution* of average parallelism! Again, this can be explained by the fact that increasing the speculation level leads to a decrease of both the total number of instructions and the critical path length, implying that the average parallelism, which is the quotient of the two, changes very little. Of course, this does not mean that increasing the speculation level is not interesting: reducing the length of the critical path and the total number of instructions getting executed can only have a positive effect on the utilization of the underlying machine.

5.4.4 Handling of really lazy programs

In this section, we briefly reexamine some of the experiments performed in the preceding sections, but, this time, restricting them to programs requiring a lazy interpreter to work correctly. As we will see, laziness is clearly a major inhibition to parallelism, even when speculation is allowed.

First, Figures 5.49 and 5.50 show that our optimizing compiler seems preferable to a pure lazy evaluator, in terms of reducing both the critical path length (Fig. 5.49) and the total number of instructions (Fig. 5.50).

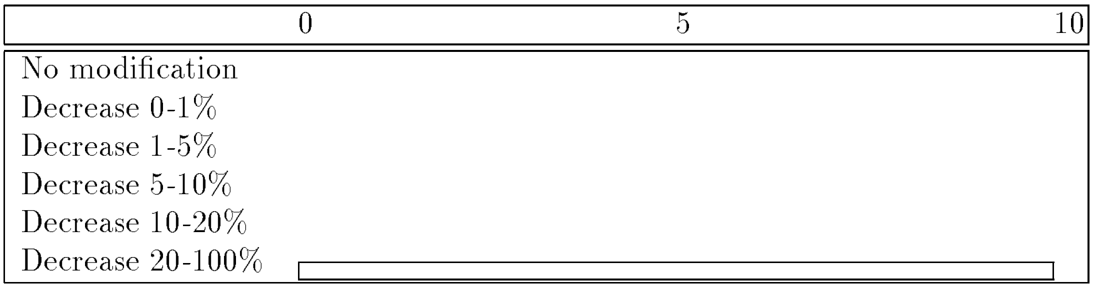


Figure 5.49: Pure vs. optimized lazy (lazy benchmarks): Effect on critical path length

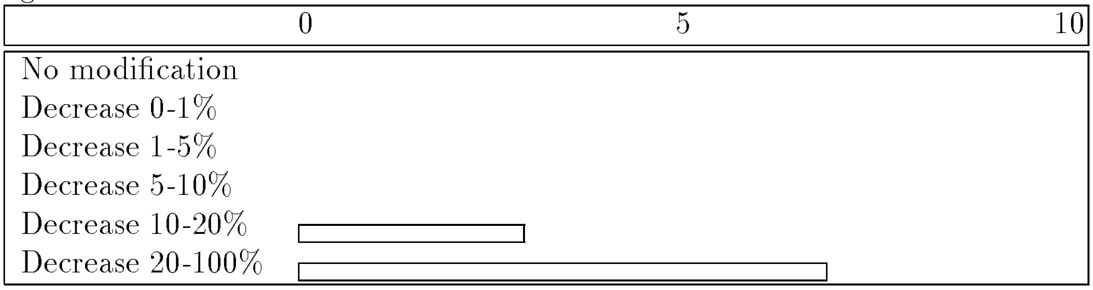


Figure 5.50: Pure vs. optimized lazy (lazy benchmarks): Effect on total number of instructions

An alternative presentation of these results is given in Figures 5.51 and 5.52: in this case, for each of the real lazy benchmark, the ratio between the result obtained for the optimized lazy variant — with a parallel printing process — (indicated by $\diamond$) vs. the non-optimized implementation (indicated by $\star$) is presented. From these figures, it can be seen that, for real lazy programs, the optimized implementation can lead to reductions of the critical path length and of the total number of instructions executed somewhere between 20-40%.

Figures 5.53 and 5.54 show that our optimizing compiler also seems preferable to a compiler based on the evaluation transformers model, in terms of reducing both the critical path length (Fig. 5.53) and the total number of instructions (Fig. 5.54).

For parallel speculation, Figures 5.55 5.56 and 5.57 show that increasing the speculation level has very little effect on the reduction of the critical path length.

As for the effect on the average parallelism, presented in Figures 5.58, 5.59, 5.60 and 5.61, we see that, for really lazy programs, very little parallelism can be obtained. [16]

[16] The only case where the average parallelism is larger than 10 instructions per time step is the `NB_SOLUTIONS_HELLER` benchmark. This program computes the total number of solutions

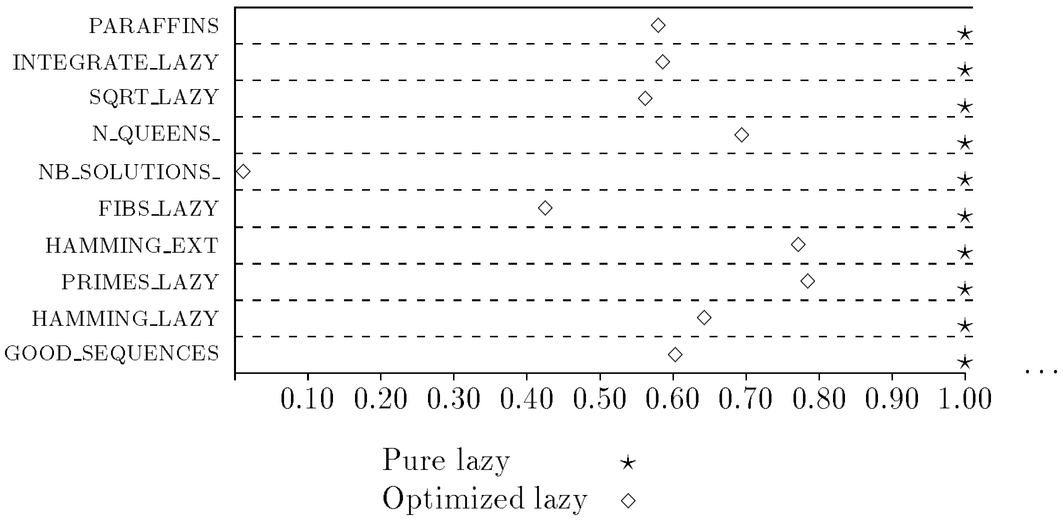


Figure 5.51: Ratio of critical path length for various benchmarks using pure (unoptimized) vs. optimized lazy implementation

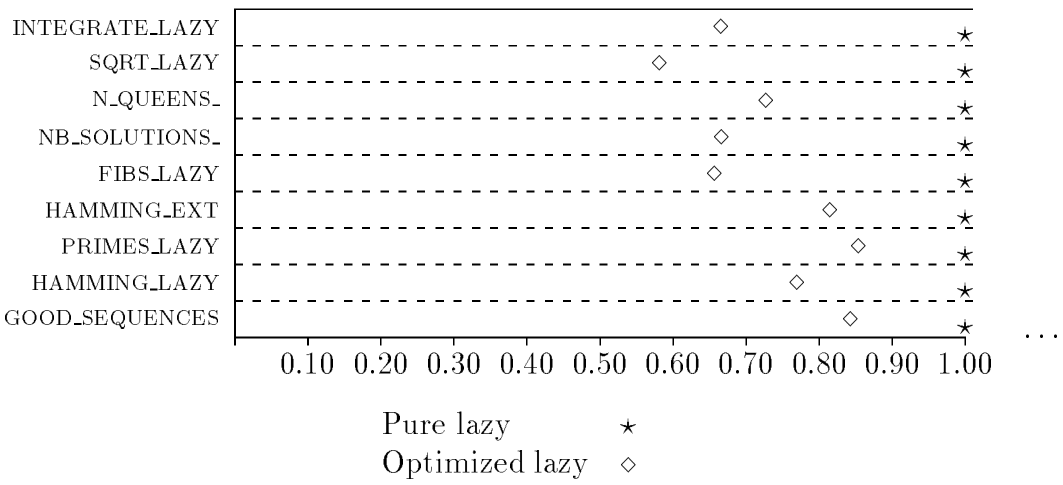


Figure 5.52: Ratio of total number of instructions executed for various benchmarks using non-optimized vs. optimized lazy implementation

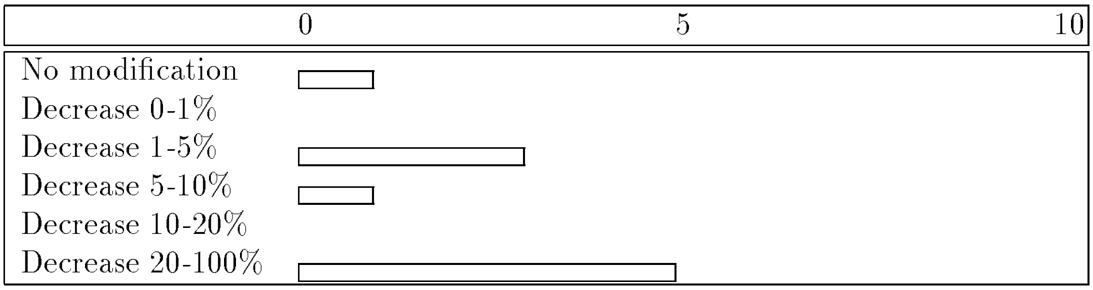


Figure 5.53: Evaluation transformers style vs. optimized lazy compiler (lazy benchmarks): Effect on critical path length

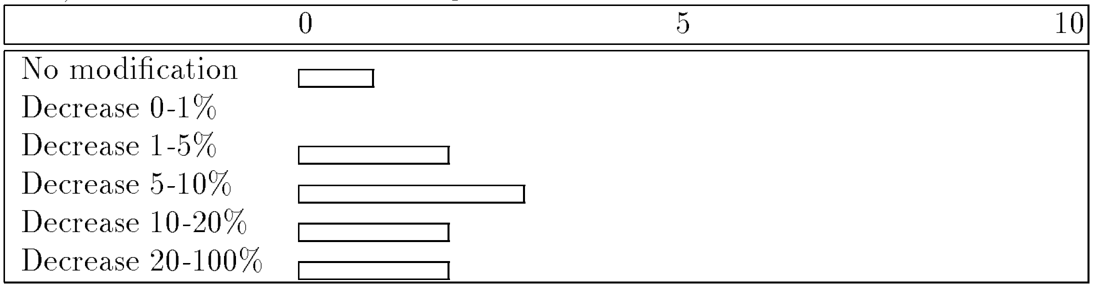


Figure 5.54: Evaluation transformers style vs. optimized lazy compiler (lazy benchmarks): Effect on total number of instructions

What can be concluded from all of this? Well, it seems that, even with speculation, laziness is clearly a major inhibition to parallelism. In other words, there is a major price one must be willing to pay when using laziness in order to manipulate potentially infinite structures: the price of parallelism.

5.4.5 Finite results and head-and-tail strictness

An important factor limiting the precision of the strictness analysis is the fact that we can only use an initial demand which is head-strict (i.e., for which all tails are delayed); this is required because the result produced by the program might be potentially infinite, so its infinite unfolding must be prevented. On the other hand, whenever the program produces a result which is finite, then there is no danger in using an initial demand which is *head-and-tail strict*, i.e., for which none of the heads and none of the tails are delayed: although this might perform more work

to the six queens problem. In this case, the initial demand ($) is transformed into a tail-strict demand for all the solutions; that is, none of the tails of the solutions have to be delayed, which is quite stronger than having only a head-strict initial demand. When a head-strict demand is used as the initial demand to compute *all* the solutions to the N queens problem (`N_QUEENS_HELLER` benchmark) instead of only computing the number of solutions, then all tails get delayed and the average parallelism fall to 3.5 instructions per time step!

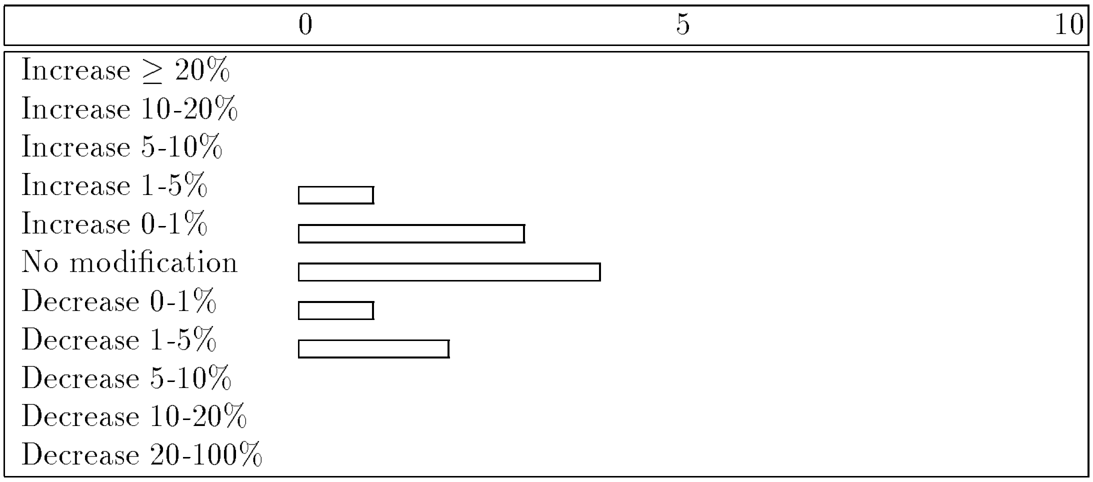


Figure 5.55: No speculation vs. speculation level = 1 (parallel speculation/lazy benchmarks): Effect on critical path length

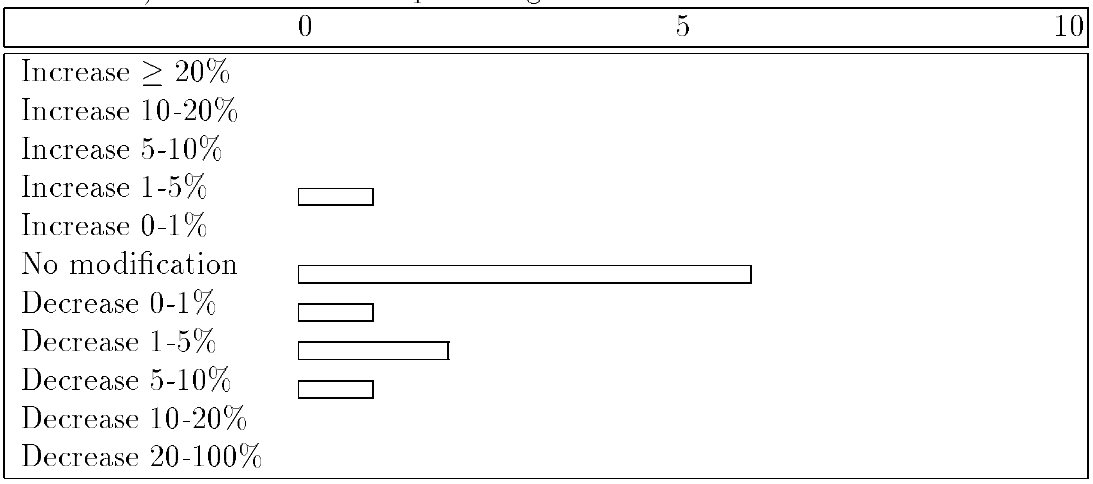


Figure 5.56: Speculation level = 1 vs. 2 (parallel speculation/lazy benchmarks): Effect on critical path length

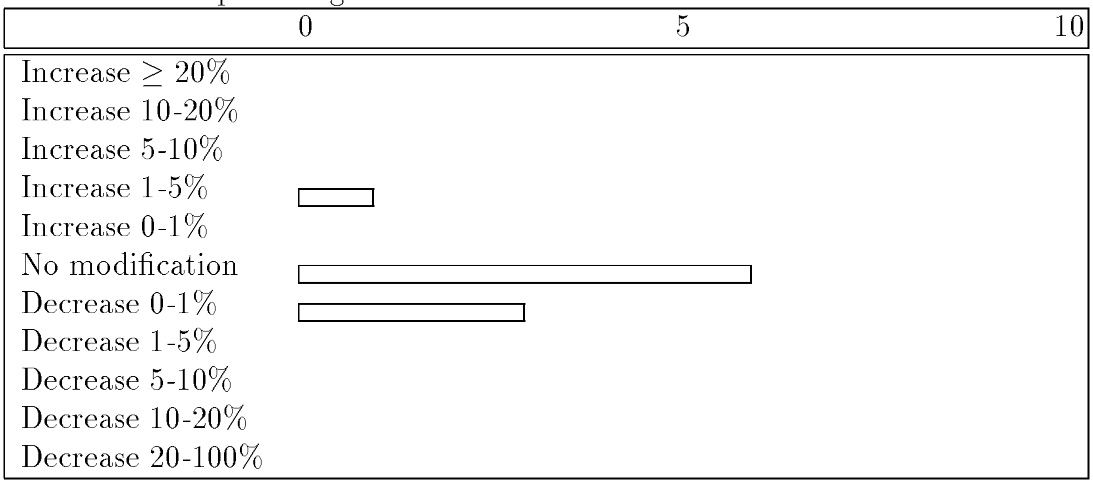


Figure 5.57: Speculation level = 2 vs. 3 (parallel speculation/lazy benchmarks): Effect on critical path length

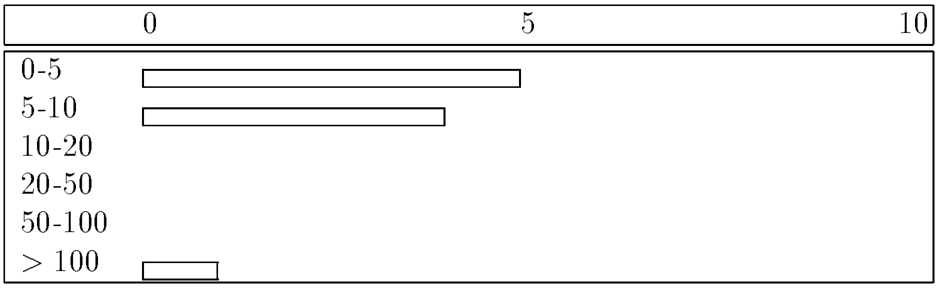


Figure 5.58: Average parallelism (lazy benchmarks): Parallel speculation level = 0

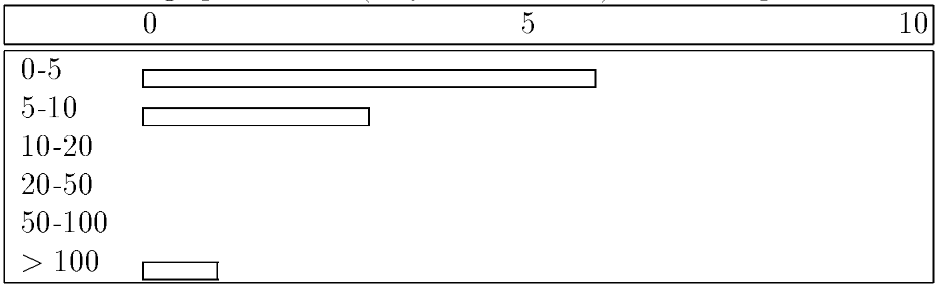


Figure 5.59: Average parallelism (lazy benchmarks): Parallel speculation level = 1

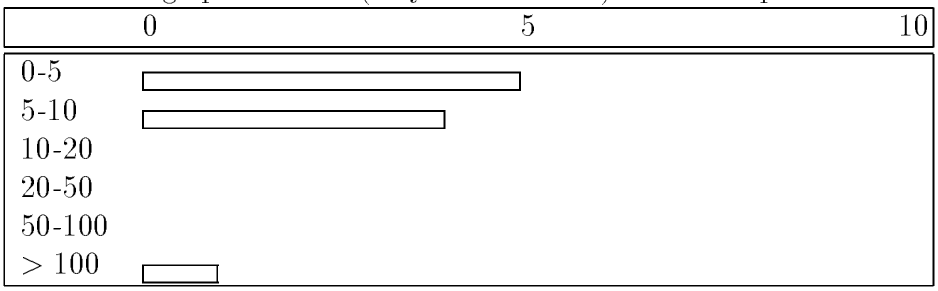


Figure 5.60: Average parallelism (lazy benchmarks): Parallel speculation level = 2

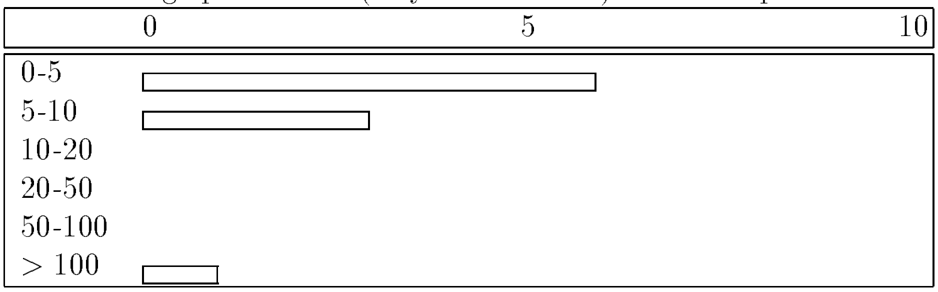


Figure 5.61: Average parallelism (lazy benchmarks): Parallel speculation level = 3

than a pure lazy interpreter, the amount of additional work that will be performed will effectively be bounded.

Figure 5.62 shows the effect, on the reduction of the critical path length, of using an initial head-and-tail strict demand instead of a head-strict one. In this case, the printing level that was used ensured that all programs' result got fully printed; of course, only benchmarks which produced finite results have been used. Clearly, as can be seen, the effect on the reduction of the critical path length is dramatic. [17]

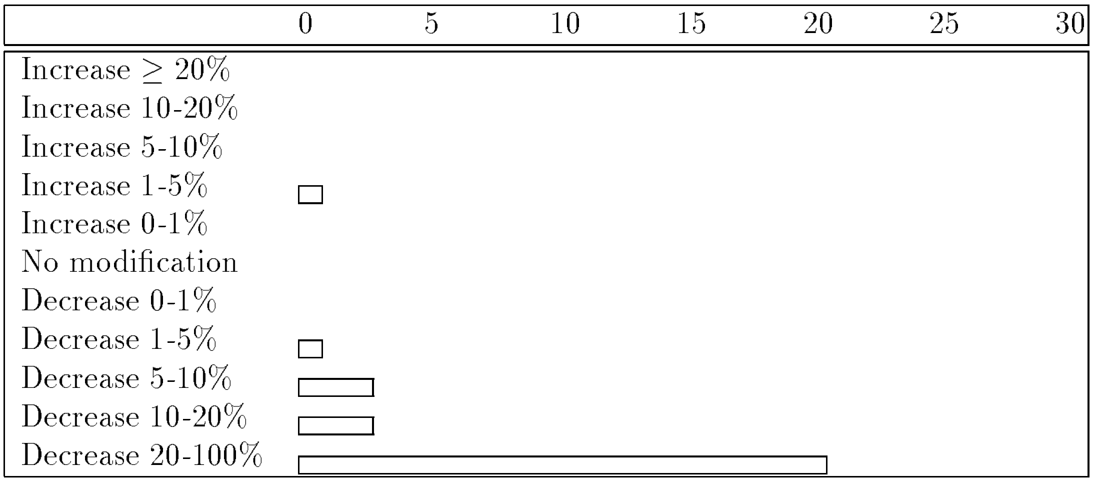


Figure 5.62: Head-strict vs. head-and-tail-strict initial demand: Effect on critical path length

The effect on the average parallelism can be seen in Tables 5.63 and 5.64. Using a head-and-tail-strict initial demand clearly improves the distribution of the average parallelism. For comparison, Figure 5.65 presents the average parallelism for the lenient implementation, given the same set of benchmarks. It can be seen that using a head-and-tail-strict initial demand brings the distribution of the average parallelism for the optimized lazy approach quite close to the one for the lenient implementation.

Clearly, these results suggest an additional strategy to increase the parallelism within a lazy implementation: provide a mean to indicate to the compiler that a program's result is not infinite, so that an initial head-and-tail-strict demand can be used. This could be done either through some form of programmer's annotations and/or through some form of *termination analysis*, i.e., some form of compile-time

[17] Note that there is a single exception: `TRANSFORM_S_`. In this case, the critical path length was increased by 3.58%, whereas the total number of instructions executed effectively got reduced by 5.45%. The exact reason for this phenomena is not understood and occurs only when direct thunks are used.

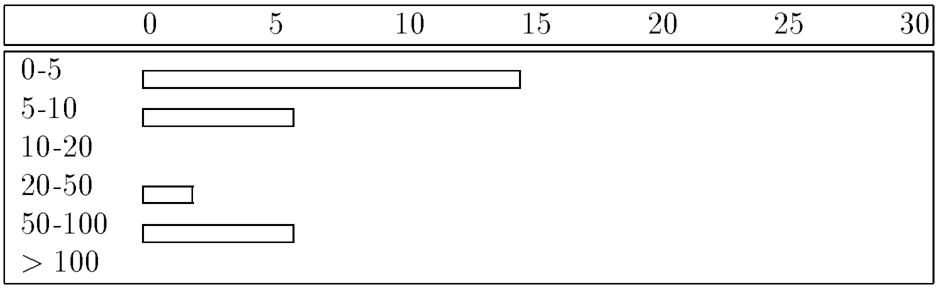


Figure 5.63: Average parallelism for optimized lazy implementation with initial head-strict demand

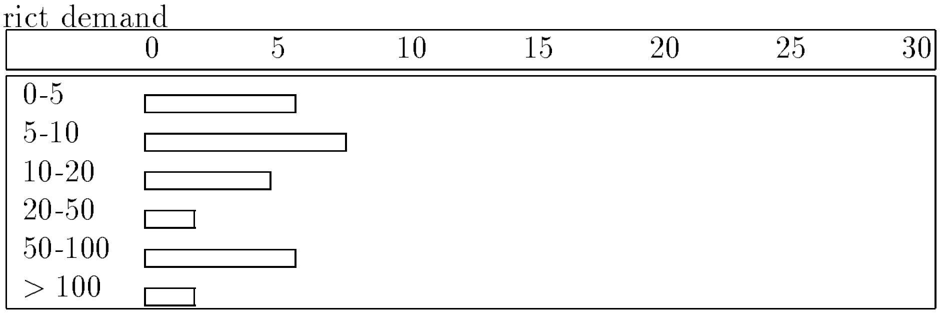


Figure 5.64: Average parallelism for optimized lazy implementation with initial head-and-tail-strict demand

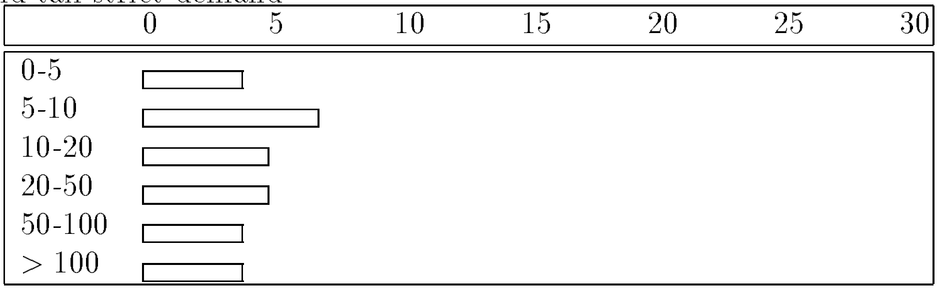


Figure 5.65: Average parallelism for lenient compiler

analysis that would detect whether a program is sure to produce a result which is not infinite. The exact details of this analysis are beyond the scope of the present thesis.

5.5 Conclusion

As mentioned earlier, one of the initial motivation for this research was the fact that, in the case of the lenient approach, there is often *too much* parallelism that is generated. This problem can be remediated by using some form of *throttling mechanism* [AC86b, CA88]: such a mechanism limits the amount of parallelism generated by a program by giving a bound on the maximum number of loop iterations that can be unfolded. One motivation for our research was to investigate whether laziness could be seen as a form of throttling mechanism.

Clearly, as the experiments in the preceding section show, laziness is indeed such a throttling mechanism; however, it is too *drastic*, because it seems that not enough parallelism can be obtained, especially when dealing with programs really requiring a lazy interpreter (Cf. Section 5.4.4).

Compile-time speculation was introduced in an attempt to allow more parallelism to be obtained while preserving the major property of laziness, i.e., the ability to deal with potentially infinite structures. Such speculation can be seen as analogous to throttling, although *in the opposite direction*: throttling is trying to bring the parallelism down when there might be too much of it, whereas speculation is trying to bring it up when there might not be enough of it. There is an important difference, however, between the two mechanisms: whereas throttling is essentially a run-time mechanism, speculation is a compile-time mechanism. This is quite important and probably accounts for the fact that speculation, when implemented within our prototype compiler, was not very successful. Compile-time speculation is quite costly to implement because it requires, in order to perform a safe and precise enough strictness analysis, that *a large domain of analysis be used*; as the speculation level is increased, so must the compile-time domain of demands. Given the cost incurred by computing fixed point within large domains of demands, using large speculation level ($>$ 3-4) would be totally impractical for our prototype compiler.

Of course, another important difference between our results and those related to throttling for leniency is the type of programs being dealt with: in [AC86b, CA88], the programs that were generating too much parallelism were scientific programs using mainly loops and arrays; as for our benchmark programs, they deal with various types of dynamic data structures, not regular arrays.

One interesting element that should be stressed again is that increasing the speculation level, at least for the values we have used (0, 1, 2, 3) has very little impact on the *distribution* of the average parallelism. This phenomena was explained earlier: it results from the fact that when more precise strictness analysis results are obtained, then both the critical path length and the total number of instructions are reduced, so the quotient remains relatively constant. In fact, the general tendency is that the average parallelism slightly decreases as the speculation level is increased, because the total number of instructions executed generally decreases faster than the critical path length.

Given all these results, this leads to two important questions:

1. Given that we would want to implement a lazy functional language for a parallel machine, what type of compiler should be built?
2. Is there any future for lazy functional languages?

Let us briefly discuss the latter question first; the former will be briefly discussed below.

It is far from clear what is the future of lazy functional programming languages. The real use of laziness is to manipulate potentially infinite data structures. Many of the earlier claims that were made about the need for laziness for dealing with certain types of problems [Bir84, All89, All92, All93] were not justified, as most of these programs only required non-strictness/leniency, not laziness.

As for programs really requiring lazy evaluation, i.e., which are simpler to express using a lazy functional language and/or manipulate potentially infinite data structures, there seems to be few of them: a few months ago, a posting was made on *comp.lang.functional* asking for examples of such programs; many people asked for the results of the survey, but few provided examples of such programs, and most of the examples received were classical ones as can be found in [Hug84, Hen80, Tur81].

One possible line of research might be to investigate forms of compile-time analysis and/or type systems that would allow lazy and non-lazy programs parts to be integrated together, as was done for the integration of *impure* constructs with functional language [GL86, SRI91, ORH93].

As for the type of compiler that should be built in order to implement a lazy functional language for a parallel machine, here are some elements of response, given our experience with our prototype compiler:

- From a theoretical point of view, the idea of abstract conditions is interesting as it allows a backward approach to obtain results as good as those that would be obtained by a forward approach when dealing with certain special cases, e.g., those involving correlation between arguments. In practice, however, there seems to be few cases which can really benefit from the additional power gained by such abstract conditions; thus, considering the costs incurred by their manipulation, i.e., identification of complementary conditions, a more realistic compiler should probably avoid implementing this idea.

- Domain of demands and speculation: This is probably *both* the major strength *and* weakness of our backward approach. In the absence of speculation, simply increasing the size of the domain from 1 to 2 seems to bring interesting benefits. However, in order to deal with large levels of speculation, which is interesting in order to reduce the critical path length and the total number of instruction, large domains of demands have to be used. Clearly, even with an improved and better tuned implementation, the costs associated with computing fixed point within a large domain of demands would probably be prohibitive. Thus, although the notion of bounded speculation is interesting, it is not quite clear how it could be efficiently implemented within a realistic compiler.

- Run-time evaluators: Unless evidence to the contrary is found, run-time evaluators do not seem to be very useful: in all the programs we examined, there were very few cases which could really benefit from the use of run-time evaluators. Given the additional overhead required to deal with such run-time demands, which would have to be incurred for all cases, run-time evaluators are probably not worth being implemented either.

- Programs with finite results: Knowing when the result of a program is finite is probably one of the most, if not *the* most, important factor in obtaining more precise strictness results, as it allows the compiler to use an initial demand which is much stronger. This is probably the most important line of research that should be pursued further, so that some efficient form of termination analysis would be included in a compiler for a lazy language.

Chapter 6

Conclusion

This thesis has discussed the implementation of lazy functional programming languages for parallel fine-grain (dataflow-like) architectures. This was done by introducing a new model of backward strictness analysis based on abstract demand propagation. Although there were previous approaches to strictness analysis that were based on demand propagation, abstract demand propagation was seen to be interesting in that a formal interpretation of the notions of demands and demand propagation could be given, thus establishing the soundness of the approach. Furthermore, it also showed that an approach based on demand propagation *per se* could also express and analyze the elusive property of head-strictness, which was made possible using the notion of negative demand combined with the use of cyclic demands.

A prototype compiler based on the abstract demand propagation model and targeted for a parallel dataflow machine has also been implemented. The compiler extended the basic demand propagation model with a number of additional features — e.g., abstract conditions, run-time demands — that improved the precision of the strictness analysis process. More importantly, the notion of conditional demands realized by the compiler showed that it was possible for a backward approach to obtain, in certain special cases involving correlation between arguments, results which were as good as those that could be obtained by a forward approach.

Finally, it has been shown through a number of experiments that, even though the performance improvements made possible by the strictness analysis process were interesting — i.e., reduction of the total number of instructions executed and reduction of the critical path length — little parallelism could effectively be extracted from lazy programs. Although the resulting parallelism could be slightly improved using (bounded) speculative laziness, the resulting increased compilation costs made it improbable that large quantity of parallelism can effectively be obtained from lazy programs.

A possible line of research suggested in Chap. 5 was to explore the use of termination analysis, in addition to strictness analysis, in order to increase the quantity of parallelism that could be identified from programs written in lazy functional languages. Defining and implementing such an analysis would certainly be useful; however, it must be realized that this would not bring much help for programs that really require laziness, i.e., those which produce results which are potentially infinite. In these latter cases, termination analysis would not bring any improvement, so the available parallelism would still remain low. Thus, parallelism and laziness can be seen as largely incompatible, that is, one must be willing to sacrifice parallelism in order to really take advantage of laziness. Investigation into language mechanisms that would allow lazy segments of programs to be combined with non-lazy ones would be interesting in that respect, as they could restrict laziness to where it is really needed. However, such an integration would have to be possible in a clean and modular way, contrary to the approach of [Hel89] where the resulting program behavior was not always obvious.

On the practical side, an important line of research would be to extend the compiler so that it would deal with a source language that includes higher-order functions. This would require a slightly different framework, because backward approaches, although interesting for the analysis of data structures, do not have as much success with the analysis of higher-order functions [Hug88]. Thus, the compiler would have to be extended to include a combination of both forward and backward analysis. How the approach of demand propagation could be blended with a forward framework remains a topic of further investigation.

On the theoretical side, further investigation of the relationships between forward and backward approaches to strictness analysis could be pursued. [1] There are two levels on which the relationships could be examined. The first one concerns the

[1]An investigation of the exact relationship between projection analysis and abstract demand propagation would also be interesting. Clearly, there are closed links between the two. Our conjecture is that our domain of demands, when restricted to demands of size at most 1, is isomorphic to the domain of projections described in [HW87]. This would remain to be formalized, together with some investigation into the precision of the results that can be obtained from each approach.

expressive power of forward vs. backward approaches. Some relationships have been formally established in the scalar case [NM92] and some tentatives have also been done in the case of data structures [HL92]. It would be interesting to see if using the notions of demands and demand propagation could help formalize such relationships. The second level concerns the costs of the analyses. Dealing with sets of demands is what makes abstract demand propagation costly. However, contrary to forward approaches, the information for the arguments can be computed all at once, whereas forward approaches require the strictness of each argument to be computed independently. Given forward and backward analyses of equivalent expressive power, it would be interesting to know which approach really is more efficient.

Still on the theoretical side: the notion of demand propagation has been presented, within this thesis, using a purely denotational setting, which was motivated by the view of demand propagation as approximate inverse computation. However, it is clear that demand propagation can also be viewed, as was done in the presentation of the prototype compiler (Chap. 4), in a more operational manner. Formally establishing the congruence between the operational and the denotational view would be an interesting problem.

Probably the most interesting line of further research, however, would be to use the approach of abstract demand propagation for problems other than strictness analysis. For example, using an appropriate form of abstract demand propagation to handle the problem of thread partitioning for multi-threaded architecture would be interesting. A notion of demand-set partitioning is already present in the work of [SCvE91], albeit in a primitive form as recursive functions can not be analyzed; using a domain of demands simpler than the one used in our prototype compiler but more complex than the one in [SCvE91] could allow good results to be obtained while still obtaining an efficient compiler implementation. [2] Investigation into the applicability of backward abstract demand propagation to other forms of compile-time analysis could also be interesting, e.g., the problem of parallelizing dynamic data structures described in [Hen90].

[2]Complex abstract conditions and conditional demands are an important factor in making the compilation costs so high in our prototype compiler. Avoiding such conditions, thus precluding the need to deal with conditional demands and the associated costs of finding complementary conditions, would make an efficient implementation of the compiler possible and would not have too much of a negative impact on the precision of the analysis.

Appendix A

A quick overview of the VDM notation

A.1 Introduction

VDM — which stands for *Vienna Development Method* — is a methodology for designing software systems. It consist of two parts: a notation and a design methodology. The notation, called *Meta-IV*, can be used for documenting designs and as a *specification language*. This notation also includes the usual notation of first-order predicate logic. The design methodology is based on a systematic development process. This process involves producing a series of specifications, each one being more concrete, i.e., closer to an implementation. In addition, it also involves ensuring the correctness of the refinement process, which is done by showing that *proof obligations* are fulfilled. In this thesis, only the Meta-IV notation is used, so only this part of VDM is presented in more detail.

VDM is based on the principle of *model-oriented specifications*. In this approach, classes of objects (data types) are explicitly constructed, i.e., are specified by defining a *model* for them. This is in contrast with the *algebraic* approach where abstract data types are specified implicitly, which is done by specifying the relationships that must be obeyed between the different operations [GH78, GTWW78].

Models in VDM can be constructed using the predefined domain constructors, e.g., sets, sequences, finite maps, records, or some other user-defined domains. The predefined domain constructors, which are presented in more detail below, can be seen as generic abstract data types, i.e., as data structures to which are associated predefined operations for manipulating them.

The specification of the operations on classes of objects can be given in a model-oriented way, i.e., in a constructive manner. This is done using the operations on the predefined domains and by *coding* the function in a high-level programming-like notation. Function specifications can also be given using pre and post-conditions (cf. Section A.8).

The following sections present the basic domain constructors of VDM together with their associated operations. Not all the elements of VDM are introduced; only the major ones used in the thesis. The notation is taken from [Bjo78, Jon78, BJ82] and has been produced using a package of LaTeX macros for VDM made publicly available by M. Wolczko, from the University of Manchester. For more detail on VDM in general, the following references should be consulted: [BJ82, Bjo87].

A.2 Basic domains

The basic predefined domains of VDM are the ones usually found in most programming languages:

- $\mathbb{B}$: boolean values, i.e., $\{\mathsf{true}, \mathsf{false}\}$;
- $\mathbb{N}$: natural numbers, i.e., $\{0, 1, 2, \ldots\}$;
- $\mathbb{N}_1$: $\{1, 2, \ldots\}$;
- $\mathbb{Z}$: integers, i.e., $\{\ldots, \text{-}2, \text{-}1, 0, 1, 2, \ldots\}$;

A.3 Sets

The set domain constructor of VDM corresponds to the usual mathematical notion of *powerset*. Given a set of objects O, then $D = O\text{-}\mathsf{set}$ defines D as the powerset of O, which means the objects of D are (finite) subsets of O. Objects from this domain can be constructed in two manners:

- explicit enumeration: $\{2, 4, 6, 8\}$;
- implicit formation: $\{2 * x \mid 1 \leq x \leq 4\}$.

The predefined operations on such set objects are the usual operations: $\cup, \cap, -, \in$ $, \subset, \subseteq, \mathsf{card}, \bigcup$, etc. The last operation ($\bigcup$) represents the distributed union: given $D = \{S_1, \ldots, S_n\}$, then $\bigcup D$ is $S_1 \cup \ldots \cup S_n$.

A.4 Sequences

VDM's sequences correspond to the notion of *lists*. Given a set of objects O, then $S = O^*$ defines S as the set of sequences (including the empty one, denoted by $[\,]$) made up of objects from O. Again, objects from this domain can be constructed in two manners:

- explicit enumeration: $[2, 4, 6, 8]$;
- implicit formation: $[2 * x \mid 1 \leq x \leq 4]$.

The predefined operations on sequences are the following:

- hd, e.g., $\mathsf{hd}\,[1, 2, 3]$ is 1;
- tl, e.g., $\mathsf{tl}\,[1, 2, 3]$ is $[2, 3]$;
- len, e.g., $\mathsf{len}\,[1, 2, 3]$ is 3;
- $\curvearrowright$, sequence concatenation, e.g., $[1, 2] \curvearrowright [3, 4]$ is $[1, 2, 3, 4]$;
- inds, which returns the set of indices of a list, e.g., $\mathsf{inds}\,[2, 4, 6]$ is $\{1, 2, 3\}$;
- elems, which returns the *set* of elements from the sequence, e.g., $\mathsf{elems}\,[1, 2, 1, 2, 3]$ is $\{1, 2, 3\}$;
- dconc, which takes a sequence of sequences, and returns their (distributed) concatenation, e.g., $\mathsf{dconc}\,[[1, 2], [2, 3], [3, 4]]$ is $[1, 2, 2, 3, 3, 4]$;
- $(\_)$, sequence selection, e.g., $[2, 4, 6](2)$ is 4.

A.5 Finite maps

VDM's finite maps correspond to the mathematical notion of functions as set of pairs, i.e., a function from D to R is viewed as a subset of the cartesian product of D and R ($D \times R$). Given sets of objects D and R, then $M = D \xrightarrow{m} R$ defines M as the set of finite maps from D to R. The empty map is denoted by $\{\,\}$ and objects from this domain can be constructed in two manners:

- explicitly: $\{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6, 4 \mapsto 8\}$;
- implicitly: $\{x \mapsto 2 * x \mid 1 \leq x \leq 4\}$.

Assume we have the following finite map objects:

- $m_1 = \{1 \mapsto 2, 2 \mapsto 3\}$;
- $m_2 = \{3 \mapsto 4\}$;
- $m_3 = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4\}$;

Then the definitions of the predefined operations on finite maps are:

- dom, returns the domain of the map, e.g., $\mathsf{dom}\, m_1$ is $\{1, 2\}$;
- rng, returns the range of the map, e.g., $\mathsf{rng}\, m_1$ is $\{2, 3\}$;
- card, returns the cardinality of the domain, e.g., $\mathsf{card}\, m_3$ is 3;
- $(\_)$, map application, e.g., $m_1(2)$ is 3;
- $\dagger$, map override, which *replaces* the value associated with elements (which must already be defined), e.g., $m_3 \dagger \{1 \mapsto 6, 3 \mapsto 4\}$ is $\{1 \mapsto 6, 2 \mapsto 3, 3 \mapsto 4\}$;
- $\cup$, map union, which adds new elements to the domain, e.g., $m_1 \cup \{3 \mapsto 4\}$ has the same value as m_3;
- $\lhd$, map restriction, which restricts the domain of the map, e.g., $m_3 \lhd \{1, 2\}$ is the same value as m_1;
- $⩤$, map removal, which removes elements from the domain, e.g., $m_3 ⩤ \{1, 2\}$ is the same value as m_2.

A.6 Abstract syntax

New domains are defined using an *abstract syntax*, similar to the BNF notation. A first way to define a domain is as the non-discriminated union of other domains. This is done in the following manner:

$D = D_1 + \ldots + D_i + \ldots + D_n$

As in BNF, an optional element can be specified using $[\_]$. Such a domain then includes the special element nil. In other words, the following holds:

$[D] = D + \{\mathsf{nil}\}$

Another way to define domains is as product of other domains, which corresponds to records in programming languages. A domain of records is introduced using "::"

in place of "=" when defining a new domain in the abstract syntax. For example, given the domains $O_1, ..., O_n$, a new record domain can be defined by:

$$
\begin{aligned}
D \ :: \ & S_1 \ : \ O_1 \\
& \ldots \\
& S_n \ : \ O_n
\end{aligned}
$$

Given objects o_1, ..., o_n from the appropriate domains, an object from domain D can be created by $mk\text{-}D(o_1, \ldots, o_n)$. Such a *mk-* operation is implicitly introduced when a new record domain is defined. The set corresponding to domain D is then the following:

$$D = \{mk\text{-}D(o_1, \ldots, o_n) \mid o_i \in O_i \wedge 1 \leq i \leq n\}$$

Also, given an object d from domain D, d can be decomposed into its constituent components either explicitly using the associated selectors ($S_1, \ldots, S_n$ in the definition) or implicitly:

- explicit decomposition, e.g., $S_i(d)$ selects from d the component from domain O_i; [1]
- implicit, e.g., **let** $mk\text{-}D(o_1, \ldots, o_n) = d$ **in** ....

An implicit decomposition can be used any place an object from the appropriate domain can appear, e.g., in a **let**, as a formal parameter in a function definition, as a value in a **case** statement, etc.

Finally, for any domain D, a predicate *is-D* is also defined. This predicate determines if an arbitrary object is in fact a member of the specified domain. This predicate must be used in place of $\in$, whose use should be restricted to set domains, i.e., to test whether o from domain O is an elements of D, which must be defined as an O-**set**.

A.7 Control structures

In the present paper, only a few of the control structures of VDM (called *combinators* in VDM terminology) are used. Furthermore, we restrict ourselves to the functional combinators.

The combinators we use are the following:

[1] Alternatively, a field can be accessed using a Pascal-like dot-notation, e.g., $d.O_i$.

- If-expressions:

  if e_1 then e_2 else e_3

- Let-expressions:

  let $e_1 = e_2$ in e_3

- Case-expressions:

  cases e_1 of

  $value_1 \rightarrow result_1$

  ...

  $value_n \rightarrow result_n$

  others *default-result*

  end

- LISP-style sequential cond-expressions:

  cond

  $predicate_1 \quad \rightarrow result_1$

  ...

  $predicate_n \quad \rightarrow result_n$

  otherwise *default-result*

  end

Note that, in order to correctly interpret the various semantics definitions, we assume the following semantics for the pattern-matching operation on sequences, i.e., we assume that pattern-matching is strict:

let $[e_j \mid 1 \leq j \leq n] = \perp$ in $e \equiv \perp$

A.8 Explicit vs. implicit function specifications

As mentioned earlier, there are two ways to specify functions: explicitly and implicitly. In the first case, some explicit algorithm is presented that illustrates what the function is supposed to do. In the second case, the implicit specification indicates what the function does by giving the condition that will hold after the function has been executed (the *post*-condition) assuming the proper condition holds when the function is called (the *pre*-condition). Below are small examples illustrating the difference between the two styles of specification for the function *fact*, function which computes the factorial of a non-negative number n and returns an undefined result when n is negative.

- Explicit specification: An algorithm describing what *fact* computes is presented. Note that the real algorithm used to implement the function could be different, e.g., it could use an explicit loop instead of recursion.

$$
\begin{array}{lll}
fact : \mathbb{Z} \to \mathbb{Z} & & \\
fact(n) & \triangleq & \textsf{if } n = 0 \\
 & & \textsf{then } 1 \\
 & & \textsf{else } n * fact(n-1)
\end{array}
$$

- Implicit specification: Here, r denotes the result produced by *fact*, which is then characterized by the post-condition $(r = n!)$.

$$
\begin{array}{lll}
fact : \mathbb{Z} \to \mathbb{Z} & & \\
fact(n) & \triangleq & r \\
\textsf{pre } n \geq 0 & & \\
\textsf{post } r = n! & &
\end{array}
$$

Appendix B

An abstraction of the demand propagation semantics for data structures

This appendix presents the details required in order to define an abstraction of the exact demand propagation semantics presented in Section 3.3.3. Like for scalars, this is an abstraction in that information is lost when dealing with conditionals, i.e., case-expressions. When dealing with such conditionals, the result of all branches will be *combined* together to obtain a less precise but safe result. Again, abstract demands will correspond, through a concretization function, to *sets* of exact demands.

B.1 Abstract demands and demand environments

The basic demands, i.e., those which are used as components of vectors of demands, are presented in Fig. B.1. The names used in that figure stand for the following abbreviations and are used for the following purpose (further explanation are given in the subsequent paragraphs):

- *ADmdRef*: Abstract Demand Reference [1]: Used to denote names of/references to demands.
- *ANDmd* = Abstract Negative Demand: Negative demand @.

[1] In VDM, the type *Token* represents an unspecified set of values whose only property is that they can be compared for equality/non-equality. Thus, values of this type can simply be seen as identifiers or pointers.

- $APDmd$ = Abstract Positive Demand: Some positive data structure demand, similar to the exact demands of Section 3.3.3 but with abstract demand subcomponents.
- $ABDmd$ = Abstract Basic Demand: A demand which is either negative or positive.
- $ADmd$ = Abstract Demand: A set of basic demands, i.e., of demands which are either negative or positive.
- $ADmdEnv$ = Abstract Demand Environment: An environment (global state) that associates demand references with abstract demands.

Cyclic demands, which are used to finitely represent demands for potentially infinite data structures (cf. p. 84) are made possible by the use of abstract demand environments ($\sigma, \sigma' \in ADmdEnv$). A demand environment σ can map an abstract demand reference ($\delta \in ADmdRef$) to an abstract demand value ($ADmd$), value which is a *set of basic* abstract *demands* ($ABDmd$). Any such basic demand is either a negative demand (@) or a positive one ($APDmd$), where a positive demand, as for exact demands, indicates a data constructor name together with the appropriate (references to abstract) demands for the components.

B.1.1 Well-formed demand references and demand environments

Certain restrictions have to be imposed upon demand environments in order that they be considered well-formed. These restrictions are described by the data type invariant *inv-ADmdEnv*. Informally, this invariant can be explained as follows:

(i) Any demand environment always contain the predefined references for the bottom, negative and scalar demand.

(ii) The cardinality of the environment is finite: this will be important when termination is discussed. [2]

(iii) Any demand reference part of the environment must be well-formed. Cf. below for the description of *is-wfAD*.

[2] In VDM, this finiteness restriction is usually left implicit, as a map always has to be finite.

$$ADmdRef = Token$$

$$ABDmd = ANDmd + APDmd$$

$$ANDmd = \{@\}$$

$$\begin{array}{lll} APDmd :: & cons & : Cons\text{-}name \\ & components & : ADmdRef^* \end{array}$$

$$ADmd = ABDmd\text{-}\mathsf{set}$$

$$ADmdEnv = ADmdRef \xrightarrow{m} ADmd$$

where

$$\begin{array}{l} inv\text{-}ADmdEnv(\sigma) \quad \triangleq \\ \quad \{\phi, @, \$\} \subseteq \mathsf{dom}\,\sigma \quad (*\ i\ *) \\ \quad \wedge\ \exists k \in \mathbb{N} \cdot \mathsf{card}\,(\mathsf{dom}\,\sigma) \leq k \quad (*\ ii\ *) \\ \quad \wedge\ \forall \delta \in \mathsf{dom}\,\sigma \cdot is\text{-}wfAD(\delta, \sigma) \quad (*\ iii\ *) \\ \quad \wedge\ \forall \delta_1, \delta_2 \in \mathsf{dom}\,\sigma \cdot \delta_1 \sqsubseteq_\sigma \delta_2 \wedge \delta_2 \sqsubseteq_\sigma \delta_1 \ \Rightarrow\ \delta_1 = \delta_2 \quad (*\ iv\ *) \end{array}$$

Figure B.1: The domain for abstract demands (data structures)

(iv) Demands are uniquely represented, i.e., there are no two distinct demands which are equivalent to each other. This will ensure that the ordering defined between demand references (cf. below) is effectively a partial ordering.

Demand references are considered well-formed, relative to a demand environment σ, when the following predicate holds:

Definition B.1 *(Well-formed abstract demands)*

$$\begin{array}{l} is\text{-}wfAD(\delta, \sigma) \quad \triangleq \\ \quad \delta \in \mathsf{dom}\,\sigma \quad (*\ \mathrm{i}\ *) \\ \quad \wedge\ \exists k \in \mathbb{N} \cdot \mathsf{card}\,\sigma(\delta) \leq k \quad (*\ \mathrm{ii}\ *) \\ \quad \wedge\ \forall mk\text{-}APDmd(cons, components) \in \sigma(\delta) \cdot \\ \quad\quad \forall \delta' \in \mathsf{elems}\,components \cdot is\text{-}wfAD(\delta', \sigma) \quad (*\ \mathrm{iii}\ *) \\ \quad \wedge\ \forall bd_1, bd_2 \in \sigma(\delta) \cdot \\ \quad\quad is\text{-}APDmd(bd_1) \wedge is\text{-}APDmd(bd_2) \wedge cons(bd_1) = cons(bd_2) \\ \quad\quad \Rightarrow\ bd_1 = bd_2 \quad (*\ \mathrm{iv}\ *) \end{array}$$

In words, a demand reference δ is well-formed relative to σ when the following hold:

(i) δ is part of the domain of σ.

(ii) δ is associated with a finite number of demand values. [3]

(iii) Any positive demand contained in δ is such that all its component demands refer to well-formed demand references.

(iv) For any constructor name, at most a single basic positive demand using that constructor is contained in δ. This is imposed in order to ensure a canonical representation for demands. As a set of basic demands really represents an upper-bound of the various demands it contains, demands with the same constructor should be combined into their appropriate upper-bound. For example, the following demand would not be valid: $\{\mathsf{Cons}\,\$\,@, \mathsf{Cons}\,@\,@\}$. Instead, the following demand should be used: $\{\mathsf{Cons}\ \hat{}\$\ @\}$.

B.1.2 Ordering for abstract demands and demand environments

Given a demand environment, we define an ordering between any two *ADmdRef* as described by Definitions B.2, B.3 and B.4. A demand reference is associated, through a demand environment, with an abstract demand, the latter being a set of basic abstract demands. These three levels thus give rise to three levels of ordering, i.e., the definition has been explicitly factored for each of the appropriate level. First, for demand references (Defn B.2), we have $\delta_1 \sqsubseteq_\sigma \delta_2$ if their corresponding abstract demands are also in the same relationship, i.e., $\sigma(\delta_1) \sqsubseteq_\sigma \sigma(\delta_2)$. As for abstract demands (Defn. B.3), because we are dealing with sets of basic demands, $ad_1 \sqsubseteq_\sigma ad_2$ holds if for each bd_1 in ad_1, there is an appropriate basic demand bd_2 in ad_2 which is at least as defined ($bd_1 \sqsubseteq_\sigma bd_2$). Finally, for basic demands (Defn B.4), $bd_1 \sqsubseteq_\sigma bd_2$ holds if both are negative or if both are positive demands with the same constructors and with their components properly ordered.

[3]Like maps, this is also implicit in VDM when using the $\_$-**set** constructor.

Definition B.2 *(Ordering between demand references)*

$$\sqsubseteq : ADmdRef \to ADmdRef \to ADmdEnv \to \mathbb{B}$$

$$\delta_1 \sqsubseteq_\sigma \delta_2 \quad \triangleq \quad \delta_1 \in \mathsf{dom}\,\sigma \wedge \delta_2 \in \mathsf{dom}\,\sigma \wedge \sigma(\delta_1) \sqsubseteq_\sigma \sigma(\delta_2)$$

Definition B.3 *(Ordering between abstract demands)*

$$\sqsubseteq : ADmd \to ADmd \to ADmdEnv \to \mathbb{B}$$

$$ad_1 \sqsubseteq_\sigma ad_2 \quad \triangleq \quad \forall bd_1 \in ad_1 \cdot \exists bd_2 \in ad_2 \cdot bd_1 \sqsubseteq_\sigma bd_2$$

Definition B.4 *(Ordering between basic abstract demands)*

$$\sqsubseteq : ABDmd \to ABDmd \to ADmdEnv \to \mathbb{B}$$

$$\begin{array}{l} bd_1 \sqsubseteq_\sigma bd_2 \quad \triangleq \\ \qquad bd_1 = @ \wedge bd_2 = @ \\ \qquad \vee \\ \qquad bd_1 = mk\text{-}APDmd(K, [\delta_1,\ \ldots,\ \delta_k]) \wedge bd_2 = mk\text{-}APDmd(K, [\delta'_1, \ldots, \delta'_k]) \\ \qquad\quad \wedge\, \delta_j \sqsubseteq_\sigma \delta'_j \;\; (j = 1, \ldots, k) \end{array}$$

It is easy to see that the ordering between demand references is reflexive and transitive; on the other hand, the anti-symmetry property is respected only if the fourth clause (iv) of *inv-ADmdEnv* is respected. For example, given $\sigma = \{\$ \mapsto mk\text{-}APDmd(\$, [\,]), \$' \mapsto mk\text{-}APDmd(\$, [\,])\}$, then we would have $\$ \sqsubseteq_\sigma \$'$ and $\$' \sqsubseteq_\sigma \$$, but with $\$ \neq \$'$. Both this uniqueness property and the one concerning the canonical representation will have to be ensured, by construction, by all the operations to be defined on abstract demands.

Because of the way in which demand environments will be manipulated, i.e., incremental construction (cf. below), the ordering between demand environments will be defined as presented in Defn. B.5. Essentially, this simply says that σ_2 is an extension of σ_1, i.e., when the domain of σ_2 is restricted to the domain of σ_1 ($\mathsf{dom}\,\sigma_1 \lhd \sigma_2$), then it is equivalent to σ_1, in other words, the initial environment is left undisturbed. [4]

Definition B.5 *(Ordering between demand environments)*

$$\sqsubseteq : ADmdEnv \to ADmdEnv \to \mathbb{B}$$
$$\sigma_1 \sqsubseteq \sigma_2 \quad \triangleq \quad \mathsf{dom}\,\sigma_1 \lhd \sigma_2 = \sigma_1$$

For example, given the following demand environments (written in a simplified notation), we would effectively have that $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \sigma_2$:

$$\sigma_0 = \{\phi \mapsto \{\,\}, @ \mapsto \{@\}, \$ \mapsto \{\$\}\}$$
$$\sigma_1 = \sigma_0 \cup \{HTS \mapsto \{\mathsf{Cons}\ \$\ HTS\}\}$$
$$\sigma_2 = \sigma_1 \cup \{TS \mapsto \{\mathsf{Cons}\ @\ TS\}\}$$

B.2 Operations on abstract demands

B.2.1 Demand environment transformers

As mentioned earlier (cf. p. 86), using a demand environment to represent (possibly cyclic) demands brings some additional complexities in defining the various operations because of the need to carry around the global state, i.e., the demand environment.

In our semantic functions, we will be using the approach and syntax of *monads for state transformations* [Wad92, Lau93]; a short presentation of this approach, adapted for demand environments, is presented in Appendix C (p. 227).

Any operation using a demand environment and creating a new environment will have its result type be an instance of the type of *Demand environment Transformers*, defined as follows:

Definition B.6 *(Demand environment transformer)*

[4] This, clearly, is also a partial ordering. Reflexivity and transitivity are immediate. As for anti-symmetry, $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \sqsubseteq \sigma_1$ implies that $\mathsf{dom}\,\sigma_1 = \mathsf{dom}\,\sigma_2$, which leads to the required result as $\sigma_1 = \mathsf{dom}\,\sigma_1 \lhd \sigma_2 = \mathsf{dom}\,\sigma_2 \lhd \sigma_2 = \sigma_2$. Note that, equivalently, this ordering could simply have been defined as $\sigma_1 \sqsubseteq \sigma_2 = \mathsf{dom}\,\sigma_1 \subseteq \mathsf{dom}\,\sigma_2$.

$$DT\ t = ADmdEnv \rightarrow t \times ADmdEnv$$

where

$$inv\text{-}(DT\ t)\,(dt) \quad \triangleq \quad \forall \sigma \cdot dt\ \sigma = (v, \sigma') \ \Rightarrow\ \sigma \sqsubseteq \sigma'$$

A demand environment transformer of type $DT\ t$ is thus a function which, given a demand environment, produces a value of type t paired with a (possibly) new demand environment. The invariant indicates that the demands which were part of the initial environment are left unchanged, as mentioned earlier. [5]

B.2.2 Explicit definitions of $\oplus$ and $\cup$ for abstract demands

In this section, the explicit definitions of $\oplus$ and $\cup$ for abstract demands are presented. Note that the definition of $\cup$ is presented first, as it is somewhat simpler to understand; furthermore, some of the definitions needed for $\cup$ are later used within the definition of $\oplus$.

In order to deal correctly with cyclic demands, memoized forms of the functions have to be introduced. This memoization is obtained using the parameter *seen*, of type $MemoMap_2$, which keeps track, while traversing the sub-components of demand structures, of which pairs of arguments have already been encountered and to which resulting demand reference their combination has been associated. The definition of a $MemoMap_2$ is the following, where the invariant simply indicates that, because the operator which is being memoized is a commutative binary operator, the order of the arguments is irrelevant (so the set could be a singleton set if $\delta_1 = \delta_2$):

$$MemoMap_2 = ADmdRef\text{-}\mathsf{set} \xrightarrow{m} ADmdRef$$

where

$$inv\text{-}MemoMap_2(seen) \quad \triangleq \quad \forall pair \in \mathsf{dom}\, seen \cdot 1 \leq \mathsf{card}\, pair \leq 2$$

[5] It is straightforward to see that, using only the constructs presented in App. C, all the demand transformers used in the abstract semantics effectively satisfy the invariant. Finally, note that an implication of this invariant is also that the bottom element of type $DT\,t$ is the demand transformer $return\ t_\perp$.

In addition, an auxiliary predicate ($\sim$) has to be introduced whose role is to determine whether two basic demands can be properly merged together, i.e., whether they represent demands for compatible values. Essentially, two basic demands are compatible if one of them is a negative demand or if both are positive demands with the same constructor.

$$\sim : ABDmd \rightarrow ABDmd \rightarrow \mathbb{B}$$

$$\begin{aligned} bd_1 \sim bd_2 \quad &\triangleq \\ &bd_1 = @ \vee \\ &bd_2 = @ \vee \\ &bd_1 = \textit{mk-APDmd}(K, comps_1) \wedge bd_2 = \textit{mk-APDmd}(K, comps_2) \end{aligned}$$

Union ($\cup$)

The $\cup$ function computes an upper-bound (in fact, the *least* upper bound, although this property is not required in the proof) of two demand references. Because cyclic demands have to be handled, memoizing is used (parameter *seen* of type $MemoMap_2$). The $\cup$ function on abstract demands works as follows: a call to $union^*$ is made, [6] using an initial empty memo-map. Function $union^*$ first checks if the two demand references to be $\cup$-ed have been encountered before: if so, the associated result is returned. If the pair of arguments has never been encountered before, then a new demand reference is obtained to represent the result ($\delta \leftarrow \textit{get-new-ref};$) — this is required because the result of the union might be cyclic — and an appropriate binding is made in the memo-map to memoize the fact that the union of δ_1 and δ_2, if ever needed, is to be δ ($seen' = seen \cup \{\{\delta_1, \delta_2\} \mapsto \delta\}$). The basic demands associated with each of δ_1 and δ_2 are then fetched and combined into a new set ($\sigma(\delta_1) \cup \sigma(\delta_1)$) which is then normalized ($\textit{normalize-up}^*$) to ensure that at most a single constructor appears in the result demand; in addition, calling $\textit{normalize-up}^*$ also ensures that if the result is a cyclic demand, then an explicit cycle will be created, which will happen because the components will be $\cup$-ed until all possible paths have been traversed. [7]

[6] In the following, any function name ending with * indicates a function being memoized, i.e., taking an additional parameter *seen* of type $MemoMap_2$.

[7] The *fetch* operator is used to retrieve the current state, i.e., the current demand environment. It is defined in App. C.2, p. 229.

$\cup : ADmdRef \rightarrow ADmdRef \rightarrow DT\ ADmdRef$

$\cup\ \delta_1\ \delta_2\ \sigma \quad \triangleq$

$\quad union^*\ \delta_1\ \delta_2\ \{\ \}\ \sigma$

pre $\delta_1 \in \mathsf{dom}\,\sigma \wedge \delta_2 \in \mathsf{dom}\,\sigma$

$union^* : ADmdRef \rightarrow ADmdRef \rightarrow MemoMap_2 \rightarrow DT\ ADmdRef$

$union^*\ \delta_1\ \delta_2\ seen \quad \triangleq$

$\quad$ if $\{\delta_1, \delta_2\} \in \mathsf{dom}\, seen$

$\quad$ then $return\ seen(\{\delta_1, \delta_2\})$

$\quad$ else $\delta \leftarrow get\text{-}new\text{-}ref;$

$\quad\quad$ let $seen' = seen \cup \{\{\delta_1, \delta_2\} \mapsto \delta\}$ in

$\quad\quad \sigma \leftarrow fetch;$

$\quad\quad ad' \leftarrow normalize\text{-}up^*\ (\sigma(\delta_1) \cup \sigma(\delta_2))\ seen';$

$\quad\quad \delta' \leftarrow bind\text{-}ref\ \delta\ ad';$

$\quad\quad return\ \delta'$

$normalize\text{-}up^* : ADmd \rightarrow MemoMap_2 \rightarrow DT\ ADmd$

$normalize\text{-}up^*\ ad\ seen \quad \triangleq$

$\quad return\ normalize\text{-}up\text{-}rec^*\ ad\ seen$

$normalize\text{-}up\text{-}rec^* : ADmd \rightarrow MemoMap_2 \rightarrow DT\ ADmd$

$normalize\text{-}up\text{-}rec^*\ ad\ seen \quad \triangleq$

$\quad$ if $\exists bd', bd'' \in ad \cdot bd' \neq bd'' \wedge bd' \sim bd''$

$\quad$ then let bd', bd'' be such that $bd', bd'' \in ad \wedge bd' \neq bd'' \wedge bd' \sim bd''$ in

$\quad\quad$ let $mk\text{-}APDmd(K, [\delta'_1, \ldots, \delta'_k]) = bd'$ in

$\quad\quad$ let $mk\text{-}APDmd(K, [\delta''_1, \ldots, \delta''_k]) = bd''$ in

$\quad\quad \delta_l \leftarrow union^*\ \delta'_l\ \delta''_l\ seen; \quad (l = 1, \ldots, k)$

$\quad\quad$ let $ad' = ad - \{bd', bd''\} \cup \{mk\text{-}APDmd(K, [\delta_l \mid 1 \leq l \leq k])\}$ in

$\quad\quad return\ normalize\text{-}up\text{-}rec^*\ ad'\ seen$

$\quad$ else $return\ ad$

There are a number of properties concerning the above operations on abstract demands which will be important in the proof of safety of the abstract demand propagation semantics. These properties are presented in more detail in App. H.3.1. Informal proofs are also presented, which should be sufficient to convince the reader that these properties effectively hold. The properties of interest are the following:

- Given well-formed demand references and environments, both *normalize-up* and $\cup$ terminate and produce a result which is normalized, i.e., having at most a single demand with any given constructor (cf. p. 301).
- $\cup$ effectively computes an upper-bound of its input arguments (cf. p. 302).

Merge ($\oplus$)

We now present the explicit definition of the merge operator. Note that, because of the need to deal with cyclic demands, memoization also has to be used.

$$\oplus : ADmdRef \rightarrow ADmdRef \rightarrow DT\ ADmdRef$$

$$\oplus\ \delta_1\ \delta_2\ \sigma \quad \underline{\triangle}$$
$$\quad merge^*\ \delta_1\ \delta_2\ \{\,\}\ \sigma$$

$$\mathsf{pre}\ \delta_1 \in \mathsf{dom}\,\sigma \wedge \delta_2 \in \mathsf{dom}\,\sigma$$

$$merge^* : ADmdRef \rightarrow ADmdRef \rightarrow MemoMap_2 \rightarrow DT\ ADmdRef$$

$$merge^*\ \delta_1\ \delta_2\ seen \quad \underline{\triangle}$$
$$\quad \mathsf{if}\ \{\delta_1, \delta_2\} \in \mathsf{dom}\, seen$$
$$\quad \mathsf{then}\ return\ seen(\{\delta_1, \delta_2\})$$
$$\quad \mathsf{else}\ \delta\ \leftarrow\ get\text{-}new\text{-}ref;$$
$$\qquad \mathsf{let}\ seen' = seen \cup \{\{\delta_1, \delta_2\} \mapsto \delta\}\ \mathsf{in}$$
$$\qquad \sigma\ \leftarrow\ fetch;$$
$$\qquad \mathsf{let}\ compatibles = [(bd_1, bd_2) \mid bd_1 \in \sigma(\delta_1) \wedge bd_2 \in \sigma(\delta_2) \wedge bd_1 \sim bd_2]\ \mathsf{in}$$
$$\qquad \mathsf{let}\ k = \mathsf{len}\, compatibles\ \mathsf{in}$$
$$\qquad \mathsf{let}\ (bd_1^l, bd_2^l) = compatibles(l) \quad (l = 1, \ldots, k) \quad \mathsf{in}$$
$$\qquad bd_l\ \leftarrow\ merge\text{-}bd^*\ bd_1^l\ bd_2^l\ seen; \quad (l = 1, \ldots, k)$$
$$\qquad ad'\ \leftarrow\ normalize\text{-}up^*\ \{bd_l \mid 1 \leq l \leq k\}\ \{\,\};$$
$$\qquad \delta'\ \leftarrow\ bind\text{-}ref\ \delta\ ad';$$
$$\qquad return\ \delta'$$

$$merge\text{-}bd^* : ABDmd \to ABDmd \to MemoMap_2 \to DT\ ABDmd$$

$$\begin{array}{l}
merge\text{-}bd^*\ bd_1\ bd_2\ seen \quad \triangleq \\
\quad \textsf{if}\ bd_1 = @ \\
\quad \textsf{then}\ return\ bd_2 \\
\quad \textsf{else if}\ bd_2 = @ \\
\quad\quad \textsf{then}\ return\ bd_1 \\
\quad\quad \textsf{else let}\ mk\text{-}APDmd(K, [\delta'_1, \ldots, \delta'_k]) = bd_1\ \textsf{in} \\
\quad\quad\quad \textsf{let}\ mk\text{-}APDmd(K, [\delta''_1, \ldots, \delta''_k]) = bd_2\ \textsf{in} \\
\quad\quad\quad \delta_l \leftarrow merge^*\ \delta'_l\ \delta''_l\ seen; \quad (l = 1, \ldots, k) \\
\quad\quad\quad return\ mk\text{-}APDmd(K, [\delta_l \mid 1 \le l \le k]) \\
\textsf{pre}\ bd_1 \sim bd_2
\end{array}$$

B.2.3 Auxiliary operations for handling abstract demands

A number of auxiliary operations are needed in order to define the abstract semantics.

Predicates and selectors

This section presents the specification of the various predicates and selectors manipulating abstract demands, i.e., the operations which only examine the demand environment and do not need to modify it. Note that, in all cases, the following pre-condition is tacitly assumed: $\delta \in \textsf{dom}\,\sigma$.

Function *bottom?* determines whether δ represents the bottom abstract demand, i.e., is associated with an empty set of basic demands.

$$bottom? : ADmdRef \to ADmdEnv \to \mathbb{B}$$

$$bottom?\ \delta\ \sigma \quad \triangleq \quad \sigma(\delta) = \{\,\}$$

Function *scalar?* simply determines whether δ is an abstract demand for a scalar (integer) value, i.e., is associated with a positive demand having constructor \$.

$$scalar? : ADmdRef \to ADmdEnv \to \mathbb{B}$$

$$scalar?\ \delta\ \sigma \quad \triangleq \quad mk\text{-}APDmd(\$, [\,]) \in \sigma(\delta)$$

Function *possibly-negative?* checks whether δ contains, among one of its possibilities, a negative demand (@).

$$possibly\text{-}negative? : ADmdRef \rightarrow ADmdEnv \rightarrow \mathbb{B}$$

$$possibly\text{-}negative?\ \delta\ \sigma \quad \triangleq \quad @ \in \sigma(\delta)$$

Function *has-cons?* checks whether δ contains a positive demand using constructor *cons*. When this is the case, then function *select-cons* can be used to retrieve this positive demand.

$$has\text{-}cons? : ADmdRef \rightarrow Cons\text{-}name \rightarrow ADmdEnv \rightarrow \mathbb{B}$$

$$has\text{-}cons?\ \delta\ K\ \sigma \quad \triangleq \quad \exists bd \in \sigma(\delta) \cdot is\text{-}APDmd(bd) \wedge cons(bd) = K$$

$$select\text{-}cons : ADmdRef \rightarrow Cons\text{-}name \rightarrow ADmdEnv \rightarrow APDmd$$

$$select\text{-}cons\ \delta\ K\ \sigma \quad \triangleq \quad bd$$

$\mathsf{pre}\ has\text{-}cons?\ \delta\ K\ \sigma$

$\mathsf{post}\ bd \in \sigma(\delta) \wedge is\text{-}APDmd(bd) \wedge cons(bd) = K$

Constructors

This section presents the specification of operations that produce an abstract demand and, thus, may need to modify the demand environment. Like for predicates and selectors, the following pre-condition is also tacitly assumed: $\delta \in \mathsf{dom}\,\sigma$.

Function *make-positive*, given δ associated with an abstract demand containing a negative demand, produces a new abstract demand with the negative demand removed, thus containing only positive demands (or, possibly, being empty, i.e., *bottom?*, if the negative demand was the only basic demand).

$$make\text{-}positive : ADmdRef \rightarrow DT\ ADmdRef$$

$$make\text{-}positive\ \delta\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

$\mathsf{pre}\ possibly\text{-}negative?\ \delta\ \sigma$

$\mathsf{post}\ \sigma'(\delta') = \sigma(\delta) - \{@\}$

Function *add-negative* returns a new demand reference associated with the same positive demands as δ, in addition to containing for sure a negative demand.

$$add\text{-}negative : ADmdRef \rightarrow DT\ ADmdRef$$

$$add\text{-}negative\ \delta\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

$$\textsf{post}\ \sigma'(\delta') = \sigma(\delta) \cup \{@\}$$

Note that the following (inverse) relationship holds between *make-positive* and *add-negative*:

$$\begin{array}{l}\forall \delta, \sigma \cdot possibly\text{-}negative?\ \delta\ \sigma \ \Rightarrow \\ \textsf{let}\ (\delta', \sigma') = make\text{-}positive\ \delta\ \sigma\ \ \textsf{in} \\ \textsf{let}\ (\delta'', \sigma'') = add\text{-}negative\ \delta'\ \sigma'\ \ \textsf{in} \\ \delta(\sigma) = \delta''(\sigma'')\end{array}$$

Function $mkVariant_{AD}$ is used is used to create an abstract demand appropriate for a case-expression guard, i.e., which simply indicates the required constructor without any further constraints for the components. (Note the special case for constructors *True* and *False*.)

$$mkVariant_{AD} : Cons\text{-}name \rightarrow DT\ ADmdRef$$

$$mkVariant_{AD}\ K \quad \triangleq \quad (\delta', \sigma')$$

$$\begin{array}{ll}\textsf{post}\ \sigma'(\delta') = & \textsf{if}\ K \in \{True, False\} \\ & \quad \textsf{then}\ \{mk\text{-}APDmd(\$, [\,])\} \\ & \quad \textsf{else}\ \{mk\text{-}APDmd(K, [@ \mid 1 \leq j \leq arity(K)])\}\end{array}$$

Finally, function $mkComponent_{AD}$ is used to create a demand appropriate for a data structure selector, i.e., fetching the l^{th} component of a data structure with tag K under demand δ. Note that if δ is *bottom*?, then so will the resulting demand.

$$mkComponent_{AD} : Cons\text{-}name \rightarrow \mathbb{N} \rightarrow ADmdRef \rightarrow DT\ ADmdRef$$

$$mkComponent_{AD}\ K\ l\ \delta\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

$$\begin{array}{l}\textsf{post}\ \textsf{if}\ bottom?\ \delta\ \sigma \\ \quad \textsf{then}\ bottom?\ \delta'\ \sigma' \\ \quad \textsf{else}\ \textsf{let}\ comps = [@ \mid 1 \leq j < l] \curvearrowright [\delta] \curvearrowright [@ \mid l < j \leq arity(K)]\ \ \textsf{in} \\ \qquad \sigma'(\delta') = \{mk\text{-}APDmd(K, comps)\}\end{array}$$

B.3 Vectors of abstract demands

Like in the exact non-standard semantics, we will be computing vectors, but now composed of abstract demands. The domain for demand vectors is defined in Fig. B.2. Note that the ordering on vectors of demands is simply the pointwise extension of the basic abstract demands ordering.

$$ADmdVec_k = \{[d_1, \ldots, d_k] \mid d_i \in ADmdRef\}$$

$$ADmdVec = \sum_{k \geq 1} ADmdVec_k \quad \text{(coalesced sum)}$$

Figure B.2: The domains for vectors of abstract demands used in the abstract non-standard semantics

The merge and union ($\oplus$ and $\cup$) operators also have to be extended to deal with vectors of abstract demands ($\overline{\oplus}$ and $\tilde{\cup}$): these will simply be defined as the element-wise extension of $\oplus$ and $\cup$ (taking into account the need to manipulate the demand environment). The definitions are presented below. [8]

$$\tilde{\oplus} : ADmdVec_k \to ADmdVec_k \to DT\ ADmdVec_k$$

$$\begin{aligned} &\tilde{\oplus}\, [\delta_1, \ldots, \delta_k]\, [\epsilon_1, \ldots, \epsilon_k] \quad \triangleq \\ &\quad \rho_l \leftarrow \oplus\, \delta_l\, \epsilon_l; \quad (l = 1, \ldots, k) \\ &\quad return\, [\rho_l \mid 1 \leq l \leq k] \end{aligned}$$

$$\tilde{\cup} : ADmdVec_k \to ADmdVec_k \to DT\ ADmdVec_k$$

$$\begin{aligned} &\tilde{\cup}\, [\delta_1, \ldots, \delta_k]\, [\epsilon_1, \ldots, \epsilon_k] \quad \triangleq \\ &\quad \rho_l \leftarrow \cup\, \delta_l\, \epsilon_l; \quad (l = 1, \ldots, k) \\ &\quad return\, [\rho_l \mid 1 \leq l \leq k] \end{aligned}$$

B.4 The abstract (non-standard) demand propagation semantics

[8] Note that, although they have not been indicated explicitly, the appropriate pre-conditions on the well-formedness of demand references must hold, i.e., the vectors must use demand references which are well-defined relative to the received demand environment.

$$\widetilde{Fve} = FuncName \xrightarrow{m} ADmdRef \rightarrow DT\ ADmdVec$$

$$\widetilde{\mathcal{M}} : Prog \rightarrow \widetilde{Fve}$$

$$\begin{array}{l}
\widetilde{\mathcal{M}}[\![p]\!] \ \triangleq \\
\quad \mu\, \widetilde{fve} \cdot \{ \\
\quad\quad f_i \mapsto \lambda\delta \cdot \\
\quad\quad\quad \sigma \leftarrow fetch; \\
\quad\quad\quad \textsf{if } \delta \in \textsf{dom}\,\sigma \\
\quad\quad\quad \textsf{then } \delta^r \leftarrow round\text{-}up\ \delta; \\
\quad\quad\quad\quad\quad [\delta_1, \ldots, \delta_{n_i}] \leftarrow \widetilde{\mathcal{E}}_{n_i}[\![e_i]\!]\, \delta^r\, \widetilde{fve}; \\
\quad\quad\quad\quad\quad \delta_l^r \leftarrow round\text{-}up\ \delta_l; \quad (l = 1, \ldots, n_i) \\
\quad\quad\quad\quad\quad return\ [\delta_1^r, \ldots, \delta_{n_i}^r] \\
\quad\quad\quad \textsf{else } return\ [\phi \mid 1 \leq l \leq n_i] \\
\quad \} \\
\quad\quad \textsf{where } p \ = \ \{f_i(x_1, \ldots, x_{n_i}) = e_i; \ldots\}
\end{array}$$

Figure B.3: Abstract non-standard semantics for programs

The definitions for the abstract semantics are presented in Fig. B.3, B.4 and B.5. Again, the term $\widetilde{\mathcal{M}}[\![p]\!]$, i.e., the meaning of program p, will be denoted by $\widetilde{fve}_{[\![p]\!]}$. That meaning is still a function environment; however, such an environment now associates to a function name f_i a function which, given some demand, will produce a demand environment transformer of type $DT\ ADmdVec$, i.e., encapsulating a result which is a demand vector; of course, the demand must be appropriate relative to the demand environment ($\delta \in \textsf{dom}\,\sigma$) otherwise the bottom element is returned. For now, let's ignore the call to *round-up*, i.e., let's simply assume that this is an identity demand transformer which returns both the argument value and the demand environment unchanged; its role will be explained in Section B.5.

Much like the exact non-standard semantics (cf. Figs. 3.20 and 3.21), the meaning of expressions is given by two functions: $\widetilde{\mathcal{E}}_k$ (Fig. B.4), which is the top-level function that deals with any demand, and $\widetilde{\mathcal{E}}_k^+$ (Fig. B.5), which only deals with (non-empty set of) positive abstract demands.

The first step in $\widetilde{\mathcal{E}}_k$ is to check whether the demand δ contains some negative demand: if it does, i.e., if there is a possibility that the result will not be needed, then a new demand not containing any negative demand is created ($\delta^+ \leftarrow make\text{-}positive\,\delta$) and $\widetilde{\mathcal{E}}_k$ is called recursively with this new demand (δ^+). Once the result is obtained, we must now account for the fact that none of bound variables may be needed if the

$$
\begin{array}{l}
\widetilde{\mathcal{E}}_k : Expr \rightarrow ADmdRef \rightarrow \widetilde{Fve} \rightarrow DT\ ADmdVec_k \\[6pt]
\widetilde{\mathcal{E}}_k[\![e]\!]\ \delta\ \widetilde{fve} \quad \triangleq \\
\qquad \sigma\ \leftarrow\ fetch; \\
\qquad \textsf{if}\ possibly\text{-}negative?\ \delta\ \sigma \\
\qquad \textsf{then}\ \delta^{+}\ \leftarrow\ make\text{-}positive\ \delta; \\
\qquad\qquad [\delta_1,\ \ldots,\ \delta_k]\ \leftarrow\ \widetilde{\mathcal{E}}_k[\![e]\!]\ \delta^{+}\ \widetilde{fve}; \\
\qquad\qquad \delta_l^{-}\ \leftarrow\ add\text{-}negative\ \delta_l; \quad (l = 1, \ldots, k) \\
\qquad\qquad return\ [\delta_l^{-} \mid 1 \leq l \leq k] \\
\qquad \textsf{else if}\ bottom?\ \delta\ \sigma \\
\qquad\qquad \textsf{then}\ return\ [\phi \mid 1 \leq j \leq k] \\
\qquad\qquad \textsf{else}\ \widetilde{\mathcal{E}}_k^{+}[\![e]\!]\ \delta\ \widetilde{fve}
\end{array}
$$

Figure B.4: Abstract non-standard semantics for expressions (arbitrary demand)

result was not, so a negative demand is added to each of the demand for the variables ($\delta_l^{-}\ \leftarrow\ add\text{-}negative\ \delta_l$). When the demand for the result is empty ($bottom?\ \delta\ \sigma$), i.e., no result is acceptable, then no bound variables can satisfy that demand, which is why $[\phi \mid 1 \leq j \leq k]$ is returned. Otherwise, a call to $\widetilde{\mathcal{E}}_k^{+}$ is made knowing that δ now effectively represents a non-empty set of positive demands.

The structure of function $\widetilde{\mathcal{E}}_k^{+}$ (Fig. B.5) is very similar to the one for the exact non-standard semantics, although demand environment transformers operations are now used (with the appropriate syntactic sugar). The only major difference is the semantics of a case-expression: as in Section 3.2.5 (Fig. 3.12, p. 60), the demands from all branches are combined together ($\widetilde{\bigcup}\,\{\widetilde{\delta}_l \mid 1 \leq l \leq a_{ij}\}$) in order to account for all possibilities.

B.5 Ensuring termination of the abstract semantics

B.5.1 Creation of cyclic demands through rounding-up

If, in the above semantics functions, the *round-up* operation is ignored, an important problem is encountered: computation of the fixed point solution using the Kleene ascending chain approach can not be done in a finite number of steps as the abstract

$$\widetilde{\mathcal{E}}_k^{+} : Expr \rightarrow ADmdRef \rightarrow \widetilde{Fve} \rightarrow DT\ ADmdVec_k$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\widetilde{\mathcal{E}}_k^{+}[\![e]\!]\,\delta\,\widetilde{fve} \quad \underline{\triangle}} \\
\quad \textsf{cases } [\![e]\!] \textsf{ of} & \\
\quad [\![c]\!] & \rightarrow \sigma \leftarrow fetch; \\
 & \quad \textsf{if } scalar?\ \delta\ \sigma \\
 & \quad \textsf{then } return\ [@ \mid 1 \leq j \leq k] \\
 & \quad \textsf{else } return\ [\phi \mid 1 \leq j \leq k] \\
\quad [\![x_i]\!] & \rightarrow return\ [@ \mid 1 \leq j < i] \curvearrowright [\delta] \curvearrowright [@ \mid i < j \leq k] \\
\quad [\![e_1 + e_2]\!] & \rightarrow \sigma \leftarrow fetch; \\
 & \quad \textsf{if } scalar?\ \delta\ \sigma \\
 & \quad \textsf{then } \widetilde{\delta}_l \leftarrow \widetilde{\mathcal{E}}_k[\![e_l]\!]\,\delta\,\widetilde{fve}; \quad (l = 1, 2) \\
 & \qquad\quad \widetilde{\bigoplus}\,\{\widetilde{\delta}_1, \widetilde{\delta}_2\} \\
 & \quad \textsf{else } return\ [\phi \mid 1 \leq i \leq k] \\
\quad [\![f_i\,(e_1,\ \ldots,\ e_{n_i})]\!] & \rightarrow [\delta_l \mid 1 \leq l \leq n_i] \leftarrow \widetilde{fve}[\![f_i]\!]\,\delta; \\
 & \quad \widetilde{\delta}_l \leftarrow \widetilde{\mathcal{E}}_k[\![e_l]\!]\,\delta_l\,\widetilde{fve}; \quad (l = 1, \ldots, n_i) \\
 & \quad \widetilde{\bigoplus}\,\{\widetilde{\delta}_l \mid 1 \leq l \leq n_i\} \\
\quad [\![K_{ij}\,e_1\ \cdots\ e_{a_{ij}}]\!] & \rightarrow \sigma \leftarrow fetch; \\
 & \quad \textsf{if } has\text{-}cons?\ \delta\ K_{ij}\ \sigma \\
 & \quad \textsf{then } mk\text{-}APDmd(K_{ij}, [\delta_1,\ \ldots,\ \delta_{a_{ij}}]) \leftarrow select\text{-}cons\ \delta\ K_{ij}; \\
 & \qquad\quad \widetilde{\delta}_l \leftarrow \widetilde{\mathcal{E}}_k[\![e_l]\!]\,\delta_l\,\widetilde{fve}; \quad (l = 1, \ldots, a_{ij}) \\
 & \qquad\quad \widetilde{\bigoplus}\,\{\widetilde{\delta}_l \mid 1 \leq l \leq a_{ij}\} \\
 & \quad \textsf{else } return\ [\phi \mid 1 \leq j \leq k] \\
\quad [\![K_{ij}\_l(e)]\!] & \rightarrow \delta' \leftarrow mkComponent_{AD}\ K_{ij}\ l\ \delta; \\
 & \quad \widetilde{\mathcal{E}}_k[\![e]\!]\,\delta'\,\widetilde{fve} \\
\quad [\![\{\texttt{case}\ e_0\ \texttt{of}\ K_{ij} \Rightarrow e_j; \ldots\}]\!] & \rightarrow \delta_l^{v} \leftarrow mkVariant_{AD}\ K_{ij}; \quad (l = 1, \ldots, a_{ij}) \\
 & \quad \widetilde{\delta}_l^{e_0} \leftarrow \widetilde{\mathcal{E}}_k[\![e_0]\!]\,\delta_l^{v}\,\widetilde{fve}; \quad (l = 1, \ldots, a_{ij}) \\
 & \quad \widetilde{\delta}_l^{e} \leftarrow \widetilde{\mathcal{E}}_k[\![e_l]\!]\,\delta\,\widetilde{fve}; \quad (l = 1, \ldots, a_{ij}) \\
 & \quad \widetilde{\delta}_l \leftarrow \widetilde{\bigoplus}\,\{\widetilde{\delta}_l^{e_0}, \widetilde{\delta}_l^{e}\}; \quad (l = 1, \ldots, a_{ij}) \\
 & \quad \widetilde{\bigcup}\,\{\widetilde{\delta}_l \mid 1 \leq l \leq a_{ij}\} \\
\quad \textsf{end} &
\end{array}
$$

Figure B.5: Abstract non-standard semantics for expressions (positive demand) domain is not finite. Furthermore, even though cyclic demands can be represented, there is currently no way to create such demands. Thus, although it is possible that a cyclic demand effectively represents the least fixed point solution, it will not be obtained using an approach based on the Kleene ascending chain.

Let's examine a small example to illustrate this problem. Suppose we are given the following program, which computes the sum of the elements of a list l:

$$p = \{sum(l) = \{\texttt{case}\, l\, \texttt{of}\ Nil \Rightarrow 0; Cons \Rightarrow Cons\_1(l) + sum(Cons\_2(l)); \}; \}$$

Using the Kleene ascending chain, the demand made for the argument of l, under a positive scalar demand, would be represented by the limit of the sequence of approximations presented in Fig. B.6.

$$\begin{array}{l}
d_0 \mapsto \{\,\} \\
d_1 \mapsto \{\mathsf{Nil}\} \\
d_2 \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ d_1\} \\
d_3 \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ d_2\} \\
\ldots \\
d_k \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ d_{k\text{-}1}\} \\
\ldots
\end{array}$$

Figure B.6: Sequence of approximations for the *sum* function

Clearly, in our domain, this limit does not exist, as we require all demand environments to be finite. Furthermore, even if the finiteness requirement was dropped, this would not solve the problem because although the limit would now exist it could not be computed in a finite number of steps, which is required in order for our compiler to always terminate.

In the abstract model of demand propagation presented in the current chapter this problem will be solved using the *round-up* function. [9] First, in order to always ensure termination, *round-up* will always produce a demand which is a member of a *finite subdomain* of the general domain of abstract demands, subdomain whose size will be specified by a globally defined parameter $\mathcal{K}_{max}$. Second, *round-up*, when appropriate to do so, will produce a result which is effectively cyclic. Before examining these two characteristics in more detail, here is the specification of round-up: [10]

Definition B.7 *(Rounding-up of demands)*

$$\textit{round-up} : \textit{ADmdRef} \to \textit{DT ADmdRef}$$

$$\textit{round-up}\ \delta\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

$\mathsf{pre}\ \delta \in \mathsf{dom}\,\sigma$

$\mathsf{post}\ \mathsf{if}\ \textit{size-at-most}\ \delta\ \sigma\ \mathcal{K}_{max}$

$\quad \mathsf{then}\ \delta' = \delta \wedge \sigma' = \sigma$

$\quad \mathsf{else}\ \delta \sqsubseteq_{\sigma'} \delta' \wedge \textit{size-at-most}\ \delta'\ \sigma'\ \mathcal{K}_{max}$

As just mentioned, we assume there is some parameter $\mathcal{K}_{max} \geq 1$ which is globally defined, parameter which determines, much like the precision when doing computer arithmetic, the maximum size that demands are allowed to be. The predicate *size-at-most* indicates, at its name says, whether the demand has a size which is at most $\mathcal{K}_{max}$. Its exact definition is presented in Section B.5.2, p. 223. For example, given the demand environment presented in Fig. 3.22, the size of the various demands would then be the following: $\{\phi \mapsto 0, @ \mapsto 0, \$ \mapsto 1, HTS \mapsto 1, HS \mapsto 2, HS' \mapsto 1\}$; thus, except for HS, all of them would be of *size-at-most* 1.

The exact definition of the size of a demand is not really important; all that matters is that this allows the search for a fixed point to be performed solely within

[9] In Chap. 4, Section 4.4, an alternative strategy will be presented, strategy that will make it possible, in a number of cases, to directly create the required cyclic demands without using *round-up*.

[10] The **post**-condition effectively says that round-up is a *closure* operator. Such operators are similar to projections, although in the upward direction, which is the required direction in our domain for the result to be safe. Remember (cf. Section A.8 p. 202) that a post-condition is used to give an implicit specification of a function, i.e., to indicate the properties that hold of the result without presenting any explicit algorithm for producing that result. Thus, in this case, the result produced by *round-up* is (δ', σ'), i.e., a demand reference paired with a new demand environment, whose properties are described by the post-condition.

a subdomain of our initial domain of demands which is of *finite* size by restricting the search to demands which are of size smaller or equal to $\mathcal{K}_{max}$. This, then, is the first role of *round-up*. [11]

The second role of *round-up* is to create cyclic demands when appropriate to do so. Although this is not immediate in the implicit specification of *round-up* presented above, it is effectively the case that, when the argument demand δ is *too large* relative to $\mathcal{K}_{max}$, then the resulting demand δ', i.e., the one satisfying the post-condition, must necessarily be cyclic for $\delta \sqsubseteq_{\sigma'} \delta'$ to hold. For example, given the sequence of approximations of Fig. B.6 and a maximum size parameter of $\mathcal{K}_{max} = 1$, then at step 2 the resulting demand $(d_2 \mapsto \{Nil, Cons\,\$\,d_1\})$ would become too large and would thus need to be rounded-up: among all the demands of *size-at-most* 1, the only ones that would be safe would be cyclic ones, the smallest among them being HTS, i.e.:

$HTS \mapsto \{Nil, Cons\ \$\ HTS\}$

The explicit/constructive definition of *round-up* will be omitted, as the implicit specification presented above will be sufficient for our purpose. A more detailed explanation of the rounding strategy used in the prototype compiler can be found in Section 4.3.

B.5.2 Maximum size of a demand

The following function determines whether a demand reference δ relative to a demand environment σ is of a size which is at most some specific value. Note that, in some cases, the size of a cyclic demand in the prototype compiler is different, i.e., it is one less than what would be inferred by the *size-at-most* function below; this difference is due to a slightly different representation used for abstract demands. [12]

To implement this function, memoizing is used explicitly (using *seen*) in order to prevent infinite loop when checking the size of cyclic demands. The following two

[11] Note that this does not mean all demands manipulated by the semantics functions are of size smaller or equal to $\mathcal{K}_{max}$: only those involved in the fixed point computation need to be. An interesting analogy, examined further in Section 4.3 p. 123, is with computer arithmetic: in order to obtain a precision of $\mathcal{K}_{max}$ digits, it is necessary that, in certain situations, *more than* $\mathcal{K}_{max}$ digits be used.

[12] In the prototype compiler, the set of demands whose size is less or equal to 1 makes it possible to represent any demand which can be represented using the projections of Hughes and Wadler [HW87].

functions are also assumed to be defined, as checking the size of a demand must be done based on its associated type:

- *type-of*(δ, σ): returns the parent type of demand δ, as determined by its associated value in environment σ.
- *type-of-component*: returns the parent type of the i^{th} slot of the indicated constructor (as determined by the original type definition).

Function *size-at-most* is the top-level function called for a newly encountered type of component. It calls function *size-at-most**, which is the memoized version of it (parameter *seen*) and which searches for demand references of the same type as the top-level one, in order to really bound the size of demands.

$size\text{-}at\text{-}most : ADmdRef \rightarrow ADmdEnv \rightarrow \mathbb{N} \rightarrow \mathbb{B}$

$size\text{-}at\text{-}most\ \delta\ \sigma\ k_{max} \quad \triangleq$

$\quad \mathsf{if}\ \delta \in \{\phi, @\}$

$\quad \mathsf{then\ true}$

$\quad \mathsf{else}\ size\text{-}at\text{-}most^*\ \delta\ \sigma\ \{\,\}\ type\text{-}of(\delta, \sigma)\ k_{max}\ k_{max}$

$\mathsf{pre}\ k_{max} \geq 1$

$size\text{-}at\text{-}most^*\ \delta\ \sigma\ seen\ type\ k\ k_{max} \quad \triangleq$

$\quad \mathsf{if}\ k = 0$

$\quad \mathsf{then}\ \delta \in seen$

$\quad \mathsf{else\ if}\ \delta \in seen$

$\quad\quad \mathsf{then\ true}$

$\quad\quad \mathsf{else\ let}\ seen' = seen \cup \{\delta\}\ \mathsf{in}$

$\quad\quad\quad \forall mk\text{-}APDmd(K, comps) \in \sigma(\delta) \cdot$

$\quad\quad\quad\quad \forall i \in \mathsf{inds}\ comps \cdot$

$\quad\quad\quad\quad\quad \mathsf{if}\ type = type\text{-}of\text{-}component(K, i)$

$\quad\quad\quad\quad\quad \mathsf{then}\ size\text{-}at\text{-}most^*\ comps(i)\ \sigma\ seen'\ type\ (k-1)\ k_{max}$

$\quad\quad\quad\quad\quad \mathsf{else}\ size\text{-}at\text{-}most\ comps(i)\ \sigma\ k_{max}$

B.6 Safety and concretization sets

This section, using the notion of safe approximation introduced earlier ($\lhd$: cf. Defn. 3.40, p. 89), formalizes the idea that a safe approximation really means that *the abstract result*, when viewed as a set of exact demands, *contains the exact one*. In order to do this, a concretization function *conc* which, given an abstract demand, returns a *set of* exact demands is defined. Essentially *conc*, given some environment σ, takes the union of the concretization sets obtained for each the basic demands associated with δ in σ using $conc'$, i.e., $\bigcup_{bd \in \sigma(\delta)} conc'\ bd\ \sigma$; of course, the demand reference must be well-defined relative the environment, which is indicated by the pre-condition ($\delta \in \mathsf{dom}\, \sigma$).

As for the concretization set associated with a basic demand bd ($conc'\ bd\ \sigma$), it is either the singleton set containing only a negative demand if bd is a negative abstract demand or else it is obtained by creating appropriate exact demands ($\{mk\text{-}EPDmd(K, [d_1 \cdots d_k]) \mid \ldots\}$) composed from the various demands associated with the concretization of the components of bd ($d_j \in conc\ \delta_j\ \sigma$):

Definition B.8 *(Concretization function for abstract demands)*

$$conc : ADmdRef \rightarrow ADmdEnv \rightarrow \mathcal{P}(EDmd)$$

$$conc\ \delta\ \sigma \quad \triangleq$$
$$\bigcup_{bd \in \sigma(\delta)} conc'\ bd\ \sigma$$

$$\mathsf{pre}\ \delta \in \mathsf{dom}\, \sigma$$

$$conc' : ABDmd \rightarrow ADmdEnv \rightarrow \mathcal{P}(EDmd)$$

$$conc'\ bd\ \sigma \quad \triangleq$$
$$\mathsf{if}\ bd = @$$
$$\mathsf{then}\ \{@\}$$
$$\mathsf{else\ let}\ mk\text{-}APDmd(K, [\delta_1, \ldots, \delta_k]) = bd\ \ \mathsf{in}$$
$$\{mk\text{-}EPDmd(K, [d_1, \ldots, d_k]) \mid d_1 \in conc\ \delta_1\ \sigma; \ldots; d_k \in conc\ \delta_k\ \sigma\}$$

Let's illustrate the unfolding of the concretization function (which is recursively defined) using a simple example. Suppose we have the demand environment σ presented in Fig. 3.22, i.e., which associates HTS with the following demand:

$$HTS \mapsto \{\mathsf{Nil}, \mathsf{Cons}\ \$\ HTS\}$$

Then the following sequence of approximations would be obtained for the concretization set associated with demand HTS:

$$
\begin{aligned}
&conc_0\ HTS\ \sigma = \{\ \} \\
&conc_1\ HTS\ \sigma = \{Nil\} \\
&conc_2\ HTS\ \sigma = \{Nil, Cons\ \$\ Nil\} \\
&conc_3\ HTS\ \sigma = \{Nil, Cons\ \$\ Nil, Cons\ \$\ (Cons\ \$\ Nil)\} \\
&\quad \ldots \\
&conc_k\ \mathtt{HTS}\ \sigma = \{\$^j\ .\ Nil \mid 0 \leq j < k\} \\
&\quad \ldots
\end{aligned}
$$

The limit of this sequence would then associate to HTS the set $\{\$^j\ .\ Nil \mid j \geq 0\}$, i.e., the set containing all demands for lists of finite length whose heads are positively demanded.

Based on the concretization function, the following lemma formalizing the notion of *safely approximates* can now be stated, i.e., δ safely approximates d means that d is in the concretization set of δ:

Lemma B.1

$$\delta \in \mathsf{dom}\,\sigma \wedge d \lhd_\sigma \delta \ \Rightarrow\ d \in conc\ \delta\ \sigma$$

Proof: The proof is by an induction on the structure of d. The details are presented in App. H.2.2, p. 307. □

Appendix C

Demand environment transformers

This appendix briefly describes the notion of *demand environment transformers* using the approach presented in [Lau93], i.e., the monads approach. Note that this is not a general presentation of the concept of monads but only a short presentation sufficient to understand the various definitions presented in App. B. For a more detailed presentation, cf. [Wad92] or [Lau93].

C.1 Motivating example

Let's first present a simple example illustrating the complexity introduced by the need to manipulate a global state, in this case, the abstract demands environment. Suppose we have an operation *op* which, given two demands, produces a new demand. When *op* does not deal with an environment, it can simply be defined as follows:

$op : ADmd \rightarrow ADmd \rightarrow ADmd$

$op\ \delta_1\ \delta_2 \quad \triangleq \quad \delta'$

post (* Some appropriate specification relating δ_1, δ_2 and δ' *)

On the other hand, when the operation uses and creates a modified demand environment, the definition has to be modified as follows:

$$op : ADmdRef \rightarrow ADmdRef \rightarrow ADmdEnv \rightarrow ADmdRef \times ADmdEnv$$

$$op\ \delta_1\ \delta_2\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

post (∗ Some appropriate specification relating $\delta_1, \delta_2, \sigma, \delta'$ and σ' ∗)

Suppose we now want to define an operation op_3 that combines 3 demands using operation op. Without the need to carry the demand environment around, this operation could simply be expressed as follows: [1]

$$op_3 : ADmd \rightarrow ADmd \rightarrow ADmd \rightarrow ADmd$$

$$\begin{aligned} & op_3\ \delta_1\ \delta_2\ \delta_3 \quad \triangleq \\ & \quad \mathsf{let}\ \delta' = op\ \delta_1\ \delta_2\ \ \mathsf{in} \\ & \quad \mathsf{let}\ \delta'' = op\ \delta'\ \delta_3\ \ \mathsf{in} \\ & \quad \delta'' \end{aligned}$$

However, when op might modify the demand environment, then a presentation using an explicit environment would look as follows, because the demand environment would have to be passed around and returned as part of the result:

$$\begin{aligned} & op_3 : ADmdRef \rightarrow ADmdRef \rightarrow ADmdRef \rightarrow ADmdEnv \\ & \quad \rightarrow ADmdRef \times ADmdEnv \end{aligned}$$

$$\begin{aligned} & op_3\ \delta_1\ \delta_2\ \delta_3\ \sigma \quad \triangleq \\ & \quad \mathsf{let}\ (\delta', \sigma') = op\ \delta_1\ \delta_2\ \sigma\ \ \mathsf{in} \\ & \quad \mathsf{let}\ (\delta'', \sigma'') = op\ \delta'\ \delta_3\ \sigma'\ \ \mathsf{in} \\ & \quad (\delta'', \sigma'') \end{aligned}$$

Clearly, this makes the definitions more complex to read and understand. In the next section, we will see how the monad of demand environment transformers, together with some syntactic sugar, can greatly simplify these definitions.

C.2 Demand environment transformers operations and syntactic sugar

The basic notion is that of Demand environment Transformer: this is a function which, given a demand environment, returns some value paired up with a (possibly)

[1] Of course, even simpler would be $op_3\ \delta_1\ \delta_2\ \delta_3 \triangleq op(op(\delta_1, \delta_2), \delta_3)$!

modified demand environment. [2]

$$DT\ t = ADmdEnv \rightarrow t \times ADmdEnv$$

where

$$inv\text{-}(DT\ t)\,(dt) \quad \triangleq \quad \forall \sigma \cdot dt\ \sigma = (v, \sigma') \ \Rightarrow \ \sigma \sqsubseteq \sigma'$$

There are a number of useful operations associated with demand environment transformers which we now present. The first function, *return*, given some value v, simply returns the value v paired with the unchanged demand environment:

$$return : t \rightarrow DT\ t$$

$$return\ v\ \sigma \quad \triangleq \quad (v, \sigma)$$

Function *fetch* makes it possible to examine the state of the current demand environment, as it simply returns the current environment paired with itself, i.e., unchanged:

$$fetch\ : DT\ ADmdEnv$$

$$fetch\ \sigma \quad \triangleq \quad (\sigma, \sigma)$$

A more interesting operation is the one that combines/composes demand transformers. Its motivation stems from the fact that, as we saw above, when combining two operations that may modify the environment, these operations must be composed so that the second operation uses the new environment produced by the first operation. Here is the binding construct, which really is a form of syntactic sugar used to create a new demand environment transformer from existing ones (e and r): [3]

$$\text{“}\_ \leftarrow \_; \_\text{”} : DT\ t_1 \rightarrow (t_1 \rightarrow DT\ t_2) \rightarrow DT\ t_2$$

$$\text{“}x \leftarrow e; r\text{”} \quad \triangleq \quad \lambda \sigma_0 \cdot \mathsf{let}\ (x, \sigma_1) = e\ \sigma_0\ \mathsf{in}\ \text{“}r\text{”}\ \sigma_1$$

[2]The invariant simply indicates that the demands part of the initial environment are left unchanged. For more detail, cf. Defn. B.5, p. 209.

[3]“e” represents the removal of the syntactic sugar in order to obtain a regular expression.

Given a demand environment transformer e, we apply it to environment σ_0 to produce some value, which we bind to x, and some new demand environment σ_1, which is then passed to the demand environment transformer "r".

The syntactic sugar also has to be extended to the other language constructs. In the case of an if-construct, we suppose that the predicate does not observe or modify the environment, although the branches can do so. [4]

$$\text{“}\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\text{”} \quad \triangleq \quad \lambda\sigma_0 \cdot \mathsf{if}\ e_0\ \ \mathsf{then}\ \text{“}e_1\text{”}\ \sigma_0\ \ \mathsf{else}\ \text{“}e_2\text{”}\ \sigma_0$$

Similarly, we also suppose that a regular let-binding does not modify the demand environment, although its body can:

$$\text{“}\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\text{”} \quad \triangleq \quad \lambda\sigma_0 \cdot \mathsf{let}\ x = e_0\ \mathsf{in}\ \text{“}e_1\text{”}\ \sigma_0$$

Finally, other expressions simply map into themselves without further modification.

$$\text{“}e\text{”} \quad \triangleq \quad e$$

C.3 The example revisited

Using demand environment transformers and the syntactic sugar introduced above our simple example can now be expressed as follows:

$$op : ADmdRef \rightarrow ADmdRef \rightarrow DT\ ADmdRef$$

$$op\ \delta_1\ \delta_2\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

post (∗ Some appropriate specification relating $\delta_1, \delta_2, \sigma, \delta'$ and σ' ∗)

$$op_3 : ADmdRef \rightarrow ADmdRef \rightarrow ADmdRef \rightarrow DT\ ADmdRef$$

$$\begin{aligned}
&op_3\ \delta_1\ \delta_2\ \delta_3 \quad \triangleq \\
&\quad \delta' \ \leftarrow\ op\ \delta_1\ \delta_2; \\
&\quad \delta'' \ \leftarrow\ op\ \delta'\ \delta_3; \\
&\quad return\ \delta''
\end{aligned}$$

[4] Although not explicitly mentioned, an analogous transformation would apply for case-expressions; again, we would assume that no environment transformation is performed within the predicate.

Et voila!

C.4 Additional operations for handling cyclic demands

Two additional operations will be required in order to deal with cyclic demands. The first one of these is used to generate a new demand reference whose value is still unbound (i.e., bound to nil) but can later be bound, possibly but not necessarily, to create an explicit cycle:

$$\mathit{get\text{-}new\text{-}ref} : DT\ ADmdRef$$

$$\mathit{get\text{-}new\text{-}ref}\ \sigma \quad \triangleq \quad (\delta, \sigma')$$

$$\mathsf{post}\ \delta \notin \mathsf{dom}\,\sigma \wedge \sigma' = \sigma \cup \{\delta \mapsto \mathsf{Nil}\}$$

The second operation is used to associate a value (i.e., a set of basic demands) to a demand reference. Combined with the first operation, this makes it possible to create explicitly cyclic demands. Note that the post-condition is there to ensure the uniqueness of demand references, i.e., preserve the invariant which says that no demand equivalent to an existing one must exist (cf. Section B.1.1, p. 206). In words, the post-condition thus says that if some equivalent demand δ' already exists, then it is returned (paired with the old environment); otherwise, the newly created demand is returned (paired with the modified environment).

$$\mathit{bind\text{-}ref} : ADmdRef \rightarrow ADmd \rightarrow DT\ ADmdRef$$

$$\mathit{bind\text{-}ref}\ \delta\ ad\ \sigma \quad \triangleq \quad (\delta', \sigma')$$

$$\mathsf{pre}\ \delta \in \mathsf{dom}\,\sigma \wedge \sigma(\delta) = \mathsf{Nil}$$

$$\begin{array}{l}\mathsf{post}\ \mathsf{let}\ \sigma' = \sigma \cup \{\delta \mapsto ad\}\ \mathsf{in} \\ \quad \mathsf{if}\ \exists \delta' \in \mathsf{dom}\,\sigma \cdot \delta' \neq \delta \wedge \delta \sqsubseteq_{\sigma'} \delta' \wedge \delta' \sqsubseteq_{\sigma'} \delta \\ \quad \mathsf{then}\ \mathsf{let}\ \delta' = \iota\, \delta' \in \mathsf{dom}\,\sigma \cdot \delta' \neq \delta \wedge \delta \sqsubseteq_{\sigma'} \delta' \wedge \delta' \sqsubseteq_{\sigma'} \delta\ \mathsf{in} \\ \quad\quad (\delta', \sigma) \\ \quad \mathsf{else}\ (\delta, \sigma')\end{array}$$

Appendix D

Formal specification of the domain of demands used by the prototype compiler

A formal specification of the domain of demands used within the prototype compiler is presented in Fig. D.1 and is explained in the following paragraphs.

First, note that some of the elements, i.e., those not dealing with abstract conditions, are quite similar to those presented in Fig. B.1 (p. 206) except that there is no explicit notion of negative demand; instead, *Certainty-indicator*s are used, as described in Section 4.3, p. 114. More precisely, both domains use an explicit map to associate some form of demand reference with an appropriate value, which is a set of positive demands.

Positive demands are used to represent demands for arbitrary user-defined algebraic types, together with scalar demands. Here are some examples:

- A demand for a scalar value: $\mathit{mk\text{-}Positive\text{-}demand}(\mathsf{Scalar}, [\,])$;
- A demand for an empty list: $\mathit{mk\text{-}Positive\text{-}demand}(\mathsf{Nil}, [\,])$
- A demand for the head of a cons-cell, with no need for the tail:
  $\mathit{mk\text{-}Positive\text{-}demand}(\mathsf{Cons}, [$
  $\quad\mathit{mk\text{-}Delayed\text{-}demand}(\mathsf{Certain}, \$),$
  $\quad\mathit{mk\text{-}Delayed\text{-}demand}(\mathsf{Uncertain}, sd_{\perp})])$

The last example shows how negative demands are implicitly represented: in that example, $sd_{\perp}$ represents a strict demand reference which is associated with

$$Certainty\text{-}indicator = \{\mathsf{Certain}, \mathsf{Uncertain}\}$$

$$\begin{aligned} Delayed\text{-}demand :: \; & certainty\text{-}indicator : Certainty\text{-}indicator \\ & strict\text{-}demand \quad\; : Strict\text{-}demand\text{-}ref \end{aligned}$$

$$\begin{aligned} Positive\text{-}demand :: \; & constructor \; : Cons\text{-}name \\ & components : Delayed\text{-}demand^{*} \end{aligned}$$

$$Strict\text{-}demand\text{-}ref = Token$$

$$Strict\text{-}demand = Positive\text{-}demand\text{-}\mathsf{set}$$

$$Strict\text{-}demand\text{-}map = Strict\text{-}demand\text{-}ref \xrightarrow{m} [Strict\text{-}demand]$$

$$Unknown\text{-}condition = Token$$

$$\begin{aligned} Predicate\text{-}condition :: \; & predicate \quad\; : Instruction \\ & constructor : Cons\text{-}name \end{aligned}$$

$$\begin{aligned} Apply\text{-}condition :: \; & apply\text{-}node : Instruction \\ & condition \quad : Predicate\text{-}condition \end{aligned}$$

$$Abstract\text{-}condition = Unknown\text{-}condition + Predicate\text{-}condition + Apply\text{-}condition$$

$$\begin{aligned} Conditional\text{-}demand :: \; & conditions \quad\;\; : Abstract\text{-}condition\text{-}\mathsf{set} \\ & strict\text{-}demand : Strict\text{-}demand\text{-}ref \end{aligned}$$

$$Demands = Conditional\text{-}demand\text{-}\mathsf{set}$$

where

$$inv\text{-}Demands(ds) \;\; \triangleq \;\; \forall d_1, d_2 \in ds \cdot conditions(d_1) \neq conditions(d_2)$$

$$Approximate\text{-}demands = Demands$$

where

$$\begin{aligned} & inv\text{-}Approximate\text{-}demands(ds) \;\; \triangleq \\ & \quad ds = \{mk\text{-}Conditional\text{-}demand(conds, sd)\} \wedge \\ & \quad conds = \{\,\} \vee (conds = \{ac\} \wedge is\text{-}Unknown\text{-}condition(ac)) \end{aligned}$$

Figure D.1: Formal specification of the domains used to define demands an empty set of positive demands, i.e., which represents the bottom element of *Strict-demands*; the presence of the Uncertain indicator in the delayed demand of which it is part then indicates that @ is among the possibilities; thus, an uncertain demand means that the object may be not needed, i.e., a negative demand is also part of the set of associated demands.

The use of logical variable style for computing fixed point is modelized using references: [1] a *Strict-demand-ref* is essentially a variable identifier whose binding — a value of type *Strict-demand* — is obtained through an appropriate mapping of type *Strict-demand-map*; an unbound *Strict-demand-ref* simply maps into nil. Such demands are also used to represent cyclic demands. For example, the head-and-tail strict demand for linear lists ($\mathsf{Rec}\, x :: \{\mathsf{Nil}, \mathsf{Cons}\,\$\, x\}$) could be represented as follows, where *sd*'s value effectively refers to *sd* itself:

$$
\begin{aligned}
&sd \mapsto \{\\
&\quad \textit{mk-Positive-demand}(\mathsf{Nil}, [\,]),\\
&\quad \textit{mk-Positive-demand}(\mathsf{Cons}, [\\
&\qquad \textit{mk-Delayed-demand}(\mathsf{Certain}, \$),\\
&\qquad \textit{mk-Delayed-demand}(\mathsf{Certain}, sd)])\}
\end{aligned}
$$

Similarly, a head-strict demand ($\mathsf{Rec}\, x :: \{\mathsf{Nil}, \mathsf{Cons}\,\$\ \hat{}x\}$) could be represented as follows, using a single demand reference instead of 2 as in the domain of Chap. 3:

$$
\begin{aligned}
&sd \mapsto \{\\
&\quad \textit{mk-Positive-demand}(\mathsf{Nil}, [\,]),\\
&\quad \textit{mk-Positive-demand}(\mathsf{Cons}, [\\
&\qquad \textit{mk-Delayed-demand}(\mathsf{Certain}, \$),\\
&\qquad \textit{mk-Delayed-demand}(\mathsf{Uncertain}, sd)])\}
\end{aligned}
$$

Predicate-conditions are used to represent conditions governing the execution of basic blocks associated with if and case branches: the predicate instruction represents the α abstract condition which has been used in the examples of Section 4.3; as for the constructor name, it indicates which branch has effectively been selected. *Apply-conditions* are associated with the tagging of abstract condition that must be done when inter-procedural conditions are allowed, as described in Section 4.3, thus the need to keep track of the apply instruction that gave rise to the associated call.

[1] Cf. Section 4.4.

The demands which are used to annotate program graphs, as mentioned in Section 4.3, are sets of conditional demands. Such demands are described by the type *Demands*. Note the invariant associated with that type: each set of conditions appears at most once within the set. Finally, *Approximate-demands* are used for recursive definitions and when inter-procedural abstract conditions are not allowed; they are simply *Demands* with a cardinality of 1 where the associated abstract conditions is either true (i.e., $= \{ \}$) or a unique unknown condition.

Appendix E

Examples of strictness analysis results

This appendix presents a number of examples showing the kinds of results that can be obtained by the strictness analysis process implemented by the prototype compiler, using the domain of (conditional) demands presented in App. D.

List-manipulating functions

Figure E.1 presents a number of Id function definitions that manipulate lists. The `list` type is defined by the following algebraic type:

```
Type list = Nil | Cons N list;
```

Assuming the indicated initial demand (between parentheses), the strictness information that would be inferred for the arguments of these various functions is presented in Fig. E.2. In words, these results indicate the following:

- `len`: The spine of the list may be evaluated and none of the elements are needed (tail-strictness).
- `last`: The spine of the list may be evaluated but all component elements must be delayed, as the last element will be needed but none of the others will (another form of tail-strictness).
- `before0`: This function, which returns the elements of `l` which appear before the first 0, is head-strict even when its result is fully needed, as the tail of `l` following the first 0 will not be needed.

```
Def len l =
   { Case l Of
      Nil = 0
   |  Cons x xs = 1 + len xs };

Def last l =
   { Case l Of
      Nil = error "Last of empty list"
   |  Cons x xs = { Case xs Of Nil = x | Cons y ys = last xs } };

Def before0 l =
   { Case l Of
      Nil = Nil
   |  Cons x xs = If x == 0 Then Nil Else Cons x (before0 xs) };

Def double l =
   { Case l Of
      Nil = Nil
   |  Cons x xs = Cons (2 * x) (double xs) };

Def append l1 l2 =
   { Case l1 Of
      Nil = l2
   |  Cons x xs = Cons x (append xs l2) };
Def reverse l =
   { Case l Of
      Nil = Nil
   |  Cons x xs = append (reverse xs) (Cons x Nil) };
```

Figure E.1: Examples of function definitions dealing with lists `len` (\$):

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ x\})\}$

`last` (\$):

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})\}$

`before0` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$

`before0` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$

`double` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$

`double` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})\}$

`append` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$:

`l1` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$

`l2` $\Rightarrow$ $\{? \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$

`append` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$:

`l1` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})\}$

`l2` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})\}$

`reverse` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})\}$

`reverse` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})\}$

`reverse_tr` $(\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$:

`l` $\Rightarrow$ $\{T \rightarrow (\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})\}$

Figure E.2: Results of strictness analysis under various demands for list-manipulating functions

- `double`: The strictness of this function, which multiplies by 2 each of its elements, is context-dependent, i.e., head-strict in a head-strict context vs. head-and-tail strict in a head-and-tail strict context.
- `append`: The strictness of this function is context-dependent, much like double. However, when the initial demand is only head-strict, then the second argument is not strict (the associated condition is unknown, i.e., ?).
- `reverse`: The strictness of this function is also context dependent, although it is always at least inferred to be tail-strict, because obtaining the first element of the result requires getting the last element of `l`, thus requiring the unfolding of its spine.

Tree-manipulating functions

Suppose we have the the following algebraic type for binary trees with information kept both at the leaves and at the nodes:

```
Type tree = Leaf N | Node tree N tree;
```

Figure E.3 presents the Id code for a number of functions that manipulate such trees. The strictness results that would be computed by the compiler are presented in Fig. E.4. In words, these results can be explained as follows:

- `nbLeaves`: Only the spine of the tree can be evaluated for sure, as the tree is simply traversed to identify the leaves (tail-strict).
- `sumLeaves`: The spine of the tree and all the leaves can be evaluated, but not the information kept at the nodes.
- `sumTree`: The tree can be fully evaluated (head-and-tail strict).
- `has0Tree`: Similar to function `before0` for lists, i.e., head-strict.
- `leftMostElement`: The right sub-trees are never needed and a leaf, when finally encountered, can effectively be evaluated right away.

Examples illustrating the influence of the size of the domain of demands

The following is a small example where the demand inferred would depend on the size of the domain of demands (option `finite-demands-domain-size`). Function

```
Def nbLeaves t =
   { Case t Of
      Leaf x = 1
   |  Node t1 x t2 = (nbLeaves t1) + (nbLeaves t2) };
Def sumLeaves t =
   { Case t Of
      Leaf x = x
   |  Node t1 x t2 = (sumLeaves t1) + (sumLeaves t2) };
Def sumTree t =
   { Case t Of
      Leaf x = x
   |  Node t1 x t2 = x + (sumTree t1) + (sumTree t2) };
Def has0Tree t =
   { Case t Of
      Leaf x =
         x == 0
   |  Node t1 x t2 =
         If x == 0 Then
            True
         Else
            If (has0Tree t1) Then True Else (has0Tree t2) };
Def leftMostElement t =
   { Case t Of
      Leaf x = x
   |  Node t1 x t2 = leftMostElement t1 };
```

Figure E.3: Examples of function definitions dealing with trees

`nbLeaves` ($):
  `t` $\Rightarrow$ $(\mathsf{Rec}\ x{::}\ \{\mathsf{Leaf}\ @,\ \mathsf{Node}\ x\ @\ x\})$

`sumLeaves` ($):
  `t` $\Rightarrow$ $(\mathsf{Rec}\ x{::}\ \{\mathsf{Leaf}\ \$,\ \mathsf{Node}\ x\ @\ x\})$

`sumTree` ($):
  `t` $\Rightarrow$ $(\mathsf{Rec}\ x{::}\ \{\mathsf{Leaf}\ \$,\ \mathsf{Node}\ x\ \$\ x\})$

`has0Tree` ($):
  `t` $\Rightarrow$ $(\mathsf{Rec}\ x{::}\ \{\mathsf{Leaf}\ \$,\ \mathsf{Node}\ \hat{}x\ \$\ \hat{}x\})$

`leftMostElement` ($):
  `t` $\Rightarrow$ $(\mathsf{Rec}\ x{::}\ \{\mathsf{Leaf}\ \$,\ \mathsf{Node}\ x\ @\ @\})$

Figure E.4: Results of strictness analysis under various demands for tree-manipulating functions `tl` retrieves the tail of its argument `l`, whereas `f` computes the sum of the tail of its argument:

```
Def tl l =
   { Case l Of
      Nil = Nil
   |  Cons x xs = xs };
Def f l =
   sum (tl l);
```

When the size of domain is greater than 1, it becomes possible to infer that all heads and all tails *of the tail of* `l` will effectively be needed (cf. below). As was seen in Section F.1, the role of the size of the domain of demands can be an important factor in obtaining precise strictness analysis results.

`f ($) (size = 1):`

`l` $\Rightarrow$ $(\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})$

`f ($) (size = 2):`

`l` $\Rightarrow$ $\{\mathsf{Nil}, \mathsf{Cons}\ @\ (\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\})\}$

Below is another example where increasing the size of the domain would lead to better results. Function `sumEvenElements` computes the sum of the elements of list `l` which are at even position (assuming the first element to be at position 1):

```
Def sumEvenElements l =
   { Case l Of
      Nil = 0
   |  Cons x1 xs1 =
         { Case xs1 Of
            Nil = 0
         |  Cons x2 xs2 = x2 + sumEvenElements xs2 }} ;
```

Depending on the size of the domain, the strictness results presented below would be obtained. When a domain of size 1 is used, all elements must be delayed; on the other hand, when a domain of size 2 is used, then a more precise cyclic demand can be created, indicating that the elements must alternatively be needed and not needed.

`sumEvenElements ($) (size = 1):`

`l` $\Rightarrow$ $(\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ x\})$

`sumEvenElements ($) (size = 2):`

`l` $\Rightarrow$ ($\mathsf{Rec}\ x{::}\ \{\mathsf{Nil}, \mathsf{Cons}\ @\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ x\}\}$)

Looping/non-terminating expressions

Our compiler, as described in Section 3.2.7, can infer some cases of non-terminating programs. For example, given the following definition:

```
Def f x y =
   If x == 0 Then f (x-1) y Else f (x+1) y;
```

the compiler can effectively signal that the following function is sure not to terminate, as it would infer the following demands:

`f ($):`

`x` $\Rightarrow$ $\{T \rightarrow \perp\}$

`y` $\Rightarrow$ $\{T \rightarrow \perp\}$

Below is a similar example which uses the fact that the bottom demand is associated with a looping expression:

```
Def loop_if_nil l =
   { Case l Of Nil = loop_if_nil l | Cons x xs = 0 };
```

When given a `Nil` value, this function gets into an infinite loop. Thus, the only way it can produce a definite result is by being fed a cons-cell, that is:

`loop_if_nil ($):`

`l` $\Rightarrow$ $\{T \rightarrow \mathsf{Cons}\ @\ \ @\}$

Small examples involving abstract conditions

When allowing intra-procedural abstract conditions, the following function would effectively be recognized as being strict in all its arguments:

```
Def f x y z =
   (If x==0 Then y Else z) + (If x==0 Then z else y);
```

That is, the following strictness result would be produced:

`f ($):`

x $\Rightarrow$ $\{T \rightarrow \$\}$

y $\Rightarrow$ $\{T \rightarrow \$\}$

z $\Rightarrow$ $\{T \rightarrow \$\}$

As another example, suppose we have the functions below:

```
Def f x y z = If x Then y Else z;
Def g x y = f x y y;
```

Assuming inter-procedural abstract conditions are allowed, then the following strictness results would be obtained:

f (\$):

x $\Rightarrow$ $\{T \rightarrow \$\}$

y $\Rightarrow$ $\{\alpha_{True} \rightarrow \$\}$

z $\Rightarrow$ $\{\alpha_{False} \rightarrow \$\}$

g (\$):

x $\Rightarrow$ $\{T \rightarrow \$\}$

y $\Rightarrow$ $\{T \rightarrow \$\}$

Appendix F

Factors influencing the precision of the strictness analysis process

This appendix examines in detail the major factors/parameters that influence the precision of the strictness analysis process and, thus, the efficiency of the resulting programs. Each of the major factor is briefly introduced and some statistics showing its effect on the benchmarks' execution are presented.

All our benchmarks are executed, after having been compiled with various setting of the compile-time options, under an *idealized execution model*, i.e., assuming an infinite number of processors with no communication latency. There are three key parameters that we measure and analyze under this model:

- the length of the critical path, i.e., the minimum number of time steps required to execute the program;
- the total number of instructions executed;
- the average parallelism, i.e., the average number of instructions executed per time step (= total number of instructions / critical path length).

We present our experimental data relative to the critical path length and the total number of instructions using histograms. Each bucket of an histogram cumulates the number of benchmark programs for which the use of a given option instead on another option lead to a certain reduction of the analyzed statistics. In most cases, we will simply present the data relative to the critical path length, as the data for the total number of instructions usually result in the same histogram. In a few cases where the resulting histograms are sufficiently different, both will be presented.

Before delving into the details, here is an overview of the results to come:

- Increasing the size of the domain of demands has an important effect when going from 1 to 2; however, further increases bring less important improvements.
- Intra-procedural abstract conditions seem to be sufficient, i.e., the more complex inter-procedural abstract conditions do not bring very important improvements.
- A small number of distinct versions seems to be sufficient in most cases. However, because limiting the number of versions increases the cost of compilation, it is preferable to impose no such limit.
- Run-time evaluators have a negligible impact on the reduction of the critical path length and simply increase the total number of instructions executed.
- Using an initial head-strict demand instead of a simple strict one has an important impact on the efficiency of the resulting program.
- Head-strictness in functions, when combined with an initial head-strict demand, also has an important positive impact.
- When strictness analysis is performed, direct implementation of data structure thunks is preferable over indirect implementation.

F.1 Influence of the size of the domain of demands

A first factor influencing the precision of the strictness analysis process is the size of the domain of demands. As the domain gets larger, the precision of the analysis should be improved, because the domain contains more elements and thus results in fewer approximations. In other words, when the domain is small, many abstract elements get collapsed into a single element, which represents the least common denominator of all of them, i.e., their least upper bound, thus losing information.

Figures F.1, F.2, and F.3 present histograms showing the effect of increasing the size of the domain of demands on the reduction of the critical path length. Each histogram indicates the number of cases (benchmark programs), when going from size k to size k', for which the reduction of the critical path length had either no effect, or was reduced by the indicated percentage. The most radical effect occurs when going from size 1 to size 2. When going from size 2 to 3, or from size 3 to 5, there are fewer cases where this leads to a reduction of the critical path length.

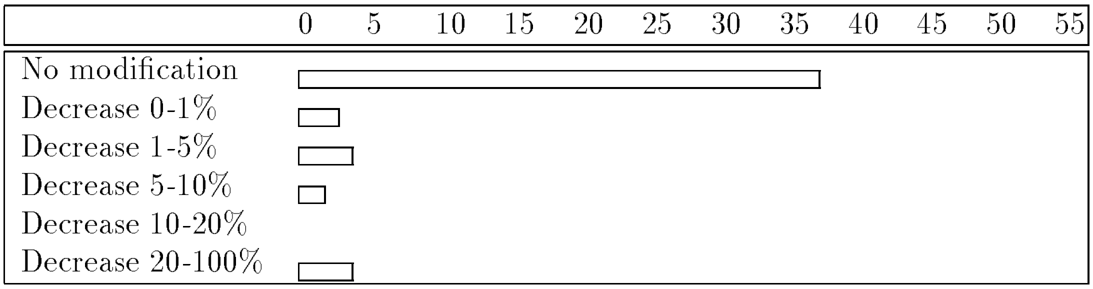


Figure F.1: Size 1 vs. 2: Effect on critical path length

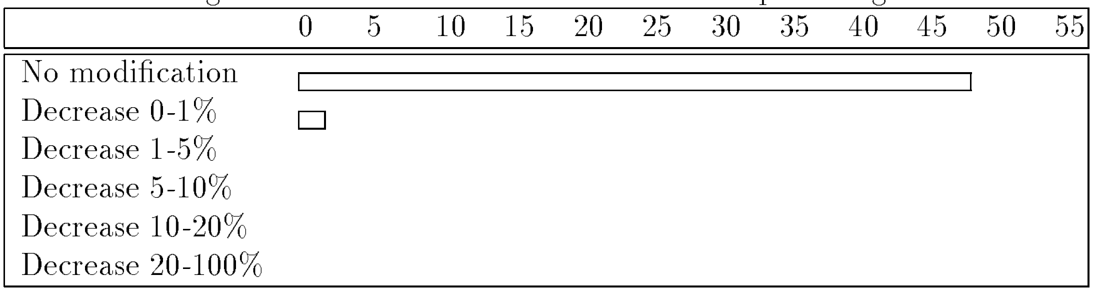


Figure F.2: Size 2 vs. 3: Effect on critical path length

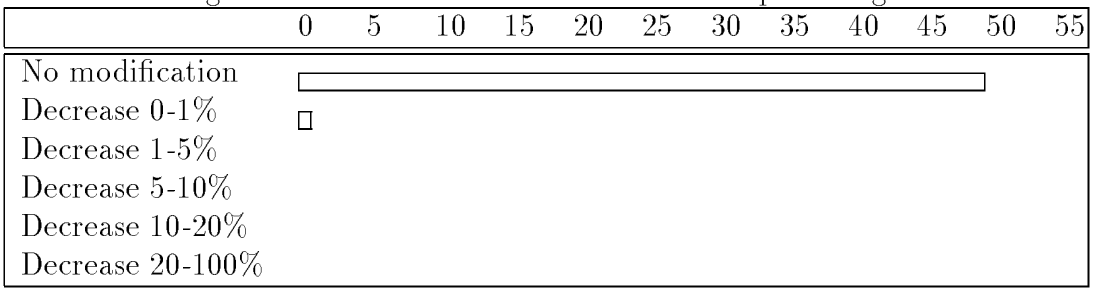


Figure F.3: Size 3 vs. 5: Effect on critical path length

To understand these results, the use of a finite domain of demands and the process of *rounding-up* has to be briefly explained. The reason why a domain of demands of *finite* size is used is to ensure the termination of the strictness analysis process. Although demands larger than the size of the domain are allowed to exist temporarily during the analysis process, these demands are not allowed to cross function boundaries, i.e., they can only be used to annotate expressions local to a function body. All demands used for the result of a function or returned as demands for its arguments are always *rounded-up*, i.e., mapped into a safe element of the selected finite domain: demands which are already member of the domain, i.e., whose size is smaller than the maximum allowed size, are left unchanged; demands whose size is larger than the allowed maximum are mapped onto a less precise, but safe, approximation.

Let's examine a simple example — the `tl` function — which is used in a number of programs. Its definition is given by the following Id code:

```
Def tl l =
   { case l of
      Nil = Nil
   |  Cons _ xs = xs };
```

Suppose we need to analyze the expression "`tl e`" demanded by the following (head-strict) demand: $(\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})$. When the size of the domain is 2, then we can infer that expression "`e`" will be demanded with the following demand:

$$\{Nil,\ Cons\ @\ (\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \$\ \hat{}x\})\}$$

When the size is 1, however, then because of the rounding that occurs, only the following can be inferred:

$$(\mathsf{Rec}\ x{:}{:}\ \{\mathsf{Nil}, \mathsf{Cons}\ \hat{}\$\ \hat{}x\})$$

Thus, all the information about the tail of `e` being head-strict is lost, so that all components effectively get delayed.

A similar process happens when analyzing the expression "`tl (tl e)`". In this case, the rounding-up that leads to a loss of information occurs when going from size 3 to size 2. As there are much fewer cases in the programs analyzed where this effectively happens, going from size 2 to 3 has thus much less influence.

It should be noted that using a larger domain augments the precision of the analysis but also increases its cost: the demands to be manipulated are larger and so is the domain on which the fixed point iteration process is performed, thus increasing the cost of analyzing recursive functions. For this reason, given the little effect that using a domain of size 3 or larger seems to have, the subsequent experiments will use, unless specified otherwise, a domain of size 2.

F.2 Influence of the type of abstract conditions

As described earlier in Section 4.3, our compiler defines three different types of abstract-conditions: `:None, :Intra-procedural, :Inter-procedural`. Each type should lead to more precise strictness analysis results.

In Figures F.4 and F.5, we present histograms showing the effect of using various forms of abstract conditions on the reduction of the critical path length. More precisely, the tables present the following data:

- Figure F.4: This shows the number of cases for which the critical path length was reduced by the indicated percentage when going from using `:None` to `:Intra-procedural` abstract conditions, for a domain of size 2.
- Figure F.5: Similarly, when going from using `:Intra-procedural` to `:Inter-procedural` abstract conditions, for a domain of size 2.

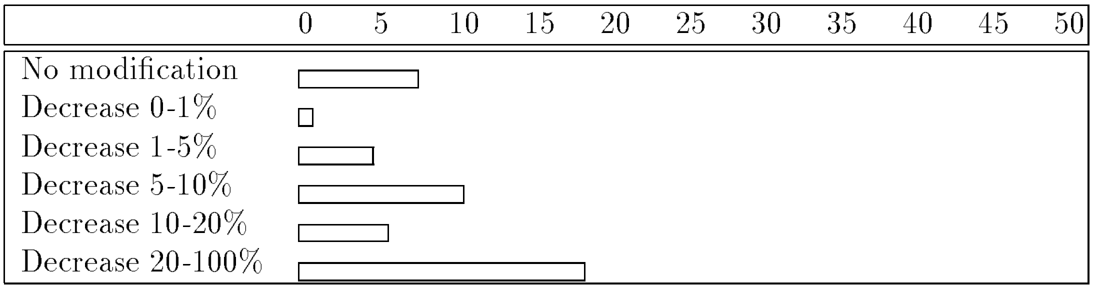


Figure F.4: None vs. intra-procedural abstract conditions: Effect on critical path length

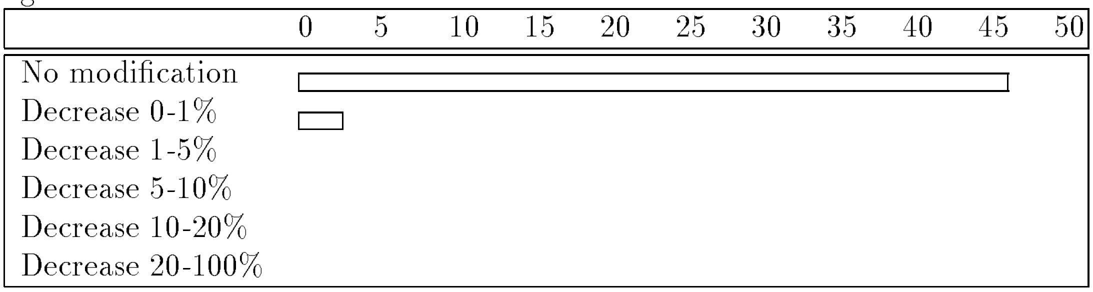


Figure F.5: Intra-procedural vs. inter-procedural abstract conditions: Effect on critical path length

Clearly, the use of `:Intra-procedural` abstract conditions has an important effect when compared with the `:None` case. This was to be expected because using option `:None` means that the only arguments of functions which can be inferred to be strict are those which do not appear within *any* conditionals. All others are inferred to be non-strict. On the other hand, when using the `:Intra-procedural` option, many arguments appearing within conditionals can be inferred to be strict, e.g., in the case of an `if` instruction, (some of) the arguments appearing in both branches of the `if`.

As for the `:Inter-procedural` abstract conditions, their use seems to bring very little improvement. Such abstract conditions are useful, as discussed earlier in Section 4.3, to handle cases where some form of *correlation* between arguments occur. Clearly, such cases can not be expected to occur frequently, thus the small improvement obtained when using `:Inter-procedural` abstract conditions.

Again, as was the case when using a larger domain of demands, using more complex abstract conditions increases the cost of the analysis. For example, when using `:Inter-procedural` abstract conditions, all such conditions must be preserved within the demand annotations, even when going from the analysis of one function to another, i.e., the conditions are preserved across function boundaries. Furthermore, they also accumulate through the various functions, and thus get very complex. Because of this increased cost and the little advantage they seem to bring, subsequent experiments will only use `:Intra-procedural` abstract conditions.

F.3 Influence of the number of function versions

One important feature of our compiler, as mentioned in Section 4.4, is that it can generate multiple function versions, each specialized to a particular context of execution, i.e., for a particular demand made for its result. The conjecture is that by allowing such multiple clones/versions to exist, the resulting program will be made more efficient, because each specialized version can be optimized using more precise information. Clearly, if only a single function version is allowed to exist, then this version must account for all the possible contexts of use, i.e., it must account for the least-upper bound of all possible demands made for the result of that function, which may lead to an important loss of precision.

Figures F.6, F.7, F.8, and F.9 present histograms showing the effect of increasing the number of function versions on the reduction of the critical path length. [1] As can be seen from these figures, increasing the maximum number of versions allowed for each function has a positive effect on the reduction of the critical path length. [2]

[1] Note that the `:maximum-nb-of-versions` parameter applies for each function definition. In other words, setting the maximum to 2 allows the compiler to create up to 2 versions *for each* distinct function definition.

[2] Note that there was a single exception, omitted from the histograms: When compiling the `FFT` benchmark, using 3 versions instead of 2 slightly increased the critical path length by 0.28% (i.e., it goes from 6805 time units to 6824); however, the total number of instructions is effectively reduced by 2.45%. This phenomena is not fully understood and occurs only when using direct

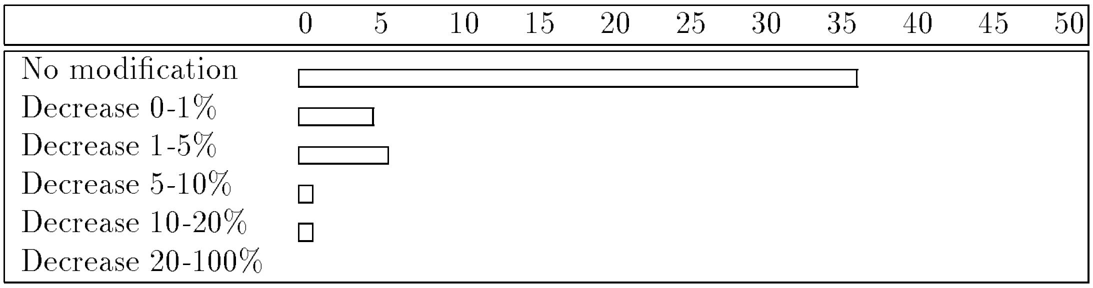


Figure F.6: Single version vs. 2 versions: Effect on critical path length

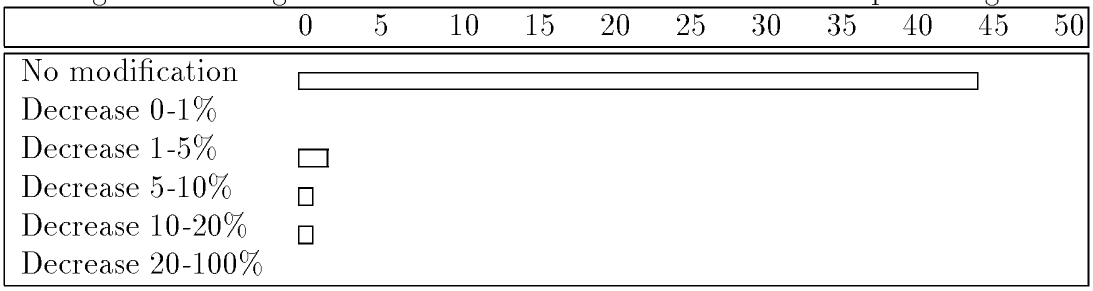


Figure F.7: 2 versions vs. 3 versions: Effect on critical path length

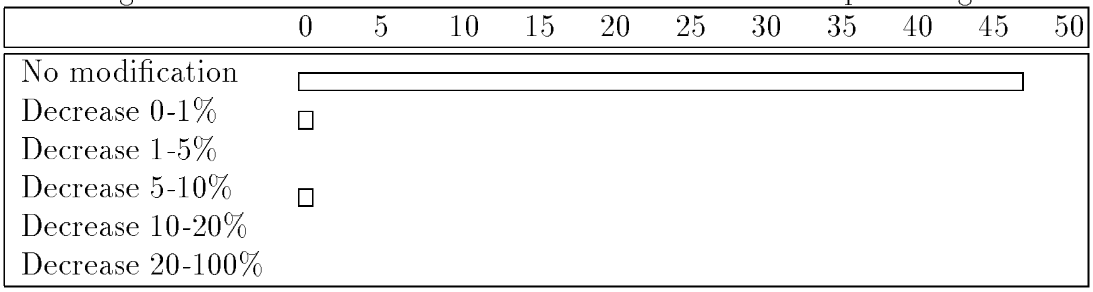


Figure F.8: 3 versions vs. 5 versions: Effect on critical path length

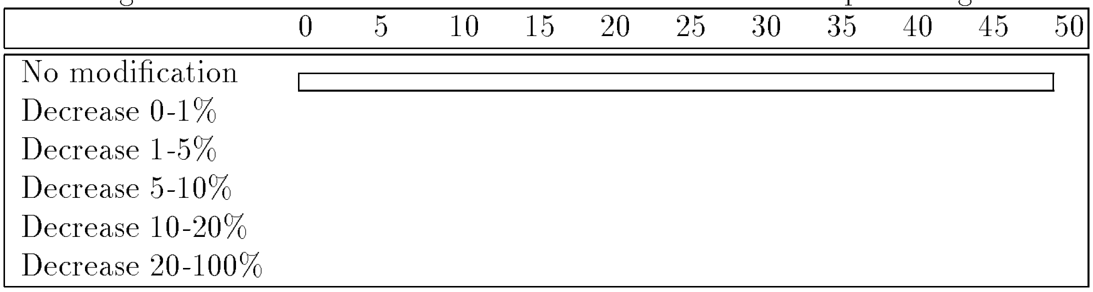


Figure F.9: 5 versions vs. unlimited number of versions: Effect on critical path length

Clearly, the function versions/specializations approach seems quite important in order to obtain more efficient programs. It seems that, as long as there is no speculation (cf. Section 5.4), the number of versions required for each function remains rather small: among all our benchmark programs, the maximum number of versions required for any program was 4. Furthermore, in contrast to the previous parameters, increasing the `:maximum-nb-of-versions` parameter *simplifies* the task of the compiler, as it means that there will be fewer occasions where existing versions will have to be merged. [3] Because of this, in the subsequent sections, we will always allow for an unlimited number of function versions.

F.4 Influence of the use of run-time evaluators

In certain cases where precise information can not be inferred at compile-time, it is still possible to infer certain strictness information that will hold at run-time, as was described in Section 4.6. In such cases, run-time evaluators can be used. Figure F.10 presents an histogram showing the effect of using run-time evaluators on the reduction of the critical path length. These results came as quite a surprise, as they imply that simple run-time evaluators such as those defined and used by our compiler (and also used in Burn's approach [Bur87, Bur91b]) bring no significant improvement to the efficiency of the resulting program: there was a *single* case where the use of run-time evaluators brought a reduction of the length of the critical path. In fact, in some cases, their use increased the length of the critical path.

The major reason why allowing run-time evaluators does not bring much improvement is because, as was realized by examining the resulting programs generated by the compiler, there are very few cases (less than 10% of the benchmarks) where run-time evaluators effectively get used. Nonetheless, there are indeed cases where they are used. This can be seen, in Figure F.11, by examining the effect on the total number of instructions that get executed when run-time evaluators are allowed. Clearly, as was to be expected, using run-time evaluators increases the total number of instructions executed. [4]

---

thunks.

[3]The process of function versions merging, when only a limited number of versions is allowed, can be quite costly: it requires invalidating an existing function and, transitively, all versions using the invalidated ones; then, when this has been done, the strictness analysis of all functions using those which have just been invalidated has to be redone from scratch!

[4]Recall from Section 4.6 that run-time evaluators simply force the evaluation of data structures

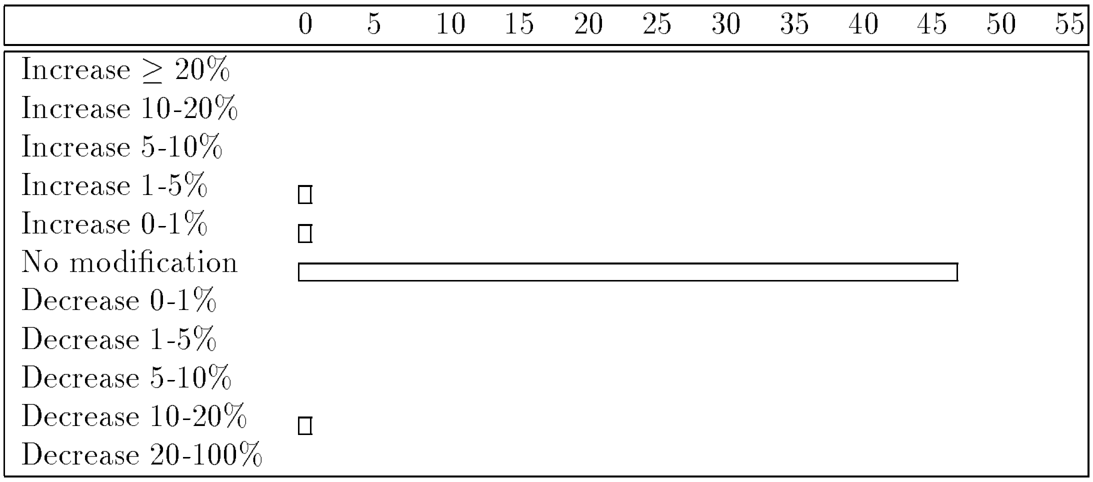


Figure F.10: Without vs. with run-time evaluators: Effect on critical path-length

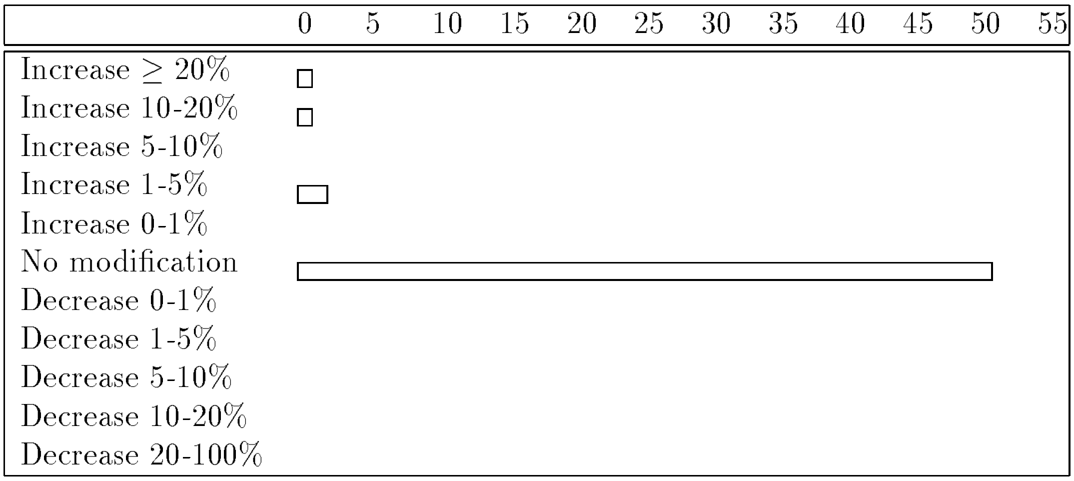


Figure F.11: Without vs. with run-time evaluators: Effect on the total number of instructions

So, why is it that the use of run-time evaluators does not bring any reduction of the critical path length but does increase the total number of instructions? The first part of the answer is that in many cases where run-time evaluators were used, they were quite simple and brought very little advantage from the parallelism point of view. The major example is in `fft`, where the run-time evaluators used by the resulting program simply force the evaluation of the two components of complex numbers, thus causing additional overhead for very little benefit.

Another reason is that, in order to allow for a simple implementation, run-time evaluators were implemented in such a way that they always traverse/force the indicated data structure, regardless of its current evaluation status. Clearly, this leads to useless additional work being done. Such a case occur in the `quickSort` benchmark. At some point in this program, the evaluation level of one of the subexpression goes from $\{Nil,\ Cons\ \hat{}\$\ HTS\}$ to HTS. Thus, although the initial demand was not head-and-tail strict (the first element had to be delayed), the final demand effectively becomes head-and-tail strict, meaning that a Head+Tail run-time evaluator can be used. In this case, even though it is safe to do so, traversing the whole data structure to force its full evaluation is clearly useless work, as it is already almost fully evaluated, except for its first head.

One possible strategy to prevent such unnecessary additional work would be, as is done in some implementation of the evaluation transformers model of Burn [Bur87, Bur91b], to associate each data structure with an indicator that would keep track of its current evaluation status. In the `quickSort` case, after the head had been forced, it would be realized that the tail is already evaluated to HTS, so the run-time evaluator would simply stop its traversal of the data structure. There are two reasons why this strategy has not been implemented in our lazy compiler. First of all, there are very few cases where run-time evaluators effectively get used. Next, using the evaluation status/flag approach would increase the overhead for all data structures manipulation: some space would have to be allocated to store the flag; the flag would have to be checked/updated by all data structure operations. Given the limited number of occasions where run-time evaluators are useful, introducing this additional overhead would seem to be quite wasteful and thus has not been implemented.

as soon as they become needed for sure, hopefully a short while *before* they effectively become needed. Thus, triggering the execution of a run-time evaluator incurs additional overhead, which is allowed in the hope that the critical path length will be reduced.

For the reasons indicated above, in all experiments except the current one, run-time evaluators were not used.

F.5 Influence of the initial demand

As discussed earlier (cf. Section 2.3), the classical notion of head-strictness is related to the printing process performed by a sequential lazy interpreter when printing a result. Because the data structure being (sequentially) printed might be potentially infinite, we must ensure, while printing it, that we have finished printing its prefix portion before starting the evaluation of the tail. In a compiler, such as ours, that can express head-strictness, we can always assume that the initial demand made for the result of a program is at least a head-strict demand. On the other hand, if the compiler can not express head-strictness, then all it can assume as the initial demand is a *simple-strict* demand: it would be unsafe/wrong to assume that all heads and all tails are required (head-and-tail strict), because if the result was an infinite data structure, then it would mean that an infinite sequence of computation would be triggered, even if only a finite portion of the result is examined/needed. [5]

For example, suppose we have the following program:

```
Def from k = Cons k (from (k+1));
Def program1 = from 0;
```

If a head-and-tail strict demand was used as the initial printing demand, then it would mean that neither the head nor the tail of the `Cons` would have to be delayed, thus resulting in an infinite sequence of recursive calls to `from`. Clearly, in order to deal correctly with finite portions of potentially infinite data structures, this should be avoided.

The two types of initial demands, in the case of linear lists, are defined as follows:

- Simple-strict demand: $(\textsf{Rec}\,x{::}\ \{\textsf{Nil}, \textsf{Cons}\ \hat{}\$\ \hat{}x\})$;
- Head-strict demand: $(\textsf{Rec}\,x{::}\ \{\textsf{Nil}, \textsf{Cons}\ \$\ \hat{}x\})$;

In the former case (simple-strict) although the top-level data structure does not need to be delayed, every tails *and* every heads must be delayed. In the latter case (head-strict) although all tails must also be delayed, none of the heads need to be.

[5]Of course, if we know for sure that the result is *not* infinite, then we can use a stronger initial demand: Cf. Section 5.4.5.

Figure F.12 presents an histogram showing the effect, on the reduction of the critical path length, of using a head-strict demand instead of a simple strict demand as the initial demand for the result of the program. [6]

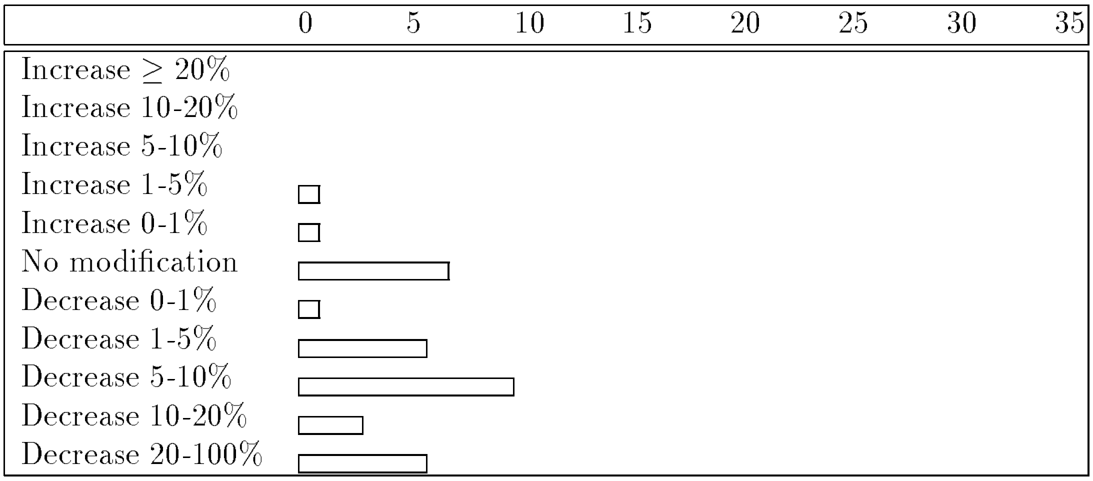


Figure F.12: Simple-strict vs. head-strict initial demand: Effect on critical path length

Using a head-strict demand leads, in most cases, to a reduction of the critical path length. This can be explained quite simply by the fact that using a stronger demand for the result of a function allows the compiler to infer more precise/stronger information for its arguments, thus obtaining more precise strictness information and allowing the removal of a larger number of delays.

But also note that there was two exceptions, i.e., two cases for which using a head-strict demand lead to an increase of the critical path length: *i*) `PRIMES_LAZY`: $+\Delta\,0.6\%$; *ii*) `MM`: $+\Delta\,4.3\%$.

In the cases of `PRIMES_LAZY`, the slight increase is analogous to the one that would occur for the following program:

```
Def program0 = Cons (fact 1000000) Nil;
```

In this simple example, if the demand made for `program0` is a simple-strict demand, then the expression (*fact* 1000000) used as argument to `Cons` does not need to be evaluated in order to produce the required answer – "*Cons* #`Thunk` #`Thunk`". On the other hand, if the demand is a head-strict one, then (*fact* 1000000) has

[6] Note that, in this experiment, only programs for which the result was a non-scalar value were analyzed: when dealing with scalar values, a simple strict demand is the most precise demand that can be made, i.e., a simple strict demand is equivalent to a head-and-tail strict one, thus, *a fortiori*, is as strong as a head-strict one.

to be evaluated to produce the required answer – "*Cons* 100000! #`Thunk`" – thus resulting in more work being performed. Clearly, this is correct behavior: as we are demanding *more* of the answer to be produced, we can expect more work being done; when the result printed for `PRIMES_LAZY` is examined, we can effectively see that one more element gets evaluated/printed when the initial demand is a head-strict one instead of simple-strict.

In fact, given this interpretation, one could wonder why there are not more cases for which using a head-strict demand leads to a larger critical path length. The answer to that question was already answered above: in most cases, the additional work having to be done is clearly less important than the gain obtained from being able to infer more precise strictness information, and thus removing more delayed expressions.

Note that the case for `MM` is slightly different: in this case, the slight increase seems to be due to the use of direct thunks instead of indirect ones (cf. Section F.7). When direct thunks are used, using a head-strict demand instead of a simple-strict demand, although it reduces the total number of instructions executed (by 2.9%), slightly increases the critical path length (by 4.3%). On the other hand, when indirect thunks are used (which, in both cases, increases the critical path length and the total number of instructions compared to the use of direct thunks), then both the critical path length and the total number of instructions are reduced (by 2.6% and 4.1% resp.) when going from a simple-strict demand to a head-strict one. The exact cause of this phenomena is not understood.

F.6 Influence of head-strictness

In this section we examine the effect of *head-strictness* on the reduction of the critical path length and on the total number of instructions executed. In order to do this, an additional compilation option was defined for the lazy compiler: `:et`. This option allows the compiler to (somewhat) emulate the strictness results that would be obtained using a compiler based on the evaluation transformers model [Bur87, Bur91b]. [7] In such a compiler, only three types of demands can be expressed: simple-, tail-, and head-and-tail- strict. Head-strict demands, i.e., demands delaying the tail of a data structure but allowing its other components not to be delayed, are not possible. Within our compiler, emulating this mode of analysis is quite simple: it simply requires, whenever a demand has to be returned as the demand for the arguments of a function, to round it up, i.e., delay any non-tail component which has been inferred to be strict by our backward compiler.

Figure F.13 shows the effect, on the reduction of the critical path-length, of using only demands equivalent to those of the evaluation transformers model vs. our lazy compiler, i.e., of not having vs. allowing head-strict demands, whereas Figure F.14 shows the effect on the total number of instructions. Note that, in both cases, the initial (printing) demand was not allowed to be head-strict, i.e., although we allowed head-strict demands to be inferred for arguments, only a simple-strict demand was assumed for the result of the program. Also, in both cases, the following setting of the parameters were used:

- Type of abstract conditions: Intra-procedural;
- Size of domain = 1;
- No run-time evaluators;
- Maximum of 3 versions per function;
- Data structure components implemented using direct thunks.

As can be seen, although there are indeed a number of cases for which this leads to a reduction of the length of the critical path, the effect is not dramatic. However, it must be realized that this behavior is to be expected, for the following reason: when the initial demand is *not* head-strict, then there is very little chance that head-strictness will get propagated further. In other words, having a simple-strict initial demand leads, in most cases, to very few cases where a head-strict demand can be inferred for the arguments; on the other hand, having a head-strict initial demand makes it possible that this demand will get propagated further into head-strict

[7] We say *somewhat* because we are still using our backward approach, and not a forward one. Although there are cases where the backward approach is preferable, e.g., for dealing with data structures, there are also some cases where the forward approach would obtain slightly better results. But also not that those cases would be the same type of cases for which the use of inter-procedural abstract conditions lead to an improvement, i.e., very few cases.

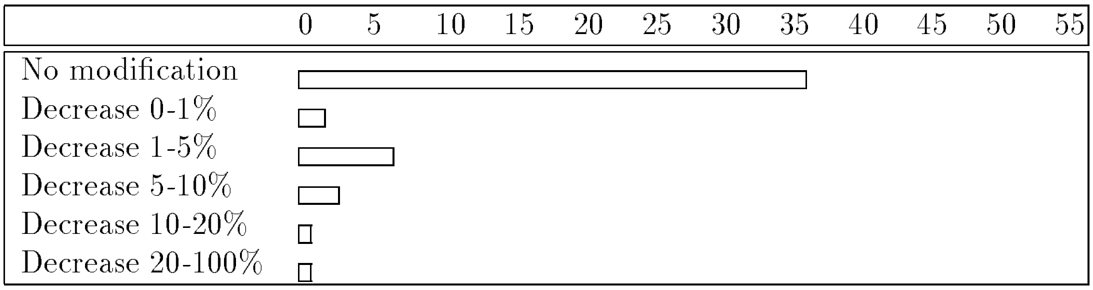


Figure F.13: Without vs. with head-strictness: Effect on critical path-length

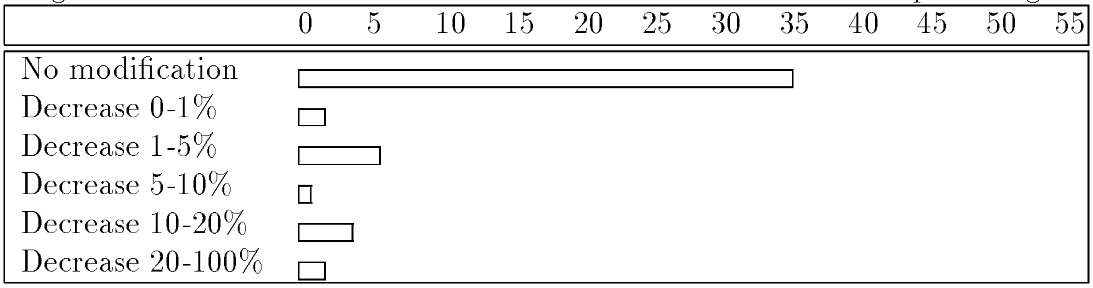


Figure F.14: Without vs. with head-strictness: Effect on total number of instructions

demands. Thus, allowing for head-strict demands makes very little sense *unless* the initial demand for the result of the program is also allowed to be head-strict.

Figures F.15 and F.16 show the effect, respectively on the length of the critical path and on the total number of instructions, of not having head-strict demands at all vs. allowing head-strict demands both as initial demand and as demands inferred for arguments. Clearly, in this case, there are more cases for which this leads to a reduction of the critical path length. [8]

Finally, we present, in Figures F.17 and F.18, a comparison between what would be obtained by a compiler based on the evaluation transformers model vs. our own optimizing compiler (i.e., by using a domain of size 2 and allowing an unlimited number of function versions). Clearly, these results tend to support the claim that the notion of head-strictness is effectively important in order to obtain good strictness analysis results and improve the efficiency of the resulting programs, although the effect is not dramatic.

[8] Note that, as in Section F.5, the same two exceptions occur: `PRIMES_LAZY` and `MM`. The behavior of these exceptions was discussed earlier.

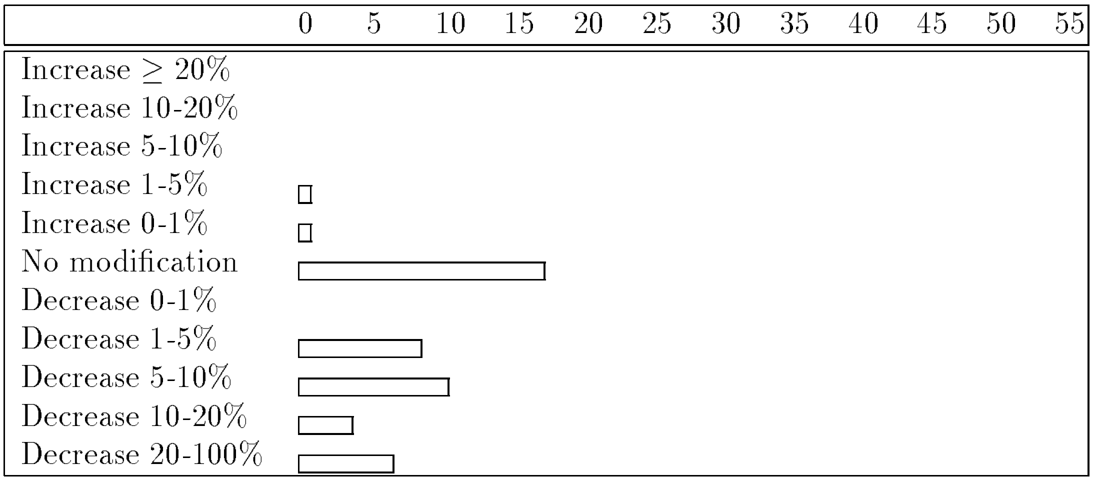


Figure F.15: Without head-strictness vs. with head-strictness + head-strict initial demand: Effect on critical path length

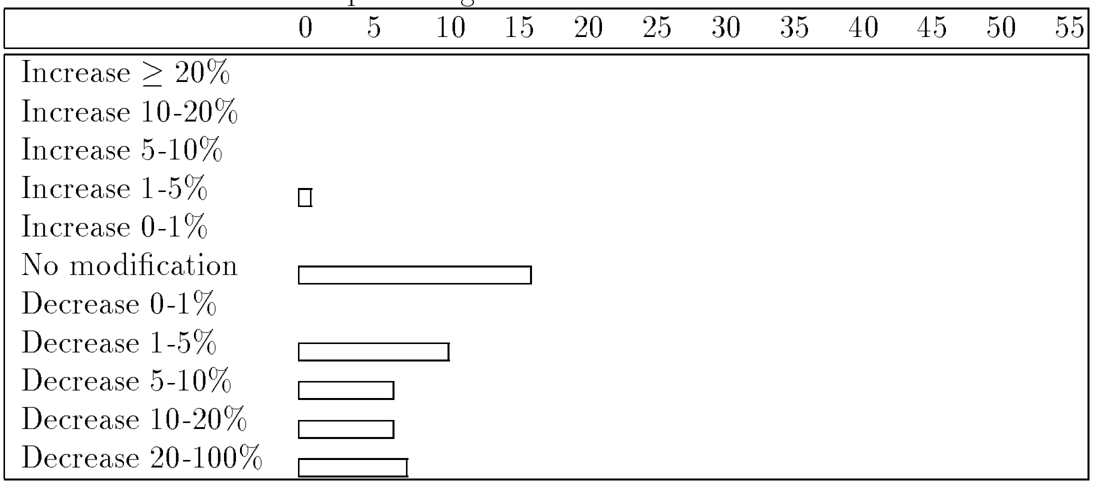


Figure F.16: Without head-strictness vs. with head-strictness + head-strict initial demand: Effect on total number of instructions

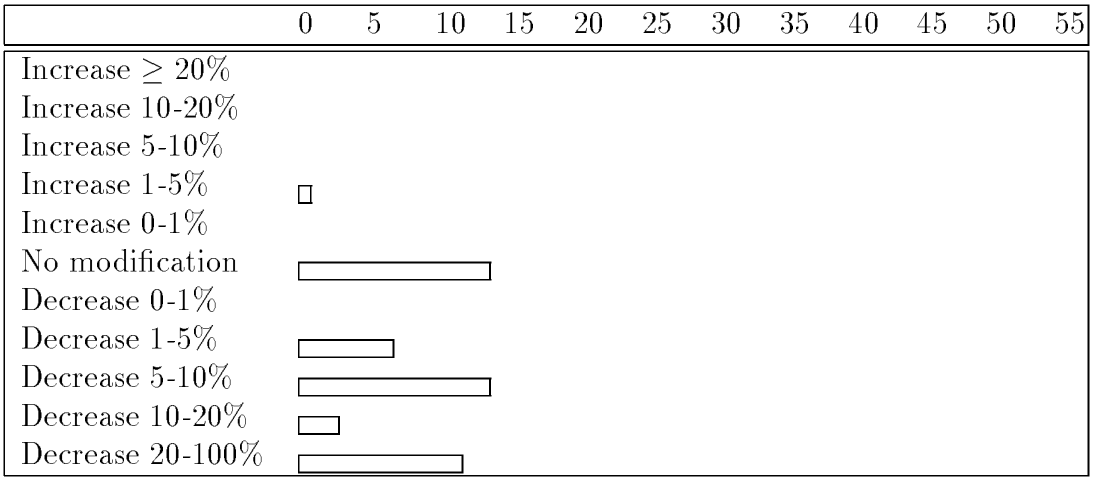


Figure F.17: Evaluation transformers style vs. optimized lazy compiler: Effect on critical path length

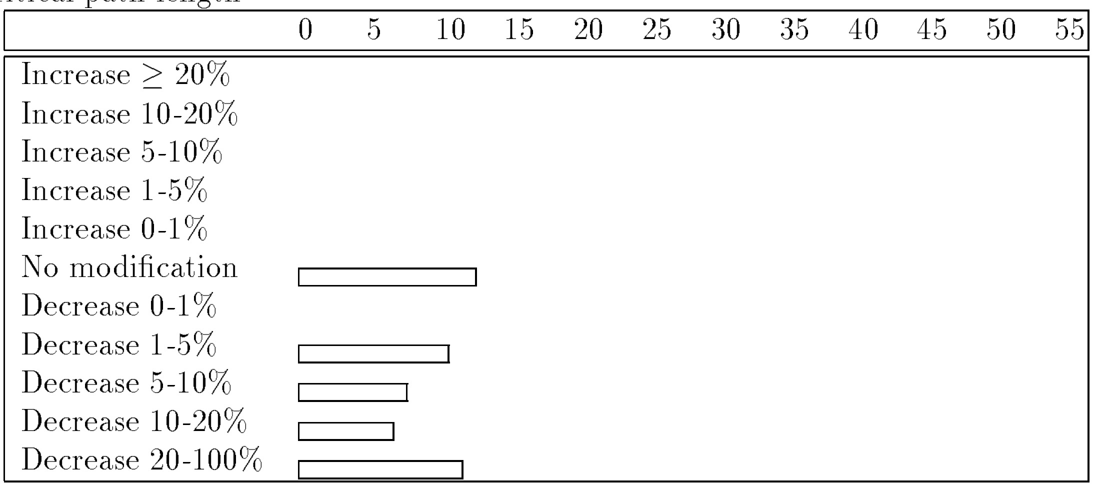


Figure F.18: Evaluation transformers style vs. optimized lazy compiler: Effect on total number of instructions

F.7 Influence of the implementation of data structures

The final factor we examine concerns the internal implementation of data structures within the lazy compiler. In the standard Id compiler that we used as the basis of our lazy compiler, data structure slots can be associated with two different types of thunk/delayed expression: [9]

- Direct thunk: When a component of a data structure is associated with a direct thunk, the thunk gets evaluated as soon as a fetch is performed on the component. Further accesses to the component simply fetch the value directly.
- Indirect thunk: When a component is associated with an indirect thunk, the component does not store directly the value or delayed expression; instead, it stores a *pointer* to the value or delayed expression. To trigger the evaluation of a delayed component, the pointer has to be fetched and then has to be explicitly forced. Further accesses to the value then require an explicit indirection (additional fetch) to be performed. [10]

The major advantage of direct thunks over indirect thunks is that accessing the resulting value can be done directly, without any indirection overhead. However, there are cases where direct thunks are more costly to use, because of the fact that any access would trigger the evaluation of the associated thunk. This phenomena occurs when a data structure is being traversed recursively by a function and the structure occurs in a non-strict argument position. Here is a simple example:

```
Def take l k =
   if k == 0 then
      Nil
   else
      { case l of
         Nil = Nil
      |  Cons x xs = Cons x (take xs (k-1)) };
```

In this case, the argument `xs` in the recursive call to `take` has to be delayed. Now, if direct thunks are being used, then we can not simply fetch the component

[9] Note that within the prototype lazy compiler, non-strict arguments to functions are always implemented using indirections, i.e., using single slot L-structure (cf. Sect. 4.7).

[10] At the source language level [Nik88], these two types of delayed expressions, in the case of lists, would be denoted respectively by "`e1 #:# e2`" and "`{# e1} :  {# e2}`".

associated with `xs` and transmit it to the recursive call, as this would mean that the associated thunk would effectively get evaluated. What has to be done, when using direct thunks, is to *create a new thunk* whose only task is to embed the fetching of the `xs` component. On the other hand, if indirect thunks are being used, then we can simply transmit the indirection (pointer to the thunk) to the recursive call, without any risk of triggering non-needed computation. Considering that the cost of creating a new thunk is relatively expensive (on the order of 20 instructions), we can expect that there are effectively cases where being able to transmit existing indirections instead of creating new thunks should be advantageous.

Figure F.19 presents an histogram showing the effect, on the reduction of the critical path length, of using direct thunks instead of indirect ones, whereas Figure F.20 shows the effect on the total number of instructions.

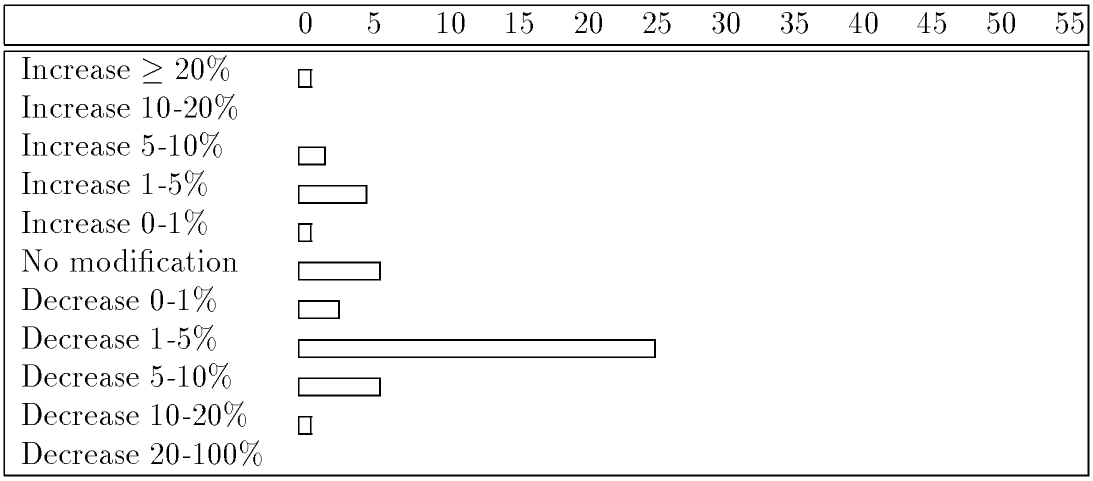


Figure F.19: Indirect vs. direct thunks: Effect on critical path length

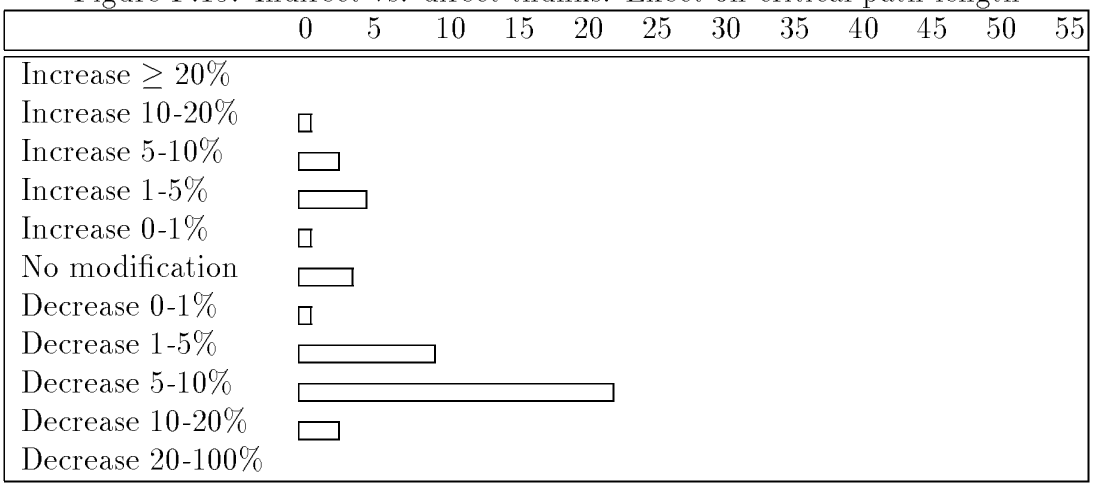


Figure F.20: Indirect vs. direct thunks: Effect on total number of instructions

It is interesting to compare these results with those that are obtained when using a purely lazy variant of the compiler, i.e., when no optimizations are performed and thus no delays get removed at compile-time: this is presented in Figures F.21 and F.22, where it is shown how using indirect thunks instead of direct ones has an important effect on the reduction of both the critical path length and the total number of instructions. In this case, indirect thunks are almost always more efficient. [11] These results are easily explained: when using the purely lazy variant, arguments must always be delayed; using indirect thunks thus precludes the need for creating new thunks when transmitting data structure components as arguments to functions, because the indirection can always be transmitted instead.

From the above result, it would seem that, as the resulting program contains more non-strict contexts, the more advantageous indirect thunks are. Inversely, as more contexts can be made strict (through strictness analysis) without the need for any delay, then the more direct thunks become advantageous. Given the fact that direct thunks seem slightly more advantageous and in order not to inhibit the advantages of obtaining more precise strictness analysis results, the optimizing lazy compiler will use direct thunks. [12]

[11]It is important to note, in the case of Figures F.21 and F.22, that the order of comparison is the inverse of the one used for Figures F.19 and F.20: in the former case, the reductions are indicated when going from direct thunks to indirect ones; in the latter case, the reductions are indicated when going from the use of indirect thunks to direct ones.

[12]It is also interesting to note, in support of our thesis, that the only case for which the increase of the critical path length, when using direct thunks, was larger than 10% (50.01%) was for the `HAMMING_EXT` benchmark. Poor strictness analysis results are obtained when analyzing this program; but note that this program can only be executed correctly using a lazy implementation, as it produces a potentially infinite data structure. As for the total number of instructions, the only case for which the increase was larger than 10% (10.72%) is for the `COMPRESSPATH` program. Although this program does not require a lazy interpreter, the strictness results that can be inferred are not very strong either.

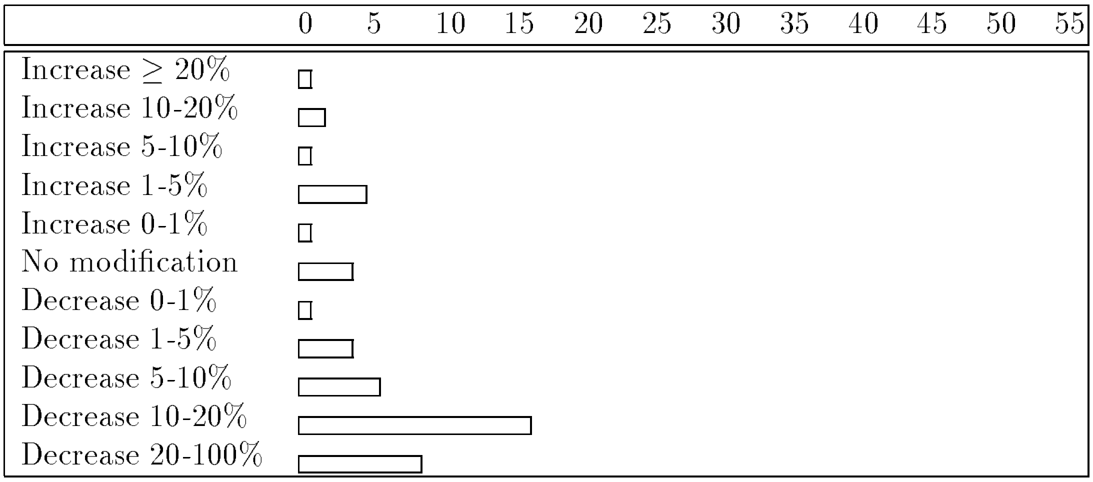


Figure F.21: Direct vs. indirect thunks (purely lazy variant): Effect on critical path length

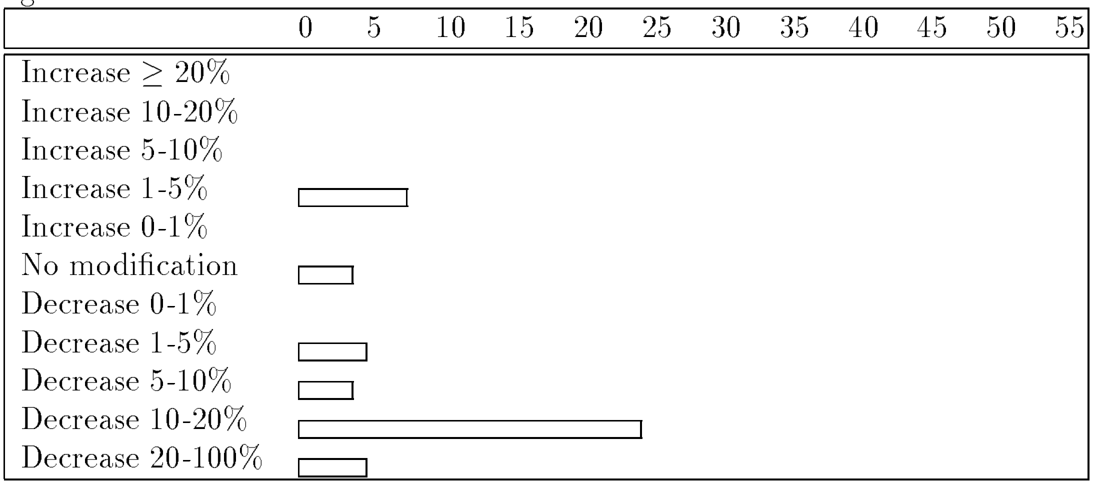


Figure F.22: Direct vs. indirect thunks (purely lazy variant): Effect on total number of instructions

Appendix G

Formal specification of compile-time speculation and run-time printing/forcing

G.1 The notion of answer

In this appendix, we specify more formally some of the notions introduced in Chapter 5: printing/forcing process, parallel head-strictness, compile-time speculation level vs. run-time printing level, etc. Note that the goal of these specifications is to clarify the various concepts, not to provide any formal proofs.

To simplify the presentation, we assume our programs always produce answers which are data structures defined through algebraic type definitions; this is not really a restriction: basic scalar types such as booleans, integers, etc., can always be viewed as predefined algebraic types, i.e., as enumeration of nilary constructors.

Type definitions are given by definitions such as the following, with $k \geq 1$ and $n_j \geq 0$:

$$Type\ t = Con_1\ t_{11}\ \ldots\ t_{1n_1}\ |\ \ldots\ |\ Con_k\ t_{k1}\ \ldots\ t_{kn_k};$$

We define an answer to a program to be as presented below, where we assume, to simplify the specification, that the original source type is available. Essentially, this specification expresses the idea that an answer can be a delayed expression, in which case a thunk is still associated with it; it also expresses the fact that an answer to a program might be a non-terminating computation: in this case, *a.being-evaluated*?

would simply remain **true** forever. As for the invariant, it expresses the idea that an expression's value is available if and only if it is not delayed and is not being evaluated, together with additional (less important) facts about the associated thunk.

$$
\begin{array}{lrcl}
\mathit{Answer} :: & \mathit{type} & : & \mathit{Type\text{-}name} \\
& \mathit{constructor} & : & [\mathit{Constructor\text{-}name}] \\
& \mathit{components} & : & [\mathit{Answer}^*] \\
& \mathit{delayed?} & : & \mathbb{B} \\
& \mathit{being\text{-}evaluated?} & : & \mathbb{B} \\
& \mathit{thunk} & : & [\mathit{Expression}]
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{inv\text{-}Answer}(a) \quad \triangleq \\
\quad (a.\mathit{constructor} \neq \mathsf{nil}) \wedge (a.\mathit{components} \neq \mathsf{nil}) \\
\qquad \Leftrightarrow \ \neg\, a.\mathit{delayed?} \wedge \neg\, a.\mathit{being\text{-}evaluated?} \\
\quad \wedge\ a.\mathit{thunk} \neq \mathsf{nil} \ \Leftrightarrow \ a.\mathit{delayed?} \\
\quad \wedge\ a.\mathit{delayed?} \ \Rightarrow \ \neg\, a.\mathit{being\text{-}evaluated?}
\end{array}
$$

G.2 The GITA printer

The result printed by the regular GITA interpreter, when given an answer a, can be specified as described below. As we assume that many activities can be done concurrently, when encountering an answer which is still in the process of being evaluated, then the *gita-print* function simply calls itself recursively until the answer effectively becomes available. If it is not a delayed expression, it first prints its constructor and then recursively prints its associated components ("<" and ">" are printed to delimit the scope of constructors).

$gita\text{-}print(a) \quad \triangleq$

if $a.being\text{-}evaluated?$

then $gita\text{-}print(a)$

else if $a.delayed?$

then print "$\#Thunk$"

else print " $<$ "

print $a.constructor$

for i from 1 to len $a.components$ do

$gita\text{-}print(a.components(i))$

print " $>$ "

G.3 Level of evaluation of an answer

We now define a number of predicates indicating the level to which an answer has been evaluated. These, combined with the specification given in the previous paragraph, will allow us to infer what would effectively be printed for a given answer. As for the *gita-print* function, any of the predicates defined below, when given an answer which is still in the process of being evaluated, will simply call itself recursively until evaluation is over, at which time the predicate's answer can be obtained; if $a.being\text{-}evaluated?$ remains true forever, i.e., the answer is $\perp$, then so is the predicate's result.

The first predicate indicates when an answer is only evaluated to WHNF, as is the case when the initial demand is a simple-strict demand:

$forced\text{-}to\text{-}whnf(a) \quad \triangleq$

if $a.being\text{-}evaluated?$

then $forced\text{-}to\text{-}whnf(a)$

else $\neg\, a.delayed? \wedge \forall i \in$ inds $a.components \cdot a.components(i).delayed?$

The next predicates are used to indicate that an answer has been forced to a specific level of evaluation, either through a run-time forcing/printing process or through a compile-time speculation level (this is explained in more detail below). It is these predicates which express the fact that, for a potentially infinite data structures, only a finite number of recursive unfoldings has been performed and that the remaining ones are still delayed.

$$
\begin{array}{l}
\textit{forced-to-level}(a,k) \quad \triangleq \\
\qquad \textit{forced-to-level}'(a,k,k) \\
\\
\textit{forced-to-level}'(a,k,k_{max}) \quad \triangleq \\
\qquad \textsf{if } a.\textit{being-evaluated?} \\
\qquad \textsf{then } \textit{forced-to-level}'(a,k,k_{max}) \\
\qquad \textsf{else if } k < 0 \\
\qquad\quad \textsf{then } a.\textit{delayed?} \\
\qquad\quad \textsf{else } \neg\, a.\textit{delayed?} \wedge \\
\qquad\qquad \forall i \in \textsf{inds } a.\textit{components} \cdot \\
\qquad\qquad\quad \textsf{if } a.\textit{components}(i).\textit{type} = a.\textit{type} \textsf{ then} \\
\qquad\qquad\qquad \text{(* Recursive component: We decrease the remaining evaluation level. *)} \\
\qquad\qquad\qquad \textit{forced-to-level}'(a.\textit{components}(i), k-1, k_{max}) \\
\qquad\qquad\quad \textsf{else} \\
\qquad\qquad\qquad \textit{forced-to-level}(a.\textit{components}(i), k_{max})
\end{array}
$$

G.4 Specification of the compilers and interpreters

We will next formalize the idea that when a program is lazily executed without any additional forcing but the one induced by the demand known at compile-time, then not much is computed/evaluated. In addition, we are going to clarify the distinction between the initial demand, which is specified at compile-time, and the printing level, which is specified at run-time. In order to do this, we will specify, in an abstract manner, the interface to the compiler and run-time program interpreter.

Two compilers will be specified: using the first one (*basic-compiler*), the user can only specify the type of initial demand which has to be used, i.e., simple-strict or parallel head-strict; using the second one (*speculative-compiler*), the user can indicate a speculation level to be used in order to compile the program. As for the interpreters, two of them will also be specified: one (*basic-interpreter*) which simply produces the answer obtained by running the compiled program and another (*forcing-interpreter*) which runs the compiled program and, if required, forces its evaluation according to the printing level specified by the user. The interfaces for these functions is presented in Figure G.1. Note that "$: ss$" stands for "Simple-strict" whereas "$: phs$" stands for "Parallel head-strict".

$$\textit{basic-compiler}: \textit{Program} \rightarrow \textit{Initial-demand} \rightarrow \textit{Compiled-program}$$

$$\textit{Initial-demand} = \{: ss, \, : phs\}$$

$$\textit{basic-interpreter}: \textit{Compiled-program} \rightarrow \textit{Answer}$$

$$\textit{speculative-compiler}: \textit{Program} \rightarrow \textit{Speculation-level} \rightarrow \textit{Compiled-program}$$

$$\textit{Speculation-level} = \mathbb{N}$$

$$\textit{forcing-interpreter}: \textit{Compiled-program} \rightarrow \textit{Printing-level} \rightarrow \textit{Answer}$$

$$\textit{Printing-level} = \mathbb{N}$$

Figure G.1: Specification of the interface of compilers and interpreters

The basic compiler and the basic interpreter

When the initial demand is a simple-strict demand, then all that is evaluated is the top-level of the answer, i.e., the associated constructor with all components delayed. This is specified as follows:

Specification 1

$$\textit{basic-interpreter}(\textit{basic-compiler}(pgm, \, : ss)) = a \;\Rightarrow\; \textit{forced-to-whnf}(a)$$

When parallel head-strictness is introduced, a little more is asked of the answer, so the answer is evaluated to a slightly deeper level. However, without forcing or speculation, this evaluation is not very deep either.

Specification 2

$$\textit{basic-interpreter}(\textit{basic-compiler}(pgm, \, : phs)) = a \;\Rightarrow\; \textit{forced-to-level}(a, 0)$$

The forcing interpreter

In order that more of the answer be printed, we can use the *forcing-interpreter*, which allows the user to specify, at execution-time, a level to which it requires the answer to be printed.

In the case of the parallel printing process, we get the following specification:

Specification 3

$$\forall k \geq 0 \cdot$$
$$\quad \mathit{forcing\text{-}interpreter}(\mathit{basic\text{-}compiler}(\mathit{pgm},\ :\mathit{phs}),\ k) = a$$
$$\quad \Rightarrow$$
$$\quad \mathit{forced\text{-}to\text{-}level}(a, k)$$

The speculative compiler

Finally, we can also indicate, at compile-time, a speculation level. This ensures that, even without any forcing, more of the answer will be obtained than when using the basic compiler (without speculation). Note that the effect of the speculation level is specified only for parallel head-strictness. It is defined to have the following effect:

Specification 4

$$\forall k \geq 0 \cdot$$
$$\quad \mathit{basic\text{-}interpreter}(\mathit{speculative\text{-}compiler}(\mathit{pgm}, k)) = a$$
$$\quad \Rightarrow$$
$$\quad \mathit{forced\text{-}to\text{-}level}(a, k)$$

Compile-time speculation + run-time forcing

Of course, speculative compilation and run-time forcing can be combined. In this case, it is possible that more of the answer effectively gets evaluated than required by the printing level itself, because of the speculation. However, there is indeed a bound to this additional level of evaluation, which justifies the name *bounded speculative laziness*:

Specification 5

$$\forall k \geq 0, \mathit{pl} \geq 0 \cdot$$
$$\quad \mathit{forcing\text{-}interpreter}(\mathit{speculative\text{-}compiler}(\mathit{pgm}, k), \mathit{pl}) = a$$
$$\quad \Rightarrow$$
$$\quad \exists j \in \{0, \ldots, k\} \cdot \mathit{forced\text{-}to\text{-}level}(a, \mathit{pl} + j)$$

Appendix H

Proofs

H.1 Proofs of Chapter 2

Lemma 2.2 $x \in \mathtt{HS}_0 \;\Leftrightarrow\; has_0\, x = \perp$

In order to prove this lemma, we will need some auxiliary lemmas and facts. First, we need to prove that the predicate we are using is an *admissible* predicate, so that our structural induction proof, which will hold for finite lists, will also hold for the limit case, i.e., the infinite lists.

The predicate we need to prove admissible is the following:

$has_0\, l = \perp \;\Rightarrow\; l \in \mathtt{HS}_0$

Instead, we will prove the following more general lemma, of which our required predicate is just a special case:

Lemma H.2 *Given f a continuous function and α a c.p.o., then the following predicate is admissible (i.e., continuous):*

$P(l)$: $f\, l = \perp \;\Rightarrow\; l \in \alpha$

Proof: Proving the admissibility of P, means proving the following:

Let $l_1 \sqsubseteq l_2 \sqsubseteq \cdots$ be a chain;
Let $l = \sqcup_{i=1}^{\infty} l_i$ be the limit of the chain;
Then
$(\forall i \cdot P(l_i)) \;\Rightarrow\; P(l)$

There are two cases:

1. $f(l) \neq \perp$:

This implies that $P(l) = \textsf{true}$, so that the lemma is indeed true.

2. $f(l) = \perp$:

By continuity of f, this means:

$f(\sqcup_{i=1}^{\infty} l_i) = \sqcup_{i=1}^{\infty} f(l_i) = \perp$

This implies:

$\forall i \cdot f(l_i) = \perp$

Thus, we have:

$\forall i \cdot P(l_i) =$

$\forall i \cdot f(l_i) = \perp \;\Rightarrow\; l_i \in \alpha =$

$\forall i \cdot \textsf{true} \;\Rightarrow\; l_i \in \alpha =$

$\forall i \cdot l_i \in \alpha$

together with:

$P(l) =$

$f(l) = \perp \;\Rightarrow\; l \in \alpha =$

$\textsf{true} \;\Rightarrow\; l \in \alpha =$

$l \in \alpha$

This then becomes equivalent to the following:

$\forall i \cdot l_i \in \alpha \;\Rightarrow\; l \in \alpha$

But then, this is straightforward from the fact that α is a c.p.o.

$\Box$ (End of proof of Lemma H.2)

We are now ready to prove Lemma 2.2.

Proof:

1. If part, i.e., $x \in \mathtt{HS}_0 \;\Rightarrow\; has_0\, x = \perp$: The proof will be omitted, as it is clear that it holds from a straightforward examination of the definition of has_0 and of $\mathtt{HS}_0$.

2. Only if part, i.e., $\forall l \cdot has_0\, l = \perp \;\Rightarrow\; l \in \mathtt{HS}_0$: The proof is by structural induction on l.

Base cases:

i) $l = \perp$: In this case, we do have that $has_0\, l = \perp \wedge l \in \mathtt{HS}_0$.

ii) $l = \textsf{nil}$: In this case, $has_0\, l \neq \perp$, so the predicate is vacuously true.

Inductive case:

iii) $l = (x\colon xs)$:

Assume the lemma holds for xs.

There are various cases depending on the value of x:

- $x = \bot$:

  $has_0(\bot\colon xs) = \bot$ and $(\bot\colon xs) \in \mathtt{HS}_0$, for any xs.
- $x = 0$:

  $has_0(0\colon xs) = 0 \neq \bot$, so there is nothing to check.
- $x \in \{\text{-}1, \text{-}2, \ldots, 1, 2, \ldots\}$:

  $has_0(x\colon xs) = has_0\ xs$

  (i) $has_0\ xs \neq \bot$

  $\Rightarrow\ has_0\ xs \neq \bot$

  (ii) $has_0\ xs = \bot$

  By the induction hypothesis, this means that $xs \in \mathtt{HS}_0$.

  By Lemma H.3 (cf. below), we then have that $(x\colon xs) \in \mathtt{HS}_0$.

□ (End of proof of Lemma 2.2)

Lemma H.3 $x \neq \bot \wedge x \neq 0 \wedge x \in \mathtt{HS}_0 \ \Rightarrow\ (x\colon xs) \in \mathtt{HS}_0$

Proof:

From the definition of $\mathtt{HS}_0$, we have:

$$\{(x\colon xs) \mid x \neq \bot, x \neq 0, xs \in \mathtt{HS}_0\} =$$
$$\mathbb{Z}^0 . \mathbb{Z}_\bot^{0\,*} . \bot \cup \mathbb{Z}^0 . \mathbb{Z}_\bot^{0\,\omega} \cup$$
$$\mathbb{Z}^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^* . \bot \cup \mathbb{Z}^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^\omega \cup$$
$$\mathbb{Z}^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^* . \mathsf{nil}$$

But then, this set is clearly a subset of the following:

$$\mathbb{Z}_\bot^0 . \mathbb{Z}_\bot^{0\,*} . \bot \cup \mathbb{Z}_\bot^0 . \mathbb{Z}_\bot^{0\,\omega} \cup$$
$$\mathbb{Z}_\bot^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^* . \bot \cup \mathbb{Z}_\bot^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^\omega \cup$$
$$\mathbb{Z}_\bot^0 . \mathbb{Z}_\bot^{0\,*} . \bot . \mathbb{Z}_\bot{}^* . \mathsf{nil}$$

In turn, this set is a subset of $\mathtt{HS}_0$, because for any X:

$X . X^* \subset X^*$ and $X . X^\omega \subset X^\omega$

□ (End of proof of Lemma H.3)

Lemma 2.3 $\forall i, j \in \mathbb{Z} \cdot i \neq j \;\Rightarrow\; \mathtt{HS}_i \neq \mathtt{HS}_j$

Proof: Straightforward, as it suffices to show that some element is a member of one set but of not of the other. For example, whenever $i \neq j$, we have:

$(i \,.\, \perp \,.\, \mathsf{nil}) \notin \mathtt{HS}_i$ but $(i \,.\, \perp \,.\, \mathsf{nil}) \in \mathtt{HS}_j$

□ (End of proof of Lemma 2.3)

Lemma 2.4 *For any two abstract values a and b:*

$$conc(a) \neq conc(b) \;\Rightarrow\; a \neq b$$

Proof: Straightforward, from the definition of *conc*, using the contraposite. Assume $a = b$. Then, we have the following:

$$\begin{aligned} & conc(a) \\ & \quad = \{v \mid abs(v) \sqsubseteq a\}^{\circ} \\ & \quad = \{v \mid abs(v) \sqsubseteq b\}^{\circ} \\ & \quad = conc(b) \end{aligned}$$

□ (End of proof of Lemma 2.4)

H.2 Proofs of Chapter 3

H.2.1 Proofs of Section 3.2

Proofs of Section 3.2.4

Fact 3.4

$$\forall v, d_1, d_2 \cdot satisfy(v, d_1) \wedge satisfy(v, d_2) \;\Leftrightarrow\; satisfy(v, d_1 \oplus d_2)$$

Proof:

Let $d = d_1 \oplus d_2$ with $conc(d) = \{v_1 \sqcup v_2 \mid v_1 \in conc(d_1) \wedge v_2 \in conc(d_2) \wedge v_1 \sim v_2\}$.
1. $\boxed{\Rightarrow}$: Take some v such that $satisfy(v, d)$. This means there are some v_1, v_2 such that $v \sqsupseteq v_1 \sqcup v_2$, with $v_1 \sim v_2$ and $v_1 \sqcup v_2 \in conc(d)$, and $v_1 \in conc(d_1)$ and $v_2 \in conc(d_2)$, by definition of $\oplus$.

By definition of $\sqcup$, we then have $v \sqsupseteq v_1$ and $v \sqsupseteq v_2$. Because $v_1 \in conc(d_1)$, then $satisfy(v, d_1)$ holds, and similarly for $satisfy(v, d_2)$.
2. $\boxed{\Leftarrow}$: From $satisfy(v, d_1)$, we have that $v \sqsupseteq v_1$ for some $v_1 \in conc(d_1)$ and similarly for some v_2 in d_2.

From our domain of value, any two values having an upper bound effectively have a *least* upper bound. Thus, this means $v_1 \sim v_2$, so that $v \sqsupseteq v_1 \sqcup v_2$. But from the definition of $\oplus$, $v_1 \sqcup v_2 \in conc(d_1 \oplus d_2)$, so that $satisfy(v, d_1 \oplus d_2)$.
$\square$ (End of proof of Fact 3.4)

Remark: Before proving the next theorem (Theorem 3.5), which uses fixed point induction, it should be ensured that the predicates being used are all *admissible*, i.e., *inclusive*. [1] In the case of the predicate *is-inverse* (Defn 3.16, p. 51) this can be obtained from an application of a proposition by Manna, Ness & Vuillemin [Sch86, p. 130], using equivalences such as those mentioned in [Sch86, p. 130]. As for the predicate $\textit{is-inverse}_{bve,fve}$ (Defn 3.15, p. 50), the fact that there are no infinite chain in the domain *DmdVec* ensures that the predicate is effectively inclusive: the least upper-bound of any chain will be one of the element of the chain, elements for which the predicate is assumed to hold, thus implying that it also holds for the least upper-bound.

Theorem 3.5

For any program p:

$\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$

Proof: The proof is by fixed point (computational) induction on $\overline{fve}^{(k)}_{[\![p]\!]}$.

- *Base case*:
  - $\overline{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \bot\}$

[1] Cf. [Sch86, p. 128] for the definition of an inclusive predicate.

In this case, we clearly have that $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}^{(0)}_{[\![p]\!]})$ holds, because, for any f_i, the following always holds, so the antecedent of $\textit{is-inverse}_{bve,fve_{[\![p]\!]}}$ is always false:

$$\overline{fve}^{(0)}_{[\![p]\!]}[\![f_i]\!]\; d\,(v_1,\,\ldots,\,v_{n_i}) = \perp \neq [d_1\,\cdots\,d_{n_i}]$$

- *Inductive case*: Assume that we have $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}^{(k)}_{[\![p]\!]})$.

  We now show that $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}^{(k+1)}_{[\![p]\!]})$ effectively holds. We have:

  - $\overline{fve}^{(k+1)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1,\,\ldots,\,z_{n_i}) \cdot \overline{\mathcal{E}}_{n_i}[\![e_i]\!] d[z_j \mid 1 \leq j \leq n_i] fve_{[\![p]\!]}\, \overline{fve}^{(k)}_{[\![p]\!]}\}$.

  Now, take some arbitrary $f_i \in \mathsf{dom}\, fve_{[\![p]\!]}$, $d \in Dmd$, $v_1, \ldots, v_{n_i} \in Value$. Given the above definitions, we thus need to show the following:

  $$\begin{array}{l}\textit{is-inverse}_{bve,fve_{[\![p]\!]}}([\![f_i(x_1,\,\ldots,\,x_{n_i})]\!], d, \overline{fve}^{(k+1)}_{[\![p]\!]}[\![f_i]\!]\; d\,(v_1,\,\ldots,\,v_{n_i}))\\ \quad \text{where } bve = [v_j \mid i \leq j \leq n_i]\end{array}$$

  By definition of $\overline{fve}_{[\![p]\!]}$, we have:

  $$\overline{fve}^{(k+1)}_{[\![p]\!]}[\![f_i]\!]\; d\,(v_1,\,\ldots,\,v_{n_i}) = \overline{\mathcal{E}}_{n_i}[\![e_i]\!]\; d\,[v_j \mid 1 \leq j \leq n_i]\, fve_{[\![p]\!]}\, \overline{fve}^{(k)}_{[\![p]\!]}$$

  By definition of $\mathcal{E}$ and $fve_{[\![p]\!]}$, we have:

  $$\begin{array}{l}\mathcal{E}[\![f_i(x_1,\,\ldots,\,x_{n_i})]\!][v_j \mid 1 \leq j \leq n_i] fve_{[\![p]\!]}\\ = fve_{[\![p]\!]}[\![f_i]\!](\mathcal{E}[\![x_1]\!][v_j \mid 1 \leq j \leq n_i] fve_{[\![p]\!]}, \ldots, \mathcal{E}[\![x_{n_i}]\!][v_j \mid 1 \leq j \leq n_i] fve_{[\![p]\!]})\\ = fve_{[\![p]\!]}[\![f_i]\!]\,(v_1,\,\ldots,\,v_{n_i})\\ = \mathcal{E}[\![e_i]\!][v_j \mid 1 \leq j \leq n_i] fve_{[\![p]\!]}\end{array}$$

  From the equivalences presented above and the definition of *is-inverse*, what we need to prove is equivalent to the following:

  $$\begin{array}{l}\textit{is-inverse}_{bve,fve_{[\![p]\!]}}(e_i, d, \overline{\mathcal{E}}_{n_i}[\![e_i]\!]\; d\; bve\; fve_{[\![p]\!]}\, \overline{fve}^{(k)}_{[\![p]\!]})\\ \quad \text{with } bve = [v_j \mid 1 \leq j \leq n_i]\end{array}$$

  By the induction hypothesis ($\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}^{(k)}_{[\![p]\!]})$) and Lemma 3.6 (cf. below), we have the required result.

□ (End of proof of Theorem 3.5)

Before proving Lemma 3.6, we need to present two useful facts. Note that a generalization of these two facts to an arbitrary number of elements could also be stated (and, in fact, will be used within later proofs). Both would hold directly from the commutativity and associativity of $\oplus$.

Fact H.8

$\overline{d_1} \,\overline{\oplus}\, \overline{d_2} = \bot \;\Leftrightarrow\; \overline{d_1} = \bot \vee \overline{d_2} = \bot$

Proof: Straightforward from the definition of $\overline{\oplus}$. $\square$

Fact H.9

$\overline{d_1} \,\overline{\oplus}\, \overline{d_2} = [d_1, \ldots, d_k] \neq \bot$

$\Leftrightarrow$

$(\overline{d_1} = [d'_1, \ldots, d'_k] \neq \bot) \wedge (\overline{d_2} = [d''_1, \ldots, d''_k] \neq \bot) \wedge$
$(\forall j \in \{1, \ldots, k\} \cdot d'_j \oplus d''_j = d_j \neq \bot)$

Proof: Straightforward from the definition of $\overline{\oplus}$. $\square$

Lemma 3.6

$\forall e, d, bve, fve, \overline{fve} \cdot$
$\quad is\text{-}inverse(fve, \overline{fve})$
$\quad\quad \Rightarrow$
$\quad is\text{-}inverse_{bve,fve}(e, d, \overline{\mathcal{E}}_k[\![e]\!]\, d\; bve\; fve\; \overline{fve})$
$\quad\quad$ where
$\quad\quad\quad k = \mathsf{len}\; bve$

Proof: The proof is by structural induction on e, with a case analysis for the possible values of d.

There are two cases depending on the value of d:

A) $d = @$:

$\overline{\mathcal{E}}_k[\![e]\!]\, @\, bve\, fve\, \overline{fve} = [d_1, \ldots, d_k] \;\Rightarrow\; d_j = @ \;\; (j = 1, \ldots, k)$ (By defn. of $\overline{\mathcal{E}}_k$)

By definition of $conc(@)$, for any $v \in conc(@)$, we have $v = \bot$. Thus, we have, by monotonicity and by definition of $\bot$ (i.e., $v_j \sqsupseteq \bot$):

$\mathcal{E}[\![e]\!][\bot \mid 1 \leq j \leq k]\, fve \sqsupseteq \bot \;\Rightarrow\; \mathcal{E}[\![e]\!][v_j \mid 1 \leq j \leq k]\, fve \sqsupseteq \bot$

Thus, there clearly exist $u_j \in conc(@)$ with $u_j \sqsubseteq v_j$ which make it possible to obtain a result at least as defined as $v = \bot$.

$B)$ $d = \$$:

Base cases:

1. $[\![e]\!] = [\![c]\!]$:

$-\mathcal{E}[\![c]\!][v_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v \wedge v \in conc(\$) = \mathbb{Z} \;\Rightarrow\; v = c \qquad (\mathbb{Z}_\bot \text{ is flat})$

$-\overline{\mathcal{E}}_k[\![c]\!]\$[\bot \mid 1 \leq j \leq k] fve\overline{fve} = [d_1, \ldots, d_k] \;\Rightarrow\; d_j = @ \qquad (\text{By defn. of } \overline{\mathcal{E}}_k)$

So, by definition of $\mathcal{E}$, we do have:

$\exists u_1 \in conc(d_1), \ldots, u_k \in conc(d_k) \cdot$
$\quad \mathcal{E}[\![c]\!][u_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v \;\wedge\; u_j \sqsubseteq v_j \;\; (j = 1, \ldots, k)$

2. $[\![e]\!] = [\![x_i]\!]$:

$-\mathcal{E}[\![x_i]\!][v_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v \;\Rightarrow\; v_i \sqsupseteq v \qquad (\text{By definition of } \mathcal{E})$

$-\overline{\mathcal{E}}_k[\![x_i]\!]\$[v_j \mid 1 \leq j \leq k]\, fve\, \overline{fve} = [d_1, \ldots, d_k] \;\Rightarrow\; d_i = \$, d_j = @\, (\text{for } j \neq i)$

$-$Let $u_i = v_i, u_j = \bot(\text{for } j \neq i)$.
Then:
$\mathcal{E}[\![x_i]\!][u_j \mid 1 \leq j \leq k]\, fve = v_i \sqsupseteq v$
$\quad$ with $u_j \sqsubseteq v_j \qquad (u_i \sqsubseteq v_i, \bot \sqsubseteq v_j)$
$\quad$ and $u_j \in conc(d_j) \qquad (\bot \in conc(@), v_i \in conc(\$) = \mathbb{Z})$

Inductive cases:

3. $[\![e]\!] = [\![e_1 + e_2]\!]$:

$-$Suppose $\overline{\mathcal{E}}_k[\![e_1 + e_2]\!]\; \$\; bve\; fve\; \overline{fve} = [d_1, \ldots, d_k]$, then:
$\quad \exists [d_1', \ldots, d_k'], [d_1'', \ldots, d_k''] \cdot$
$\quad\quad \overline{\mathcal{E}}_k[\![e_1]\!]\; \$\; bve\; fve\; \overline{fve} = [d_1', \ldots, d_k']$
$\quad\quad \wedge\, \overline{\mathcal{E}}_k[\![e_2]\!]\; \$\; bve\; fve\; \overline{fve} = [d_1'', \ldots, d_k'']$
$\quad\quad \wedge [d_1, \ldots, d_k] = [d_1', \ldots, d_k'] \overline{\oplus} [d_1'', \ldots, d_k''] \qquad (\text{By defn. of } \overline{\mathcal{E}}_k \text{ and Fact } H.9)$

–Take some $v \in conc(\$)$, $v_1, \ldots, v_k \in \mathbb{Z}_\perp$ such that the following holds:

$\mathcal{E}[\![e_1 + e_2]\!][v_j \mid 1 \le j \le k]\, fve \sqsupseteq v$

Then we will have, for some v', v'' such that $v' + v'' = v$, by definition of $\mathcal{E}$:

$\mathcal{E}[\![e_1]\!][v_j \mid 1 \le j \le k]\, fve \sqsupseteq v'$

which implies, by induction hypothesis:

$\exists u'_1 \in conc(d_1), \ldots, u'_k \in conc(d_k) \cdot$

$\mathcal{E}[\![e_1]\!][u'_j \mid 1 \le j \le k]\, fve \sqsupseteq v'$

with $u'_j \sqsubseteq v_j \quad (j = 1, \ldots, k)$

And similarly for $e_2 / v'' / u''_j$

–$u'_j \sqcup u''_j$ exists, because $u'_j \sqsubseteq v_j \wedge u''_j \sqsubseteq v_j$.

Let $u_j = u'_j \sqcup u''_j$. Then we will have, by monotonicity:

$\mathcal{E}[\![e_1]\!][u_j \mid 1 \le j \le k]\, fve \sqsupseteq v'$ and $\mathcal{E}[\![e_2]\!][u_j \mid 1 \le j \le k]\, fve \sqsupseteq v''$

which implies:

$\mathcal{E}[\![e_1 + e_2]\!][u_j \mid 1 \le j \le k]\, fve \sqsupseteq v$

–Because we also have the properties below, we get the required result:

$u_j \sqsubseteq v_j \qquad (u'_j \sqsubseteq v_j \wedge u''_j \sqsubseteq v_j \;\Rightarrow\; u_j = u'_j \sqcup u''_j \sqsubseteq v_j)$

$u_j \in conc(d_j) \qquad$ (By Fact 3.3)

4. $[\![e]\!] = [\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]$:

Here, $\overline{\mathcal{E}}_k[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\$\ bve\ fve\ \overline{fve} = [d_1, \ldots, d_k] \;\Rightarrow\; \mathcal{E}[\![e_1]\!]\ bve\ fve \ne \perp$, i.e., it evaluates to some integer value v such that either $v \ne 0$ or $v = 0$. In each case, the reasoning would then be identical to the case for $[\![e]\!] = [\![e_1 + e_2]\!]$.

5. $[\![e]\!] = [\![f_i(e_1, \ldots, e_{n_i})]\!]$:

–Suppose $\overline{\mathcal{E}}_k[\![f_i(e_1, \ldots, e_{n_i})]\!]\ \$\ bve\ fve\ \overline{fve} = [d_1, \ldots, d_k]$

Take any $v \in conc(\$)$ such that $\mathcal{E}[\![f_i(e_1, \ldots, e_{n_i})]\!] bve\ fve \sqsupseteq v$.

Then, by definition of $\mathcal{E}$, we will have:

$\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!][\mathcal{E}[\![e_j]\!]\ bve\ fve \mid 1 \le j \le n_i]\, fve \sqsupseteq v$

–Given the induction hypothesis $\textit{is-inverse}(fve, \overline{fve})$, we then have:

$\textit{is-inverse}_{bve,fve}(f_i(x_1, \ldots, x_{n_i}), \$, \overline{fve}[\![f_i]\!]\ \$\ (v_1, \ldots, v_{n_i}))$

where $bve = [v_j \mid 1 \le j \le n_i]$

and $v_j = \mathcal{E}[\![e_j]\!]\ bve\ fve \quad (j = 1, \ldots, n_i)$

–Suppose $\overline{fve}[\![f_i]\!]\ \$\ (v_1,\ \ldots,\ v_{n_i}) = [d_1,\ \ldots,\ d_{n_i}]$

By definition of *is-inverse*, we then have:

(1) $\exists u_1 \in conc(d_1), \ldots, u_{n_i} \in conc(d_{n_i}) \cdot$

(2) $\mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][u_j \mid 1 \le j \le n_i]\, fve \sqsupseteq v$

and

(3) $u_j \sqsubseteq v_j = \mathcal{E}[\![e_j]\!]\ bve\ fve \quad (j = 1, \ldots, n_i)$

–By structural induction hypothesis, we have, for $j = 1, \ldots, n_i$:

$is\text{-}inverse_{bve,fve}(e_j, d_j, \overline{\mathcal{E}}_k[\![e_j]\!]\, d_j\ bve\ fve\ \overline{fve})$

Let $\overline{\mathcal{E}}_k[\![e_j]\!]\, d_j\ bve\ fve\ \overline{fve} = [d_{j1}, \ldots, d_{jk}] \quad (j = 1, \ldots, n_i)$

so that: $\overline{\bigoplus}\, \{\overline{\mathcal{E}}_k[\![e_j]\!]\, d_j\ bve\ fve\ \overline{fve}\} = [d_1, \ldots, d_k]$

(i.e., $d_l = \overline{\bigoplus}\, \{d_{jl} \mid 1 \le j \le n_i\} \quad (l = 1, \ldots, k)$)

From $u_j \in conc(d_j)$, $\mathcal{E}[\![e_j]\!]\ bve\ fve \sqsupseteq u_j$ and the definition of *is-inverse*:

(By generalization of Fact $H.8$)

$\exists u_{j1} \in conc(d_{j1}), \ldots, u_{jk} \in conc(d_{jk}) \cdot$

(4) $\mathcal{E}[\![e_j]\!][u_{jl} \mid 1 \le l \le k] \sqsupseteq u_j$

and

$u_{j1} \sqsubseteq bve(1), \ldots, u_{jk} \sqsubseteq bve(k)$

$(j = 1, \ldots, n_i)$

–Now, by monotonicity:

$\mathcal{E}[\![e_j]\!][u_{jl} \mid 1 \le l \le k]\, fve \sqsupseteq u_j \quad (j = 1, \ldots, n_i)$

and

$\mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][u_j \mid 1 \le j \le n_i]\, fve \sqsupseteq v$

$\Rightarrow$

$\mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][\mathcal{E}[\![e_j]\!][u_{jl} \mid 1 \le l \le k]\, fve \mid 1 \le j \le n_i]\, fve \sqsupseteq v$

–Let $u_l = \bigsqcup_{j=1}^{n_i} u_{jl} \quad (l = 1, \ldots, k)$

Again, by monotonicity and definition of $\mathcal{E}$, we have:

$\mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][\mathcal{E}[\![e_j]\!][u_l \mid 1 \le l \le k]\, fve \mid 1 \le j \le n_i]\, fve \sqsupseteq v$

which implies:

$\mathcal{E}[\![f_i(e_1,\ \ldots,\ e_{n_i})]\!][u_l \mid 1 \le l \le k]\, fve \sqsupseteq v$

–At this point, what we now need to show is the following:

$u_{jl} \in conc(d_{jl}) \quad (l = 1, \ldots, k; j = 1, \ldots, n_i)$

$\wedge\, [d_1 \ \cdots \ d_k] = \overline{\bigoplus} \{[d_{j1}, \ldots, d_{djk}] \mid 1 \leq j \leq n_i\}$

$\Rightarrow$

$\bigsqcup_{j=1}^{n_i} u_{jl} \in conc(d_l) \quad (l = 1, \ldots, k)$

This can be proved from the definition of $\overline{\oplus}$ and $\oplus$ (Defn. 3.11 and Fact 3.3), from which we obtain the required result.

$\Box$ (End of proof of Lemma 3.6)

Proofs of Section 3.2.6

Remark: The next theorem (Theorem 3.7) uses fixed point induction, so it must be ensured that the predicates being used are all inclusive. That this is effectively the case follows from straigthforward applications of a proposition by Manna, Ness & Vuillemin [Sch86, p. 130], together with the fact that neither the exact nor the abstract domain of demand vectors contain infinite chains, so that the least upper-bound of any chain will necessarily be one of the element of the chain, for which the predicate is assumed to hold.

Theorem 3.7

For any program p:

$\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$

Proof: The proof is by fixed point (computational) induction.

- *Base case*: Let's take the bottom element of each of the function environment domain:

  - $\overline{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1, \ \ldots, \ z_{n_i}) \cdot \bot\}$
  - $\widetilde{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda\delta \cdot [\phi \mid 1 \leq j \leq n_i]\}$

  We have $\overline{fve}^{(0)}_{[\![p]\!]} \lhd \widetilde{fve}^{(0)}_{[\![p]\!]}$, as the antecedent of $\lhd$ is false.

- *Inductive case*: Assume we have $\overline{fve}^{(k)}_{[\![p]\!]} \lhd \widetilde{fve}^{(k)}_{[\![p]\!]}$.

  Take some arbitrary f_i, $d \lhd \delta$, $u_1, \ \ldots, \ u_{n_i}$. From the definitions of $\overline{fve}_{[\![p]\!]}$ and $\widetilde{fve}_{[\![p]\!]}$, we get:

- $\overline{fve}^{(k+1)}_{[\![p]\!]} [\![f_i]\!]\; d\; (u_1,\; \ldots,\; u_{n_i}) = \overline{\mathcal{E}}_{n_i} [\![e_i]\!]\; d\; [u_j \mid 1 \leq j \leq n_i]\; fve_{[\![p]\!]}\; \overline{fve}^{(k)}_{[\![p]\!]}$.
- $\widetilde{fve}^{(k+1)}_{[\![p]\!]} [\![f_i]\!]\; \delta = \widetilde{\mathcal{E}}_{n_i} [\![e_i]\!]\; \delta\; \widetilde{fve}^{(k)}_{[\![p]\!]}$.

  By induction hypothesis, the fact that $d \lhd \delta$, and by Lemma 3.8 (cf. below), we get the required result.

□ (End of proof of Theorem 3.7)

Before presenting Lemma 3.8, we state a number of facts involving negative demands and facts stating properties of an auxiliary predicate which will be defined ($\overline{\in}$).

Fact H.12

$\overline{\mathcal{E}}_k [\![e]\!]\; @\; bve\; fve\; \overline{fve} = [@ \mid 1 \leq j \leq k]$

Proof: Direct from the definition of $\overline{\mathcal{E}}_k$. □

Fact H.13

$@ \lhd \delta \;\Rightarrow\; \widetilde{\mathcal{E}}_k [\![e]\!]\; \delta\; \widetilde{fve} = [\delta_1,\; \ldots,\; \delta_k] \wedge \forall j \in \{1, \ldots, k\} \cdot @ \lhd \delta_j$

Proof: Straightforward from the definition of $\widetilde{\mathcal{E}}_k$. □

The auxiliary definition presented below is useful in writing of the proof of Lemma 3.8. It defines a predicate relating a non-$\perp$ vector of exact demands with a vector of abstract ones. Note that, using this auxiliary definition, the predicate $\overline{d} \lhd \widetilde{\delta}$ becomes equivalent to $\overline{d} \neq \perp \;\Rightarrow\; \overline{d} \,\overline{\in}\, \widetilde{\delta}$.

Definition H.1

$$\overline{\in} : DmdVec_k \times ADmdVec_k \rightarrow \mathbb{B}$$

$$\overline{\in}\,(\overline{d}, \widetilde{\delta}) \quad \triangleq \quad \forall j \in \{1, \ldots, k\} \cdot \overline{d}(j) \lhd \widetilde{\delta}(j)$$

$$\mathsf{pre}\ \overline{d} \neq \bot$$

Fact H.14

$$\overline{d}_1 \,\overline{\in}\, \widetilde{\delta}_1 \wedge \overline{d}_2 \,\overline{\in}\, \widetilde{\delta}_2 \;\Leftrightarrow\; \overline{d}_1 \,\overline{\oplus}\, \overline{d}_2 \,\overline{\in}\, \widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2$$

Proof: Straightforward from the definitions of $\oplus$, $\overline{\oplus}$, $\widetilde{\oplus}$ and $\overline{\in}$. □

Fact H.15

$$\forall \widetilde{\delta}, \overline{d_1}, \overline{d_2}, \widetilde{\delta}_1, \widetilde{\delta}_2 \cdot$$
$$\overline{d_1} \,\overline{\in}\, \widetilde{\delta}_1 \wedge \overline{d_2} \,\overline{\in}\, \widetilde{\delta}_2 \;\Rightarrow\; \overline{d_1} \,\overline{\oplus}\, \overline{d_2} \,\overline{\in}\, \widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2 \,\widetilde{\cup}\, \widetilde{\delta}$$

Proof: By Fact H.14 we have:

$$\overline{d_1} \,\overline{\oplus}\, \overline{d_2} \,\overline{\in}\, \widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2$$

But then, by the definition of $\widetilde{\cup}$ we clearly have that:

$$\forall j \cdot (\widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2)(j) \subseteq (\widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2)(j) \cup \widetilde{\delta}(j) = (\widetilde{\delta}_1 \,\widetilde{\oplus}\, \widetilde{\delta}_2 \,\widetilde{\cup}\, \widetilde{\delta})(j)$$

□

Lemma 3.8

$$\forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd \widetilde{fve} \cdot$$
$$\forall d, \delta \cdot d \lhd \delta \cdot$$
$$\forall k, bve \cdot \mathsf{len}\, bve = k \cdot$$
$$\overline{\mathcal{E}}_k[\![e]\!]\; d\; bve\; fve\; \overline{fve} \;\; \lhd \;\; \widetilde{\mathcal{E}}_k[\![e]\!]\; \delta\; \widetilde{fve}$$

Proof: The proof is by structural induction on e, with two possibilities depending on the value of d:

1) $d = @$: We have $@ \lhd \delta$. By Facts H.12 and H.13 (cf. above) and definition of $\lhd$ (Defn. 3.23, p. 61), we do have:

$$\overline{\mathcal{E}}_k[\![e_i]\!]\; @\;\; bve\; fve\; \overline{fve} = [@ \mid 1 \leq j \leq k] \;\; \lhd \;\; \widetilde{\mathcal{E}}_k[\![e_i]\!]\; \delta\; \widetilde{fve}_{[\![p]\!]}$$

2) $d \neq @$, i.e., $d = \$$:

Base cases:

1. $[\![e]\!] = [\![c]\!]$:

   Straightforward, as $@ \lhd @$:
   $[@ \mid 1 \leq i \leq k] \lhd [@ \mid 1 \leq i \leq k]$

2. $[\![e]\!] = [\![x_i]\!]$:

   Straightforward, as $@ \lhd @$ and $d \lhd \delta$:
   $[@ \mid 1 \leq j < i] \curvearrowright [d] \curvearrowright [@ \mid i < j \leq k] \lhd [@ \mid 1 \leq j < i] \curvearrowright [\delta] \curvearrowright [@ \mid i < j \leq k]$

Inductive cases:

3. $[\![e]\!] = [\![e_1 + e_2]\!]$:

   Let $\overline{d_1}, \overline{d_2}, \tilde{\delta}_1, \tilde{\delta}_2$ be defined as follows:
   $\overline{d_j} = \overline{\mathcal{E}}_k[\![e_j]\!]\ d\ bve\ fve\ \overline{fve} \quad (j = 1, 2)$
   $\tilde{\delta}_j = \tilde{\mathcal{E}}_k[\![e_j]\!]\ \delta\ \widetilde{fve} \quad (j = 1, 2)$
   By the induction hypothesis and the definition of $\lhd$ we have:
   $(\overline{d_1} \neq \bot \ \Rightarrow\ \overline{d_1} \,\overline{\in}\, \tilde{\delta}_1) \wedge (\overline{d_2} \neq \bot \ \Rightarrow\ \overline{d_2} \,\overline{\in}\, \tilde{\delta}_2)$
   From this, we can infer:
   $(\overline{d_1} \neq \bot \wedge \overline{d_2} \neq \bot) \ \Rightarrow\ (\overline{d_1} \,\overline{\in}\, \tilde{\delta}_1 \wedge \overline{d_2} \,\overline{\in}\, \tilde{\delta}_2)$
   Now, because we have (from Fact $H.9$):
   $(\overline{d_1} \neq \bot \wedge \overline{d_2} \neq \bot) \ \Leftrightarrow\ (\overline{d_1} \,\overline{\oplus}\, \overline{d_2} \neq \bot)$
   and by using Fact H.14 (cf. above), we get the required result:
   $\overline{d_1} \,\overline{\oplus}\, \overline{d_2} \neq \bot \ \Rightarrow\ \overline{d_1} \,\overline{\oplus}\, \overline{d_2} \,\overline{\in}\, \tilde{\delta}_1 \,\tilde{\oplus}\, \tilde{\delta}_2$

4. $[\![e]\!] = [\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]$:

   There are three subcases depending on the outcome of $\mathcal{E}[\![e_1]\!]\ bve\ fve$:

   (i) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \bot$:

   In this case, we will have: $\overline{\mathcal{E}}_k[\![e]\!] d\ bve\ fve\ \overline{fve} = \bot$, so there is nothing to prove.

(ii) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \mathsf{true}$:

Let $\overline{d_1}, \overline{d_2}, \overline{d_3}, \widetilde{\delta}_1, \widetilde{\delta}_2, \widetilde{\delta}_3$ be defined as follows:

$\overline{d_i} = \overline{\mathcal{E}}_k[\![e_i]\!]\ d\ bve\ fve\ \overline{fve} \quad (i = 1, 2, 3)$

$\widetilde{\delta}_i = \widetilde{\mathcal{E}}_k[\![e_i]\!]\ \delta\ \widetilde{fve} \quad (i = 1, 2, 3)$

By the induction hypothesis and the definition of $\lhd$ we have:

$(\overline{d_1} \neq \perp \ \Rightarrow\ \overline{d_1} \mathrel{\overline{\in}} \widetilde{\delta}_1) \wedge (\overline{d_2} \neq \perp \ \Rightarrow\ \overline{d_2} \mathrel{\overline{\in}} \widetilde{\delta}_2)$

From this, we can infer:

$(\overline{d_1} \neq \perp \wedge \overline{d_2} \neq \perp) \ \Rightarrow\ (\overline{d_1} \mathrel{\overline{\in}} \widetilde{\delta}_1 \wedge \overline{d_2} \mathrel{\overline{\in}} \widetilde{\delta}_2)$

Now, because $(\overline{d_1} \neq \perp \wedge \overline{d_2} \neq \perp) \ \Leftrightarrow\ (\overline{d_1} \mathbin{\overline{\oplus}} \overline{d_2} \neq \perp)$ (Fact $H.9$) we get:

$\overline{d_1} \mathbin{\overline{\oplus}} \overline{d_2} \neq \perp \ \Rightarrow\ \overline{d_1} \mathrel{\overline{\in}} \widetilde{\delta}_1 \wedge \overline{d_2} \mathrel{\overline{\in}} \widetilde{\delta}_2$

Using Fact H.15 ($\widetilde{\delta} = \widetilde{\delta}_1 \widetilde{\oplus} \widetilde{\delta}_3$) and the definition of $\overline{\mathcal{E}}_k$, we get the required result.

(iii) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \mathsf{false}$:

Similar to case (ii).

5. $[\![e]\!] = [\![f\,(e_1,\ \ldots,\ e_n)]\!]$:

Let $\overline{d_f}, \widetilde{\delta_f}$ be defined as follows:

$\overline{d_f} = \overline{fve}[\![f]\!]\ d\ (\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots, \mathcal{E}[\![e_n]\!]\ bve\ fve)$

$\widetilde{\delta_f} = \widetilde{fve}[\![f]\!]\ \delta$

From $\overline{fve} \lhd \widetilde{fve}$ and $d \lhd \delta$ we have:

$\overline{d_f} \neq \perp \ \Rightarrow\ \overline{d_f} \mathrel{\overline{\in}} \widetilde{\delta_f}$

Now, there are two possible cases:

(i) $\overline{d_f} = \perp$

In this case, we have:

$\overline{\mathcal{E}}_k[\![f\,(e_1,\ \ldots,\ e_n)]\!]\ d\ bve\ fve\ \overline{fve} = \perp$

so the safety definition holds vacuously.

(ii) $\overline{d_f} \neq \perp$

In this case, we will have:

$\overline{d_f} = [d_i \mid 1 \leq i \leq n]$

$\widetilde{\delta_f} = [\delta_i \mid 1 \leq i \leq n]$

with $\overline{d_f} \mathrel{\overline{\in}} \widetilde{\delta_f}$ which means: $\forall i \in \{1, \ldots, n\} \cdot d_i \lhd \delta_i$

From this, the induction hypothesis and the definition of $\lhd$ we get:

$$\forall j \in \{1, \ldots, n\} \cdot$$
$$\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \neq \perp \ \Rightarrow\ \overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mathrel{\overline{\in}} \widetilde{\mathcal{E}}_k[\![e_j]\!]\ \delta_j\ \widetilde{fve}$$

From this, we can infer the following:

$$(\forall j \in \{1, \ldots, n\} \cdot \overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \neq \perp)$$
$$\Rightarrow\ (\forall j \in \{1, \ldots, n\} \cdot \overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mathrel{\overline{\in}} \widetilde{\mathcal{E}}_k[\![e_j]\!]\ \delta_j\ \widetilde{fve})$$

Using Fact H.14 and (generalized) Fact H.9, we get the required result:

$$\overline{\oplus}\ \{\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mid 1 \leq j \leq n\} \neq \perp$$
$$\Rightarrow$$
$$\overline{\oplus}\ \{\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mid 1 \leq j \leq n\} \mathrel{\overline{\in}} \widetilde{\oplus}\ \{\widetilde{\mathcal{E}}_k[\![e_j]\!]\ \delta_j\ \widetilde{fve} \mid 1 \leq j \leq n\}$$

□ (End of proof of Lemma 3.8)

Proofs of Section 3.2.7

Theorem 3.9

$$\forall p \cdot$$
$$\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\phi \mid 1 \leq j \leq n_i]$$
$$\Rightarrow$$
$$f_i \text{ is sure to loop}$$

Proof: Straightforward using Lemmas H.18 and H.20 presented below, together with the fact that $\delta \neq \phi$ and the fact that $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$ (from Theorem 3.7). □ (End of proof of Theorem 3.9)

Lemma H.18

$$\forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd \widetilde{fve} \cdot$$
$$\forall d, \delta \cdot d \lhd \delta \cdot$$
$$\widetilde{fve}[\![f_i]\!]\ \delta = [\phi \mid 1 \leq j \leq n_i]$$
$$\Rightarrow$$
$$\forall u_1, \ldots, u_{n_i} \cdot \overline{fve}[\![f_i]\!]\ d\ (u_1, \ldots, u_{n_i}) = \perp$$

Proof:

We know that $\overline{fve} \lhd \widetilde{fve}$.

This means that for any $f_i, u_1, \ldots, u_{n_i}, d \lhd \delta$ we have the following:

$\overline{d} = \overline{fve}[\![f_i]\!]\ d\ (u_1, \ldots, u_{n_i}) \lhd \widetilde{fve}[\![f_i]\!]\ \delta = \widetilde{\delta}$

By definition of $\lhd$ we have:

$\overline{d} \neq \perp \ \Rightarrow\ \overline{d}(j) \lhd \widetilde{\delta}(j)\ \ (j = 1, \ldots, n_i)$

Now, assume that $\widetilde{\delta} = [\phi \mid 1 \leq j \leq n_i]$

Clearly, as $conc(\phi) = \{\,\}$, then $\overline{d}(j) \lhd \widetilde{\delta}(j)$ can not be true.

Thus, we must have that $\overline{d} \neq \perp$ is false, i.e., $\overline{d} = \perp$!

□ (End of proof of Lemma H.18)

The following Lemma is useful to prove Lemma H.20. It indicates that when the demand computed for an argument by the exact non-standard semantics (under a positive demand) is positive, then that argument is effectively strict. More precisely, if computing the vector of demands for a function does not require the exact value of an argument — i.e., the demand computation does not depend on that argument to determine which branch of a conditional will be executed — and that argument's demand is positive, then the function is effectively strict in that position. Note that it can be seen, from the exact non-standard semantics presented in Fig. 3.7, that if an argument's value is required to resolve a conditional, than the function is effectively strict in that argument, as it will be also required by the standard semantics to resolve that same conditional. This is why only the case where a non-$\perp$ vector can be produced using a $\perp$-value for the argument is discussed by Lemma H.19.

Lemma H.19

$$\forall fve, \overline{fve} \cdot is\text{-}inverse(fve, \overline{fve}) \cdot$$
$$\forall f_i, vs = (v_1, \ldots, v_{l\text{-}1}, \bot, v_{l+1}, \ldots, v_{n_i}) \cdot$$
$$\overline{fve}[\![f_i]\!]\ \$\ vs = [d_1, \ldots, d_{l\text{-}1}] \curvearrowright [\$] \curvearrowright [d_{l+1}, \ldots, d_{n_i}]$$
$$\Rightarrow$$
$$fve[\![f_i]\!]\ vs = \bot$$

Proof:

The proof is by contradiction. Let's take some $f_i, v_1, \ldots, v_{n_i}$ such that the following holds:

$$\overline{fve}[\![f_i]\!]\ \$\ vs = \overline{d}$$
Where $\overline{d}(l) = \$ \wedge vs = (v_1, \ldots, v_{l\text{-}1}, \bot, v_{l+1}, \ldots, v_{n_i})$

From $is\text{-}inverse(fve, \overline{fve})$ (Defn. 3.16, p. 51), we must then have:

$$is\text{-}inverse_{bve,fve}([\![f_i(x_1, \ldots, x_{n_i})]\!], \$, \overline{d})$$
Where $bve = [vs(j) \mid 1 \leq j \leq n_i]$

From the definition of $is\text{-}inverse_{bve,fve}$ (Defn. 3.15, p. 50), the following must then hold:

$$\forall v \in conc(\$) \cdot$$
$$\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve \sqsupseteq v$$
$$\Rightarrow$$
$$\exists u_1 \in conc(\overline{d}(1)), \ldots, u_{n_i} \in conc(\overline{d}(n_i)) \cdot$$
$$\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!][u_j \mid 1 \leq j \leq n_i]\ fve \sqsupseteq v$$
$$\wedge$$
$$u_j \sqsubseteq v_j \ \ (j = 1, \ldots, n_i)$$

Suppose $\exists v \in conc(\$) = \mathbb{Z} \cdot \mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve \sqsupseteq v$.

Now, we have $v_l = \bot$, so that $u_l \sqsubseteq v_l \ \Rightarrow \ u_l = \bot$. However, $d_l = \$ \ \Rightarrow \ u_l = \bot \notin conc(\$) \ \Rightarrow \ \nexists u_l \in conc(\$) \cdot u_l \sqsubseteq v_l$. In turn, this must mean $\nexists v \in conc(\$) \cdot \mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve \sqsupseteq v$. Thus, we must have $\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve = \bot$!
□ (End of proof of Theorem H.19)

Lemma H.20

$$\forall p \cdot$$
$$\forall d, u_1, \ldots, u_{n_i} \cdot$$
$$\overline{fve}_{[\![p]\!]}[\![f_i]\!]\; d\; (u_1, \ldots, u_{n_i}) = \bot \;\Rightarrow\; fve_{[\![p]\!]}[\![f_i]\!](u_1, \ldots, u_{n_i}) = \bot$$

Proof:

The proof is by fixed point induction.

- *Base case*: Let's take the bottom element of each of the function environment domain:

  - $\overline{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \bot\}$
  - $fve^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda (z_1, \ldots, z_{n_i}) \cdot \bot\}$

  Thus, the lemma clearly holds in this case.

- *Inductive case*: Assume that the lemma holds for $\overline{fve}^{(k)}_{[\![p]\!]}$ and $fve^{(k)}_{[\![p]\!]}$.

  We now show that it also holds for $\overline{fve}^{(k+1)}_{[\![p]\!]}$ and $fve^{(k+1)}_{[\![p]\!]}$. These are defined as follows:

  - $\overline{fve}^{(k+1)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \overline{\mathcal{E}}_{n_i}[\![e_i]\!]\; d\; [z_j \mid 1 \leq j \leq n_i] fve^{(k)}_{[\![p]\!]}\, \overline{fve}^{(k)}_{[\![p]\!]}\}$.
  - $fve^{(k+1)}_{[\![p]\!]} = \{f_i \mapsto \lambda (z_1, \ldots, z_{n_i}) \cdot \mathcal{E}[\![e_i]\!][z_j \mid 1 \leq j \leq n_i]\, fve^{(k)}_{[\![p]\!]}\}$.

  Take some $f_i, d, u_1, \ldots, u_{n_i}$ such that $\overline{fve}^{(k+1)}_{[\![p]\!]}[\![f_i]\!]\; d(u_1, \ldots, u_{n_i}) = \bot$. Given the above definitions, this means:

  $$\overline{\mathcal{E}}_{n_i}[\![e_i]\!]\; d[u_j \mid 1 \leq j \leq n_i]\, fve^{(k)}_{[\![p]\!]}\, \overline{fve}^{(k)}_{[\![p]\!]} = \bot$$

  By Lemma H.21 (cf. below) and induction hypothesis we then have:

  $$\mathcal{E}[\![e_i]\!][u_j \mid 1 \leq j \leq n_i] fve^{(k)}_{[\![p]\!]} = \bot$$

  In turn, this means that we have the following, thus completing the proof:

  $$fve^{(k+1)}_{[\![p]\!]}\; [\![f_i]\!]\; (u_1, \ldots, u_{n_i}) = \bot$$

□ (End of proof of Lemma H.20)

Lemma H.21

Given $fve, \overline{fve}$ such that

$\forall f_i, d, u_1, \ldots, u_{n_i} \cdot$

$\overline{fve}[\![f_i]\!]\ d\ (u_1, \ldots, u_{n_i}) = \bot \ \Rightarrow\ fve[\![f_i]\!](u_1, \ldots, u_{n_i}) = \bot$

We have:

$\forall\ e, d,\ bve \cdot$

$\overline{\mathcal{E}}_k[\![e]\!]\ d\ bve\ fve\ \overline{fve} = \bot \ \Rightarrow\ \mathcal{E}[\![e]\!]\ bve\ fve = \bot$

Proof:

The proof is by structural induction on e.

There are two base cases, both of which are vacuous:

1. $[\![e]\!] = [\![c]\!]$:

   In this case, we always have $\overline{\mathcal{E}}_k[\![e]\!]\ d\ bve\ fve\ \overline{fve} \neq \bot$

2. $[\![e]\!] = [\![x_i]\!]$:

   In this case, we also always have $\overline{\mathcal{E}}_k[\![e]\!]\ d\ bve\ fve\ \overline{fve} \neq \bot$

The inductive cases go as follows:

3. $[\![e]\!] = [\![e_1 + e_2]\!]$:

Suppose that $\overline{\mathcal{E}}_k[\![e_1 + e_2]\!]\ d\ bve\ fve\ \overline{fve} = \bot$

From the definition of $\overline{\mathcal{E}}_k$ and Fact $H.8$ (cf. p. 277), this implies the following:

$(\overline{\mathcal{E}}_k[\![e_1]\!]\ d\ bve\ fve\ \overline{fve} = \bot) \vee (\overline{\mathcal{E}}_k[\![e_2]\!]\ d\ bve\ fve\ \overline{fve} = \bot)$

By the induction hypothesis, this in turn implies:

$(\mathcal{E}[\![e_1]\!]\ bve\ fve = \bot) \vee (\mathcal{E}[\![e_2]\!]\ bve\ fve = \bot)$

Finally, by definition of $\mathcal{E}$ we get the required result:

$\mathcal{E}[\![e_1 + e_2]\!]\ bve\ fve = \bot$

4. $[\![e]\!] = [\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]$:

Suppose that $\overline{\mathcal{E}}_k[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ d\ bve\ fve\ \overline{fve} = \perp$

There are a number of possible subcases:

(i) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \perp$:

Clearly, in this case, from the definition of $\mathcal{E}$, we also will get:

$\mathcal{E}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ bve\ fve = \perp$

(ii) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \textsf{true}$:

Clearly, this means that we have the following:

$\overline{\mathcal{E}}_k[\![e_2]\!]\ d\ bve\ fve\ \overline{fve} = \perp$

This must be so because $\overline{\mathcal{E}}_k[\![e_1]\!]\ d\ bve\ fve\ \overline{fve} \neq \perp$.

(Otherwise, by the induction hypothesis, we would get:

$\mathcal{E}[\![e_1]\!]\ bve\ fve = \perp$

but this would contradict (ii).)

In turn, this implies the following, through the induction hypothesis:

$\mathcal{E}[\![e_2]\!]\ bve\ fve = \perp$

Then, by definition of $\mathcal{E}$ we thus get that:

$\mathcal{E}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ bve\ fve = \perp$

(iii) $\mathcal{E}[\![e_1]\!]\ bve\ fve = \textsf{false}$: Same reasoning as (ii).

5. $[\![e]\!] = [\![f_i\,(e_1,\ \ldots,\ e_{n_i})]\!]$:

Suppose that $\overline{\mathcal{E}}_k[\![f_i\,(e_1,\ \ldots,\ e_{n_i})]\!]\ d\ bve\ fve\ \overline{fve} = \perp$

By definition of $\overline{\mathcal{E}}_k$ we have the following two possibilities:

1) $\overline{fve}[\![f_i]\!]\ d(\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\ bve\ fve) = \perp$

By the hypothesis on fve and $\overline{fve}$ this means :

$fve[\![f_i]\!](\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots\ , \mathcal{E}[\![e_{n_i}]\!]\ bve\ fve) = \perp$

But then, by definition of $\mathcal{E}$ this implies the following, which is the required result:

$\mathcal{E}[\![f_i\,(e_1,\ \ldots,\ e_{n_i})]\!] = \perp$

2) $\overline{fve}[\![f_i]\!]\ d(\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots, \mathcal{E}[\![e_{n_i}]\!]\ bve\ fve) = [d_1,\ \ldots,\ d_{n_i}]$

From the definition of $\overline{\mathcal{E}}_k$, we must then have:

$\bigoplus \{\overline{\mathcal{E}}_k[\![e_j]\!]\ d_j\ bve\ fve\ \overline{fve} \mid 1 \leq j \leq n_i\} = \bot$

From (generalized) Fact $H.8$, this means there must be some l such that:

$\overline{\mathcal{E}}_k[\![e_l]\!]\ d_l\ \ bve\ fve\ \overline{fve} = \bot$

By induction hypothesis, we can then infer:

$\mathcal{E}[\![e_l]\!]\ bve\ fve = \bot$

Clearly, $d_l = \$$ (otherwise, $d_l = @$, so we would have $\overline{\mathcal{E}}_k \ldots = [@ \mid 1 \leq j \leq k]$).

Using Lemma $H.19$, we then obtain the following, leading to the required result:

$fve[\![f_i]\!](\mathcal{E}[\![e_1]\!]\ bve\ fve, \ldots, \bot, \ldots, \mathcal{E}[\![e_{n_i}]\!]\ bve\ fve) = \bot$

□ (End of proof of Lemma H.21)

Theorem 3.10

$\forall p\ \cdot$

$\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\delta_j \mid 1 \leq j \leq n_i] \wedge \delta_l = \$$

$\Rightarrow$

f_i is strict in its l^{th} argument

Proof:

Let $\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = \widetilde{\delta}$, such that $\widetilde{\delta}(l) = \$$. From $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$, we then have:

$\forall v_1,\ \ldots,\ v_{n_i} \cdot \overline{d} = \overline{fve}_{[\![p]\!]}[\![f_i]\!]\ \$\ (v_1,\ \ldots,\ v_{n_i}) \lhd \widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = \widetilde{\delta}$

There are two possibilities:

1. $\overline{d} = \bot$:

In this case, from Lemma H.20, we will have $fve_{[\![p]\!]}[\![f_i]\!](v_1,\ \ldots,\ v_{n_i}) = \bot$, so the fact that f_i is strict holds vacuously.

2. $\overline{d} \neq \bot$:

From the definition of $\lhd$, this means $\overline{d}(l) = \$$. From $\textit{is-inverse}(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$, this implies the following:

$$\textit{is-inverse}_{bve,fve_{[\![p]\!]}}([\![f_i(x_1,\ \ldots,\ x_{n_i})]\!], \$, \overline{d})$$
$$\text{with } bve = [v_j \mid 1 \leq j \leq n_i]$$

In turn, from the definition of $\textit{is-inverse}_{bve,fve_{[\![p]\!]}}$, this means:

$$\begin{array}{l} \forall v \in conc(\$) \cdot \\ \quad \mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][v_j \mid 1 \leq j \leq n_i]\,fve \sqsupseteq v \\ \quad \Rightarrow \\ \quad \exists u_j \in conc(\overline{d}(j)) \quad (j = 1, \ldots, n_i) \cdot \\ \quad\quad \mathcal{E}[\![f_i(x_1,\ \ldots,\ x_{n_i})]\!][u_j \mid 1 \leq j \leq n_i]\,fve \sqsupseteq v \\ \quad\quad \wedge \\ \quad\quad u_j \sqsubseteq v_j \quad (j = 1, \ldots, n_i) \end{array}$$

But $\overline{d}(l) = \$$ and $u_l \in conc(\$)$ implies that $u_l \neq \bot$, so that f_i is effectively strict in its l^{th} argument, as $v \in conc(\$)$ also means $v \neq \bot$.
□ (End of proof of Theorem 3.10)

Theorem 3.11

$$\begin{array}{l} \forall p \cdot \\ \quad \widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \$ = [\delta_j \mid 1 \leq j \leq n_i] \wedge \delta_l = @ \\ \quad \Rightarrow \\ \quad f_i \text{ does not need its } l^{th} \text{ argument} \end{array}$$

Proof: The proof has exactly the same structure as the proof for Theorem 3.10. The only difference is that $\overline{d}(l) = @$ and $u_l \in conc(@)$ implies that $u_l = \bot$, from which the appropriate result is obtained.
□ (End of proof of Theorem 3.11)

Proofs of Section 3.2.8

Theorem 3.13

For any program p:

$is\text{-}inverse(fve_{[\![p]\!]}, \widetilde{fve}_{[\![p]\!]})$

Proof: Take some arbitrary $f_i, d, v_1, \ldots, v_{n_i}$. There are two cases:

1. $\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \{d\} = [\phi \mid 1 \leq j \leq n_i]$: From the definition of $\widetilde{fve}_{[\![p]\!]}$, this clearly means that $d = \$$. From Theorem 3.9, we can then conclude that $fve[\![f_i]\!](v_1, \ldots, v_{n_i}) = \perp$. Because this means $\nexists v \in conc(\$) \cdot \mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve \sqsupseteq v$, then the predicate *is-inverse* holds vacuously.

2. $\widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \{d\} = [\delta_j \mid 1 \leq j \leq n_i] \neq [\phi \mid 1 \leq j \leq n_i]$: From $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$, we thus have:

$$\overline{fve}_{[\![p]\!]}[\![f_i]\!]\ d\ (v_1, \ldots, v_{n_i}) \lhd \widetilde{fve}_{[\![p]\!]}[\![f_i]\!]\ \{d\}$$

From the definition of $\lhd$, there are then two possibilities:

i) $\overline{fve}_{[\![p]\!]}[\![f_i]\!]\ d\ (v_1, \ldots, v_{n_i}) = \perp$: From $is\text{-}inverse(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$ and Lemma H.20, this means $fve[\![f_i]\!](u_1, \ldots, u_{n_i}) = \perp$, so that $\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve = \perp$ with $bve = (u_1, \ldots, u_{n_i})$. There are then two possibilities:

- $d = @$: This implies that $\delta_j = @$, so that *is-inverse* is effectively satisfied, as $\mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!][\perp \mid 1 \leq j \leq n_i]fve \sqsupseteq \perp$.
- $d = \$$: In this case, *is-inverse* is satisfied vacuously, because $\nexists v \in conc(\$) \cdot \mathcal{E}[\![f_i(x_1, \ldots, x_{n_i})]\!]\ bve\ fve \sqsupseteq v$.

ii) $\overline{fve}_{[\![p]\!]}[\![f_i]\!]\ d\,(v_1, \ldots, v_{n_i}) = [d_1, \ldots, d_{n_i}]$: From $\overline{fve}_{[\![p]\!]} \lhd \widetilde{fve}_{[\![p]\!]}$, we thus have $d_j \lhd \delta_j$. From $is\text{-}inverse(fve_{[\![p]\!]}, \overline{fve}_{[\![p]\!]})$, we have:

$is\text{-}inverse_{bve,fve_{[\![p]\!]}}([\![f_i(x_1, \ldots, x_{n_i})]\!], d, \overline{fve}_{[\![p]\!]}[\![f_i]\!]\ d\ (v_1, \ldots, v_{n_i}))$
with $bve = [v_j \mid 1 \leq j \leq n_i]$

By definition of $is\text{-}inverse_{bve,fve_{[\![p]\!]}}$ together with the fact that $d_j \lhd \delta_j$ implies $conc(d_j) \subseteq conc(\delta_j)$, we get the required result.

□ (End of proof of Theorem 3.13)

H.2.2 Proofs of Section 3.3

Proofs of Section 3.3.4

Fact 3.14

$$\forall\, ed_1, ed_2 \cdot ed_1 \sim ed_2 \cdot$$
$$conc(ed_1 \oplus ed_2) = \{v_1 \sqcup v_2 \mid v_1 \in conc(ed_1) \wedge v_2 \in conc(ed_2) \wedge v_1 \sim v_2\}$$

Proof:

There are a number of cases depending on the values of ed_1 and ed_2.

Base cases:

1. $ed_1 = @$: In this case, we have $ed_1 \oplus ed_2 = ed_2$, so that:

$$\begin{aligned} & conc(ed_1 \oplus ed_2) \\ & \quad = conc(ed_2) \\ & \quad = \{v_2 \mid v_2 \in conc(d_2)\} \\ & \quad = \{\perp \sqcup v_2 \mid v_2 \in conc(d_2)\} \\ & \quad = \{v_1 \sqcup v_2 \mid v_1 \in conc(@) \wedge v_2 \in conc(d_2)\} \end{aligned}$$

2. $ed_2 = @$: Similar to the previous case.

Inductive case:

3. $cons(ed_1) = cons(ed_2)$: Suppose $ed_1 = mk\text{-}EPDmd(K_{ij}, [d'_1, \ldots, d'_a])$ and $ed_2 = mk\text{-}EPDmd(K_{ij}, [d''_1, \ldots, d''_a])$, with $d'_j \sim d''_j$. Then, $ed_1 \oplus ed_2 = mk\text{-}EPDmd(K_{ij}, [d'_1 \oplus d''_1, \ldots, d'_a \oplus d''_a])$, so that $conc(ed_1 \oplus ed_2) = \{mk\text{-}Struct(K_{ij}, [v_1, \ldots, v_a]) \mid v_1 \in conc(d'_1 \oplus d''_1), \ldots, v_a \in conc(d'_a \oplus d''_a)\}$.

From $d'_j \sim d''_j$ and induction hypothesis, we have: $conc(d'_j \oplus d''_j) = \{v'_j \sqcup v''_j \mid v'_j \in conc(d'_j) \wedge v''_j \in conc(d''_j) \wedge v'_j \sim v''_j\}$. Thus: $conc(ed_1 \oplus ed_2) = \{mk\text{-}Struct(K_{ij}, [v'_1 \sqcup v''_1, \ldots, v'_a \sqcup v''_a]) \mid v'_j \in conc(d'_j) \wedge v''_j \in conc(d''_j) \wedge v'_j \sim v''_j \quad (j = 1, \ldots, a)\}$. But $v'_j \in conc(d'_j)$ if and only if $v' = mk\text{-}Struct(K_{ij}, [v'_1, \ldots, v'_a]) \in conc(d_1)$ and similarly for $v'' = mk\text{-}Struct(K_{ij}, [v''_1, \ldots, v''_a]) \in conc(d_2)$.

Now, $v'_j \sim v''_j \quad (j = 1, \ldots, a)$ if and only if $v' \sim v''$, in which case we have: $v' \sqcup v'' = mk\text{-}Struct(K_{ij}, [v'_1 \sqcup v''_1, \ldots, v'_a \sqcup v''_a])$. Thus, we get the required result: $conc(ed_1 \oplus ed_2) = \{v' \sqcup v'' \mid v' \in conc(ed_1) \wedge v'' \in conc(ed_2) \wedge v' \sim v''\}$

□ (End of proof of Fact 3.14)

Lemma 3.17

$$\forall e, d, v_j, bve, fve, \overline{fve} \cdot$$
$$\quad is\text{-}inverse(fve, \overline{fve})$$
$$\qquad \Rightarrow$$
$$\quad is\text{-}inverse_{bve,fve}(e, d, \overline{\mathcal{E}}_k[\![e]\!]\, d\ bve\ fve\ \overline{fve})$$
$$\qquad \text{where}$$
$$\qquad\quad k = \mathsf{len}\ bve$$

Proof: This is a proof by structural induction similar to the proof of Lemma 3.6, p. 277. Here, we will only examine the new cases, i.e., those dealing with data structures constructs; the cases for $[\![c]\!]$ and $[\![e_1 + e_2]\!]$, although slightly different from the scalar semantics, are straightforward to prove, because when $d \neq @$ and $d \neq \$$, then the condition for being an inverse holds vacuously, as there are no $v \in conc(d)$ for which the expression evaluates to a value $\sqsupseteq v$.

1. $[\![K\ e_1\ \cdots\ e_a]\!]$: Assume $\overline{\mathcal{E}}_k[\![K\ e_1\ \cdots\ e_a]\!]d\ bve\ fve\ \overline{fve} = [d_1,\ \ldots,\ d_k] \neq \bot$. From the definition of $\overline{\mathcal{E}}_k$, this means $d = mk\text{-}EPDmd(K, [d'_1, \ldots, d'_a])$ for some d'_l ($l = 1, \ldots, a$).

Take some $v \in conc(d)$ such that $\mathcal{E}[\![K\ e_1\ \cdots\ e_a]\!]\ bve\ fve \sqsupseteq v$. From $v \in conc(d)$, this means $v = mk\text{-}Struct(K, [v'_1, \ldots, v'_a])$ for some $v'_l \in conc(d'_l)$ $(l = 1, \ldots, a)$. By definition of $\mathcal{E}$, this then means:

$$\mathcal{E}[\![e_l]\!]\ bve\ fve \sqsupseteq v'_l \in conc(d'_l) \quad (l = 1, \ldots, a) \quad (1)$$

Now, $\overline{\mathcal{E}}_k[\![K\ e_1\ \cdots\ e_a]\!]d\ bve\ fve\ \overline{fve} = [d_1,\ \ldots,\ d_k]$ implies $\overline{\mathcal{E}}_k[\![e_l]\!]\delta'_l\ bve\ fve\ \overline{fve} = [d_{l1}, \ldots, d_{lk}] \neq \bot \quad (l = 1, \ldots, a)$, with $d_j = \overline{\bigoplus}{}_{l=1}^{a} d_{lj} \quad (j = 1, \ldots, k)$ (by Fact H.9).

By induction hypothesis, definition of *is-inverse* and (1), we have:

$$\exists u_{l1} \in conc(d_{l1}), \ldots, u_{la} \in conc(d_{la}) \cdot \mathcal{E}[\![e_l]\!][u_{lj} \mid 1 \leq j \leq k]\ fve \sqsupseteq v'_l$$
$$\wedge\ u_{lj} \sqsubseteq bve(j)$$

Let $u_j = \bigsqcup_{l=1}^{a} u_{lj}$ $(j = 1, \ldots, k)$. By Fact 3.14, we effectively have $u_j \in conc(\bigoplus_{l=1}^{a} d_{lj}) = conc(d_j)$. By monotonicity, we have:

$$\mathcal{E}[\![e_l]\!][u_j \mid 1 \leq j \leq k]\ fve \sqsupseteq v'_l$$
$$\wedge\ u_{lj} \sqsubseteq bve(j)$$

From definition of $\mathcal{E}$ and monotonicity, this then implies the required result:

$$\mathcal{E}[\![K\ e_1\ \ldots\ e_a]\!][u_j \mid 1 \leq j \leq k]\ fve \sqsupseteq mk\text{-}Struct(K, [v'_1, \ldots, v'_a]) = v$$

2. $[\![K\_l(e)]\!]$: Assume $\overline{\mathcal{E}}_k[\![K\_l(e)]\!]d\ bve\ fve\ \overline{fve} = [d_1 \cdots d_k] \neq \bot$. Take some $v \in conc(d)$ such that $\mathcal{E}[\![K\_l(e)]\!]\ bve\ fve \sqsupseteq v$. By definition of $\mathcal{E}$, the fact that $d \neq @$, which means $v \neq \bot$, this implies the following hold: $cons(\mathcal{E}[\![e]\!]\ bve\ fve) = K$ and $components(\mathcal{E}[\![e]\!]\ bve\ fve)(l) \sqsupseteq v$.

From this, we can infer, from the definition of $\sqsupseteq$ that $\mathcal{E}[\![e]\!] bve fve \sqsupseteq mk\text{-}Struct(K, [\bot \mid 1 \leq j < l] \frown [v] \frown [\bot \mid l < j \leq k])$. By definition of $(mkComponent_{ED}\ K\ l\ d)$ and $conc$, this means $\mathcal{E}[\![e]\!]\ bve\ fve \sqsupseteq v'$ for some $v' \in conc(mkComponent_{ED}\ K\ l\ d)$.

By defn., $\overline{\mathcal{E}}_k[\![K\_l(e)]\!]d bve fve \overline{fve} = [d_1, \ldots, d_k]$ means $\overline{\mathcal{E}}_k[\![e]\!](mkComponent_{ED} K l d) bve fve \overline{fve}$ $= [d_1, \ldots, d_k] \neq \bot$. In turn, by induction hypothesis and definition of *is-inverse*, this means:

$$\exists u_j \in conc(d_j) \cdot \mathcal{E}[\![e]\!][u_j \mid 1 \leq j \leq k] \sqsupseteq v'$$
$$\wedge\ u_j \sqsubseteq v_j = bve(j)$$

But $\mathcal{E}[\![e]\!][u_j \mid 1 \leq j \leq k]\ fve \sqsupseteq v'$ also means that $cons(\mathcal{E}[\![e]\!][u_j \mid 1 \leq j \leq k]\ fve) = K$ and $components(\mathcal{E}[\![e]\!][u_j \mid 1 \leq j \leq k]\ fve)(l) \sqsupseteq v$. Thus, we effectively have the required result: $\mathcal{E}[\![K\_l(e)]\!][u_j \mid 1 \leq j \leq k]\ fve)(l) \sqsupseteq v$.

3. $[\![\{\texttt{case}\ e_0\ \texttt{of}\ K_i \Rightarrow e_i; \ldots\}]\!]$: Suppose $\overline{\mathcal{E}}_k[\![\{\texttt{case}\ e_0\ \texttt{of}\ K_i \Rightarrow e_i; \ldots\}]\!]d\ bve\ fve\ \overline{fve}$ $= [d_1, \ldots, d_k] \neq \bot$. This means there is some i such that:

$$cons(\mathcal{E}[\![e_0]\!]\ bve\ fve) = K_i$$
$$\overline{\mathcal{E}}_k[\![e_0]\!](mkVariant_{ED}\ K_i)\ bve\ fve\ \overline{fve} = [d_1^{e_0}, \ldots, d_k^{e_0}] \neq \bot \quad (1)$$
$$\overline{\mathcal{E}}_k[\![e_i]\!]d\ bve\ fve\ \overline{fve} = [d_1^{e_i}, \ldots, d_k^{e_i}] \neq \bot \quad (2)$$
$$\text{with } d_j = d_j^{e_0} \oplus d_j^{e_i} \quad (j = 1, \ldots, k)$$

Take some $v \in conc(d)$ ($v \neq \bot$, as $d \neq @$) such that $\mathcal{E}[\![\{\texttt{case}\ e_0\ \texttt{of}\ K_i \Rightarrow e_i; \ldots\}]\!]\ bve fve \sqsupseteq v$. As we have $cons(\mathcal{E}[\![e_0]\!]\ bve fve) = K_i$, this means $\mathcal{E}[\![e_i]\!]\ bve fve \sqsupseteq v$. In conjunction with (2) and induction hypothesis, this implies the following hold:

$$\exists u_j^{e_i} \in conc(d_j^{e_i}) \cdot \mathcal{E}[\![e_i]\!][u_j^{e_i} \mid 1 \leq j \leq k]\ fve \sqsupseteq v$$
$$\wedge\ u_j^{e_i} \sqsubseteq bve(j) \quad (j = 1, \ldots, k)$$

Now, $cons(\mathcal{E}[\![e_0]\!]\ bve\ fve) = K_i$ means $\mathcal{E}[\![e_0]\!]\ bve\ fve \sqsupseteq v^{e_0}$, for some $v^{e_0} \in conc(mkVariant_{ED}\ K_i)$, by definition of $mkVariant_{ED}$ and $conc$. This implies, in conjunction with (1) and induction hypothesis, that:

$$\exists u_j^{e_0} \in conc(d_j^{e_0}) \cdot \mathcal{E}[\![e_0]\!][u_j^{e_0} \mid 1 \leq j \leq k]\ fve \sqsupseteq v^{e_0}$$
$$\wedge\ u_j^{e_0} \sqsubseteq bve(j) \quad (j = 1, \ldots, k)$$

Let $u_j = u_j^{e_0} \sqcup u_j^{e_i}$ $(j = 1, \ldots, k)$. By Fact 3.14, we then have $u_j \in conc(u_j^{e_0} \oplus e_j^{e_i})$. By monotonicity, we have $\mathcal{E}[\![e_0]\!][u_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v_{e_0}$ and $\mathcal{E}[\![e_i]\!][u_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v$. Thus, by definition of $\mathcal{E}$, we get the required result: $\mathcal{E}[\![\{\texttt{case}\ e_0\ \texttt{of}\ K_i \Rightarrow e_i; \ldots\}]\!][u_j \mid 1 \leq j \leq k]\, fve \sqsupseteq v$. □ (End of proof of Lemma 3.17)

Proofs of Section 3.3.6

Remark: The next theorem (Theorem 3.18) uses fixed point induction, so it should be ensured that the predicates being used are all *admissible*. Let's briefly examine each of the appropriate predicates:

- Safe abstract demands (Defn. 3.40, p. 89): Suppose we have an exact demand d [2] and an infinite chain of demand references and demand environments for which the appropriate relationship holds, i.e., they all contain a safe approximation of d. Given the ordering on demand references and demand environments (cf. Defns. B.2–B.4, p. 208 and B.5, p. 209), then the least upper-bound will also contain an appropriate element safe for d, thus satisfying the existential quantifier of Defn 3.40.

- Safe abstract demand vectors (Defn. 3.41, p. 90): Given that the previous predicate is inclusive, then inclusivity of safety for demand vectors is immediate: if at some point $\overline{d} \neq \bot$, then safe relationships between the vector components will exist and will be preserved for the least upper-bound, given the ordering between demand vectors (i.e., component-wise extension: cf. p. 217).

- Safe abstract function environments (Defn. 3.42, p. 90): Given that the two previous predicates are inclusive, the inclusivity of this predicate can be obtained from an application of the proposition by Manna, Ness & Vuillemin [Sch86, p. 130].

Theorem 3.18 (Safety of abstract non-standard semantics)

For any program p:

$\forall \sigma \cdot$

$$\overline{fve}_{[\![p]\!]} \lhd_\sigma \widetilde{fve}_{[\![p]\!]}$$

[2] Given our domain of exact demands, there can be no infinite chain of exact demands.

Proof: The proof is by fixed-point (computational) induction using Lemma 3.19 (cf. below).

Base case:

$\overline{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda d, (z_1, \ldots, z_{n_i}) \cdot \bot\}$

$\widetilde{fve}^{(0)}_{[\![p]\!]} = \{f_i \mapsto \lambda\delta \cdot return\ [\phi \mid 1 \leq j \leq n_i]\}$

Clearly, for any σ, we have $\overline{fve}^{(0)}_{[\![p]\!]} \lhd_\sigma \widetilde{fve}^{(0)}_{[\![p]\!]}$, as $\bot \lhd_\sigma \delta$, for any δ.

Inductive case:

Take some arbitrary $f_i, v_1, \ldots, v_{n_i}, d, \delta, \sigma$ such that $d \lhd_\sigma \delta$. From the post-condition of *round-up*, we have $\delta \sqsubseteq_{\sigma'} \delta^r$, where σ' is the new environment obtained after the rounding-up of δ. Thus, from Fact H.34, we have $d \lhd_{\sigma'} \delta^r$. From the definition of $\overline{fve}^{(k+1)}_{[\![p]\!]}$ and $\widetilde{fve}^{(k+1)}_{[\![p]\!]}$, and from Lemma 3.19, together with the induction hypothesis, we have that the results produced by $\overline{\mathcal{E}}_{n_i}$ and $\widetilde{\mathcal{E}}_{n_i}$ are safe relative to each other. Now, again from Fact H.34, i.e., the rounding-up of the components of the resulting vector, the result returned by $\widetilde{fve}^{(k+1)}_{[\![p]\!]}$ is still safe for the one produced by $\overline{fve}^{(k+1)}_{[\![p]\!]}$.

$\square$ (End of proof of Theorem 3.18)

Lemma 3.19 *(Safety of expression semantics: arbitrary demand)*

$$
\begin{array}{l}
\forall \sigma \cdot \\
\quad \forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd_\sigma \widetilde{fve} \cdot \\
\quad\quad \forall d \cdot \forall \delta \in \mathsf{dom}\,\sigma \cdot d \lhd_\sigma \delta \cdot \\
\quad\quad\quad \forall k, bve \cdot \mathsf{len}\, bve = k \cdot \\
\quad\quad\quad\quad \mathsf{let}\ \overline{d} = \overline{\mathcal{E}}_k[\![e]\!]\ d\ bve\ fve\ \overline{fve}\ \ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ (\widetilde{\delta}, \sigma') = \widetilde{\mathcal{E}}_k[\![e]\!]\ \delta\ \widetilde{fve}\ \sigma\ \ \mathsf{in} \\
\quad\quad\quad\quad \overline{d} \lhd_{\sigma'} \widetilde{\delta}
\end{array}
$$

Proof: Straightforward case analysis based on the definitions of $\overline{\mathcal{E}}_k$ and $\widetilde{\mathcal{E}}_k$.

1. $d = @$: In this case, the exact semantics produces $[@ \mid 1 \leq j \leq k]$. But then, $@ \lhd_\sigma \delta$ means that *possibly-negative*? $\delta\ \sigma$ holds, from which we will effectively have that $@ \lhd_{\sigma'} \delta'_l$ (for some σ'), because of the post-condition of *add-negative*.

2. $d \neq @$: This means that $d = \textit{mk-EPDmd}(K, comps)$. But $d \lhd_\sigma \delta$ means there is some $\textit{mk-APDmd}(K, comps') \in \sigma(\delta)$ such that $\forall j \in \textsf{inds}\, comps \cdot comps(j) \lhd_\sigma comps'(j)$.

There are then two possibilities:

i) *possibly-negative?* $\delta\,\sigma$: From the definition of *make-positive*, we still have $\delta \sqsubseteq_{\sigma'} \delta^+$ (for some σ'). A recursive call to $\widetilde{\mathcal{E}}_k$ will then be made, in which case we will have that *possibly-negative?* $\delta\,\sigma'$ is false and so is *bottom?* $\delta\,\sigma'$, i.e., a call to $\widetilde{\mathcal{E}}_k^+$ will be made. From Lemma 3.20, we will then have the required result.

ii) $\neg$ *possibly-negative?* $\delta\,\sigma$: Because $d \lhd_\sigma \delta$, then *bottom?* $\delta\,\sigma$ is false. From Lemma 3.20, we will get the required result.

$\Box$ (End of proof of Lemma 3.19)

Lemma 3.20 *(Safety of expression semantics: positive demand)*

$$\begin{array}{l}
\forall \sigma \cdot \\
\quad \forall \overline{fve}, \widetilde{fve} \cdot \overline{fve} \lhd_\sigma \widetilde{fve} \cdot \\
\quad\quad \forall d \cdot \forall \delta \in \textsf{dom}\,\sigma \cdot d \lhd_\sigma \delta \cdot \\
\quad\quad\quad \forall k, bve \cdot \textsf{len}\, bve = k \cdot \\
\quad\quad\quad\quad \textsf{let}\ \overline{d} = \overline{\mathcal{E}}_k^+ [\![e]\!]\ d\ bve\ fve\ \overline{fve}\ \ \textsf{in} \\
\quad\quad\quad\quad \textsf{let}\ (\widetilde{\delta}, \sigma') = \widetilde{\mathcal{E}}_k^+ [\![e]\!]\ \delta\ \widetilde{fve}\ \sigma\ \ \textsf{in} \\
\quad\quad\quad\quad \overline{d} \lhd_{\sigma'} \widetilde{\delta}
\end{array}$$

Proof: The proof is by structural induction on e, using Lemma 3.19 for the inductive cases. Note that all we really need to check are the cases where $\overline{d} \neq \bot$, otherwise the lemma holds vacuously as the antecedent is false.

Base cases:

1. $[\![e]\!] = [\![c]\!]$: Immediate, as $d = \$$ and $d \lhd_\sigma \delta$ clearly implies that *scalar?* $\delta\,\sigma$ is true.

2. $[\![e]\!] = [\![x_i]\!]$: Immediate from the definition of $\lhd_\sigma$ for vectors, as $d \lhd_\sigma \delta$.

Inductive cases:

3. $[\![e]\!] = [\![e_1 + e_2]\!]$: As for $[\![e]\!] = [\![c]\!]$, $d = \$$ means *scalar?* $\delta\,\sigma$ will be true. By Lemma 3.19 and Fact H.36, we get the required result.

4. $[\![e]\!] = [\![f_i(e_1, \ldots, e_{n_i})]\!]$: Using the fact that $\overline{fve} \lhd \widetilde{fve}$, we get that $d_l \lhd_{\sigma'} \delta_l$. By Lemma 3.19, we get $\overline{d}_l \lhd_{\sigma''} \widetilde{\delta}_l$. From generalization of Fact H.36 to an arbitrary number of arguments, we then get the required result.

5. $[\![e]\!] = [\![K_{ij}\, e_1 \; \cdots \; e_{a_{ij}}]\!]$: If $cons(d) = K_{ij}$, then from $d \lhd_\sigma \delta$ we must clearly have that *has-cons?* $\delta\ K_{ij}\ \sigma$ is true. But then, from the definition of *select-cons* and $\lhd_\sigma$, we must have $d_l \lhd_\sigma \delta_l$ (Fact H.37). At this point, the rest of the proof becomes like the function call case.

6. $[\![e]\!] = [\![K_{ij}\_l(e)]\!]$: Straightforward using Lemma 3.19 and Fact H.39.

7. $[\![e]\!] = [\![\{\texttt{case}\ e_0\ \texttt{of}\ K_{ij} \Rightarrow e_j; \ldots\}]\!]$: Using Fact H.38, Lemma 3.19 and Fact H.36 we have that $\overline{d} \lhd_{\sigma'} \widetilde{\delta}_l$ for one of the $l \in \{1, \ldots, a_{ij}\}$. From Fact H.41, we then get the required result.

□ (End of proof of Lemma 3.20)

H.3 Proofs of Appendix B

H.3.1 Properties of operations defined in Section B.2

Below are a number of properties concerning the operations on abstract demands (cf. App. B.2) which are used in the proof of safety of the abstract demand propagation semantics (cf. App. H.2.2, p. 298).

The first property states that, given any two well-formed demand references, including cyclic ones, the process of computing their $\cup$ is sure to terminate in a finite number of steps: [3]

Fact H.30 *(Union and normalize-up terminate)*
Both $\cup$ and normalize-up* terminate and produce a result which is normalized, i.e., having at most a single demand with any given constructor.

Proof: If neither δ_1 and δ_2 are cyclic, then the process of computing the union of the sub-components is clearly finite. Now, in each recursive call to *normalize-up-rec** — called by *normalize-up** — a pair of demands with the same constructors are

[3] Note that a demand is *normalized* when it has, for any given constructor, at most a single positive demand using that constructor. Cf. clause (*iv*) of the *is-wfAD* predicate, p. 206.

replaced by a single demand with that constructor: after a finite number of recursive calls there will be, for any constructor, at most a single demand, so the recursive calls will terminate and *normalize-up** will effectively produce its result. As for the result of *union** (and thus of $\cup$), it will also be normalized because it will either be the one produced by *normalize-up** or an existing δ' which, by the invariant on demand environment (*inv-ADmdEnv*) will also be normalized.

If either one of δ_1 or δ_2 or both are cyclic, then the traversal of the subcomponents is sure, at some point, to encounter a pair of arguments which has been encountered before, because there is only a finite number of possible combination of such pairs (cf. clause (*ii*) of *inv-ADmdEnv*, Fig. B.1 p. 206, together with the definition of *is-wfAD*); thus, at some point $\{\delta_1, \delta_2\} \in \mathsf{dom}\, seen$ will become true and the recursion will terminate. Also, as explained in the preceding paragraph, *normalize-up** will produce a normalized result after a finite number of steps, so that the result of *union** will also be produced and be normalized. $\Box$ (End of proof of Fact H.30)

The next property indicates that $\cup$ effectively computes an upper-bound of its arguments:

Fact H.31 *(Union is an upper-bound)*

$$\delta_1 \in \mathsf{dom}\, \sigma \wedge \delta_2 \in \mathsf{dom}\, \sigma \wedge (\delta', \sigma') = \cup\, \delta_1\, \delta_2\, \sigma$$
$$\Rightarrow$$
$$\delta_1 \sqsubseteq_{\sigma'} \delta' \wedge \delta_2 \sqsubseteq_{\sigma'} \delta'$$

Proof: In *union**, take some $bd' \in \sigma(\delta_1)$. If $bd' = @$, then clearly $@ \in (\sigma(\delta_1) \cup \sigma(\delta_2))$. As *normalize-up** has no effect on negative demand, $@$ will also be $\in ad'$ after normalization. Finally, after *bind-ref*, either $\delta' = \delta$, in which case the result is immediate, or $\delta \sqsubseteq_{\sigma'} \delta'$ (for some appropriate σ' and for some δ' for which we also have $\delta' \sqsubseteq_{\sigma'} \delta$) so that $@$ is still in the result.

If $bd' = mk\text{-}APDmd(K, [\delta'_1, \ldots, \delta'_k])$, then bd' is also $\in (\sigma(\delta_1) \cup \sigma(\delta_2))$. If there is no $bd'' \in \sigma(\delta_2)$ such that $cons(bd'') = K$, then bd' will still be in the normalized demand, as *normalize-up** will have no effect on it. If there is some $bd'' \in \sigma(\delta_2)$ such that $bd'' = mk\text{-}APDmd(K, [\delta''_1, \ldots, \delta''_k])$ then, within *normalize-up-rec**, at some iteration the pair bd', bd'' will be selected for normalization. This means that, at some point, ad' (in *normalize-up-rec**) will be made to contain $mk\text{-}APDmd(K, [\delta_l \mid 1 \leq l \leq k])$, with bd' and bd'' removed. Now, as we have $\delta_l = union^*\ \delta'_l\ \delta''_l\ seen$ we get (using an appropriate induction hypothesis) that $\delta'_l \sqsubseteq_{\sigma'} \delta_l$, so that we would effectively have some positive demand, in the resulting normalized union, more defined than bd' (once a new positive demand is added after normalization, it remains there forever). □ (End of proof of Fact H.31)

The following fact will be used in the proof of Fact H.35, p. 304, which states that the abstract merge is a safe approximation of the exact one:

Fact H.32 *The function normalize-up* produces an abstract demand ad' which is normalized and is an upper-bound of its argument ad.*

Proof: The fact that the result is normalized has been mentioned above and is quite evident from the examination of *normalize-up-rec**: when the recursion terminates (and it will: cf. above) then the if-condition will be true, i.e., there exists no two distinct positive demands having the same constructor.

As for being an upper-bound, take some $bd' \in ad$. If there is no $bd'' \in ad$ such that $bd' \neq bd''$ and $bd' \sim bd''$, then bd' will clearly be $\in ad'$. If there is some $bd'' \in ad$ such that $bd' \neq bd''$ and $bd' \sim bd''$ then, by construction, and assuming *union** effectively produces an upper-bound, ad' will effectively contain a positive demand $mk\text{-}APDmd(K, [\delta_1, \ldots, \delta_k])$ such that $bd' \sqsubseteq_{\sigma'} mk\text{-}APDmd(K, [\delta_1, \ldots, \delta_k])$ (for some σ'). This demand will either remain in the final result or it may, at some later point if some other positive demand with the same constructor also exists, be replaced by a demand which will also be an upper-bound. thus ensuring the proper result. □

The next fact states that the safety relationship between an exact and abstract demands continues to hold when the demand environment is refined:

Fact H.33 *(A more refined environment is safe)*

$$d \lhd_\sigma \delta \wedge \sigma \sqsubseteq \sigma' \;\Rightarrow\; d \lhd_{\sigma'} \delta$$

Proof: Immediate from the definition of $\sigma \sqsubseteq \sigma'$ (Defn B.5, p. 209) as δ will be unchanged. □

The following fact, used, among other things, in the proof of Theorem 3.18, indicates that rounding-up preserves safety, i.e., as long as a more defined result is produced, then safety is ensured.

Fact H.34 *(Upward is safe)*

$d \lhd_\sigma \delta \wedge \delta \sqsubseteq_\sigma \delta' \;\Rightarrow\; d \lhd_\sigma \delta'$

Proof: Straightforward from the definitions of $\lhd_\sigma$ (Defn 3.40, p. 89) and $\sqsubseteq_\sigma$ (Defn. B.2, p. 208), using induction on the structure of d. In the base case, i.e., $d = @$, from $\lhd_\sigma$, we have that $@ \in \sigma(\delta)$. But then, from $\delta \sqsubseteq_\sigma \delta'$, $@$ must also be $\in \sigma(\delta')$. A similar reasoning holds for the inductive case: $d = \textit{mk-EPdmd}(K, \textit{comps})$, means that δ must contain some appropriate $\textit{mk-APDmd}(K, \textit{comps}')$, such that $\textit{comps}(j) \lhd_\sigma \textit{comps}'(j)$. But $\delta \sqsubseteq_\sigma \delta'$ means δ' must contain some $\textit{mk-APDmd}(K, \textit{comps}'')$ such that $\textit{comps}'(j) \sqsubseteq_\sigma \textit{comps}''(j)$, which leads to the required result based on the induction hypothesis.
□ (End of proof of Fact H.34)

The following fact states that the result obtained from merging two abstract demands is safe relative to the merging of any two exact demands which are safely approximated by the abstract ones:

Fact H.35 *(Merge is safe)*

$d_1 \lhd_\sigma \delta_1 \wedge d_2 \lhd_\sigma \delta_2 \wedge d_1 \sim d_2$
$\Rightarrow$
$\mathsf{let}\ d' = d_1 \oplus d_2\ \mathsf{in}$
$\mathsf{let}\ (\delta', \sigma') = \oplus\ \delta_1\ \delta_2\ \sigma\ \mathsf{in}$
$d' \lhd_{\sigma'} \delta'$

Proof: There are a number of possible cases:
1. $d_1 = @$: From the definition of $\oplus$, we will have that $d' = d_1 \oplus d_2 = d_2$. From $d_1 \lhd_\sigma \delta_1$, we must then have $@ \in \sigma(\delta_1)$. From the definition of $\sim$ between $ABDmd$, the set *compatibles* will then contain a pair $(@, bd_2)$ for each $bd_2 \in \sigma(\delta_2)$. From *merge-bd**, as each one of these pairs will produce a basic demand equal to bd_2, we will then have (in *merge**) that $bd_2 \subseteq \{bd_l \mid 1 \leq l \leq k\}$, thus *a fortiori* $bd_2 \sqsubseteq_{\sigma'} \{bd_l \mid 1 \leq l \leq k\}$ (for some σ' s.t. $\sigma \sqsubseteq \sigma'$). From the fact that *normalize-up** produces an

upper-bound (Fact H.32, p. 303), we will still have that $\sigma''(\delta_2) \sqsubseteq_{\sigma''} ad'$ (for some σ'' such that $\sigma \sqsubseteq \sigma' \sqsubseteq \sigma''$), which leads to the required result given the post-condition of bind-ref.

2. $d_2 = @$: Similar to the previous case.

3. $d_1 \neq @ \wedge d_2 \neq @$: From $d_1 \sim d_2$, we have that $d_1 = \mathit{mk\text{-}EPDmd}(K, [d'_1, \ldots, d'_k)]$ and $d_2 = \mathit{mk\text{-}EPDmd}(K, [d''_1, \ldots, d''_k])$ with $d'_j \sim d''_j$ $(j = 1, \ldots, k)$. Now, from $d_1 \lhd_\sigma \delta_1$, there is then some $\mathit{mk\text{-}APDmd}(K, [\delta'_1, \ldots, \delta'_k]) \in \sigma(\delta_1)$ with $d'_j \lhd_\sigma \delta'_j$, and similarly for $\mathit{mk\text{-}APDmd}(K, [\delta''_1, \ldots, \delta''_k]) \in \sigma(\delta_2)$ with $d''_j \lhd_\sigma \delta''_j$.

By definition of $\oplus$ on exact demands, let d be $d_1 \oplus d_2 = \mathit{mk\text{-}EPDmd}(K, [d'_j \oplus d''_j \mid 1 \leq j \leq k])$. The two demands $\mathit{mk\text{-}APDmd}(K, [\delta'_1, \ldots, \delta'_k])$ and $\mathit{mk\text{-}APDmd}(K, [\delta''_1, \ldots, \delta''_k])$ being compatible, in *merge**, these two demands will be combined using *merge-bd**, i.e., one of the bd_l will be obtained from their merging. Clearly, from the definition of *merge-bd**, this new demand will be equal to some $\mathit{mk\text{-}APDmd}(K, [\delta'''_1, \ldots, \delta'''_k])$ such that δ'''_j was obtained from $\mathit{merge}^*\, \delta'_j\, \delta''_j$. By induction hypothesis, we then have that this δ'''_j is such that $d'_j \oplus d''_j \lhd_{\sigma'} \delta'''_j$ (for an appropriate $\sigma' \sqsupseteq \sigma$). Thus, we have that $d = d_1 \oplus d_2$ is safely approximated by some positive demand in $\{bd_l \mid 1 \leq l \leq k\}$. As *normalize-up** creates an upper-bound (Fact H.32, p. 303), then d will still be safely approximated by some positive demand in ad', so it will effectively be safely approximated by the demand reference returned by *bind-ref*.
□ (End of proof of Fact H.35)

The next fact states a similar relationship for the merge of vectors of demands:

Fact H.36 *(Vector merge is safe)*

$\overline{d}_1 \lhd_\sigma \widetilde{\delta}_1 \wedge \overline{d}_2 \lhd_\sigma \widetilde{\delta}_2$
$\Rightarrow$
let $\overline{d}' = \overline{d}_1 \,\overline{\oplus}\, \overline{d}_2$ in
let $(\widetilde{\delta}', \sigma') = \widetilde{\oplus}\ \widetilde{\delta}_1\, \widetilde{\delta}_2\, \sigma$ in
$\overline{d}' \lhd_{\sigma'} \widetilde{\delta}'$

Proof: From the definition of $\widetilde{\oplus}$ (App. B.3, p. 217), we have $\rho_l = \oplus\, \delta_l\, \epsilon_l\, \sigma'$, for some environment σ'. Thus, from Fact H.35 (safe merge) and Fact H.33 (a more refined environment is safe), we have that each component of the result vector is effectively safe for the corresponding component of the exact demand vector. Thus, given the definition of safety of vectors (Defn. 3.41), which is defined element-wise, we thus have the appropriate result. □

The next fact states that a certain relationship exists between a definite positive exact demand and any abstract demand safe for it, that is, the abstract demand must have some associated positive demand with the indicated constructor and with safe components:

Fact H.37 *(Property of select-cons)*

$$\mathit{mk\text{-}EPDmd}(K_{ij}, [d_1,\ \ldots,\ d_{a_{ij}}]) \lhd_\sigma \delta$$
$$\Rightarrow$$
$$\mathit{has\text{-}cons?}\ \delta\ K_{ij}\ \sigma$$
$$\wedge$$
$$\mathsf{let}\ \mathit{mk\text{-}APDmd}(K_{ij}, [\delta_1,\ \ldots,\ \delta_{a_{ij}}]) = \mathit{select\text{-}cons}\ \delta\ K_{ij}\ \sigma\ \mathsf{in}$$
$$d_l \lhd_\sigma \delta_l \quad (l = 1, \ldots, a_{ij})$$

Proof: From the definition of $\lhd_\sigma$, this means there is some $\mathit{mk\text{-}APDmd}(K_{ij}, [\delta_1, \ldots, \delta_{a_{ij}}])$ such that $d_l \lhd_\sigma \delta_l$ $(l = 1, \ldots, a_{ij})$. This clearly means, by definition, that $\mathit{has\text{-}cons?} \delta K_{ij} \sigma$ holds. Furthermore, by definition of $\mathit{select\text{-}cons}\ \delta\ K_{ij}\ \sigma$, we effectively have the required abstract positive demand (because $d_l \lhd_\sigma \delta_l$). □

The next fact states that an appropriate safety relationship hold between $\mathit{mkVariant}_{ED}$ and $\mathit{mkVariant}_{AD}$.

Fact H.38 *($\mathit{mkVariant}_{AD}$ is safe)*

$$\mathsf{let}\ d' = \mathit{mkVariant}_{ED}\ K\ \mathsf{in}$$
$$\mathsf{let}\ (\delta', \sigma') = \mathit{mkVariant}_{AD}\ K\ \sigma\ \mathsf{in}$$
$$d' \lhd_{\sigma'} \delta'$$

Proof: Direct from the definitions of $\mathit{mkVariant}_{ED}$ and $\mathit{mkVariant}_{AD}$. □

As for the next fact, it states a similar relationship between $\mathit{mkComponent}_{ED}$ and $\mathit{mkComponent}_{AD}$ as long as $d \lhd_\sigma \delta$

Fact H.39 *($\mathit{mkComponent}_{AD}$ is safe)*

$$d \lhd_\sigma \delta$$
$$\Rightarrow$$
$$\text{let } d' = mkComponent_{ED}\ K\ l\ d \text{ in}$$
$$\text{let } (\delta', \sigma') = mkComponent_{AD}\ K\ l\ \delta\ \sigma \text{ in}$$
$$d' \lhd_{\sigma'} \delta'$$

Proof: Direct from the definitions of $mkComponent_{ED}$ and $mkComponent_{AD}$, together with the fact that $d \lhd_{\sigma'} \delta$ holds (because of Fact H.33). □

The next fact states that if an exact demand is safely approximated by an abstract one, then taking the union of the abstract one with any other (well-formed) abstract demand will also be a safe approximation.

Fact H.40 *(Union is safe)*

$$d \lhd_\sigma \delta_1 \wedge \delta_2 \in \mathsf{dom}\,\sigma$$
$$\Rightarrow$$
$$\text{let } (\delta', \sigma') = \cup\ \delta_1\ \delta_2\ \sigma \text{ in}$$
$$d \lhd_{\sigma'} \delta'$$

Proof: Immediate from Fact H.31 (the union is an upper-bound) and Fact H.34 (an upper-bound is safe). □

The next fact states that a similar relationship holds for the union of vectors of demands, fact which is used in the proof of Lemma 3.20.

Fact H.41 *(Vector union is safe)*

$$\overline{d} \lhd_\sigma \widetilde{\delta}_1 \wedge \mathsf{len}\,\widetilde{\delta}_1 = \mathsf{len}\,\widetilde{\delta}_2 = k \wedge \forall l \in \{1, \ldots, k\} \cdot \widetilde{\delta}_2(l) \in \mathsf{dom}\,\sigma$$
$$\Rightarrow$$
$$\text{let } (\widetilde{\delta}', \sigma') = \ \widetilde{\cup}\ \widetilde{\delta}_1\ \widetilde{\delta}_2\ \sigma \text{ in}$$
$$\overline{d} \lhd_{\sigma'} \widetilde{\delta}'$$

Proof: Similar to the proof for Fact H.36, p. 305. □

H.3.2 Proofs of Section B.6

The main lemma of Section B.6 relates the notion of safety (Defn. 3.40, p. 89) for abstract demands with the notion of membership in the associated concretization set (cf. Section B.6, p. 225):

Lemma B.1 *(Safety as membership in the concretization set)*

$$\delta \in \mathsf{dom}\, \sigma \wedge d \lhd_{\sigma} \delta \;\Rightarrow\; d \in conc\; \delta\; \sigma$$

Proof: The proof is by induction on the structure of d.

Base case: $d = @$. From $d \lhd_{\sigma} \delta$, this means $@ \in \sigma(\delta)$. Clearly then, from the definition of *conc* and $conc'$, $conc\; \delta\; \sigma$ will contain @.

Inductive case: $d = mk\text{-}EPDmd(K, [d_1, \ldots, d_k])$. From $d \lhd_{\sigma} \delta$, this means there is some $bd = mk\text{-}EPDmd(K, [\delta_1, \ldots, \delta_k]) \in \sigma(\delta)$ with $d_j \lhd_{\sigma} \delta_j$. By induction hypothesis, this then means that $d_j \in conc \delta_j\, \sigma$. In turn, $mk\text{-}EPDmd(K, [d_1, \ldots, d_k])$ $\in conc'\; bd\; \sigma$, with $bd \in \sigma(\delta)$, so we do have the required result.

□ (End of proof of Lemma B.1)

Bibliography

[AC86a] Arvind and D.E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[AC86b] Arvind and D.E. Culler. Managing resources in a parallel machine. In J.V. Woods, editor, *Fifth Generation Computer Architecture*, pages 103–121. Elsevier Science Publishers, 1986.

[ACGS91] B.S. Ang, A. Caro, S. Glim, and A. Shaw. An introduction to the Id compiler. Computation Structures Group Memo 328, Laboratory for Computer Science, MIT, 1991.

[ACM88] Arvind, D.E. Culler, and G.K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. Computation Structures Group Memo 279, Laboratory for Computer Science, MIT, 1988.

[AH87] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 9–31. Ellis Horwood Ltd., 1987.

[AI87] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing. Computation Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987.

[AJ89] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, g \rangle$-machine. In *Proc. of the Fourth International Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 202–213. ACM Press, 1989.

[All89] L. Allison. Circular programs and self-referential structures. *Software — Practice and Experience*, 19(2):99–109, Feb. 1989.

[All92] L. Allison. Lazy dynamic-programming can be eager. *Info. Proc. Letters*, 43(4):207–212, 1992.

[All93] L. Allison. Application of recursively defined data structures. *Australian Comp. Journal*, 25(1):14–20, 1993.

[AN87] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. Computation Structures Group Memo 271, Laboratory for Computer Science, MIT, 1987.

[AN90] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[ANP89] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, October 1989.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, Aug. 1978.

[BEJ88] D. Bjorner, A.P. Ershov, and N.D. Jones. *Partial Evaluation and Mixed Computation*. IFIP/North-Holland, Amsterdam, 1988.

[BH86] A. Bloss and P. Hudak. Variations on strictness analysis. In *Proc. of the 1986 ACM Conf. on Lisp and Funct. Progr.*, pages 132–142, 1986.

[BH88] A. Bloss and P. Hudak. Path semantics. In *Mathematical Foundations of Programming Languages Semantics*, pages 476–489. Springer-Verlag, LNCS-298, 1988.

[BHA85] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, pages 42–62. Springer-Verlag, LNCS-217, 1985.

[BHY88] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, 1988.

[Bir84] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[BJ82] D. Bjorner and C.B. Jones. *Formal Specification and Software Development*. Prentice-Hall International Series in Computer Science, 1982.

[Bjo78] D. Bjorner. Programming in the meta-language: A tutorial. In *The Vienna Development Method: The Meta-Language*, pages 24–217. Springer-Verlag, LNCS-61, 1978.

[Bjo87] D. Bjorner. *VDM '87, VDM — A Formal Method at Work*. Springer-Verlag, LNCS-252, 1987.

[BP89] M. Beck and K. Pingali. From control flow to dataflow. Technical Report TR-89-1050, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[BPR88] G.L. Burn, S.L. Peyton Jones, and J.D. Robson. The spineless G-Machine. In *Proc. of the 1988 ACM Conf. on Lisp and Funct. Progr.*, pages 244–258, 1988.

[Bur87] G.L. Burn. Evaluation transformers — A model for the parallel evaluation of functional languages (extended abstract). In *Functional Prog. Lang. and Comp. Arch.*, pages 446–470. Springer-Verlag, LNCS-274, 1987.

[Bur88] G.L. Burn. A shared memory parallel G-Machine based on the evaluation transformer model of computation. In *Proc. of the Workshop on Implementation of Lazy Functional Languages*, pages 301–330. Chalmers University of Technology, 1988.

[Bur89a] G. Burn. Overview of a parallel reduction machine project II. In *PARLE '89, vol. I*, pages 385–396. Springer-Verlag, LNCS-365, 1989.

[Bur89b] G.L. Burn. *Deriving a Parallel Evaluation Model for Lazy Functional Languages Using Abstract Interpretation*, volume Languages for Parallel Architectures, pages 111–160. J. Wiley and Sons., 1989.

[Bur90a] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *Proc. of the ACM Symp. on Principles of Prog. Lang.*, pages 151–156, 1990.

[Bur90b] G.L. Burn. Using projection analysis in compiling lazy functional languages. In *Proc. of the 1990 ACM Conf. on LISP and Funct. Prog.*, pages 227–241, 1990.

[Bur91a] G.L. Burn. The evaluation transformer model of reduction and its correctness. In *TAPSOFT '91, vol. 2*, pages 458–482. Springer-Verlag, LNCS-494, 1991.

[Bur91b] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1991.

[CA88] D.E. Culler and Arvind. Resource requirements of dataflow programs. In *Proc. of the 15th Annual International Symp. on Computer Architecture*, pages 141–150, 1988.

[CP85] C. Clack and S.L. Peyton Jones. Strictness analysis — A practical approach. In *Functional Prog. Lang. and Comp. Arch.*, pages 35–49. Springer-Verlag, LNCS-201, 1985.

[CP86] C. Clack and S.L. Peyton Jones. The four-stroke reduction engine. In *Proc. of the 1986 ACM Conf. on Lisp and Funct. Prog.*, pages 220–232, 1986.

[CSS+89] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. of the Fourth International Conf. on Architectural Support for Prog. Lang. and Oper. Systems*, pages 164–175, 1989.

[Den85] J.B. Dennis. Data flow computation. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 345–398. Springer-Verlag, 1985.

[DG92] J. Dennis and G. Gao. Evolution of multithreaded architectures. ACAPS Technical Memo 29, School of Computer Science, McGill University, Montreal, 1992.

[DK82] A.L. Davis and R.M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, Feb. 1982.

[Dyb87] P. Dybjer. Inverse image analysis. In *Automata, Languages and Programming*, pages 21–30. Springer-Verlag, LNCS-267, 1987.

[Dyb88] P. Dybjer. Inverse image analysis generalizes strictness analysis. In *Proc. of the Workshop on Implementation of Lazy Functional Languages*, pages 109–128. Chalmers University of Technology, 1988.

[EM91] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Automata, Languages and Programming*, pages 47–59. Springer-Verlag, LNCS-510, 1991.

[FB93] S. Finne and G. Burn. Assessing the evaluation transformer model of reduction on the spineless g-machine. In *FPCA '93*, pages 331–340. ACM Press, 1993.

[FW87] J. Fairbairn and S. Wray. TIM: A simple lazy abstract machine to execute supercombinators. In *Functional Prog. Lang. and Comp. Arch.*, pages 34–45. Springer-Verlag, LNCS-274, 1987.

[Geo89] L. George. An abstract machine for parallel graph reduction. In *Proc. of the Fourth International Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 214–229. ACM Press, 1989.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–53, 1978.

[GH87] B. Goldberg and P. Hudak. Alfalfa: Distributed graph reduction on a hypercube multiprocessor. In *Graph Reduction*, pages 94–113. Springer-Verlag, LNCS-279, 1987.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proc. of the 1986 ACM Conf. on Lisp and Funct. Prog.*, pages 28–38, 1986.

[Gol88a] B. Goldberg. Buckwheat: Graph reduction on a shared memory multiprocessor. In *Proc. of the 1988 ACM Conf. on Lisp and Funct. Prog.*, pages 40–51, 1988.

[Gol88b] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Dept. Of Computer Science, Yale University, 1988.

[GTWW78] J.A. Goguen, J. Thatcher, E. Wagner, and J.B. Wright. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, R. Yeh (ed.).* Prentice-Hall, 1978.

[Hal87] C.V. Hall. *Strictness Analysis Applied to Programs with Lazy List Constructors.* PhD thesis, Dept. Of Computer Science, Indiana University, 1987.

[Hel89] S.K. Heller. Efficient lazy data-structures on a dataflow machine. Technical Report MIT/LCS/TR-438, Laboratory for Computer Science, MIT, Feb. 1989.

[Hen80] P. Henderson. *Functional Programming — Application and Implementation.* Prentice-Hall International Series in Computer Science, 1980.

[Hen90] L.J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[HG85] P. Hudak and B. Goldberg. Efficient distributed evaluation of functional programs using serial combinators. In *Proc. of the 1985 International Conf. on Parallel Processing*, pages 831–839, 1985.

[HGW91] P.H. Hartel, H. Glaser, and J. Wild. On the benefits of different analyses in the compilation of a lazy functional language. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 123–145. University of Southampton, 1991.

[HKL92] G. Hogen, A. Kindler, and R. Loogen. Automatic parallelization of lazy functional programs. In *ESOP '92 — European Symp. on Programming*, pages 254–268. Springer-Verlag, LNCS-582, 1992.

[HL91] R.J.M Hughes and J. Launchbury. Towards relating forwards and backwards analyses. Presented at Workshop on Computing, Glasgow Functional Programming, 1991.

[HL92] R.J.M. Hughes and J. Launchbury. Reversing abstract interpretations. In *ESOP '92 — European Symp. on Programming*, pages 269–286. Springer-Verlag, LNCS-582, 1992.

[HL93] P.H. Hartel and K.G. Langendoen. Benchmarking implementations of lazy functional languages. In *FPCA '93*, pages 341–349. ACM Press, 1993.

[HR87] P.G. Harrison and M.J. Reeve. The parallel graph reduction machine, ALICE. In *Graph Reduction*, pages 181–202. Springer-Verlag, LNCS-279, 1987.

[HS88] P. Hudak and Anderson S. Haskell solution to the language session problem at the 1988 salishan high-speed computing conference. Technical Report YALEU/DCS/RR–627, Dept. of Computer Science, Yale University, Jan. 1988.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood Ltd., 1987.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), Sept. 1989.

[Hug84] J. Hughes. Why functional programming matters. Programming Methodology Group Memo PMG-40, CTH Gothenburg, 1984.

[Hug85] J. Hughes. Strictness detection in non-flat domains. In *Programs as Data Objects*, pages 112–135. Springer-Verlag, LNCS-217, 1985.

[Hug87] J. Hughes. Analyzing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 63–102. Ellis Horwood Ltd., 1987.

[Hug88] J. Hughes. Backwards analysis of functional programs. In *Partial Evaluation and Mixed Computation*, pages 187–208. IFIP/North-Holland, 1988.

[Hug89] J. Hughes. Abstract interpretation — Tutorial, Sept. 1989. Tutorial presented at FPCA '89.

[Hun90] S. Hunt. PERs generalise projections for strictness analysis. Research Report DOC 90/14, Imperial College, October 1990.

[HW87] R.J.M. Hughes and P. Wadler. Projections for strictness analysis. In *Functional Prog. Lang. and Comp. Arch.*, pages 385–407. Springer-Verlag, LNCS-274, 1987.

[HW89] C.V. Hall and D.S. Wise. Generating function versions with rational strictness patterns. *Science of Computer Programming*, 12(1):39–74, June 1989.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda-calculus. In *Proc. of the 13th Annual ACM Symp. on Principles of Prog. Lang.*, pages 97–109, 1986.

[JL90] S.B. Jones and D. LeMetayer. A new method for strictness analysis on non-flat domains. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, pages 1–11. Stringer-Verlag, 1990.

[Joh83] T. Johnsson. The G-machine: An abstract machine for graph reduction. In *Declarative Programming Workshop*, pages 1–20. University College, London, 1983.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Prog. Lang. and Comp. Arch.*, pages 190–203. Springer-Verlag, LNCS-201, 1985.

[Joh87] T. Johnsson. Target code generation from G-machine code. In *Graph Reduction*, pages 119–159. Springer-Verlag, LNCS-279, 1987.

[Jon78] C.B. Jones. The meta-language: A reference manual. In *The Vienna Development Method: The Meta-Language*, pages 218–277. Springer-Verlag, LNCS-61, 1978.

[Kam92] S. Kamin. Head-strictness is not a monotonic abstract property. *Info. Proc. Letters*, 41(4):195–198, 1992.

[Kel85] R.M. Keller. Rediflow architecture prospectus. Technical Report UUCS-85-105, University of Utah, 1985.

[KM87] T.-M. Kuo and P. Mishra. On strictness and its analysis. In *Conf. Record of the 14th ACM Symp. on Principles of Prog. Lang.*, pages 144–155, 1987.

[KM89] T.-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. of the 1989 Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 260–272, 1989.

[KPR+92] O. Kaser, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Fast parallel implementation of lazy languages – the EQUALS experience. In *Proc. of the 1992 ACM Conf. on Lisp and Funct. Prog.*, pages 335–344, 1992.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Lau88] J. Launchbury. Projections for specialisation. In *Partial Evaluation and Mixed Computation*, pages 299–315. IFIP/North-Holland, 1988.

[Lau93] J. Launchbury. Lazy imperative programming. In *SIPL '93: ACM SIGPLAN Workshop on State in Programming Languages*, pages 46–56. YALEU/DCS/RR-968, June 1993.

[Les87] D. Lester. The G-Machine as a representation of stack semantics. In *Funct. Prog. Lang. and Comp. Arch.*, pages 46–59. Springer-Verlag, LNCS-274, 1987.

[LH92] K. Langendoen and P.H. Hartel. FCG: A code generator for lazy functional languages. In *Compiler Construction*, pages 278–296. Springer-Verlag, LNCS-641, 1992.

[Lin86] G. Lindstrom. Static evaluation of functional programs. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 196–206, 1986.

[Lin88] G. Lindstrom. Static analysis of functional programs with logical variables. In *Prog. Lang. Implementation and Logic Programming*, pages 1–19. Springer-Verlag, LNCS-348, 1988.

[LLY87] G. Lindstrom, George. L., and D. Yeh. Generating efficent code from strictness annotations. In *TAPSOFT '87*, pages 140–154. Springer-Verlag, LNCS-250, 1987.

[LM91] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher-order lazy languages. In *FPCA '91*, pages 328–351. Springer-Verlag, LNCS-523, 1991.

[Maa92] A. Maaβen. Parallel programming with data structures and higher order functions. *Science of Computer Programming*, 18(1):1–38, April 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. of the International Symp. on Programming*, pages 269–281. Springer-Verlag, LNCS-83, 1980.

[NA90] R.S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[Nik88] R.S. Nikhil. Id (Version 88.0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, March 1988.

[Nik90] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.

[Nik91] R.S. Nikhil. Id language reference manual (Version 90.1). Computation Structures Group Memo 284-2, Laboratory for Computer Science, MIT, July 1991.

[NM92] M. Neuberger and P. Mishra. A precise relationship between the deductive power of forward and backward strictness analysis. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 127–138. ACM Press, 1992.

[NPA92] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, 1992.

[ORH93] M. Odersky, M. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *Proc. of the 20th ACM Symp. on Principles of Prog. Lang.*, pages 43–56, 1993.

[PC87] S.L. Peyton Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 246–265. Ellis Horwood Ltd., 1987.

[PCS87] S.L. Peyton Jones, C. Clack, and J. Salkild. GRIP — A parallel graph reduction machine. *ICL Technical Journal*, 5(3):595–599, May 1987.

[PCS89] S.L. Peyton Jones, C. Clack, and J. Salkid. High-performance parallel graph reduction. In *PARLE '89, vol. I*, pages 193–206. Springer-Verlag, LNCS-365, 1989.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

[Pin86] K. Pingali. *Demand-Driven Evaluation on Dataflow Machines.* PhD thesis, MIT, 1986.

[Pin88] K. Pingali. Lazy evaluation and the logic variable. In *Proc. of the 1988 International Conf. on Supercomputing*, pages 560–572, 1988.

[PJP89] K. Pingali, R. Jagadeesan, and P. Panangaden. A fully abstract semantics for a functional language with logic variables. Technical Report 89-969, Cornell University, 1989.

[Sch86] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[SCvE91] K.E. Schauser, D.E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *FPCA '91*, pages 50–72. Springer-Verlag, LNCS-523, 1991.

[SPR90] R.C. Sekar, S. Pawagi, and I.V. Ramakrishnan. Small domains spell fast strictness analysis. In *Proc. of the ACM Symp. on Principles of Prog. Lang.*, pages 169–183, 1990.

[SRI91] V. Swarup, U.S. Reddy, and E. Ireland. Assignments for applicative languages. In *FPCA '91*, pages 192–214. Springer-Verlag, LNCS-523, 1991.

[Ste84] G.L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[Sto81] J.E. Stoy. The congruence of two programming languages definitions. *Theoretical Computer Science*, 13:151–174, 1981.

[TBH82] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. *Computing Surveys*, 14(1):93–143, March 1982.

[TCS92] K.R Traub, D.E. Culler, and K.E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. of the 1992 ACM Conf. on Lisp and Funct. Prog.*, pages 324–334, 1992.

[TGH93] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. Speculative execution and branch prediction on parallel machines. In *Conf. Proceedings 1993 Intl. Conf. on Supercomputing*, pages 77–86, July 1993.

[Tra86] K.R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's thesis, Dept. of Electrical Eng. and Comp. Sci., MIT, 1986.

[Tra88] K.R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

[Tra91] K.R. Traub. Multi-thread code generation for dataflow architectures from non-strict programs. In *FPCA '91*, pages 73–101. Springer-Verlag, LNCS-523, 1991.

[Tre89] G. Tremblay. A survey of graph reduction. Unpublished note, Dec. 1989.

[Tre90] G. Tremblay. Some issues for the implementation of lazy evaluation on dataflow architectures. Unpublished note, March 1990.

[Tre91] G. Tremblay. Strictness analysis based on forward abstract interpretation can not express head-strictness. ACAPS Technical Note 33, School of Computer Science, McGill University, Montréal, Qué., September 1991.

[Tur81] D.A. Turner. The semantics elegance of applicative languages. In *Proc. of the 1981 Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 85–98, 1981.

[Wad87] P. Wadler. Strictness analysis on non-flat domains. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood Ltd., 1987.

[Wad92] P. Wadler. The essence of functional programming. In *Proc. of the 19th ACM Symp. on Principles of Prog. Lang.*, pages 1–14, 1992.

[WG88] Y.H. Wei and J.L. Gaudiot. Demand-driven interpretation of FP programs on a data-flow multiprocessor. *IEEE Trans. on Computers*, 37(8):946–966, Aug. 1988.

[Wri91] D.A. Wright. A new technique for strictness analysis. In *TAPSOFT '91, vol. 2*, pages 235–258. Springer-Verlag, LNCS-494, 1991.

[WW87] P. Watson and I. Watson. Evaluating functional programs on the Flagship machine. In *Functional Prog. Lang. and Comp. Arch.*, pages 80–97. Springer-Verlag, LNCS-274, 1987.

[Yeh90] D. Yeh. *Static Analysis of Multiparadigm Languages.* PhD thesis, The University of Utah, Dec. 1990.

[YO88] J. Young and P. O'Keefe. Experience with a type evaluator. In *Partial Evaluation and Mixed Computation*, pages 573–581. IFIP/North-Holland, 1988.

[You89] J. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.* PhD thesis, Yale University, May 1989.