Auxiliary Computations

A Framework for a Step-Wise, Non-Disruptive Introduction of Static Guarantees to Untyped Programs Using Partial Evaluation Techniques

Stephan Andreas Herhut

January 2010

A thesis submitted to the University of
Hertfordshire in partial fulfilment of the
requirements of the degree of

Doctor of Philosophy

The programme of research was carried out in
the School of Computer Science,
Faculty of Science, Technology and Creative Arts,
University of Hertfordshire.

Abstract

Type inference can be considered a form of partial evaluation that only evaluates a program with respect to its type annotations. Building on this key observation, this dissertation presents a uniform framework for expressing computation, its dynamic properties and corresponding static type information. By using a unified approach, the static phase divide between values and types is lifted. Instead, computations and properties can be freely assigned to the static or dynamic phase of computation. Even more, moving a property from one world to the other does not require any program modifications.

This thesis builds a bridge between two worlds: That of statically typed languages and the dynamically typed world. The former is wanted for the offered static guarantees and detection of a range of defects. With the increasing power of type systems available, the kinds of errors that can be statically detected is growing, nearing the goal of proving overall program correctness from the program's source code alone. However, such power does come for a price: Type systems are becoming more complex, restrictive and invasive, to the point where specifying type annotations becomes as complex as specifying the algorithm itself.

Untyped languages, in contrast, may provide less static safety but they have simpler semantics and offer a higher flexibility. They allow programmers to express their ideas without worrying about provable correctness. Not surprisingly, untyped languages have a strong following when it comes to prototyping and rapid application development.

Using the framework presented in this thesis, the programmer can have both: Prototyping applications using a dynamically typed approach and gradual refinement of prototypes into programs with static guarantees.

Technically, this flexibility is achieved with the novel concept of *auxiliary computations*. Auxiliary computation are additional streams of computation. They model, next to the data's computation, the computation of property of data. These streams thereby may depend on the actual data that is computed, as well as on further auxiliary computations. This expressiveness brings auxiliary computations into the domain of dependent types.

Partial evaluation of auxiliary computations is used to infer static knowledge from auxiliary computations. Due to the interdependencies between auxiliary computations, evaluating only those parts of a program that contribute to a property is non trivial. A further contribution of this work is the use of *demands on computations* to narrow the extent of partial evaluation to a single property. An algorithm for demand inference is presented and the correctness of the inferred demands is shown.

Acknowledgements

Foremost, I would like to thank my principal supervisor and mentor Sven-Bodo Scholz. He caught my interest in programming language design in the first place and kept it alive ever after. Without his guidance, encouragement and support, this thesis would not have been possible. Furthermore, my thanks go to my secondary supervisor Alex Shafarenko for insightful discussions and the provision of the well-equipped research environment I was lucky to benefit from.

Many thanks also go to the research team working on SaC. They always made me feel part of a bigger movement, even as my work progressed away from SaC. In particular, I would like to thank Clemens Grelck for his interest in my research and the various hours spent on feedback for my papers on SaC. Robert Bernecky deserves my gratitude for introducing me to the magic of array programming, prototyping in dynamic languages and APL.

I am grateful to my examiners, Bruce Christianson, Olivier Danvy and Kevin Hammond for their time, interest and helpful feedback. It was a pleasure to present my work and defend my thesis before them.

A PhD is more than research and a thesis: It is a journey. I thank my fellow PhD students for their support, friendliness and patience during my travel and for distracting me when necessary and deviating me to new roads. Jun Li introduced me to the Chinese side of life and opened my eyes in many regards. Frank Penczek endured many long evenings discussing research life and beyond. And Michael Hicks and Daniel Rolls provided the counterbalance with their British perspective, even on German philosophers.

During the final stage of my PhD, many people provided me with a place to write. Amongst others, I would particularly like to thank the people at Starbucks Haymarket in Edinburgh, the Blum family and Anja Rosenthal for their hospitality.

Finally, I would like to express my deep gratitude to my family: My parents, who always supported me throughout my studies and never lost their belief in me. My brother and sister, for their understanding and help in busy times. And to Janine and Mikko, for making me smile every day.

Contents

List of Figures

List of Symbols

LRec	basic expression language with records
$\mathscr{L}_t$	lowering scheme for resolving tags in LRec
$\mathscr{L}_e$	lowering scheme for resolving implicit equality constraints in LRec
$\mathscr{R}_e$	auxiliary rewriting scheme used in $\mathscr{L}_e$
$\mathscr{L}_i$	lowering scheme for resolving implicit labels for values in LRec
LRec$_\mathrm{C}$	core expression language with records
$\mathscr{L}_c$	lowering scheme for rewriting function guards in LRec
$\mathscr{F}\mathscr{V}$	set of free variables of an expression in LRec
$\mathscr{L}$	set of all labels in LRec
$\mathscr{V}$	set of all values
$\mathscr{R}$	set of all record values
dom	domain of record value
range	range of record value
elem	element of record value
clos	transitive closure
$\prec$	strict partial order on labels
$\mathscr{F}$	function environment for evaluation
$\mathscr{E}$	variable environment for evaluation
$\perp$	empty function environment
$\Downarrow$	relation for full evaluation of expressions in LRec$_\mathrm{C}$
$\leftarrow$	insertion of a new binding into the variable environment
$\overset{v}{=}$	equality on values
$\vec{\prec}$	lifting of $\prec$ to pattern
$\mathscr{P}$	powerset of a set

$\lvert \cdot \rvert$	cardinality of a set or tuple
$\mathscr{V}_p$	set of all partial values
$\mathscr{R}_p$	set of all partial record-values
dom_p	domain of a partial record-value
$\overline{\mathrm{dom}}_p$	anti-domain a of partial record-value
range_p	range of a partial record-value
elem_p	element of partial record-value
$\downarrow$	relation for partial evaluation of expressions in LREC$_\mathrm{C}$
$\sqsubseteq$	relation between values and corresponding partial values
$\overset{\rightarrow}{\sqsubseteq}$	relation between variable environments and corresponding partial variable environments
$\uparrow$	lifted demand
$\vdash$	demand context
δ	meta variable for demands
$\mathscr{S}$	set of demand contexts
$\mathscr{D}$	set of all demands
$\mathscr{D}_r$	set of all record demands
elem_d	sub-demand of a demand
dom_d	domain of a demand
$\overset{\delta}{\sqsubseteq}$	demand satisfaction
$\uplus$	union of demands
LREC$_\mathrm{D}$	core expression language with demand annotations
$\mathscr{A}$	demand inference scheme
$\mathscr{E}_A$	variable environment for demand inference
δ_c	demand context during demand inference
$\triangleleft$	join of demand contexts
$\overset{d}{\leftarrow}$	insertion of a new binding into the variable environments with demands
$\Uparrow$	lifting of demands
nest	nesting of demands
$\overline{\mathrm{empty}}$	predicate for non-empty demands

empty	predicate for empty demands
extract	extraction of a demand from a set of demand annotations
$\mathscr{E}_\delta$	variable environment with demands
$\overset{\overset{\delta_c}{\rightarrow}}{\sqsubseteq}$	demand satisfaction for variable environments
$\overset{d}{\equiv}$	matching of function environments and function environments with demands
$\mathscr{F}_A$	function environment for demand inference

1. Introduction

"Well-typed programs don't go wrong." [Milner, 1978] This has been the main selling argument for statically typed languages over the last decades. And indeed, although not true in general, static type systems are powerful enough to rule out common program errors at compile time. With the advent of dependent type systems, the set of errors caught at compile time has even further increased. Even the ultimate goal, *i.e.*, putting the Curry-Howard isomorphism to work by embedding the correctness proof inside the actual program, has been reached for a set of problems [Sheard, 2004, 2005].

Nonetheless, dynamically typed languages, *i.e.*, languages that perform compliance checks at runtime that would be dealt with by the static type system at compile time, are still widely used and popular. This is commonly attributed to three main advantages dynamically typed languages have over their statically typed counterparts:

expressiveness Well-typedness is only an approximation of program correctness. All programs a static type system accepts are correct with respect to the properties assured by the concrete type system. However, the inverse is not true. Static type systems reject programs that behave correctly at runtime. A classical example is the lambda expression $\lambda x.(xx)$, which cannot usually be assigned a type. However, in dynamically typed languages, expressions as the one above can easily be specified and executed.

genericity In untyped languages, genericity comes for free. As a simple example, consider the identity function $\lambda x.x$. In a dynamically typed language, the above can be implemented and applied to any value. A statically typed language needs to support techniques like type polymorphism to allow for generic functions as simple as the identity. As a consequence, in statically typed languages like C [Kernighan and Ritchie, 1988] or JAVA [Gosling et al., 2005] the above function cannot be specified. The same holds for more complex functions, *e.g.*, adding two matrices. In a dynamically typed language, it suffices to map the plus operation to each element of the matrix. As long as there is a plus operation available for the given elements, the program will just evaluate to the expected result. In statically typed languages, complex type annotations are required to make this work. In HASKELL [Peyton-Jones, 2003], for example, the programmer has to encode the desired genericity by means of type classes. In JAVA, the desired behaviour can be achieved using a hierarchy of classes. However, in all cases, considerably more work is required.

simplicity Dynamically typed languages have simpler semantics. The meaning of a

> program is completely described by a single set of evaluation rules. In particular, dynamically typed languages do not need to cater for limitations of the type system. In statically typed languages, the programmer has to additionally understand and master the particularities of a given type system. The more complex these become, the more difficult understanding the actual meaning of a program gets. This additional complexity has led to the wide-spread perception that programming in languages with sophisticated type systems, *e.g.*, functional programming languages like HASKELL or ML [Milner et al., 1997], is difficult and requires particularly skilled programmers.

Thus, overall, dynamically typed languages are of advantage whenever implementation time and thus cost is an issue or where programmers are interested in results without investing time in understanding the underlying type theory behind a given programming language. Therefore, not surprisingly, dynamically typed languages are used commonly for prototyping and in numerical applications.

For the former, the correctness of the application is not the major concern. Instead, the goal of a prototype usually is to prove the viability of a concept. Traditionally, once the prototype has proven to be successful, the problem at hand is re-implemented as a production-quality application. In this stage of a project, correctness is more of a concern. Thus, for the final implementation, statically typed languages provide an advantage. Their static guarantees help sieve out bugs early in the development cycle and reduce the maintenance costs during the life time of the application. However, software projects increasingly adapt what is referred to as an *agile development process* [Beck et al., 2001; Larman and Basili, 2003; Martin, 2003]. Instead of discarding the initial prototype and starting from scratch, in agile development the prototype is refined in multiple stages to a final product. The main advantage of this process is that the application is available earlier and that no development effort is lost. On the other hand, agile development does not allow the programmer to easily change the implementation language during the development process. Instead, the language chosen for the prototype has to be used for the final application, as well. This leads to a dilemma: Choosing a dynamically typed language aids the rapid development in the early stages of the program but impairs the correctness requirements and maintenance cost in later stages. Using a statically typed language, on the other hand, slows down initial progress but helps achieve the goals in later stages. To solve this dilemma, a language that supports both dynamically and statically typed programming is required.

A similar situation is given in the field of numerical applications. Here, MATLAB [MathWorks, 2009] is a widely used language for implementing initial prototypes. Being dynamically typed, it allows for rapid program development. Furthermore, it allows numerical scientists to concentrate on what they do best: specify the mathematical algorithm for their specific problem. In this setting, program correctness is not the driving force behind the desire to use more statically typed languages. Instead, in this scenario, the reduced overhead of statically typed languages is what really matters. As dynamically typed languages employ no static analysis whatsoever, all conformity checks have to be performed at runtime. In the above example of adding two matrices, the runtime behaviour of a naïve implementation is determined mostly by checking the conformity of each pair of elements to be added. But even for less naïve implementations, conformity checks at runtime have a significant impact on program performance. Thus, once the suitability of an algorithm has been proven, commonly numerical applications are re-implemented in a more statically typed language like FORTRAN [Adams et al., 1997] or HPF [Hig, 1994]. Such a re-implementation, however, comes at a huge cost. The entire application needs to be translated to a new language, potentially introducing algorithmic flaws due to transscription errors. Thus, such a translation needs to be performed with extreme care, and requires expertise not only on the target language but on the problem at hand, as well. Often, it would suffice to exploit static information for computationally expensive kernels of the application. This task might even be performed not by the numerical scientist that implemented the actual algorithm but by a specialised programmer who has experience in performance tuning rather than numerical applications. Applying agile development techniques in this setting, *i.e.*, to refine the dynamic prototype into a semi-static implementation over time, would require a language that allows to gradually refine a dynamically typed implementation with static knowledge.

As these examples show, combining dynamic typing and static typing in a single language offers clear advantages.

- in early stages of the development, the programmer can concentrate on implementing the actual algorithm
- once the implementation has stabilised, static type annotations can be added, increasing overall program robustness and program performance
- this task can be performed by a specialist, potentially even with limited knowledge of the program domain
- as only parts of the program need to be amended, less initial development effort is wasted

In this thesis, I explore how such a combination of static and dynamic typing can be achieved whilst retaining much of the benefits of dynamic typing.

1.1. Existing Approaches

Unsurprisingly,given its promising advantages, the idea to combine static and dynamic typing in a single language is by no means new. Existing languages like Microsoft's Visual Basic.NET provide static typing and type inference for certain constructs of the language. A more complete approach is taken by Cartwright and Siek with their soft typing [Cartwright and Fagan, 1991] and gradual typing [Siek and Taha, 2006] approaches. In soft typing, a classical type system is applied to a dynamically typed language. However, instead of rejecting ill-typed programs, dynamic checks are inserted into the code to offload type checking to the runtime. In gradual typing, the opposite approach is taken. Initially, all expressions are assigned with a dynamic type, which is then gradually refined by programmer specified type annotations and inferred static knowledge. At the interface between dynamically typed and statically typed codes, appropriate runtime checks are inserted.

At first glance, such gradual introduction of types solves the problem. However, both approaches apply standard type systems that are limited to a predefined set of static information. Coming back to our previous example of adding two matrices, these systems are able to check whether the element types of the two matrices match. Thus, costly dynamic checks can be avoided and errors are found at compile time, if sufficient static knowledge is available. However, more complex properties cannot be modelled. In our example, these systems would not be able to statically check whether the two matrices have a matching shape, *i.e.*, whether the number of elements per axis matches. The conformity of the shapes, however, is an important property required by the semantics of matrix addition.

To eliminate runtime checks for shape constraints like the one above and catch corresponding errors at compile time, more powerful type systems are needed. For example, Zenger's indexed types [Zenger, 1997, 1998] and extensions thereof [Trojahner and Grelck, 2009] can be used to model these properties. Indexed types allow the programmer to use values like integers within the type language. In our example, the type of the matrix would include, additionally to the type of the matrix's elements, the extent along each axis. As these properties are value dependent, not all problems can be modelled in Zenger's type system. In particular those cases, where the shape of a matrix depends on a value, which is statically not known, *e.g.*, a value read at runtime or the result of a complex computation, the well-typedness of a program cannot be statically shown and thus the program is rejected. This strictness, applied to dynamic languages, would rather dramatically impact the expressiveness of these languages, which is one of their key benefits.

To overcome these limitations, techniques like hybrid type checking [Flanagan, 2006] have been developed. In hybrid type checking, a dependent type system like Zenger's indexed types is used. However, instead of rejecting programs where insufficient static knowledge is available to prove their correctness statically, in hybrid typing the missing static knowledge is assumed and appropriate runtime checks are inserted. Thus, where possible the program is proven to be correct and only where needed runtime checks remain.

For many interesting properties, a fixed type system like Zenger's indexed types, even if extended by ideas from hybrid typing, is still not good enough. As an example, once more consider element-wise adding two matrices. Knowing that one of the arguments has certain structural properties can dramatically improve runtime performance. For instance, if one of the arguments to the addition is the unit matrix, adding the two matrices requires only adding `1` to all elements along the diagonal of the non-unit argument. In the context of highly optimising APL [International Standards Organization, 1993] interpreters, exploiting knowledge similar to the unit matrix property from the example above has been shown to yield dramatic runtime improvements for real-world applications [Bernecky, 1998]. To be able to express diverse structural properties like the one above, programmers need to be able to adapt the type system to their needs. Fully dependent languages like Cayenne [Augustsson, 1998] or decidable subsets like Epigram [McBride, 2004; McBride and McKinna, 2004] allow for this. However, in the context of untyped languages, even if combined with approaches like hybrid typing, these languages hardly fulfil the simplicity property of dynamically typed languages.

1.2. My Approach

All the approaches named above that combine static and dynamic typing have in common that they, in some form, extend a dynamic programming language with a type system or a statically typed language with dynamic components. Whichever way, the resulting language ultimately entails a type system. In this thesis, I explore a different approach.

The key insight that has triggered the research presented in this thesis is that, on an abstract level, *type inference can be considered a form of partial evaluation that evaluates the program with respect to its type annotations.* From that perspective, a statically typed programming language consists of two languages in one: An implementation language to write the actual computation in and a further type language to express type annotations and constraints with. The former is evaluated at runtime, whereas the latter is computed statically at compile time. Furthermore, both languages usually have their own syntax and semantics.

Using two separate languages has the advantage that each can be designed to best serve its purpose. For example, a huge body of research covers how to design a type language such that all valid expressions in that language can be evaluated, *i.e.*, such that all valid expressions terminate. On the downside, however, the programmer has to learn both languages. Furthermore, even though both languages can be designed specifically for their use, the expressiveness of the combined end result is determined by the expressiveness of both parts.

Dependently typed programming languages begin to unify those two languages. As in their setting types may contain values, quite naturally parts of the implementation language are required in the type language, as well. When followed through, this trend will ultimately lead to a single programming language used for both, the implementation and the corresponding type annotations. Experimental approaches like Epigram already come rather close to this setting.

Even though the languages for implementation and type annotations are increasingly unified, types and values are still kept separate. In particular, in all languages that feature a type system of some sort, the distinction between type annotations, which are statically computed, and the actual implementation of an algorithm that is computed at runtime is predetermined.

Yet, is such a separation strictly required or can the implementation and type language be fully unified, to the point where types are just ordinary expressions? And if such an approach is feasible, what features does a corresponding language need to have such that it can be used in an agile setting, *i.e.*, how can a language support a seamless transition from dynamic to more statically checked implementations?

In this thesis I provide an answer to these two questions. I present a language that allows the programmer to encode properties of data alongside the actual computation. These properties are similar to types but differ in two significant aspects: Firstly, the properties are expressed in the dynamic programming language itself. Thus, the same syntax and semantics apply to them. Secondly, the annotations are not computed a priori statically. Instead, they are part of the dynamic program. Furthermore, to support an agile development style, I incorporate a pattern matching based programming style that, in the spirit of subtyping as known from object oriented languages, allows one to ignore additional properties where they are not needed.

Using a single language for the program and its properties simplifies the programmer's view. However, it imposes a challenge when trying to prove properties statically: As properties are expressed in the same language as computations on the actual data, both become indistinguishable. Thus, it is not clear which parts to evaluate to prove a certain property. It is not even clear anymore which properties a program encodes.

To solve the latter problem, I introduce the concept of *auxiliary computations.* An auxiliary computation is a named additional stream of computation alongside the main stream of computation, *i.e.*, alongside the computation that computes the actual result data. A single computation thereby may be accompanied by multiple auxiliary computations. As these additional streams of computation are named, they can be distinguished from each other and from the computation of the main value. Thus, to compute a property of an expression, it suffices to compute the corresponding auxiliary computation.

This step, however, is not as trivial as it may at first glance seem. Auxiliary computations need not be separate from each other. Like in dependent types, where types may depend on values, different auxiliary computations may depend on each other or even on a value from the main stream of computation. Allowing for such dependencies vastly improves the expressiveness of auxiliary computations, much like dependent types are more expressive than regular types. This expressiveness, however, comes at a price. It is, in this setting, still not clear which parts of a program need to be computed to show a certain property. Even more, it is not clear whether it is at all possible to guide partial evaluation such that only desired properties are computed.

> The main contribution of this thesis, supplemental to the concept of auxiliary computations and the design of a corresponding programming model, is therefore to show that partial evaluation can be instrumented such that only desired properties of a program are evaluated.

I do not propose an entire new language in this thesis. Instead, I have developed my approach as an extension to existing dynamically typed languages, in the hope that this will foster its adaptation in a wide range of languages. My approach can be applied to any pure language, as long as an existing framework for evaluation exists.

1.3. Overview

The remainder of this thesis is structured as follows. The next chapter gives an extended discussion of the above examples and motivates the design of an extension to a dynamically typed language to support auxiliary computations and constraint checking by means of contracts. The findings are then presented more formally in Chapter 3, culminating in the syntax and semantics of LRec. An extension of LRec by contracts to encode arbitrary constraints on function arguments is described separately in Chapter 4. Based on this definition of LRec, Chapters 5 and 6 discuss the exploitation of static information by partial evaluation. Finally, Chapter 7 provides some conclusions and ideas for future research directions.

2. Design Decisions

In this chapter, I revisit my initial example presented in the introduction to show the challenges and the resulting design decisions I have made. I thereby focus mainly on the programmer's perspective. First, the following section investigates how to support auxiliary computations in general. Using the general design described there, I explore in Section 2.2 how to efficiently program with auxiliary computations and how to elegantly exploit the additional knowledge they encode. Next, Section 2.3 explores how to model constraints like the earlier mentioned requirement of shape equality in matrix addition. Finally, Section 2.4 demonstrates my design in action. With the example of an evaluator for a small expression language, I demonstrate how a fully dynamic implementation can gradually be refined into an evaluator with static guarantees. I close this chapter with some design conclusions in Section 2.5.

2.1. A Data Representation for Auxiliary Computations

The key idea behind auxiliary computations is *to make the computations of properties of data explicit in the program text.* As a running example, I will use the aforementioned addition of two matrices. This choice is motivated by two general observations. Firstly, matrices in general, although being a relatively simple data structure, are widely used in numerical applications, one of the key target areas of my approach. Secondly, despite their simple structure, matrices provide a rich set of structural properties. One particularly important property is the shape of a matrix, *i.e.*, its extent along the two axes. As an example, consider the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

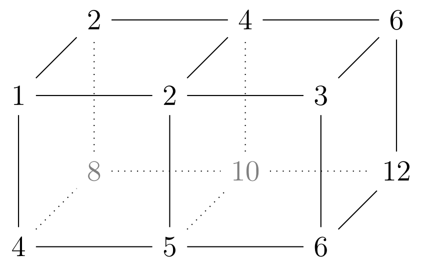
As can be seen, the above matrix has 3 rows and 4 columns. Its shape, using a row-major encoding, is thus 3×4, or encoded as a vector `[ 3, 4]`. I will use the latter encoding in all the examples provided in this thesis.

By using an explicit encoding of the shape, the actual data of the matrix, *i.e.*, its elements, can be encoded as a vector, as well. Using again row-major encoding, the above matrix can be encoded as the following vector by concatenating each row vector:

```
[ 1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12]
```

I will refer to this vector encoding as the *ravel* or *unrolling* of the matrix. Note that the shape vector `[ 3, 4]` provides a sufficient description of the structure of the matrix to reconstruct the initial matrix from its ravel.

This encoding can easily be extended to general n-dimensional arrays. All that is required is to allow shape vectors of arbitrary length. For example, a 3 dimensional array of the form:



can be encoded using a shape vector `[ 2, 3, 2]` and a ravel of

```
[ 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12]
```

In this context, row-major encoding refers to the concatenation of the elements of an array starting form the outermost axis. As with matrices, this encoding suffices to reconstruct the initial n-dimensional array from the ravel and shape vector. However, general arrays are distinguished by a further property, their *rank* or dimensionality. In principle, the current encoding suffices to represent these two structural properties of arrays: The rank can be computed as the length of the shape vector. However, it can nonetheless be of advantage to explicitly encode the rank of an array. An explicit encoding saves computing the length of the shape vector at runtime. More importantly, however, the shape of an array might not be statically known, whereas the rank in most cases is. In these cases, computing the rank of an array via its shape vector complicates the inference of static knowledge. I therefore, throughout this thesis, will use an encoding that makes both shape and rank explicit.

The encoding of n-dimensional arrays using a ravel and shape vector is neither unique to my approach nor is it the only possible encoding. Many array languages, *e.g.*, APL [Bernecky and Berry, 1993; International Standards Organization, 1993], J [Hui and Iverson, 2004] and SaC [Scholz, 2003] to name few, use an encoding based on a ravel of data elements and a shape vector to describe the structure of the data. The main difference between languages like APL on the one hand and SaC on the other is what structures are supported. Whereas SaC only allows for homogeneous arrays, *i.e.*, arrays where all elements are of the same type, languages like APL and J go one step further by enabling the use of arrays as elements of an array. Such array elements are then treated as scalar elements, thus providing a means to express arbitrarily nested and even inhomogeneous arrays.

Support for inhomogeneous nested arrays can be added to the encoding above by allowing arbitrary expressions as array elements. This would complicate the discussion in the following and distract from the underlying principles of my approach. I will therefore use homogeneous arrays only. However, my approach does support nested structures, as I will show in Section 2.4.

Even though I use homogeneous, non-nested vectors to encode the ravel of an array, this does not fully rule out nested arrays. Blelloch has shown in his pioneering work on nested data-parallel programming languages in general and Nesl in particular [Blelloch, 1994; Blelloch et al., 1994; Sipelstein and Blelloch, 1991] that nested arrays can be flattened into flat arrays and a nesting descriptor. Thus, using Blelloch's technique, much of the expressiveness of the APL nesting approach can be regained in the homogeneous setting I use in this thesis. To add support of Blelloch style nested arrays to my encoding as discussed above, it suffices to use a nested vector to encode the shape and a vector to describe the rank.

As an example, consider a representation for complex numbers. A single complex number can be seen as an array, *i.e.*, a two-element vector, containing the real and imaginary components. Consequently, a vector of complex numbers then is an array of arrays, *i.e.*, a vector of two-element vectors. A four-element vector of complex numbers then has the form

$$\left(\begin{pmatrix}1\\5\end{pmatrix},\begin{pmatrix}2\\6\end{pmatrix},\begin{pmatrix}3\\7\end{pmatrix},\begin{pmatrix}4\\8\end{pmatrix}\right)$$

where the upper component of each element of the vector denotes the real part of the complex number and the lower component denotes the imaginary part. The array above, even though being nested, is perfectly homogeneous. In particular, each element of the outer vector has the same structure. In this setting, it suffices to describe the shape of the outer vector and the shape of the elements. Similarly, it suffices to encode the dimensionality once for all elements. A possible encoding thus would be using the ravel `[1,5,2,6,3,7,4,8]` and a shape vector `[[4],[2]]`. The first component of the shape vector encodes the shape of the outer structure, *i.e.*, the vector, whereas the second component gives the structure of the nested elements. Similarly, the rank of the above array can be represented as `[1,1]`.

This encoding suffices for homogeneously nested arrays. However, in the more general case of inhomogeneous nestings, the shape vector becomes slightly more complex. As an example, consider the following vector of integer vectors with differing lengths:

$$\begin{pmatrix} (1 & 2 & 3 & 4 & 5) \\ (6 & 7 & 8) & & \\ (9 & 0 & 1 & 2) & \end{pmatrix}$$

As before, the outer structure is a one-dimensional array whose elements are arrays again. However, now the structure of these inner arrays is no longer homogeneous. Nonetheless, the data elements can still be flattened into a row-major ravel of the form:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

To encode the nesting structure of the above array, it does not suffice to memoize the overall shape of the inner elements. Instead, the shape of each single element needs to be memoized. I use Blelloch's concept of segments here [Blelloch et al., 1994]. However, in contrast with Blelloch who only allows vectors as segments, I support n-dimensional segments.

To construct the shape vector, I begin with the outermost array; it has three elements, each being a vector. Thus, the outer shape of the above array is `[3]`. Consequently, the ravel needs to be segmented into 3 parts, one for each of the vectors contained in the outer array. The first part consists of the first five elements; its shape vector therefore is `[5]`. Similarly, the shape vector for the other two components can be derived as `[3]` and `[4]`, respectively. As the elements of each vector are scalar values, the ravel needs not to be further segmented. The overall shape vector for this nesting level is then computed by catenating the individual shape vectors, yielding the vector `[[5],[3],[4]]`. Lastly, I prepend the segmentation of the outer level, yielding a shape vector of `[[3],[5],[3],[4]]`.

The above shape vector suffices to fully describe the structure of the array. In particular, the rank at each level is given by the length of the corresponding component of the shape vector. Furthermore, this encoding shares the property with all previous encodings that two arrays have the same structure if their shape vectors are identical. Blelloch refers to the identity of the structure of two arrays as two arrays being member of the same paralation [BlellochBlelloch and Sabot, 1990; Sabot, 1989].

Both encodings for nested arrays can easily be expressed in my approach. It suffices to add nested vectors as data type and corresponding operations thereon. However, such additions would only complicate the presentation without offering additional insight. I will therefore refrain from such an extension here.

Finally, another possible encoding for n-dimensional arrays that shall not go unmentioned is the use of nested vectors. In the realm of array programming languages, early versions of SISAL [Cann, 1989] are the most prominent representatives using such an encoding. The underlying idea is to represent each dimension of an array as a vector of subarrays. Such an encoding gives utmost flexibility, as it allows for both nested and inhomogeneous arrays. However, in SISAL most of this flexibility is lost due to the limitations of its static type system. Nonetheless, early versions of SISAL suffered from the runtime overheads that an encoding using nested vectors entails. Therefore, for version 2.0 of SISAL true n-dimensional arrays were proposed [Cann et al., 1991]. However, to my knowledge, this proposal was never implemented.

It would be possible to use an encoding based on nested vectors in my thesis, as well. Similar to the encoding of nested arrays used by APL, all that is required is to allow for nested vectors. However, as the SISAL-style encoding is not widely used, I have opted for the encoding using a ravel and shape vector, instead.

Using that encoding for matrices, I can now start to describe adding two matrices. As a first starting point, consider the following pseudo code:

```
A_val   = [ 1, 2, 3, 4]
A_shape = [ 2, 2]
A_rank  = 2
```

```
  B_val   = [ 1, 0, 0, 1]
5 B_shape = [ 2, 2]
  B_rank  = 2
7
  S_val   = (vect_add A_val B_val)
9 S_shape = A_shape
  S_rank  = A_rank
```

The first six lines define two matrices A and B. In both cases, first the actual value in form of the ravel, denoted by the suffix _val, is defined, followed by definitions of the shape and rank, identified by the suffixes _shape and _rank, respectively. These six lines define A and B as values

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{and} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

In a next step, lines 8ff. define the computation of the element-wise addition of the two matrices. As this section is not concerned with the actual computation of the addition but merely with the modelling of properties of data, I have used a pseudo function vect_add in Line 8 to describe the actual element-wise plus operation on vectors. Next, in Line 9 the shape of the result is computed as the shape of matrix A. Using the shape of A and not the shape of B is an arbitrary choice. As element-wise addition of matrices is only defined for matrices of the same shape, both A_shape and B_shape should have the same value. For the above example, this property is clearly given. I postpone the discussion of how to generally enforce these kinds of constraints to Section 2.3. Lastly, the rank of the result is computed as the rank of one of the argument matrices in Line 10. For symmetry alone, I have chosen to use the rank of A. The same arguments as for the shape apply.

The above encoding suffices to make the computation of shape and rank explicit in the code. At first glance, this seems to come at the price of increased computational complexity, as I have added two auxiliary computations to the actual task of adding the two matrices. However, these computations need to be performed in classical languages like MATLAB, APL and SAC, as well. The only difference is that these are usually hidden inside the semantics and implementations of these languages. It is therefore worth noting that, even with the two auxiliary computations shown above, the overall computational complexity has not changed compared to traditional approaches.

Computing the shape and rank of the result by arbitrarily choosing either the shape and rank of matrix A or those of matrix B yields the correct result. However, from a design perspective this solution is not satisfactory. Even though the programmer might be aware of this property at the time he implements the above matrix addition, this property is not documented in the code itself. When looking at the code again later on, this property has to be inferred from the context. To make the code more self-documenting, it is desirable to directly express this property in the program text. I will do so by introducing a special any operator, similar to the amb operator proposed by McCarthy [McCarthy, 1961, 1962]. Informally, the any operator is evaluated by non-deterministically choosing any of its arguments. Using this additional operator, the above code can be rewritten as follows:

```
A_val   = [ 1, 2, 3, 4]
```

```
2  A_shape = [ 2, 2]
   A_rank  = 2
4  B_val   = [ 1, 0, 0, 1]
   B_shape = [ 2, 2]
6  B_rank  = 2

8  S_val   = (vect_add A_val B_val)
   S_shape = any( A_shape B_shape)
10 S_rank  = any( A_rank B_rank)
```

In the amended example above, the shape of the result is now defined in Line 9 as either the shape of matrix A or the shape of matrix B. Analogously, the rank of the result is defined. This encoding elegantly documents the fact that either value would be correct directly in the program text. Furthermore, it allows me to exploit this knowledge for inferring properties statically, as described later in this thesis. In the above example, it now suffices if the shape or rank property is known for one of the two argument matrices to deduce the shape or rank of the result.

Even though the computational complexity has not increased compared to traditional approaches, the code still got more complex and more difficult to read. This is, apart from the inevitable growth of the specification, mainly due to the loose coupling of the three streams of computation. Even though the ravel, shape and rank computation for each matrix are conceptually highly coupled, they appear as separate computations in the code. They are only grouped by a rather informal naming scheme using suffixes. This is unsatisfactory for two reasons. Firstly, it makes it more difficult for the programmer to identify related computations in the code and extract the actual stream of computation of the algorithm. Secondly, this shortcoming applies to formal reasoning and the implementation of partial evaluation, as well. An interpreter or compiler would have to identify related computations by analysing the data flow. It therefore is desirable to group the main computation and its auxiliary computations more formally, both in syntax and semantics. This gives rise to the idea of using records, *i.e.*, tuples of label value pairs, to represent data and their properties, as shown below.

```
A = { val=[ 1, 2, 3, 4], shape=[ 2, 2], rank=2}
B = { val=[ 1, 0, 0, 1], shape=[ 2, 2], rank=2}

S = { val=(vect_add A.val B.val),
      shape=any( A.shape B.shape),
      rank=any( A.rank B.rank)}
```

In the above example, and throughout this thesis, I use { and } to represent records syntactically. To extract components from a record, I use the selection operation . . Given a label as right-hand operand and a record expression as left-hand operand, the infix . operation extracts the component of the record that corresponds to the given label. Using records has two advantages. Firstly, records increase the brevity of specification and provide a tighter coupling of values and properties. Secondly, by using labels to index different streams of computation, I retain the self-documenting property of my very first approach. However, in its current form, the new syntax does not differentiate between the main stream of computation of the actual data and the additional auxiliary computations of properties. From a formal perspective, this uniform treatment is even desirable and a key idea of my approach. Namely, to use the same techniques to compute the values and properties of data. To enhance readability, I nonetheless propose to differentiate the main computation from auxiliary computations. Instead of encoding the value of a record using a `val` component, I implicitly specify this component by prefixing each record with an expression.

```
  A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
2 B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}

4 S = (vect_add !A !B){ shape=any( A.shape B.shape),
                        rank =any( A.rank B.rank)}
```

As can be seen above, this syntactical change highlights the actual computation of the value. To select this now implicit component of the record, I introduce a special selection operation `!`. This can be seen in action in Line 4 of the above example. The value of the addition of the two matrices is computed by applying the function `vect_add` to the values of the two argument matrices, syntactically referred to by `!A` and `!B`.

2.2. Programming with Auxiliary Computations

Although the encoding and language informally introduced so far suffice to express my example of adding two matrices, it would be cumbersome to require the programmer to explicitly state all auxiliary computations whenever he wants to add two matrices. To alleviate this, I will first introduce one of the most powerful abstractions found in almost all programming languages: Functions. For the following examples and throughout this thesis, I will adapt a syntax that closely resembles that of Standard ML. A full discussion of the syntax of Standard ML would be beyond the scope of this chapter. I will therefore informally introduce parts of it alongside their introduction to my examples. A formal definition of the completed syntax will be given in Chapter 3. For a full discussion of the syntax of Standard ML, I refer the reader to [Milner et al., 1997].

To start off, consider the following version of the example from the previous section, using a function definition for the addition of two matrices:

```
let
  fun add A B = (vect_add !A !B){ shape=any(A.shape B.shape),
                                  rank =any(A.rank B.rank)}
  val A       = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
  val B       = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
in
  (add A B)
end
```

To bind functions or values to identifiers, I use the `let ... in ... end` construct known from Standard ML. The `let` construct binds all expressions on the right-hand sides in the `let`-block to the corresponding identifiers on the left-hand side. To differentiate between values and functions, each binding is prepended by either the keyword `val` or the keyword `fun`, respectively. These bindings are only visible within the `let` expression, *i.e.*, within the definitions between the `let` and `in` keywords and the expression between the `in` and `end` keywords. I will refer to the latter expression in the following as the *body* of the `let` construct.

Note that the order of the definitions of functions using the `fun` keyword does not impose an order on the scope of these functions within the `let` construct. All function definitions are visible throughout the entire `let` construct. I have chosen these semantics to allow for mutually recursive function definitions. In this respect, my approach differs from Standard ML where function bindings are by default only visible after their definition. For mutually recursive function definitions, Standard ML provides a special `and` keyword. In contrast with functions, values bound using the `val` construct are only visible in the definitions further down in the program text and in the body of the `let` construct. However, in contrast to Standard ML, they are not visible in the bodies of function definitions, *i.e.*, all functions must be closed.

Lines 2 and 3 three give an example of a function definition. Generally, the keyword `fun` is directly followed by the function identifier and potentially multiple further identifiers, which represent the arguments of the function. In the above example, the function `add` expects two arguments `A` and `B`. These identifiers can be used in the right-hand side of the function definition to access the actual arguments the function is applied to. In the above example, this is used to compute the element-wise addition of two matrices, as introduced in the previous section.

Lines 4 and 5 give an example for the definition of values. Here, the keyword `val` is directly followed by only a single identifier. In the above example, the two matrices shown earlier are bound to the identifiers `A` and `B`. As with functions, this binding is only visible within the body of the `let` construct.

Using these definitions, the actual addition of the two matrices can now be written as the expression `add A B` as shown in Line 7 above. Thus, having defined the addition as a function once, the specification of the actual addition of the two matrices becomes as tense and readable as in languages without auxiliary computations. However, the programmer can still look up which properties the result carries and how they are defined by looking at the function definition in the program text.

Looking up the properties that are required from the arguments of a function is more difficult, as these are not encoded in the *signature* of the function, *i.e.*, the left-hand side of its definition. Instead, the programmer has to scan the entire right-hand side of a function definition and look at all accesses to the arguments to find this information. For the above example this might look straight forward. However, for more complex function definitions this can be a tedious task. Similarly, when initially writing the function definition or amending it later on, the programmer has no syntactic representation to look up the properties of the function's arguments.

To enhance readability in this respect, I extend the syntax for function definitions, as shown below.

```
let
```

```
2    fun add A{ shape, rank} B{ shape, rank}
       = (vect_add !A !B){ shape=any(A.shape B.shape),
4                          rank =any(A.rank B.rank)}
     val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
6    val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
   in
8    (add A B)
   end
```

Line 2 gives an example of this extended syntax for function definitions. As can be seen, each argument now additionally carries a list of all labels used to select properties from arguments in the right-hand side of the function definition. This directly exposes the required structure of the arguments to the programmer. Furthermore, when modifying the function definition, this can be used as a dictionary to the valid labels for each argument.

Having introduced a syntactical handle for the properties used in the right-hand side of function definitions allows me to add some more syntactical sugar to the language. By definition, all labels listed on the left-hand side of a function definition are used for selections on the right-hand side. To spare the need for these explicit selections, I add a notation to directly bind the value corresponding to a given label to an identifier. Syntactically, this is achieved by adding an = sign followed by an identifier directly after the label of an argument. An example is given below:

```
1  let
     fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
3      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
     val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
5    val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
   in
7    (add A B)
   end
```

In the above version, I use the newly added sugar to bind the shape and rank of argument `A` to `sA` and `rA`, respectively. Analogously, `sB` and `rB` are defined for the shape and rank of the second argument. This, as expected, completely removes the need for explicit selections from the right-hand side of the function definition, as can be seen in Line 3 above.

Having defined the overall syntax for auxiliary computations and function definitions using these, I now move on to extend the example by further properties of data. In the introduction, I gave the example of modelling the special property of unit matrices that all values except those along the diagonal of the matrix are zeros. As mentioned earlier, this additional knowledge can be used to reduce the computational complexity of the addition of two matrices, as only the elements along the diagonal need to be added. For instance, linear algebra packages like BLAS (Basic Linear Algebra Subprograms) [Blackford et al., 2002; Dongarra et al., 1988, 1990; Lawson et al., 1979] define multiple versions of their algebraic subroutines for different kinds of input data, including banded matrices. To add similar support to the above example, I first add the additional property that a value is a unit matrix to the data. As can be seen above, matrix `B` is a unit matrix. This is captured below by the additional `unit` label in the definition of `B`. A property `unit=`n encodes the additional information that the given matrix is the unit matrix of the matrix ring over $\mathbb{R}^{n \times n}$.

```
  let
2   fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
4   val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
    val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
6 in
    (add A B)
8 end
```

Line 5 shows the extended definition of `B`. Note that, even though `B` now carries an additional property, the function `add` can still be applied in Line 7 . To support an agile development style, *i.e.*, refinement of properties over time, this flexibilty is important. It allows the programmer to add additional properties and thus encode further knowledge about data without the need to adapt the entire code. However, to actually make use of the `unit` property, the definition of `add` needs to be adapted.

```
   let
2    fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
4    fun add_unit A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
       = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
6                                     rank=any(rA rB)}
     val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
8    val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
   in
10   (add_unit A B)
   end
```

Lines 4ff. define a further function `add_unit` that expects as second argument a matrix that carries the additional `unit` attribute. To compute the actual result ravel, I use a special function `diag_add`, which expects as first argument the shape of the arguments, followed by the two ravels of the matrices to add. By convention, the second argument needs to be a diagonal matrix. The shape is needed to identify the elements within the ravel that correspond to the diagonal of the matrix.

Using this additional function, adding the two matrices can then be expressed by the expression `add_unit A B` in the body of the `let` construct. Although this allows me to exploit the `unit` property to reduce the computational complexity, the above solution is not optimal. In particular, the additional property is not exploited automatically in the code. Instead, all occurrences of function applications of the function `add` to a unit matrix need to be replaced by an application of the new function `add_unit`. In this respect, the above encoding is identical to the C and FORTRAN implementations of BLAS. Yet, contrary to BLAS, my encoding at least ensures that the argument actually carries the `unit` property.

Nonetheless, for larger applications, choosing the appropriate function by hand would involve major refactoring, thereby substantially increasing the implementation cost to exploit additional properties like the `unit` property above. However, given the explicit encoding of the `unit` property, this manual refactoring is not necessary. To automate the selection of the appropriate version of a function definition, I introduce a further concept widely found in modern functional languages: Pattern matching. The code below gives an example:

```
1  let
     fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
3      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
            A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
5      = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
                                      rank=any(rA rB)}
7    val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
     val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
9  in
     (add A B)
11 end
```

Syntactical, multiple instances of a single function are expressed by providing multiple argument lists and right-hand sides for that function. In the above example, the function `add` has two definitions, given in lines 2ff. and 4ff., respectively. To pick the appropriate instance for a function application, I reuse the, so far only documentational, list of properties required for any argument as a pattern. In the above example, the first instance in lines 2ff. lists the properties `shape` and `rank` as required for the second argument. In contrast, the second instance in lines 4ff. additionally requires a further label `unit` to be present. Thus, whenever `add` is applied to a second argument that features this label, the second instance is used. Otherwise, the first instance is chosen. As can be seen in Line 10, I now use the function `add` again to compute the element-wise addition of the two matrices. However, as the second argument has a label `unit`, the second instance will be chosen.

By using pattern matching and making properties explicit, exploiting the additional knowledge now no longer requires refactoring of the body of the `let` construct. Instead, only the function definition for `add` needs to be modified. For larger applications, this reduces the implementation cost of enriching the encoded properties even more.

I postpone the formal definition of the pattern matching process to Chapter 3. However, it is worth noting here that the above pattern matching deviates from traditional pattern matching approaches by using a best match strategy. Traditionally, patterns are matched in the order of their definition and the first instance with a matching pattern is chosen. The main argument against best match usually is the difficulty in defining which pattern matches best. As I will show in Chapter 3, in my setting the best match can be given a well-defined meaning. In the above example, the second instance matches better, as the pattern matches more labels of the argument.

So far, I have only encoded a special case for adding two matrices where the second argument is a unit matrix. Below, I provide a further instance for the case that the first argument is a unit matrix.

```
 1  let
      fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
 3      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
            A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
 5      = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
                                       rank=any(rA rB)}
 7          A{ shape=sA, rank=rA, unit} B{ shape=sB, rank=rB}
        = (add B A)
 9    val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
      val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
11  in
      (add A B)
13  end
```

The additional instance in lines 7ff. uses a pattern for the first argument that matches the additional `unit` label of unit matrices. To define the addition in this case, I exploit the associativity of addition in the ring of matrices and simply define it as a recursive call of the function `add` with its arguments swapped. This suffices to exploit the additional knowledge in case the first argument is a unit matrix. For the fourth case, *i.e.*, adding two unit matrices, an instance is yet missing. Nonetheless, the element-wise addition can be computed using one of the existing instance. However, without the fourth instance, the best match is not obvious in the case where both argument are unit matrices. Both instances for unit matrices match equally well. The instance in lines 5ff. matches five labels, *i.e.*, the `shape` and `rank` labels of both arguments and the `unit` label of the second argument. The same number of labels is matched by the instance in lines 7ff., with the `unit` label being matched for the first argument instead. To disambiguate pattern matching in this situation, I use an independent matching on the arguments from left to right. Thus, in the above example, the third instance matches best if both arguments are unit matrices.

To further demonstrate the expressive power of my approach, I introduce another property of matrices below. The additional label `ldiag` encodes that a matrix is a lower diagonal matrix, *i.e.*, that all elements above-right of the diagonal of the matrix equate zero. As with unit matrices, this property can be exploited to reduce the computational complexity by only adding the lower-left elements of the matrices.

```
let
  fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
    = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
        A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
    = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
                                   rank=any(rA rB)}
  val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
  val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
  val C = [ 9, 0, 8, 7]{ shape=[ 2, 2], rank=2, ldiag=true}
in
  (add A (add B C))
end
```

I have removed all but one instance for the addition of unit matrices in order to keep the example short. In their place, I have added the definition of a further matrix `C` with value

$$\begin{pmatrix} 9 & 0 \\ 8 & 7 \end{pmatrix}.$$

As can be seen, the above matrix indeed is a lower diagonal matrix. In the program text, this is encoded using the additional label `ldiag`. Furthermore, the body of the `let` expression now computes the element-wise addition of all three matrices. Note that, even though matrix `C` carries a further attribute, the definition of the function `add` needs not to be changed.

For lower diagonality, no actual value is associated with the property. Instead, the existence of the label alone suffices to encode that the corresponding matrix has the given property. This motivates the introduction of tags, *i.e.*, labels without an associated value. Syntactically, these are expressed by omitting the `=` sign and the following expression. Again, I defer a formal discussion to Chapter 3.

Even though the label `ldiag` does not hold a value, it can still be used to define special instances for matrices with this property using pattern matching. The code below makes use of this.

```
   let
2    fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
4          A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
       = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
6                                      rank=any(rA rB)}
           A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, ldiag}
8      = (ldiag_add( any(sA sB) !A !B){ shape=any(sA sB),
                                         rank=any(rA rB)}
10   val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
     val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2}
12   val C = [ 9, 0, 8, 7]{ shape=[ 2, 2], rank=2, ldiag}
   in
14   (add A (add B C))
   end
```

I have added a further instance of the function `add` for lower diagonal matrices in lines 8 following. As can be seen, it uses an additional label `ldiag` in the pattern for the second argument to match matrices with the lower diagonal property. Again, the actual computation of the result ravel is not subject of this section. I have therefore, analogously to the `diag_add` function used for unit matrices, used a special function `ldiag_add` to compute the result ravel. Furthermore, I have used a tag, *i.e.*, a value-less label, in the definition of the matrix `C` in Line 12. By encoding the additional property and adding the additional instance, the computation in the body of the `let` construct now exploits the lower diagonal property of the matrix `C` to use a less computationally complex instance to compute the element-wise addition.

Closer inspection of the matrix `B` reveals that it is a lower diagonal matrix, as well. By definition, in fact all unit matrices are. Even more, the result of adding two lower diagonal matrices yields again a lower diagonal matrix. Thus, in the above example, adding matrices `B` and `C` should yield a matrix carrying the `ldiag` label. To achieve this, both properties, *i.e.*, that `B` is a lower diagonal matrix and that addition on matrices is complete with respect to the lower diagonal structural property, need to be made explicit. The updated code is shown below.

```
1  let
     fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
3      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
             A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
5      = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
                                       rank=any(rA rB)}
7            A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, ldiag}
       = (ldiag_add any(sA sB) !A !B){ shape=any(sA sB),
9                                       rank=any(rA rB)}
             A{ shape=sA, rank=rA, ldiag} B{ shape=sB, rank=rB, ldiag}
11     = (ldiag_add any(sA sB) !A !B){ shape=any(sA sB),
                                        rank=any(rA rB),
13                                      ldiag}
     val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
15   val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2, ldiag}
     val C = [ 9, 0, 8, 7]{ shape=[ 2, 2], rank=2, ldiag}
17 in
     (add A (add B C))
19 end
```

I have amended the definition of the matrix `B` in Line 15 by the additional `ldiag` label. Furthermore, lines 10ff. define the required additional instance of the function `add`. It differs from the previously defined instances in that it requires both arguments to carry the `ldiag` label. Furthermore, the fact that the result is a lower diagonal matrix again is expressed by using the `ldiag` property in the definition of the result of the instance, as well. As both matrices `B` and `C` carry the `ldiag` property, the expression `add B C` in Line 18 uses the newly defined instance to compute the result. Furthermore, the consecutive addition of matrix `A` to the result is computed using the instance defined in Line 7, thereby making use of the propagated `ldiag` property.

Before I close this section, I discuss one last example to motivate a final extension to the pattern matching mechanism introduced so far. For this, consider the expression `add A B` in the context of the body of the above `let` construct. As the matrix `B` carries both, the `ldiag` and `unit` label, the best match is again not well defined. Both, the instance defined in lines 4ff. and the instance defined in lines 7ff. match. The former, as the second argument to the application of `add` carries the `unit` label and the latter as the argument carries the `ldiag` property, as well. A possible solution to disambiguate the dispatch would be to define a further instance for matrices that carry both properties. However, this would unnecessarily bloat the number of instances that are required to specify addition in the above scenario. Instead, I introduce a means to express precedence on labels for pattern matching. The details are shown in the example below.

```
1  let
```

```
     rel ldiag <: unit
3    fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
5            A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, unit}
       = (diag_add any(sA sB) !A !B){ shape=any(sA sB),
7                                     rank=any(rA rB)}
             A{ shape=sA, rank=rA} B{ shape=sB, rank=rB, ldiag}
9      = (ldiag_add any(sA sB) !A !B){ shape=any(sA sB),
                                       rank=any(rA rB)}
11   val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
     val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, unit=2, ldiag}
13 in
     (add A B)
15 end
```

For brevity, I have removed the definitions of all instances and values not required for this particular example. To declare a precedence relation between two labels, I have introduced the new keyword `rel` as shown in Line 2. In general, the precedence of a label over another label is declared using a precedence expression of the form *label* `<:` *label*, where the second label is declared to take precedence over the first. In the above example, the label `unit` takes precedence over the label `ldiag`. Thus, the instance defined in lines 5ff. now is chosen whenever a matrix carries both labels. Consequently, the expression `add A B` in the body of the above `let` expression is evaluated using the instance requiring the `unit` label for the second argument. Generally, the scoping of precedence declarations is the same as for function definitions. They are visible throughout the entire `let` construct and not only below their definition in the program text.

This completes the discussion of functions and pattern matching with auxiliary computations. However, so far I have in all examples taken certain constraints for granted, *e.g.*, that the two matrices that are added have the same shape. In the next section, I will discuss how constraints of this kind can be expressed in a dynamically typed setting with auxiliary computations.

2.3. Supporting Constraints

The definition of the function `add` in the examples above silently assumes that both arguments have the same shape and rank. However, in general, this needs not be the case. So far, this will yield a runtime error only if the ravels of the two matrices have different length. In this case, the application of the add functions on the ravel vectors will fail. For matrices with different shapes but identical ravel length, *e.g.*, two matrix with shapes `[ 2, 3]` and `[ 3, 2]`, the function `add` can still be applied without yielding an error. Nonetheless, the result would not represent an element-wise sum of these two matrices. Thus, the current encoding does not suffice to catch violations of the shape equality constraint. Furthermore, from a programmers perspective, even if an error is signalled at runtime, it is not obvious where it stems from. Instead, the programmer has to deduce why the addition of the two ravels went wrong.

This deficiency is not unique to my approach. Indeed, all dynamically typed languages show this behaviour. Even for statically typed languages, if the type system cannot encode a certain constraint, such constraint might be caught neither at compile time nor at runtime. To ensure that constraint violations are caught, I propose to encode them explicitly in the code by means of *contracts*. A contract, in its general form, is a Boolean expression that encodes a certain constraint. For the example of matrix addition, this would be the equality of the `rank` and `shape` properties of both argument matrices. If the contract evaluates to false, the computation of the associated expression fails with an error. This ensures that all constraint violations, as long as they are encoded by a contract, are caught at least at runtime. Furthermore, as they are encoded in the language itself, contracts in my approach might even be resolved statically using partial evaluation.

The idea to model constraints by means of contracts is by no means new. It has been first proposed by Meyer [1990, 1992] in the context of EIFFEL and has consecutively been applied to further languages. In the object oriented domain, contracts have been adopted by modern languages like JAVA [Karaorman et al., 1999; Kramer, 1998] and PYTHON [Ploesch, 1997]. For the statically typed functional language HASKELL with its powerful type system, Xu [2006] has proposed ESC/HASKELL and a general framework for static contract checking in HASKELL [Xu et al., 2009; Xu, 2008]. ESC/HASKELL extends HASKELL with contracts to model constraints that cannot be expressed by the type system.

I introduce contracts to my language by means of *pattern guards*. A pattern guard is a Boolean expression annotated at a pattern. If the pattern matches, first the annotated guards are checked before the corresponding expression is evaluated. However, pattern guards in my approach do not influence the pattern matching process as such. If a guard fails, the matching process is not continued. Instead, the overall program evaluation fails with an error.

The use of pattern guards to express contracts in the program text is motivated by the observation that constraints usually arise in the context of function applications. Thus, annotating the contract at the function definition seems natural. Furthermore, this enhances readability and program documentation. The function signature explicitly states which constraints need to hold for an application of a function to be valid. Below is an example for the use of contracts in the definition of the addition of two matrices. For brevity, I have again removed all function instances not required for this particular example.

```
let
  fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
    | (sA = sB) (rA = rB)
    = (vect_add A! B!){ shape=any(sA sB), rank=any(rA rB)}
  val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
  val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
in
  (add A B)
end
```

As can be seen above, a contract is expressed syntactically by a | symbol after the pattern definition of an instance, followed one or more Boolean expressions. In the example above, the first expression in Line 3 ensures that both argument matrices have the same shape. Note that the scope of bindings of values of labels to identifiers introduced in the pattern extends to the contract, as well. As a further contract, the second expression in Line 3 requires that both matrices have furthermore the same rank. For matrices, this of course is always the case. However, in the context of general arrays, it might not be.

Equality constraints like the two shown above are relatively common in function definitions. To enhance readability and ease program specification, I introduce some further syntactic sugar below.

```
1  let
     fun add A{ shape=s, rank=r} B{ shape=s, rank=r}
3      = (vect_add A! B!){ shape=s, rank=r}
     val A = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
5    val B = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
   in
7    (add A B)
   end
```

The above code is semantically equivalent to the previous version. To express the equality constraint on the `shape` and `rank` labels of the two argument matrices, I have bound the corresponding labels of each argument to the same identifier, *i.e.*, the value of the `shape` label of matrices `A` and `B` is bound to the identifier `s` within the pattern in Line 2. Analogously, I have defined the identifier `r`. These identifiers are then used in the definition of the instance instead of the previous application of `any` to the formerly separate identifiers for the two arguments. The above can be de-sugared to the original encoding by adding two equality constraints and replacing the occurrences of the identifiers within the instance definition with the original `any` expressions. A formal description of this source-to-source transformation is given in the next chapter.

Using the contracts and pattern guards as introduced above, the code now explicitly captures the constraints of the function `add`. Thus, if applied to ill-formed arguments, the application of `add` now instantly fails. Furthermore, the programmer can directly deduce the constraints a function imposes on its arguments from the function's signature.

This completes the discussion of constraints in this section. Before giving a formal definition of the syntax and semantics of contracts and pattern guards in the next chapter, I first discuss a further example to show the applicability of my approach beyond arrays in the next section.

2.4. Example: Encoding a Typed Expression Language

In this section, I provide a further, more complete instance of the use of auxiliary computations. By example of an evaluator for a small expression language, I show how properties of data can gradually be introduced to gain varying degrees of static guarantees.

As a prerequisite, I first introduce a further data structure beyond arrays that is commonly found in functional programming languages: Algebraic data types. An *algebraic data type* is a structured data type defined as a sum of tagged products. As an example, I will use the definition of a small language on integers by means of an algebraic data type `Expr`. Below is an example of the definition of the algebraic data type `Expr` in the functional programming language HASKELL. A discussion of HASKELL in general is not required for the examples below and would be beyond the scope of this section. I refer the interested reader to the book by Peyton-Jones [2003] for details.

```
  data Expr = ENum Int
2           | EBool Boolean
            | ECond Expr Expr Expr
4           | EIsZero Expr
            | EDiv Expr Expr
```

The algebraic data type `Expr` defined above consists of four choices. The first, defined in Line 1, is made up of the tag `ENum` and an integer value. In the small example language encoded here, it represents integer numerals. Next, Line 2 defines Boolean numerals using the tag `EBool`. Line 3 gives an example of a nested choice. It encodes conditionals as the tag `ECond` followed by three further expressions, the predicate, then expression end else expression. To allow for slightly more complex expressions, my small sample language furthermore features a predicate function `IsZero`. This function is encoded in the algebraic data type by the tag `EIsZero` followed by the expression to check, as defined in Line 4 above. Finally, Line 5 defines an encoding for the division operation on integers by means of the tag `EDiv`, followed by the two argument expressions to the division.

Apart from the actual data type, the above definition implicitly defines a set of matching data constructors, as well. Each data constructor carries the same name as the tag of the corresponding choice and expects as arguments the values to be stored in the encoded fields. As an example, consider the encoding of the expression IsZero(5/2) shown below.

```
1 EIsZero (EDiv (ENum 5) (ENum 2))
```

Read inside out, the above expression first constructs two values of type `Expr` using the tag `ENum` for integers. Next, the resulting values are wrapped by the `EDiv` data constructor in the corresponding choice of the `Expr` data type for integer division. Lastly, this is then wrapped by the `EIsZero` in the choice corresponding to the encoding of the `IsZero` predicate function. Overall, the above thus defines a three times nested value of the `Expr` algebraic data type.

Before I can encode the above data structure in my approach, first the additional properties that are encoded apart from the value need to be identified. For example, the result of the data constructor `ENum` above carries, apart from the integer numeral, two further properties. Firstly, the, in HASKELL static, property that the result is of type `Expr`. Secondly, the result is tagged, in this case dynamically, by the tag `ENum` to encode that the result is the first choice of the `Expr` data type. Similarly, all other data constructors encode these two properties. Thus, to encode the above data in my approach, the corresponding data constructors need to encode these two properties, as well. However, as my language is untyped, there is no need for a data type declaration. Instead, I directly encode the data constructors, as shown below.

```
1 let
    fun ENum I{} = (!I){ Expr, ENum}
3   fun EBool B{} = (!B){ Expr, EBool}
    fun ECond P{ Expr} T{ Expr} E{ Expr} = (P, T, E){ Expr, ECond}
5   fun EIsZero E{ Expr} = (E){ Expr, EIsZero}
    fun EDiv A{ Expr} B{ Expr} = (A, B){ Expr, EDiv}
7 in
    EIsZero (EDiv (ENum 5) (ENum 2))
9 end
```

In the example above, I use a different language to encode the actual computations. In contrast with my previous examples, which mainly make use of vectors as data type within the actual computations embedded in the records, I in these examples use tuples, denoted by the (and) symbols above. This is a general observation. Depending on the targeted clientèle, the embedded language differs. However, the general principles presented so far still apply. Even more, the actual embedded language is orthogonal to my approach. I will give a more formal discussion of this statement in the next Chapter.

Line 2 above defines the data constructor for the `ENum` choice of the algebraic data type. It expects as an argument an integer value `I`. Note that, for brevity of the example, I have not encoded the fact that `I` indeed represents an integer. However, such an encoding could be enforced by, *e.g.*, an `Int` tag in the pattern and by furthermore adding a constructor for integer values. The data constructor yields as its result a tuple containing the integer value of the argument, denoted by the `!I` expression above, with the two additional tags `Expr` and `ENum` to encode the two properties that the result is an expression and that it represents the `ENum` choice of the encoded algebraic data type. Similarly, all other data constructors are encoded. For data constructors that yield a nesting of expressions, *e.g.*, the `ECond` data constructor, the tuple contains the records encoding the corresponding expressions.

As Line 8 shows, by using these data constructors, the sample expression IsZero(5/2) can be encoded with the exact same program text as in the HASKELL example. Thus, once the data constructors with their auxiliary computations have been defined, the remaining program text requires no further annotations.

Having encoded expressions by means of an algebraic data type, I can now define an evaluator on these expressions. I first give an example in HASKELL below.

```
eval (ENum v)       = ENum v
eval (EBool v)      = EBool v
eval (ECond p t e)  = case (eval p) of
                      | EBool v  -> if v (eval t) (eval e)
eval (EIsZero e)    = case (eval e) of
                      | (ENum v) -> EBool (v = 0)
eval (EDiv a b)     = case ( (eval a), (eval b)) of
                      | ( (ENum va), (ENum vb)) -> ENum (va/vb)

eval (EIsZero (EDiv (ENum 5) (ENum 2)))
```

The above code defines a function `eval` by pattern matching on the argument expression. Note that, in contrast with my approach, pattern matching in HASKELL uses a first match, top to bottom, left to right matching algorithm. Line 1 defines an instance of the function `eval` on the `ENum` choice of the algebraic data type. As `ENum` already represents a value, *i.e.*, a fully evaluated expression, it yields its argument as its result. Similarly, the function `eval` is defined for the `EBool` case. In lines 3ff., the definition for conditional expressions is given. To evaluate a conditional, I first evaluate the predicate expression. If it evaluates to an expression of kind `EBool`, as checked by the nested pattern matching using the `case` expression in Line 3, I then compute the result by evaluating either the then or else expression, depending on the Boolean value of the evaluated predicate expression. Similarly, the evaluation of `EIsZero` expressions is defined in lines 5 following. Here, if the argument expression evaluates to an integer number, the result is defined as an `EBool` expression whose value is computed by comparing the integer numeral to `0`. Finally, lines 7ff. define the evaluation of `EDiv` expressions. To evaluate an `EDiv` expression, both argument expressions need to evaluate to an integer numeral encoded as an `ENum` expression. If this is the case, the result is computed as an `ENum` expression whose value equates to the integer division of the values of the evaluated argument expressions.

Using the above definition of an evaluator function `eval`, I can now evaluate the sample expression provided earlier. The corresponding function call is given in Line 10 above. To evaluate the expression, first the instance given in lines 5ff. for the `EIsZero` choice is used. This instance recursively applies the `eval` function to the `EDiv` expression contained in the argument. In the above example, this expression is then evaluated using the instance given in lines 7ff. above, which again recursively evaluates the nested expressions. As these both are `ENum` expressions, both are evaluated using the instance in Line 1 and yield expressions of kind `ENum`. These match the nested pattern matching in Line 7 and thus the application of the instance in lines 7ff. yields an `ENum` expression with value 2. Finally, as the pattern in Line 5 matches, the overall result is the expression `EBool false`.

A similar definition for an evaluator can be given using my approach. The program text is given below.

```
let
  fun eval E{ Expr, ENum} = E
           E{ Expr, EBool} = E
           E{ Expr, ECond} = (cond (eval !E#0) !E#1 !E#2)
           E{ Expr, EIsZero} = (isZero (eval E#0))
           E{ Expr, EDiv} = (div (eval !E#0) (eval !E#1))
  fun cond P{ Expr, EBool} T{ Expr} E{ Expr}
    = if (!P) (eval T) (eval E)
  fun isZero E{ Expr, ENum}
    = (EBool (!E = 0))
  fun div A{ Expr, ENum} B{ Expr, ENum}
    = (ENum (!A / !B))
in
  (eval (EIsZero (EDiv (ENum 5) (ENum 2))))
end
```

The above definition, apart from syntactical differences, largely corresponds to the definition in Haskell provided earlier. One main difference is the use of additional functions to facilitate the pattern matching on the results of intermediate evaluations. This stems from the lack of a case statement and support for pattern matching within function bodies. However, this is only a syntactical inconvenience and does not impose any restrictions on the expressiveness of my approach.

A further difference is the explicit extraction of the components of the argument tuples by means of a selection compared to the pattern match in the Haskell version. Syntactically, the selection is represented by the infix operator `#`. It denotes the selection of the element at the position given by the right-hand operand from the tuple given as left-hand operand. The elements of the tuple thereby are indexed from left to right starting with 0.

Again, the application of the function `eval` in Line 2 is even syntactically identical to the corresponding expression in the Haskell code. Thus, even though the above example makes heavy use of auxiliary computations, once defined they do not interfere with the remainder of the program.

In both implementations above, the implementation of `eval` is not complete in that it cannot evaluate all expressions that can be constructed using the data constructors. As an example, consider the expression `EIsZero (EBool true)`. Although the expression is a valid application of the data constructors, the encoded expression is ill-formed. The predicate function `IsZero` cannot be applied to Boolean values. Using the above approaches, this will only be caught once the expression is actually evaluated. However, to catch errors early, it would be desirable to rule out the construction of the above expression in the first place. In my approach, this can be done by adding a further property to expressions: Their kind. The idea is to encode what kind of value the expression will evaluate to. Or, to use the vocabulary of the typed world, we assign each expression a type. The code below shows an updated version of the data constructors.

```
1  let
     fun ENum I{} = (!I){ Expr, ENum, Kind=Int}
3    fun EBool B{} = (!B){ Expr, EBool, Kind=Bool}
     fun ECond P{ Expr, Kind=kp} T{ Expr, Kind=kb} E{ Expr, Kind=kb}
5      | (kp = Bool)
       = (P, T, E){ Expr, ECond, Kind=kb}
7    fun EIsZero E{ Expr, Kind=k}
       | (k = Int)
9      = (E){ Expr, EIsZero, Kind=Bool}
     fun EDiv A{ Expr, Kind=ak} B{ Expr, Kind=bk}
11     | (ak = Int) (bk = Int)
       = (A, B){ Expr, EDiv, Kind=Int}
13 in
     (EIsZero (EBool true))
15 end
```

As can be seen in Line 2 above, the definition of the `ENum` data constructor now yields a result with the additional label `Kind`, which has value `Int`. This encodes that the resulting expression is of integer kind and will evaluate to an integer expression. Similarly, I have amended the definition of the `EBool` data constructor using the value `Bool` for the `Kind` component of its result to identify the result as an expression of Boolean kind. The two special values `Int` and `Bool` can be seen as enumeration values for all possible expression kinds. I use names instead of mere numbers here to enhance readability. However, any distinguishing pair of values would suffice.

For the `ECond` data constructor, the kind of the result expression is computed from the two argument expressions. As can be seen in lines 4ff., I use an implicit equality constraint to ensure that both, the then and else expression, are of the same kind `kb`. This kind is then used as the kind of the result. Furthermore, I use an explicit pattern guard to enforce the constraint that the first argument to the data constructor, *i.e.*, the predicate expression, needs to be of Boolean kind.

Following this pattern, I have defined an instance of the `EIsZero` data constructor in Line 9 above that requires the argument expression to be of integer kind and yields an expression of Boolean kind. For the `EDiv` constructor in Line 10, I require both arguments to be of integer kind. In this case, the result is then of kind integer, as well.

Using these modified data constructors, the construction of the expression in Line 14 above fails, even before the expression is actually evaluated. The application of the data constructor `EBool` to the argument `true` yields an expression of kind `Bool`. Thus, in the consecutive application of the `EIsZero` constructor, the pattern guard evaluates to false and thus the program fails.

It is worth noting here that, even though I have changed the encoding of the algebraic data type above, the function `eval` can still be applied. As pattern matching in my approach simply ignores superfluous labels, the instances of `eval` still match regardless of the additional kindness tag `Kind`.

My approach of enforcing well-formed expressions by adding a further property to the data constructors cannot be directly translated to HASKELL. It is possible to define an algebraic data type that carries an additional tag to model the kind of an expression, as shown below.

```
data Kind = KInt | KBool
data Expr = ENum Kind Int
          | EBool Kind Boolean
          | ECond Kind Expr Expr Expr
          | EIsZero Kind Expr
          | EDiv Kind Expr Expr
```

In the above code, I have defined a further algebraic data type `Kind` to encode the kind of an expression. All choices then carry the kind as a further element. However, as the data constructors are defined implicitly, they cannot directly be used to enforce restrictions on the kind argument of the above expressions. By default, they accept any kind argument for all expressions. This can easily be overcome by specifying user-defined constructors similar to those used in my approach. Due to the nature of pattern matching in HASKELL, it is however not possible to directly extract the kind component of an arbitrary expression. Instead an extractor function like the one below is required:

```
kind (ENum k _)       = k
kind (EBool k _)      = k
kind (ECond k _ _ _) = k
kind (EIsZero k _)    = k
kind (EDiv k _ _)     = k
```

In the above code, I provide an instance of the function `kind` for each choice in the algebraic data type `Expr`. Each instance matches the `Kind` component of the corresponding choice and returns it as its result. Using the function above, the data constructor for the `ECond` expression can be encoded as follows:

```
1 eCond p t e | (((kind e) == KBool) && ((kind e) == (kind t)))
    = ECond (kind t) p t e
```

I use a guard in Line 1 above to ensure that the arguments to the constructor have the correct kind, *i.e.*, that the predicate has kind `KBool` and that the then and else expressions are of the same kind. If this is the case, the resulting `ECond` expression has the kind of the then expression.

The above encoding is fully dynamic. As the pattern match will only be evaluated during runtime, such will be the guard that checks for the correct kinds of the arguments. To push this into the type system and thus evaluate kinds at compile time, a further possible encoding in HASKELL is to use a stratified approach. By *stratification*, I refer to the technique of splitting an algebraic data type into a nesting of multiple algebraic data types. Below is an encoding of my example language using a stratified approach in HASKELL.

```
data IKind = ENum Int
           | EDiv IKind IKind
           | EICond BKind IKind IKind
data BKind = EBool Boolean
           | EBCond BKind BKind BKind
           | EIsZero IKind
data Expr = IExpr IKind
          | BExpr BKind
```

I have split up the algebraic data type `Expr` into two kinds: Integer expressions modelled by the `IKind` algebraic data type and Boolean expressions represented by the `BKind` algebraic data type. To glue these back together as a single data type, I use a further algebraic data type `Expr`.

Using the above encoding, the implicit data constructors now enforce the correct kind for their argument expressions statically. However, as the kind property is encoded jointly with the tag of each choice, I had to introduce two tags for the conditional expression. The tag `EICond` encodes conditionals of kind integer, whereas the tag `EBCond` encodes expressions of Boolean kind. Thus, when constructing expressions, the programmer has to be aware of what kind the arguments are and choose the appropriate data constructor. However, as HASKELL is statically typed, one can use the type system as a guide in picking the right constructors. Nonetheless, the above approach does not scale for richer sets of attributes as each further attribute results in potential further duplication of tags.

As I will show in Chapter 6, the same static safety as provided by the stratified algebraic data type can be achieved with the encoding using auxiliary computation as presented earlier.

Using any of the two encodings of the kind attribute in HASKELL presented above furthermore requires a major refactoring of the implementation of the evaluator. For the first approach using the additional `Kind` value, the patterns have to be adapted. This is due to the strict pattern matching in HASKELL. When using the stratified approach, the `eval` function needs to be stratified, as well, to reflect the different structure of the data type. I omit the adapted implementations here and instead refer the interested reader to Appendix A.1. Contrarily, all versions of the algebraic data type encoded using my approach can still be evaluated using the initially designed evaluator.

As a final example for an encoding in HASKELL, I below give an example using *generalised algebraic data types* [Peyton-Jones et al., 2006], a recent addition to the language also referred to as *first class phantom types* [Cheney and Hinze, 2003]. Generalised algebraic data types, or GADTs for short, allow the programmer to enrich algebraic data types with additional type information. Thus, as the kind of an expression introduced above is statically inferable, GADTs can be used to model and enforce this additional property of data. Below, I provide an example of the definition of the `Expr` data structure as a generalised algebraic data type.

```
  data Expr :: * -> * where
2   ENum    :: Int -> Expr Int
    EBool   :: Boolean -> Expr Boolean
4   ECond   :: Expr Boolean -> Expr a -> Expr a -> Expr a
    EIsZero :: Expr Int -> Expr Bool
6   EDiv    :: Expr Int -> Expr Int -> Expr Int
```

In contrast to the definition of the simple algebraic data type, the definition of the GADT above additionally provides a kind signature for the data type. A *kind* in this setting refers to the type of types and functions on types. In the setting of GADTs, the data type `Expr` is a function from types to types, syntactically represented by `* -> *` in Line 1 above. It takes a type as argument and yields a new type, the `Expr` data type, as its result. Lines 2ff. above then define the data constructors. To accommodate the additional type information, the syntax for GADTs uses function signatures to define the different data constructors of a GADT. For example, the type constructor `ENum` in Line 2 takes an integer value as argument and yields a result of type `Expr Int`. Thus, it explicitly encodes the kind of the expression in its type. This information can then be used in the definition of further data constructors. For example, the data constructor for `EIsZero` expects an argument of type `Expr Int` and yields a value of type `Expr Bool`. By means of this type signature, I enforce that `EIsZero` is only applied to expressions of integer kind. Furthermore, GADTs in HASKELL facilitate the use of all-quantified types in the signature of data constructors. An example is given in Line 4. The data constructor `ECond` requires the first argument to be an expression of Boolean kind. Furthermore, the then and else expression can be of any kind `a`, as long as both have the same kind. The result is then of the same kind as the then and else expression, *i.e.*, it has the kind `a`.

Using the above definition of expressions, the type system of HASKELL is able to statically reject ill-formed expressions like the example provided earlier, *i.e.*, the expression

`EIsZero (EBool true)`. As the data constructor `EBool` yields an expression of type `Expr Bool`, the type of its result is not compatible with the signature of the data constructor `EIsZero` and thus the term is rejected.

However, GADTs can only be used if the information encoded can be statically inferred, *i.e.*, it needs to be expressible in the type system of HASKELL. To achieve this, I have silently limited the expressiveness of the example language in all previous encodings with expression kinds. As an example, consider the following expression:

```
EIsZero (ECond (EBool true) (ENum 0) (EBool false))
```

The above expression will be rejected by all encodings with expression kinds as the then and else arguments to the `ECond` data constructor have different kinds. The first expression is of integer kind whereas the second has kind Boolean. Thus, without inspecting the value of the predicate, it is not possible to assign a kind to the overall conditional. However, the evaluators specified earlier both evaluate the above expression to the value `EBool true`. Thus clearly the above expression is well-formed with respect to evaluation. In a truly dynamically typed spirit, it should thus not be rejected.

In my approach, the kind of the conditional expression might not be decidable statically either, depending on the available static knowledge. However, my approach is expressive enough to encode a soft approach to rejecting expressions. The idea is, in the way of soft typing, to reject clearly ill-formed expressions at construction time and thus ultimately statically, and to defer the decision for undecidable cases to evaluation time. To facilitate this rejection of clearly ill-formed expressions, I introduce a further expression kind `Any` to tag expressions whose kind is not known. The amended data constructors are given below.

```
1  let
     fun ENum I{} = (!I){ Expr, ENum, Kind=Int}
3    fun EBool B{} = (!B){ Expr, EBool, Kind=Bool}
     fun ECond P{ Expr, Kind=kp} T{Expr, Kind=kt} E{Expr, Kind=ke}
5      | (kp != Int)
       = (P, T, E){ Expr, ECond, Kind=if (kt = ke) any(kt ke) Any}
7    fun EIsZero E{ Expr, Kind=k}
       | (k != Bool)
9      = (E){ Expr, EIsZero, Kind=Bool}
     fun EDiv A{ Expr, Kind=ka} B{ Expr, Kind=kb}
11     | (ka != Bool) (kb != Bool)
       = (A, B){ Expr, EDiv, Kind=Int}
13 in
     (EIsZero (ECond (EBool true) (ENum 0) (EBool false)))
15 end
```

The two data constructors `ENum` and `EBool` have not changed, as their kind is always statically decidable. For the data constructor `ECond`, I have changed the pattern guards and the construction of the result to take the new kind `Any` into account. As can be seen in Line 4, I have removed the implicit constraint that the then and else expression must be of the same kind. Furthermore, the explicit constraint in form of the patter guard in Line 5 now only fails for predicate expressions of kind `Int`, thus allowing predicate expressions of kind `Any`. Next, I have modified the computation of the kind of the conditional expression itself. Only if both branches of the conditional are of the same kind, the conditional itself is assigned that kind. Otherwise, the kind `Any` is used for the result. Furthermore, I have updated the data constructors `EIsZero` and `EDiv` so that they accept expressions of kind `Any`, as well. However, for clearly ill-formed arguments, *i.e.*, arguments of kind `Bool` and `Int` in the case of the constructor `EIsZero` and the constructor `EDiv`, respectively, the guards will fail. Thus, such applications would be rejected at construction time.

The above data constructors still reject my initial example, *i.e.*, they reject the expression `EIsZero (EBool true)`, as expressions of kind `Bool` are rejected by the constructor `EIsZero`. However, the expression given in Line 14 above is accepted. As the then and else arguments to the `ECond` data constructor have different kinds, the data constructor yields a result of kind `Any`. This in turn is a valid argument to the data constructor `EIsZero`.

Note here that even introducing a third kind does not require the evaluator to be changed. The expressions constructed using the soft typing style encoding can still be evaluated using the initial implementation of the evaluator. For the example in Line 14 above, it yields `EBool true` as expected. Furthermore, it is not possible to use the above encoding with the additional `Any` kind in the context of GADTs in Haskell, as the constructor `ECond`, depending on the relationship between its argument types, uses a different type derivation.

This completes the discussion of my approach in the context of algebraic data types. I will return to the examples provided here in Chapter 5 to demonstrate the inference of static properties by partial evaluation.

2.5. Conclusions

In this chapter, I have introduced the concept of auxiliary computations to model additional properties of data beyond their value. To support auxiliary computations, I have motivated the use and introduced a syntax for records. Furthermore, to allow for efficient programming with auxiliary computations as records, I have introduced and motivated a best match pattern matching construct. It supports partial matches on the labels of the record representation of arguments. To increase its expressiveness, I have introduced support for a user-defined order on labels.

As my examples show, using a language with these features allows one to extend existing programs with additional encodings for additional properties in an agile fashion, *i.e.*, using a local and iterative approach. In my first example, I iteratively extended the definition of a function by additional structural properties on its arguments. The second example shows how a step-wise refinement of the data encoding can yield additional structural knowledge. In both examples, I have shown that the introduced amendments were only local and did not require a refactoring of the entire program text.

Furthermore, I have motivated and introduced contracts as a means to model arbitrary constraints on function arguments. To support contracts, I have added pattern guards to the features proposed by my approach.

Finally, to enhance readability and further ease programming in my approach, I have introduced some syntactic sugar, namely

- support for implicit labels for values,
- a notation to bind the values of labels in pattern to identifiers that can be used in the definition of a function,
- a shortcut notation for equality constraints on properties of function arguments,
- tags, *i.e.*, valueless labels, in records.

Next, I will give a formal definition of the syntax and semantics of the above language features.

3. A Formal Definition of LRec

In this chapter, I will formalise the design presented in the previous chapter by means of LRec, a first order pure functional programming language with pattern matching and records. I start off in the next section by formally specifying the syntax of LRec. Next, Section 3.2 presents as set of source-to-source transformations to stepwise de-sugar LRec programs. I close that section with the definition of LRec$_C$, a de-sugared core of LRec. Finally, Section 3.3 presents an operational semantics for LRec$_C$, before I end this chapter with some conclusions in Section 3.4.

3.1. Syntax of LRec

In the following, I present the syntax of LRec. As my thesis is concerned with presenting a minimal framework to support auxiliary computations for dynamically typed languages in general, LRec consists of only the bare essentials required for my approach. To define a complete syntax with respect to the examples presented in the previous chapter, I provide additional definitions for the applied features that are used in those examples in Appendix B.

Figure 3.1 on the following page shows the syntax of LRec in extended Backus Naur form. The production rule *expression* defines the set of syntactically valid expressions. The syntax of the first among these, the *record*, is defined as a potentially empty comma separated list of *label=expression* pairs surrounded by curly brackets. As labels are not first class objects in LRec, labels cannot be bound to identifiers. Thus, only labels are allowed in label position. A further valid expression is the selection operation. Its syntax is given by the production rule *selection* as an expression followed by the `.`-symbol and a label. As with records, the label used for selection is not an expression as labels are not first class objects. The syntax of the special selection operation on implicit value labels is defined by the *blink* production rule as a `!`-symbol followed by an expression.

Next in the set of valid expressions is the `any` operation. Its syntax is defined by rule *any* as the keyword `any` followed by a non-empty list of expressions in parentheses. I use a different syntax for the `any` construct compared to the syntax of general function applications to highlight syntactically that `any` is a built-in construct.

The next two valid expressions are *identifier* and *boolean* values. The former is needed to reference expressions bound by the `val` construct. I have not given an explicit definition of the syntax of an identifier. Any definition used by the embedded target language suffices, as long as it excludes the keywords of LRec. Boolean values are defined by the production rule *boolean* in the usual way. These are the only values included in LRec.

program	$\Rightarrow$	**expression**
expression	$\Rightarrow$	**record** $\mid$ **selection** $\mid$ **blink** $\mid$ **any**
	$\mid$	**identifier** $\mid$ **boolean** $\mid$ **conditional**
	$\mid$	**equal** $\mid$ **let** $\mid$ **application**
record	$\Rightarrow$	**expression** `{` [**element** [`,` **element**]*] `}`
element	$\Rightarrow$	**label** [`=` **expression**]
selection	$\Rightarrow$	**expression** `.` **label**
blink	$\Rightarrow$	`!` **expression**
any	$\Rightarrow$	`any` `(` [**expression**]$^+$ `)`
boolean	$\Rightarrow$	`true` $\mid$ `false`
conditional	$\Rightarrow$	`if` **expression** **expression** **expression**
equal	$\Rightarrow$	`(` **expression** `=` **expression** `)`
let	$\Rightarrow$	`let` [**definition**]* `in` **expression** `end`
definition	$\Rightarrow$	**relation**
	$\mid$	**value**
	$\mid$	**function**
relation	$\Rightarrow$	`rel` **label** `<:` **label**
value	$\Rightarrow$	`val` **identifier** `=` **expression**
function	$\Rightarrow$	`fun` [**instance**]$^+$
instance	$\Rightarrow$	[**pattern**]$^+$ [**guards**] `=` **expression**
pattern	$\Rightarrow$	**identifier** `{` [**part** [`,` **part**]*] `}`
part	$\Rightarrow$	**label** [`=` **identifier**]
guards	$\Rightarrow$	`|` [**expression**]$^+$
application	$\Rightarrow$	`(` **identifier** [*expression*]$^+$ `)`

Figure 3.1.. Syntax of LRec in extended Backus Naur form.

I have chosen to include Boolean values to be able to give a semantic meaning to the conditional construct in later sections.

The production rule *conditional* defines the syntax for conditional expressions in LRec. A conditional is represented syntactically by the keyword `if` followed by three expressions: the predicate expression, the expression for the *then* case that is evaluated if the predicate is true and the expression for the *else* case that is evaluated for a false predicate.

LRec furthermore includes an equality function on non-record values. Strictly speaking, such an equality function belongs to the applied extensions of LRec, as it is defined on all non-record values. I nonetheless include it here as it is required for the lowering of implicit equality constraints as presented in Section 3.2.3. As the only non-record values in LRec are Boolean values, the function `equal` can only be used to compare those. For an applied extension of LRec, it needs to be extended accordingly. The corresponding production rule *equal* defines the syntax of the equality function as an expression, the `=`-symbol and a further expression, surrounded by parentheses.

Next is the `let` construct defined in the production rule *let*. It consists of the keyword `let` followed by a potentially empty set of definitions. These are followed by the body of the `let` construct surrounded by the keywords `in` and `end`. LRec supports three kinds of definitions. Firstly, the production rule *relation* defines the syntax for specifying a relation on labels: The keyword `rel` is followed by a label, the keyword `<:` and a second label. Similarly to records and selections, labels in a relation definition are not first class.

The second kind of definition is the binding of expressions to identifiers as defined by the *value* production rule. Such a binding is specified syntactically by the keyword `val` followed by an identifier, the `=`-symbol and an expression.

Lastly, LRec supports function definitions within the `let` construct. Their syntax is defined by the production rule *function* as the keyword `fun` followed by a non empty set of instance definitions. The syntax of instances in turn is defined by the production rule *instance*. An instance is specified by an instance signature consisting of one or more argument pattern and an optional guard definition for contracts. Finally, the signature is followed by the `=`-symbol and an expression for the body of the instance. Below, I will discuss the different parts of the signature part of an instance from left to right.

The first component of the signature part of an instance definition are the argument patterns. The syntax of an argument pattern is given by the production rule *pattern*. An argument pattern consists of an identifier for the argument name, followed by a list of pattern components surrounded by curly brackets. Each pattern component in turn is either a label, or a label and an identifier separated by the `=`-symbol. The latter syntax is used to introduce an implicit binding of record values in the function body and to specify implicit equality constraints.

Finally, the last component of the signature part of an instance is the guard definition. Its syntax is defined by the *guard* production rule. A guard definition starts with the `|`-symbol. This symbol is followed by a non-empty list of guard expressions.

Still missing is the syntax for function applications in LRec. It is defined by the production rule *application*. A function application is represented syntactically by the `(`-symbol followed by an identifier, a non-empty set of expressions and a final `)`-symbol. I only allow identifiers in function position as LRec is a first order language and furthermore does not support nameless functions, *i.e.*, functions that are defined inline in the code.

Function definition and function application both syntactically require at least one argument. This restriction rules out argument-less functions. However, as LRec is pure, an argument-less function always represents a constant value and thus can be represented by the `val` construct, as well. Enforcing this syntactically makes this fact more obvious to the programmer. However, in the end, it is merely a matter of taste.

This completes the discussion of the syntax of LRec. In the next section, I will demonstrate how LRec can be de-sugared and reduced to a core language LRec$_\text{C}$.

3.2. Towards a Sugar-Free LRec

One goal of my thesis is to provide a language extension with as clear as possible semantics. In particular, I want to enable the programmer to fully understand the semantics without much prior knowledge. To achieve this and to keep the semantics definition of LRec as concise as possible, I first reduce LRec to a simpler core language LRec$_\text{C}$ with all syntactic sugar removed. To define the semantics of the syntactic sugar in LRec, I use a set of source-to-source transformations from LRec to LRec, which will ultimately lead to a program in LRec's sugar free core LRec$_\text{C}$.

I perform this transformation in three steps. First, Section 3.2.1 describes how to resolve tags in records by rewriting them to labels with the special `unit` constant as value. The next section translates implicit equality constraints and bindings in patterns to explicit equality constraints and selection operations. Lastly, I describe in Section 3.2.3 how to resolve the implicit label for values by rewriting record definitions, argument patterns and the `!` operation as explicit definitions for a value label and operations thereon.

Finally, I close the overall discussion on syntax by a definition of the reduced syntax of LRec$_\text{C}$ in Section 3.2.4.

3.2.1. Resolving Tags in Records

In Section 2.2, I have introduced special valueless auxiliary computations, referred to as tags, to model properties of data that do not carry a value but merely encode a property by their existence in a record. These tags can then be exploited in the pattern of instance definitions to define specialised instances for certain tags. The motivating example used in Section 2.2 was the `ldiag` property of matrices. As a reminder, a tag is defined by the following syntax:

```
1  let
     val A = true{ TAG}
3  in
     !A
5  end
```

In Line 2 above, a record with the value `true` and the additional valueless label `TAG` is bound to the identifier `A`. The selection in Line 4 then selects the value of this record.

To make record expressions more uniform in the following discussion of LREC, it is desirable to rewrite the above expression such that all labels have a corresponding value. This can be achieved by replacing the definition of a tag by a label with an arbitrary value bound to it. As pattern matching in LREC does not take values into account, this does not influence the instance selection process. However, a too naïve choice of the value of such a replacement can nonetheless have an influence on the meaning of an expression. As an example, consider the following expression using a tag:

```
1  let
     val A = true{ TAG}
3  in
     if A.TAG true false
5  end
```

Line 2 above binds a record with tag `TAG` to the identifier `A`. The value of this tag is then used in Line 4 as the predicate for a conditional. As tags do not carry a value, a natural semantics would be that the conditional cannot be evaluated, as `A.TAG` does not represent a Boolean value.

By replacing tags with ordinary labels carrying an arbitrary value this expected semantics might change. For example, consider using the value `true` as the chosen value for tags. A rewriting would then yield the code below:

```
1  let
     val A = true{ TAG=true}
3  in
     if A.TAG true false
5  end
```

Now, as the label `TAG` carries the value `true` (cf. Line 2), the expression in Line 4 can indeed be evaluated, yielding `true` as its result.

To prevent unexpected changes in program meaning as the one shown above, I introduce a special expression to LREC, which I will use as value for tags: the *unit expression*, syntactically denoted by the ~-symbol. It represents a non-value and cannot be further evaluated. Using this special value for tags ensures that introducing ~ into program texts does not change their meaning. If the value of a tag was used in the original program text, each use of the value will still fail.

For my above example, a rewriting using the ~ value would yield the following code:

```
1  let
     val A = true{ TAG=~}
3  in
     if A.TAG true false
5  end
```

Even though the tag has been replaced by a label `TAG` with value ~ (cf. Line 2), the conditional in Line 4 now cannot be evaluated. Thus, rewriting tags with labels carrying the value ~ does not change the expected program meaning.

Figure 3.2 on the next page presents a formal definition of this rewriting by means of a transformation scheme $\mathscr{L}_t$. Throughout this thesis, I describe source-to-source transformations by functions on the syntax of LREC. In this case, I define a function $\mathscr{L}_t$

(Record) $\mathscr{L}_{\mathrm{t}}[\![\ e_{val}\texttt{\{}c_1\texttt{,}\ \ldots\texttt{,}\ c_n\texttt{\}}\ ]\!]$

$\rightsquigarrow\ \mathscr{L}_{\mathrm{t}}[\![\ e_{val}\ ]\!]\texttt{\{}\mathscr{L}_{\mathrm{t}}[\![\ c_1\ ]\!]\texttt{,}\ \ldots\texttt{,}\ \mathscr{L}_{\mathrm{t}}[\![\ c_n\ ]\!]\texttt{\}}$

(ElementVal) $\mathscr{L}_{\mathrm{t}}[\![\ \mathit{label}\texttt{=}\mathit{expression}\ ]\!]$

$\rightsquigarrow\ \mathit{label}\texttt{=}\mathscr{L}_{\mathrm{t}}[\![\ \mathit{expression}\ ]\!]$

(ElementTag) $\mathscr{L}_{\mathrm{t}}[\![\ \mathit{label}\ ]\!]$

$\rightsquigarrow\ \mathit{label}\texttt{=\~{}}$

Figure 3.2.. Transformation scheme $\mathscr{L}_{\mathrm{t}}$ to resolve tags in record definitions.

that accepts a LRec program with tags yields a semantically equivalent LRec program where these have been resolved. The first rewriting rule, the rule Record, defines the rewriting of records by applying the transformation $\mathscr{L}_t$ recursively to the value expression and each element of the record.

For record elements, Figure 3.2 defines two rewriting rules: The rule ElementVal for components with values and the rule ElementTag for tags, *i.e.*, valueless components. The former rewrites a record element of the form *label=expression* by recursively applying the transformation $\mathscr{L}_t$ to the expression part of the element. The latter performs the actual rewriting. A record component of the form *label* is rewritten as a component with the special ˜ expression as value.

I have omitted the rules for driving the transformation $\mathscr{L}_t$ through the remaining expressions of LRec. For reference, the corresponding rules can be found in Figure C.1 on page 201 in Appendix C.

3.2.2. Resolving Implicit Equality Constraints and Bindings in Patterns

For convenience, LRec allows to bind elements of the argument records of a function to identifiers within the function body. This is expressed syntactically by augmenting a label in a pattern to the right by an =-symbol and an identifier. In Section 2.3 I have further motivated and introduced implicit equality constraints in function signatures. These can be expressed by binding the same identifier to different record elements within one instance signature. In this section, I present a source-to-source transformation that resolves this syntactic sugar.

A naïve way to resolve implicit bindings would be to substitute all occurrences of the corresponding identifier in expression position by an explicit selection operation.

However, such a substitution would not preserve the meaning in general. Consider the following example:

```
1  let
     fun foo A{ label=s}
3      = let
           val A = true{ label=true}
5        in
           s
7        end
   in
9    (foo true{ label=false})
   end
```

In the above code, the signature of the single instance of function `foo` binds the value of the `label` element of the argument `A` to the identifier `s`. In the body of the function, the identifier `A` is then bound to a different record, *i.e.*, the expression `true{ label=true}` (cf. Line 4). Lastly, the result of `foo` is defined as the value of `s` in Line 6. Thus, the expression in Line 9 evaluates to `false`. Using the naïve approach and substituting all occurrences of `s` by its definition, *i.e.*, the expression `A.label`, would yield the following code.

```
   let
2    fun foo A{ label}
       = let
4          val A = true{ label=true}
         in
6          A.label
         end
8  in
     (foo true{ label=false})
10 end
```

Above, I have removed the implicit binding of `s` in the definition of `foo` in Line 2. Furthermore, I have replaced all occurrences of `s`, *i.e.*, the single occurrence in Line 6, by the expression `A.label`. This, however, has changed the meaning of the expression in line 8. Now, when evaluating the application of `foo` to `true{ label=false}`, the selection in Line 6 references the value of `A` as defined in Line 4 instead of the value of the function argument. Thus, the application of `foo` yields the result `true`.

To prevent scoping related problems like the one shown above, I use a different strategy to resolve implicit bindings. Instead of substituting the identifier by its bound value, I surround the function body by a set of explicit bindings using the `let` construct. For example, transforming the original code given above yields the following result:

```
   let
2    fun foo A{ label}
       = let
4          val s=A.label
         in
6          let
```

```
        val A = true{ label=true}
 8     in
        s
10     end
      end
12 in
     (foo true{ label=false})
14 end
```

I have resolved the implicit binding of `s` by wrapping the `let` construct of the function body into a further `let` construct. This outer `let` construct explicitly binds the `label` component of `A` to the identifier `s` (cf. Line 4). Thus, the expression `s` in Line 9 now evaluates to the correct value, despite the new binding for `A` in Line 7.

It is worth noting here that in the above example it would suffice to introduce the explicit binding of `s` to the existing `let` construct, as long as the additional bindings are inserted at the top of the definition chain. However, as a function is not required to have a top-level `let` construct, this solution would not scale to the general case.

The above transformation suffices to resolve implicit bindings if each identifier is only bound once within each instance signature. However, implicit equality constraints allow the programmer to bind the same identifier more than once. In this case, the identifier corresponds to any one of the values that are bound to it. To express this, I make use of the `any` operation. Instead of defining the identifier by a single selection, I bind it to an application of `any` to the selections of all record elements that were previously bound implicitly. As an example, consider an extended version of the example above.

```
   let
 2   fun foo A{ labelX=s} B{labelY=s}
       = let
 4         val A = true{ label=true}
         in
 6         s
         end
 8 in
     (foo true{ labelX=false}) false{ labelY=false}
10 end
```

The function `foo` as defined in Line2 above now expects two arguments. By means of an implicit binding, the value of the `labelX` component of the first argument and the `labelY` component of the second argument are bound to the identifier `s`. To resolve the implicit binding above, it suffices to introduce an explicit binding of the expression `any( A.labelX B.labelY)`, as shown below.

```
   let
 2   fun foo A{ labelX} B{labelY}
       = let
 4         s = any( A.labelX B.labelY)
         in
 6         let
             val A = true{ label=true}
```

```
8          in
             s
10         end
         end
12 in
     (foo true{ labelX=false}) false{ labelY=false}
14 end
```

The explicit binding is shown in Line 4 above. In case that `A.labelX` and `B.labelY` have the same value, this transformation correctly resolves the implicit binding. What is still missing is a contract to ensure the above constraint. In general, such a contract has to ensure that all components that are bound to an identifier have the same value. To do so, it suffices to require that for each disjoint pair of components, both have the same value. As in the above example only two components are bound to `s`, this can be enforced by a single contract `(A.labelX = B.labelY)`. The resulting code is given below.

```
   let
2    fun foo A{ labelX} B{labelY} | (A.labelX = B.labelY)
       = let
4          s = any( A.labelX B.labelY)
         in
6          let
             val A = true{ label=true}
8          in
             s
10         end
         end
12 in
     (foo true{ labelX=false}) false{ labelY=false}
14 end
```

Line 2 shows the added contract in form of a guard. The above code is semantically equivalent to the expected semantics of both implicit bindings and implicit equality constraints.

Figure 3.3 on the following page gives a formal definition of the above transformation for the general case. It defines a function $\mathscr{L}_e$ that accepts a LRec program with implicit bindings and implicit equality constraints and yields a semantically equivalent LRec program where these have been resolved. I assume that all components of a pattern are bound to an identifier to simplify the specification of $\mathscr{L}_e$. This does not affect the applicability of the transformation in the general case, as unbound components can easily be transformed into components bound to a fresh, unused identifier.

Rule Instance in Figure 3.3 on the next page specifies how an instance definition of a function is transformed. To handle implicit equality constraints, I define an equivalence class $[i]_P$ over a set of bindings P with respect to the used identifier i. The set P thereby represents all (argument,label,identifier) triples collected from the instance definition. An (argument,label,identifier) triple is in P if the instance contains an implicit binding of the component of an argument referenced by the label to the identifier. The equivalence class $[i]_P$ is then defined by the equality relation on the third element of the triple, *i.e.*,

$$\text{(Instance)}\quad \mathscr{L}_{\mathsf{e}}\left[\!\!\left[\begin{array}{l} a_1\texttt{\{ } l^1_1\texttt{=}i^1_1\texttt{, } \ldots\texttt{, } l^1_{n_1}\texttt{=}i^1_{n_1}\texttt{\}} \;\cdots\; a_m\texttt{\{ } l^1_m\texttt{=}i^m_1\texttt{, } \ldots\texttt{, } l^m_{n_m}\texttt{=}i^m_{n_m}\texttt{\}} \\ \texttt{|}\ \textit{guards}\ \texttt{=}\ \textit{expression} \end{array}\right]\!\!\right]$$

$$\rightsquigarrow\quad \begin{array}{l} a_1\texttt{\{ } l^1_1\texttt{, } \ldots\texttt{, } l^1_{n_1}\texttt{\}} \;\cdots\; a_m\texttt{\{ } l^m_1\texttt{, } \ldots\texttt{, } l^m_{n_m}\texttt{\}} \\ \texttt{|}\ \textit{newguards}\ \mathscr{R}_{\mathsf{e}}[\![\mathtt{S},\ \mathscr{L}_{\mathsf{e}}[\![\ \textit{guards}\ ]\!]\ ]\!] \\ \texttt{=}\ \mathscr{R}_{\mathsf{e}}[\![\mathtt{S},\ \mathscr{L}_{\mathsf{e}}[\![\ \textit{expression}\ ]\!]\ ]\!] \end{array}$$

where S is defined as

$$S = \bigcup_{i\in I}\{(i, \{(a,l) \mid (a,l,i) \in [i]_P\})\}$$

and *newguards* is defined as the expressions contained in the set

$$G = \bigcup_{i\in I}\{\texttt{(}a_j\texttt{.}l_j\ \texttt{=}\ a_k\texttt{.}l_k\texttt{)} \mid (a_j,l_j,i),(a_k,l_k,i) \in [i]_P \wedge (a_j \neq a_k \vee l_j \neq l_k)\}$$

with the sets P and I defined as

$$P = \bigcup_{j=1}^{m}\{(a_j, l^j_k, i^j_k) \mid k \in \{1,\ldots,n_j\}\},$$

$$I = \bigcup_{j=1}^{m}\bigcup_{k=1}^{n_j}\{i^j_k\}$$

and the equivalence class $[i]_P$ defined as

$$[i]_P = \{(a,l,i') \in P \mid i = i'\}.$$

Figure 3.3.. Transformation scheme $\mathscr{L}_e$ for resolving implicit equality constraints and bindings in patterns.

the identifier. Using this equivalence class, I compute the set G of additional guards required to express the equality constraints. It contains, for each identifier i used in an implicit binding, guards to assert the equality of each disjoint pair of record components from $[i]_P$.

Furthermore, to resolve the implicit bindings, I compute the set S of pairs associating a set of record components with the identifier they are bound to. This set of bindings is then used to rewrite the existing guards and the body expression of the instance definition by means of the rewriting function $\mathscr{R}_e$. As Figure 3.4 on the facing page shows, given a set of bindings S and an expression *expression* to rewrite, $\mathscr{R}_e$ yields the expression wrapped in a new `let` construct that contains explicit bindings for all identifiers in S. To express these bindings, I use an application of the `any` operator to selection operations that correspond to the (argument,label) pairs associated with an identifier. The resulting expression is then bound to the identifier using the `val` construct.

$$\mathscr{R}_{\mathsf{e}}[\![S,\ \ \mathit{expression}\ ]\!] = \texttt{let}\ \mathit{definitions}\ \texttt{in}\ \mathit{expression}\ \texttt{end}$$

where *definitions* is defined as the definitions contained in the set

$$D = \bigcup_{(i,\{(a_1,l_1),\ldots,(a_n,l_n)\})\in S} \{\texttt{val}\ i\texttt{=any(}\ a_1\texttt{.}l_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{.}l_n\texttt{)}\}$$

Figure 3.4.. Rewriting function $\mathscr{R}_e$ for introducing explicit bindings.

Note that in rule Instance in Figure 3.3 on the preceding page, before rewriting the expressions, first the transformation scheme $\mathscr{L}_e$ is recursively applied to the guards and body expression. This ensures that implicit equality constraints and implicit bindings in function definitions within these expressions are resolved, as well. I have omitted here the rules that are needed only to drive the transformation $\mathscr{L}_e$ through arbitrary expressions. These can be found in Figure C.2 on page 204 in Appendix C.

3.2.3. Resolving Implicit Labels for Values

As a final lowering step from LRec to LRec$_\text{C}$, I in the following describe the rewriting of the implicit labels for the value component of a record into an explicit value label. In Chapter 2, I have introduced the implicit value label to syntactically highlight the actual computation in contrast to further auxiliary computations encoded in a record. For the formal discussion of LRec, this distinction is not needed and would only obfuscate the general principles of evaluation.

To start off, I first give an example of the syntax of implicit labels for values as a reminder:

```
  let
2   val A = true{ X=false}
  in
4   !A
  end
```

In the above code, the value `A` is defined in Line 2 as a record with value `true` and the auxiliary computation `X` with value `false`. In Line 4, the value component of the record bound to `A` is then selected using the selection for implicit values `!`. To rewrite the above code with explicit value labels, it suffices to bind the value of the record in Line 2 to a so far unused label and to replace the implicit selection in Line 4 by an explicit selection of that label. The resulting code is given below.

```
let
  val A = { val=true, X=false}
in
  A.val
end
```

(Record) $\mathscr{L}_{\mathtt{i}}[\![\ e_{val}\texttt{\{}l_1\texttt{=}e_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n\texttt{\}}\ ]\!]$

$\leadsto\ \texttt{\{val=}\mathscr{L}_{\mathtt{i}}[\![\ e_{val}\ ]\!]\texttt{,}\ l_1\texttt{=}\mathscr{L}_{\mathtt{i}}[\![\ e_1\ ]\!]\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}\mathscr{L}_{\mathtt{i}}[\![\ e_n\ ]\!]\texttt{\}}$

(Blink) $\mathscr{L}_{\mathtt{i}}[\![\ \texttt{!}\mathit{expression}\ ]\!]$

$\leadsto\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{expression}\ ]\!]\texttt{.val}$

(Pattern) $\mathscr{L}_{\mathtt{i}}[\![\ a\texttt{\{}l_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{\}}\ ]\!]$

$\leadsto\ \mathscr{L}_{\mathtt{i}}[\![\ a\texttt{\{val,}\ l_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{\}}\ ]\!]$

Figure 3.5.. Transformation scheme $\mathscr{L}_{\mathtt{i}}$ to resolve implicit labels for values.

As Line 2 shows, I have used the label `val` to bind the value component of the record. I will use the label `val` to refer to the value component of a record in the remainder of this thesis. This, of course, assumes that the label `val` is not used anywhere else in the program text. That restriction, however, can be easily achieved by restricting the set of labels in LRec accordingly. Line 4 above shows the rewriting of the implicit selection into an explicit selection using the `val` label.

A formal definition of this last rewriting is given by the transformation scheme $\mathscr{L}_i$ in Figure 3.5. As before, I only show the most important rules here. The remaining rules are given in Appendix C in Figure C.3 on page 205.

The rule Record rewrites a record definition with an implicit value component to a record using the label `val` by binding the value expression in front of the original record definition to the label `val` in the resulting record. Furthermore, the transformation $\mathscr{L}_i$ is recursively applied to all expressions for the components of the record, including the expression for the value.

The rewriting of selections of the value component using the `!` operator is defined by rule Blink. The original selection is replaced by an explicit selection of the `val` label. To resolve implicit value labels in the expression the selection is applied to, $\mathscr{L}_i$ is recursively applied to that expression, as well.

Lastly, in rule Pattern, I add the label Val to each pattern. Although this is not strictly required, as the pattern would match without explicitly listing the label `val`, as well, I add the label to highlight that each argument is required to have a value component. As the existence of a value component for each record value is syntactically enforced, this rewriting has no impact on the set of matching expressions.

This completes the discussion of the transformations required to de-sugar LRec. In the next section, I give a formal definition of the syntax of the de-sugared core of LRec.

3.2.4. Syntax of LRec$_C$

By applying the transformation schemes $\mathscr{L}_t$, $\mathscr{L}_e$ and $\mathscr{L}_i$ in the given order, a program text in LRec can be fully de-sugared to a corresponding program text in the core language LRec$_C$. All formal discussions in the remainder of this thesis will use LRec$_C$ as their basis. For reference, I close the discussion of the syntax of LRec with a formal definition of the syntax of its core LRec$_C$.

Figure 3.6 on the next page gives the syntax of LRec$_C$ in extended Backus Naur form. In comparison to the syntax of full LRec as shown in Figure 3.1 on page 38, LRec$_C$ does no longer contain the blink operator `!`, nor support for implicit values in the production rule *record* for records. These language features are resolved by the $\mathscr{L}_i$ transformations.

To reflect the transformation of tags to ordinary labels with values by the $\mathscr{L}_t$ transformation, I have adapted the production rule *element* for record elements accordingly.

Furthermore, the syntax for implicit bindings and thus implicit equality constraints has been removed from the production rule *pattern* for argument patterns of function instances. As the transformation $\mathscr{L}_e$ fully resolves these, it is no longer required. As the only addition, I have added the special expression `~` used by the $\mathscr{L}_t$ transformation. The corresponding production rule *unit* is now referenced in the rule for valid expressions *expression*.

Given this syntax of core LRec, I will now define its formal semantics by providing an operational semantics for LRec$_C$ in the next section.

3.3. An Operational Semantics For LRec$_C$

In this section, I provide an operational semantics for LRec. I only provide the semantic rules for the core LRec$_C$. The semantics for the syntactic sugar is given by the semantics of its transformation into LRec$_C$ provided by the transformation schemes presented in the previous sections. Furthermore, this section does not consider guards in LRec$_C$. I will discuss these and their semantics separately in Chapter 4.

The semantics are provided by a relation $\Downarrow$ between expressions of LRec$_C$ and values. $\Downarrow$ is not a function, as one expression can potentially be evaluated to different results, depending on which argument is chosen when evaluating the `any` operator. Before discussing the relation's defining rules in detail, I first give a formal definition of the set of values. Figure 3.7 on page 51 shows the production rules for all valid values in extended Backus Naur form.

A value, as defined by the production rule *value*, is either a Boolean value or a record value. Boolean values, as defined in rule *boolean*, are the two constants *true* and *false*. Record values are defined by the production rule *record*. A record is represented as a set of (label,value) pairs. Note that, in contrast with record expressions in LRec$_C$, record values can potentially be empty sets. I will use this to represent the non-value `~`.

In the following, the set $\mathscr{V}$ refers to all values that can be produced using the production rule *value* in Figure 3.7 on page 51. Furthermore, I will use $\mathscr{R}$ to denote the set of values that can be produced using the production rule *record*, *i.e.*, all record values. Lastly, $\mathscr{L}$ refers to the set of all valid labels. As in previous definitions of the syntax of LRec, I

program	⇒	**expression**
expression	⇒	**record** $\mid$ **selection** $\mid$ **unit** $\mid$ **any**
	$\mid$	**identifier** $\mid$ **boolean** $\mid$ **conditional**
	$\mid$	**equal** $\mid$ **let** $\mid$ **application**
record	⇒	`{` [**element** [`,` **element**]*] `}`
element	⇒	**label** `=` **expression**
selection	⇒	**expression** `.` **label**
unit	⇒	`~`
any	⇒	`any` `(` [**expression**]$^{+}$ `)`
boolean	⇒	`true` $\mid$ `false`
conditional	⇒	`if` **expression** **expression** **expression**
equal	⇒	`(` **expression** `=` **expression** `)`
let	⇒	`let` [**definition**]* `in` **expression** `end`
definition	⇒	**relation**
	$\mid$	**value**
	$\mid$	**function**
relation	⇒	`rel` **label** `<:` **label**
value	⇒	`val` **identifier** `=` **expression**
function	⇒	`fun` [**instance**]$^{+}$
instance	⇒	[**pattern**]$^{+}$[**guards**] `=` **expression**
pattern	⇒	**identifier** `{` [**label** [`,` **label**]*] `}`
guards	⇒	`|` [**expression**]$^{+}$
application	⇒	`(` **identifier** [*expression*]$^{+}$ `)`

Figure 3.6.. Syntax of LREC$_{\mathrm{C}}$ in extended Backus Naur form.

do not give a definition of this set of labels. Instead, any set of discriminating identifiers can be used.

As a further prerequisite, I define some functions on record values below. Firstly, I define the domain of a record as the set of its labels:

Definition 3.3.1 (Domain of a Record)**.** *The* domain *of a record value dom* $: \mathscr{R} \to \mathscr{L}$ *is defined as*

$$dom(R) := \{l \mid \exists v \in \mathscr{V} : (l, v) \in R\}.$$

This definition is motivated by the notion of records as partial functions from the set of labels to the set of values. Analogously, the range of a record is defined as the set of its values:

Definition 3.3.2 (Range of a Record)**.** *The* range *of a record value range* $: \mathscr{R} \to \mathscr{V}$ *is defined as*

$$range(R) := \{v \mid \exists l \in \mathscr{L} : (l, v) \in R\}.$$

Lastly, I define the element of a record at a given label as the value corresponding to that label:

Definition 3.3.3 (Element of a Record)**.** *The* element *of a record value elem* $: \mathscr{R} \times \mathscr{L} \to \mathscr{V}$ *is defined as*

$$elem(R, l) := \begin{cases} v & \text{if } (l, v) \in R, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Figure 3.8 on the next page provides a big-step operational semantics for LREC$_C$. To describe the semantics, I use a natural deduction system with inference rules in the style of Gentzen [1934]. I use the common notation for derivation formulae known from standard textbooks like [Pierce, 2002]. The statement $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ is to be read as: Given an evaluation environment $(\mathscr{F}, \prec, \mathscr{E})$, the term e can be evaluated to the value v. The evaluation environment thereby is a triple consisting of the function environment $\mathscr{F}$, the current partial order on labels $\prec$, which is used in pattern matching, and an environment of variable bindings $\mathscr{E}$. I assume throughout this thesis that all orders are strict, *i.e.*, that they are anti-symmetric and thus not reflexive.

value	$\Rightarrow$	**boolean** | **record**
boolean	$\Rightarrow$	`true` | `false`
record	$\Rightarrow$	`{` *[* **element** *[* `,` **element** *]** *]* `}`
element	$\Rightarrow$	`(` **label** `,` **value** `)`

Figure 3.7.. Set of legal values used as by the operational semantics.

True : $$\frac{}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{true} \Downarrow \mathit{true}}$$

False : $$\frac{}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{false} \Downarrow \mathit{false}}$$

Unit : $$\frac{}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{\~{}} \Downarrow \emptyset}$$

Var : $$\frac{(i, v) \in \mathscr{E}}{(\mathscr{F}, \prec, \mathscr{E}) : i \Downarrow v}$$

EqualTrue : $$\frac{(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow v_2 \quad v_1, v_2 \notin \mathscr{R} \quad v_1 \overset{v}{=} v_2}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{(}e_1 \texttt{ = } e_2\texttt{)} \Downarrow \mathit{true}}$$

EqualFalse : $$\frac{(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow v_2 \quad v_1, v_2 \notin \mathscr{R} \quad v_1 \overset{v}{\neq} v_2}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{(}e_1 \texttt{ = } e_2\texttt{)} \Downarrow \mathit{false}}$$

Record : $$\frac{\begin{array}{c}\forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow l_i \neq l_j \\ \forall i \in \{1, \ldots, n\} \ : \ (\mathscr{F}, \prec, \mathscr{E}) : e_i \Downarrow v_i\end{array}}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{\{}l_1\texttt{=}e_1\texttt{, } \ldots\texttt{, } l_n\texttt{=}e_n\texttt{\}} \Downarrow \{(l_1, v_1), \ldots, (l_n, v_n)\}}$$

Selection : $$\frac{(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v \quad v \in \mathscr{R} \quad l \in \mathrm{range}(v)}{(\mathscr{F}, \prec, \mathscr{E}) : e\texttt{.}l \Downarrow \mathrm{elem}(v, l)}$$

Any : $$\frac{\exists i \in \{1, \ldots, n\} \ : \ (\mathscr{F}, \prec, \mathscr{E}) : e_i \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{any(}e_1\texttt{, } \ldots\texttt{, } e_n\texttt{)} \Downarrow v}$$

CondThen : $$\frac{(\mathscr{F}, \prec, \mathscr{E}) : e_p \Downarrow \mathit{true} \quad (\mathscr{F}, \prec, \mathscr{E}) : e_t \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{if } e_p \ e_t \ e_e \Downarrow v}$$

CondElse : $$\frac{(\mathscr{F}, \prec, \mathscr{E}) : e_p \Downarrow \mathit{false} \quad (\mathscr{F}, \prec, \mathscr{E}) : e_e \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{if } e_p \ e_t \ e_e \Downarrow v}$$

Let : $$\frac{(\mathscr{F}', \prec', \mathscr{E}') : e \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \ \texttt{let } d_1 \cdots d_n \texttt{ in } e \texttt{ end} \Downarrow v}$$

$$\text{where} \quad \begin{array}{rcl} \prec' & = & \mathrm{rel}(\prec, \{d_1, \ldots, d_n\}) \\ \mathscr{F}' & = & \mathrm{fun}(\mathscr{F}, \prec', \{d_1, \ldots, d_n\}) \\ \mathscr{E}' & = & \mathrm{val}(\mathscr{F}', \prec', \mathscr{E}, (d_1, \ldots, d_n)) \end{array}$$

Figure 3.8.. An operational semantics for LRec$_\mathrm{C}$.

$$\textsc{Ap} \quad : \quad \frac{\begin{array}{c}\forall i \in \{1,\dots,n\} \ : \ (\mathscr{F},\prec,\mathscr{E}) \ : \ e_i \Downarrow v_i \in \mathscr{R} \\ \{(\mathscr{F}',\prec',\mathscr{E}',e_b)\} = \text{match}(\mathscr{F},f,(v_1,\dots,v_n)) \\ (\mathscr{F}',\prec',\mathscr{E}') : e_b \Downarrow v\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : (f \ \ e_1 \ \dots \ e_n) \Downarrow v}$$

Figure 3.8.. An operational semantics for LREC$_C$ (contd.).

To support scoping of function definitions, the function environment $\mathscr{F}$ is a triple $(\mathscr{F}',\prec',F)$ where $\mathscr{F}'$ is the function environment of the outer scope, $\prec'$ is the partial order on labels as valid at the current scoping level and F is a set of functions defined at the current scoping level. Initially, I use $\perp$ as the empty function environment. I will discuss the structure of the function set F in detail when defining the semantics of the `let` construct.

The second component of the evaluation environment is a partial order on labels $\prec \subset \mathscr{L} \times \mathscr{L}$. Initially, $\prec$ is the empty set. It is extended by the evaluation of the `rel` construct of LREC$_C$.

Finally, $\mathscr{E} \subset \mathscr{I} \times \mathscr{V}$ encodes the current bindings of values to identifiers. Above, $\mathscr{I}$ denotes the set of all legal identifiers. As with labels, I do not formally define this set here. Instead, any set of discriminating identifiers can be used. Evaluation starts out with an empty variable environment. New bindings are introduced by evaluations of the `val` construct. To insert new variable bindings into the environment, I use the operation $\leftarrow\!\!\!\!\lhd$ as defined below.

Definition 3.3.4 (Variable Insertion)**.** *Given a set $\mathscr{E} \subset \mathscr{I} \times \mathscr{V}$, the set $\mathscr{E} \leftarrow\!\!\!\!\lhd (i,v)$ where $i \in \mathscr{I}$ and $v \in \mathscr{V}$ is defined as*

$$\mathscr{E} \leftarrow\!\!\!\!\lhd (i,v) := \{(i',v') \mid (i',v') \in \mathscr{E} \wedge i \neq i'\} \cup \{(i,v)\}$$

By using $\leftarrow\!\!\!\!\lhd$ as defined above instead of simple set union, I model the shadowing of previous bindings to an identifier by a new binding.

The first inference rule in Figure 3.8 on the facing page, the rule TRUE, defines the evaluation of the expression `true` to the value *true*. Analogously, the rule FALSE specifies the evaluation of the expression `false` to the value *false*. A note on typesetting: I use `teletype` when referring to an expression and an *italic* typeface when referring to values.

Next, the evaluation of the special `~` expression is defined in rule UNIT. The `~` expression is evaluated to the empty record $\emptyset$, *i.e.*, the record that carries no value. It is important to note here that `~` is the only expression that evaluates to the empty record. All other record values at least carry the special `val` label and an according value. This is enforced by the syntax of LREC with its implicit label for values and the corresponding transformation defined in Section 3.2.3.

The rule VAR defines the evaluation of bound identifiers. An identifier can be evaluated to a value if a corresponding tuple is present in the current variable environment $\mathscr{E}$.

The semantics of the built-in equality function `=` is defined by the two rules EQUALTRUE and EQUALFALSE. The former defines the case where both expressions evaluate to equal values, whereas the latter handles the opposing case where both expression evaluate to non-equal values. In both rules, I require that the values being compared are non-record values. For the comparison, the relation $\overset{v}{=}$ is used. For Boolean values, it is defined as follows.

Definition 3.3.5 (Equivalence of Values)**.** *The equivalence relation $\overset{v}{=} \subset \mathscr{V} \times \mathscr{V}$ on values is defined as*

$$\overset{v}{=} \ := \ \{(\mathit{true}, \mathit{true}), (\mathit{false}, \mathit{false})\}$$

Obviously, $\overset{v}{=}$ as defined above is indeed an equivalence relation. Furthermore, it covers all non-record values in $\mathscr{V}$. It is worth noting here that actual implementations of LRec that feature a richer set of non-record values must extend the equality relation on values accordingly. Otherwise, equality constraints on those additional kinds of values cannot be expressed via pattern guards. I provide such amended versions of $\overset{v}{=}$ for the two extensions of LRec$_\text{C}$ used in this thesis, *i.e.*, for the extensions used in the matrix and algebraic data type examples, in Appendix B. In the following, I will use $v_1 \overset{v}{=} v_2$ as a shorthand for $(v_1, v_2) \in \ \overset{v}{=}$.

Rule Record specifies the semantics of record expressions in LRec. They evaluate to a corresponding record value, if each expression bound to a label within the record can be evaluated to a value. Furthermore, I require that each label occurs only once within the record expression.

Corresponding to the rule Record for record expressions, the rule Selection defines the semantics for the selection expression in LRec. Firstly, rule Selection requires that the first argument of an selection expression evaluates to a record value. Furthermore, the second argument of the selection expression needs to be in the domain of the record value. If both conditions hold, the selection evaluates to the element of the record value that corresponds to the label. The functions *dom* and *elem* are defined in Definitions 3.3.1 on page 51 and 3.3.3, respectively.

The definition of selection in rule Selection allows for non-deterministic results if two values are bound to the same label in a record value. However, the rule Record ensures that a corresponding record expression cannot be evaluated and thus, a non-deterministic selection can never occur.

Next in Figure 3.8 on page 52 is the rule Any that defines the semantics of the `any` expression in LRec. If any of the argument expressions of an application of `any` can be evaluated to a value, the entire `any` expression evaluates to this value. It is worth noting here that I neither require all argument expressions to evaluate to the same value nor that all argument expressions can be evaluated at all. I have chosen to only require the evaluation of one expression to allow for an efficient implementation. The fact that all arguments need to evaluate to the same value has to be encoded by the programmer using a contract, if at all desired.

The `any` operator is similar to McCarthy's `amb` operator [McCarthy, 1961, 1962] in that the result of both operators is defined as the value of any of the arguments. However, the two operations differ in an important aspect: Whereas the `amb` operator requires that all argument expressions are checked until at least one is found that does not diverge and indeed yields a value as result, the `any` operator has no such requirement. If the argument expression to an `any` operation that has been chosen for evaluation diverges, so does the `any` operation itself. Clinger has coined the terms *call by need* and *call by lazy* [Clinger, 1982] for the different semantics of `amb` and `any`, respectively.

In principle, using these semantics for the `any` operations introduces non-determinism to LREC. In particular, the `any` operator invalidates referential transparency, also commonly referred to as the Church Rosser property after Churchs and Rosser's proof of confluence [Church and Rosser, 1936] of reduction to normal form in the lambda calculus [Barendregt, 1981; Hindley and Seldin, 1986]. To regain referential transparency, which is needed for some proofs in later chapters of this thesis, one can for instance employ a technique introduced by Burton [Burton, 1988]. In his paper, Burton describes how to rewrite a non-deterministic program into a deterministic program using an infinite tree of decisions. The key idea is to parametrise each non-deterministic operation within the program by a decision, which is drawn deterministically from the infinite tree of decisions. This additional parameter is then used to decide the non-deterministic behaviour of the operation. Thus, multiple copies of the same non-deterministic expression, as they use the same element from the decision tree, will perform the same choice and thus ultimately yield the same result.

Burton's technique allows to regain referential transparency in the most general case, including the `amb` operator. However, in the context of LREC with its `any` operation, a simpler solution exists. In contrast with the `amb` operator, the `any` operation may choose any of its arguments, regardless of whether the argument diverges or not. Thus, for a deterministic implementation of LREC, it would suffice to always evaluate for instance the first argument of the `any` operation. This choice satisfies the semantics defined here without introducing non-determinism.

For the remainder of this thesis, I will not choose a particular technique to ensure determinism of evaluation. Instead, I will simply assume that evaluation is deterministic when required. In these cases, either of the above techniques can be applied. However, where not explicitly stated otherwise, I assume a non-deterministic `any` operation.

The next two rules, CONDTHEN and CONDELSE define the semantics of conditionals in LREC. Both require that the predicate can be evaluated to a Boolean value. In rule CONDTHEN, this value has to be *true*, whereas in the rule CONDELSE a value of *false* is required. In case of the former rule, the result of evaluating the conditional expression is then defined as the result of evaluating the then expression, if possible. Similarly, for CONDELSE the result of evaluating the conditional is defined as the result of evaluating the else expression, if such result exists.

The penultimate rule in Figure 3.8 on page 52 defines the semantics of the `let` construct in LREC. In short, a `let` construct evaluates to the value of its body expression under an extended environment, if such a value exists, *i.e.*, if the evaluation of the body expression does not get stuck.

I use a three step process to construct the new environment $(\mathscr{F}', \prec', \mathscr{E}')$ from the definitions $d_1 \ldots d_n$ contained in the `let` construct. First, I construct a new order on labels $\prec'$ from the previous order $\prec$ and the definitions $d_1 \ldots d_n$ using the function *rel*. Figure 3.9 on the following page shows the definition of *rel*.

The function *rel* computes a new order from a given order $\prec$ and a set of definitions

$$\mathrm{rel}(\prec, \{d_1, \ldots, d_n\}) := \prec_n$$

where

$$\begin{aligned} \prec_0 &:= \prec \\ \prec_{i+1} &:= \mathrm{rel'}(\prec_i, d_{i+1}) \end{aligned}$$

with

$$\mathrm{rel}'(\prec, d) := \begin{cases} \mathrm{clos}(\prec \cup \{(l_1, l_2)\}) & \text{if } d \equiv \texttt{rel}\ l_1\ \texttt{<:}\ \ l_2 \text{ and } (l_2, l_1) \notin \prec, \\ \prec & \text{if } d \equiv \texttt{val}\ i\ \texttt{=}\ e, \\ \prec & \text{if } d \equiv \texttt{fun}\ i_1 \ldots i_n, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Figure 3.9.. Definition of function *rel* as used in Figure 3.8 on page 52.

$\{d_1, \ldots, d_n\}$ by step-wise inclusion of all order definitions using the `rel` construct to the partial order $\prec$. The step-wise inclusion is modelled by the inductive definition of the result. For each definition d_i, a new partial order is computed using the helper function *rel′* and the previously computed partial order. For function definitions and value bindings using the `fun` and `val` constructs, respectively, the partial order remains unchanged. For a new order definition of the form `rel` l_1 `<:` l_2, the partial order is amended by a corresponding tuple. However, to ensure that the result is a partial order again, I firstly require asymmetry, *i.e.*, I require that for a definition `rel` l_1 `<:` l_2 the symmetric case (l_2, l_1) is not yet contained in the order. Furthermore, I compute the transitive closure of the previous partial order extended by the additional tuple to ensure transitivity of the result. In Figure 3.9 this is denoted by the application of the function clos to the extended order. Thus, at any stage of evaluation, $\prec$ is a partial order on labels. For invalid order definitions, *i.e.*, those that would invalidate the above property, the function *rel′* is undefined.

Using the adapted partial order $\prec'$, in the second step I compute a new function environment $\mathscr{F}'$ by means of the function *fun* shown in Figure 3.10 on the facing page. Similarly to the function *rel*, the function *fun* extracts function definitions from a set of definitions D by applying a helper function *fun′* to each definition. Contrary to the definition of *rel*, I use a simple set union to combine the extracted sets of functions to the overall set of function definitions of the entire `let` construct. As function definitions are independent of each other, no special restrictions need to be enforced when inserting them into the function environment. The result of the function *fun* is a triple, consisting of the previous function environment, the current partial order on labels and the set of functions defined in the current `let` construct. This nesting is used in function applications to model the scoping of function definitions. The current partial order on labels is included in the triple to ensure that the correct order is used in pattern matching for applications of the functions in a different context. As definitions of the partial order on labels using

$$\mathrm{fun}(\mathscr{F}, \prec, D) := (\mathscr{F}, \prec, \bigcup_{d \in D} \mathrm{fun}'(d))$$

where

$$\mathrm{fun}'(d) := \begin{cases} \{f\} \times \mathrm{inst}(\{i_1, \ldots, i_n\}) & \text{if } d \equiv \texttt{fun}\ f\ \ i_1\ \cdots\ i_n, \\ \emptyset & \text{otherwise.} \end{cases}$$

with

$$\mathrm{inst}(I) := \bigcup_{i \in I} \mathrm{inst}'(i)$$

where

$$\begin{aligned} &\mathrm{inst}'(\alpha_1\texttt{\{}\ l^1_1\texttt{,}\ \ldots\texttt{,}\ l^1_{n_1}\texttt{\}}\ \cdots\ \alpha_m\texttt{\{}\ l^m_1\texttt{,}\ \ldots\texttt{,}\ l^m_{n_m}\texttt{\}}\ \texttt{=}\ e) \\ &:= \{(((\alpha_1, \{l^1_1, \ldots, l^1_{n_1}\}), \ldots, (\alpha_m, \{l^m_1, \ldots, l^m_{n_m}\})), e)\} \end{aligned}$$

Figure 3.10.. Definition of function *fun* as used in Figure 3.8 on page 52.

the `rel` construct are visible throughout a `let` construct, I use the already extended order $\prec'$ here. I do not need to add the current variable binding environment $\mathscr{E}$ into this closure as LREC requires all functions to be closed expressions. I enforce this property in the rule AP for function applications, which I will discuss further below.

The function *fun′* computes the set of functions for a single definition. For definitions of the order on labels and the definition of values using the `rel` and `val` constructs, respectively, the resulting set of function definitions is empty. For function definitions using the `fun` construct, I construct a set of tuples where each tuple consists of the function name and one of the instances defined in the given function definition.

Instance definitions are extracted using the helper function *inst*. It computes a set of instances as the union of the sets computed by the function *inst′* applied to each instance of the set of instances I. The function *inst′* in turn computes a representation for a single instance definition as a tuple consisting of the instance's signature and the defining expression of that instance. The signature is encoded as a tuple containing tuples describing each argument. These, finally, contain the identifier used for the argument a_i and the set of labels $\{l^i_1, \ldots, l^i_{n^i}\}$ describing the pattern corresponding to that argument.

I use a tuple to represent the arguments as the order in this case matters. For the labels of argument pattern, however, the order is not relevant and thus a set is used. It is worth noting here that the function *inst* does not enforce that all instances have the same number of arguments, nor that the pattern of the instance definitions allow for a uniquely defined best match. The former is a design choice. By allowing different numbers of arguments in instance definitions, I allow for reusing the same identifier for functions of different arity. The latter is checked by the matching process used for function applications, which I will discuss in detail further below.

As a final remark on function definitions, note that the function *fun* does not enforce that each function identifier is only used once in a single `let` construct. Again, this is a design choice. Multiple definitions using the same function name will be treated as

$$\mathrm{val}(\mathscr{F}, \prec, \mathscr{E}, (d_1, \ldots, d_n)) := \mathscr{E}_n$$

where

$$\begin{aligned}
\mathscr{E}_0 &:= \mathscr{E} \\
\mathscr{E}_{i+1} &:= \begin{cases} \mathscr{E}_i \leftarrow (l, v) & \begin{array}{l}\text{if } d_{i+1} \equiv \texttt{val}\ \ l\texttt{=}e \\ \text{and } (\mathscr{F}, \prec, \mathscr{E}_i) : e \Downarrow v,\end{array} \\ \mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{fun}\ \ f\ \ i_1\ \cdots\ i_m, \\ \mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{rel}\ \ l_1\ \texttt{<:}\ \ l_2, \\ \text{undefined} & \text{otherwise.} \end{cases}
\end{aligned}$$

Figure 3.11.. Definition of function *val* as used in Figure 3.8 on page 52.

a single function definition in the context of function application as discussed further below.

The last step in constructing a new environment for evaluating the `let` construct is the processing of bindings using the `val` construct. This is expressed in rule LET of Figure 3.8 on page 52 using the function *val*. Figure 3.11 provides a definition of this function.

Value bindings using the `val` construct in LREC use a different scoping rule than function definitions and definitions of the order on labels using the `fun` and `rel` constructs, respectively. Each binding is only visible in further bindings below its definition in a `let` construct and in the body of the `let` construct. However, as functions need to be closed, value bindings have no impact on their definitions. To model the scoping of value bindings, the function *val* is defined by induction on the set of definitions. Starting out with the original variable environment $\mathscr{E}$, for each definition d_i a new variable environment $\mathscr{E}_i$ is computed. If the definition corresponds to a `rel` or `fun` construct, the environment remains unchanged. For a `val` construct, the environment is amended if the bound expression can be evaluated to a value using the current variable environment and the updated function environment and partial order on labels as computed in the previous two steps. I use the special insertion operation $\leftarrow$ defined in Definition 3.3.4 here to model the shadowing semantics of bindings: If another value has been bound to the same identifier previously, the corresponding tuple will be removed from the variable environment.

This completes the definition of the new environment $(\mathscr{F}', \prec', \mathscr{E}')$ used to evaluate the body expression of a `let` construct in rule LET in Figure 3.8 on page 52. It is important to note that the two functions *rel* and *val* are only partial functions on the syntax of LREC$_\text{C}$. The function *rel* is undefined for `rel` constructs that would invalidate the partial order property of $\prec$. For bindings of expressions that cannot be evaluated, the function *val* is undefined. In either case, the inference rule LET cannot be applied. Thus, a `let` construct that contains invalid `rel` or `val` constructs cannot be evaluated.

The last missing rule from Figure 3.8 on page 52 is the rule AP for function applications. A function application can be evaluated if each argument evaluates to a corresponding record value and furthermore the pattern matching function *match* yields a single instance. In this case, the function application evaluates to the value of the defining expression e_b of the matching instance, evaluated using a new environment $(\mathscr{F}', \prec', \mathscr{E}')$, if such a value exists. The new environment consists of the function environment $\mathscr{F}'$ and the partial order on labels $\prec'$ as they were valid in the defining context of the function. Furthermore, the new environment contains an updated variable environment $\mathscr{E}'$ that binds all actual arguments to the formal parameters of the function. This new environment is computed, alongside the matching instance, by the matching function match as defined in Figure 3.12 on the next page.

Before I discuss the formal definition of the matching process in function applications, I first define the lifting of $\prec$ from labels to entire pattern.

Definition 3.3.6 (Complete Partial Order on Pattern)**.** *Given a partial order $\prec \subset \mathscr{L} \times \mathscr{L}$ on labels, the corresponding partial order $\overset{\rightarrow}{\prec} \subset \mathscr{P}(\mathscr{L}) \times \mathscr{P}(\mathscr{L})$ on sets of labels is defined as*

$$\overset{\rightarrow}{\prec} := \{(P, P') \mid P, P' \in \mathscr{P}(\mathscr{L}) \wedge |P| = |P'| \wedge \forall l \in P \setminus P' \exists l' \in P' \setminus P : l \prec l'\}.$$

Thus, for two patterns P and P', $P \overset{\rightarrow}{\prec} P'$ if all labels l in P that are not present in P' are shadowed by at least one label in P' that is not in P. Furthermore, the order is only defined for pattern of equal length. Pattern of different length are unrelated. I use the function $\mathscr{P}$ above to denote the powerset of a set, *i.e.*, the set of all subsets of a set.

Using the order $\overset{\rightarrow}{\prec}$ on pattern, I now formally define the instance matching process *match* as used in rule AP in Figure 3.8 on page 52. The details are given in Figure 3.12 on the next page. The function match expects three arguments: a function environment $\mathscr{F}$, the function name f, and the argument tuple of the application a. It yields a set of matching instances, each encoded as a tuple consisting of a new function environment, a new order on labels, a new variable environment and, lastly, the defining expression of the corresponding function instance.

The matching process itself consists of five stages. The first stage, the function *lookup*, takes a function environment $\mathscr{F}$, a function name f and an argument tuple a. As its result, the function *lookup* yields a new function environment, a new order on labels, a set of matching instances and the function arguments. The set of matching instances I is computed by extracting all instances of function definitions that use f as function name from the top-most set of functions in the environment $\mathscr{F}$. If this set of instances is empty, the look-up process continues with the previous function environment. Finally, if the function *lookup* reaches the empty function environment $\bot$, an empty set of instances alongside an empty function environment and an empty order on labels is returned.

The function *lookup* models the shadowing of previous function definitions by a new definition using the same identifier. As soon as at least one instance has been found in a function environment, the remaining nested function environments are not searched. Thus, redefining a function in a nested scope shadows all previous definitions, regardless of their arity and argument pattern.

Furthermore, note that the function *lookup* combines all instances from all function definitions at a certain scoping level that use the same function name. Thus, a single

$$\text{match} := \text{bind} \circ \text{order} \circ \text{pattern} \circ \text{arity} \circ \text{lookup}$$

where *lookup* is defined as

$$\begin{aligned}
\text{lookup}(\bot, f, a) &:= (\bot, \emptyset, a) \\
\text{lookup}((\mathscr{F}', \prec, F), f, a)) &:= \begin{cases} ((\mathscr{F}', \prec, F), I, a) & \text{if } I := \{i \mid (f, i) \in F\} \neq \emptyset, \\ \text{lookup}(\mathscr{F}', f, a) & \text{otherwise.} \end{cases}
\end{aligned}$$

and the function *arity* is defined as

$$\text{arity}(\mathscr{F}, \prec, I, a) := (\mathscr{F}, \prec, \{(s, e) \in I \mid |s| = |a|\}, a).$$

The function *pattern* is defined as

$$\text{pattern}(\mathscr{F}, \prec, I, (a_1, \ldots, a_n)) := (\mathscr{F}, \prec, I_n, (a_1, \ldots, a_n))$$

where

$$\begin{aligned}
I_0 &:= I \\
I_{i+1} &:= \text{filter}(I_i, a_i, i)
\end{aligned}$$

with

$$\text{filter}(I, a, i) := \max_{|p_i|}\{(((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I \mid p_i \subseteq \text{dom}(a)\}.$$

Furthermore, the function *order* is defined as

$$\text{order}(\mathscr{F}, \prec, I, (a_1, \ldots, a_n)) := (\mathscr{F}, \prec, I_n, (a_1, \ldots, a_n))$$

where

$$\begin{aligned}
I_0 &:= I \\
I_{i+1} &:= \text{filter}'(\prec, I_i, i + 1)
\end{aligned}$$

with

$$\begin{gathered}
\text{filter}'(\prec, I, i) \\
:= \{(((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I \mid \forall(((\alpha'_1, p'_1), \ldots, (\alpha'_n, p'_n)), e') \in I : p_i \overset{\rightarrow}{\not\prec} p'_i\}.
\end{gathered}$$

Finally, *bind* is defined as

$$\text{bind}(\mathscr{F}, \prec, I, a) := \{(\mathscr{F}, \prec, \text{bind}'(P, a), e) \mid (P, e) \in I\}$$

where

$$\text{bind}'(((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), (a_1, \ldots, a_n)) := \{(\alpha_1, a_1), \ldots, (\alpha_n, a_n)\}.$$

Figure 3.12.. Best-match pattern matching function *match* as used in Figure 3.8 on page 52.

definition with multiple instances and multiple definitions using the same function name with some instances each yield the same set of instances.

As the second stage, the function *arity* filters out all instances whose number of argument patterns does not match the number of arguments. The function expects a function environment $\mathscr{F}$, an order on labels $\prec$, a list of instances I and, lastly, an argument tuple as arguments. It yields the function environment $\mathscr{F}$, the order $\prec$, a filtered list of instances and the argument tuple. In the definition of the function *arity*, I use $|\cdot|$ to denote the cardinality of a tuple, *i.e.*, $|(x_1, \dots, x_n)| := n$.

Next, the third matching stage, the function *pattern*, filters the set of instances to those that match the arguments' patterns. Given a function environment $\mathscr{F}$, an order on labels $\prec$, a list of instances I and an argument tuple, the function returns the function environment $\mathscr{F}$, the order on labels $\prec$, a filtered set of instances I_n and the argument tuple.

The function *pattern* implements a left-to-right best match pattern matching on the arguments. I use an inductive definition of the result set of instances. Initially, the set of resulting instances I_0 is the set of instances supplied to the function *pattern* as argument. Starting from this set, I stepwise refine the set of instances by filtering out instances whose pattern do not match a given argument.

To implement left-to-right pattern matching, the refinement starts with the first argument, leaving only those instances in the result set whose first argument pattern matches the first argument. This is expressed by the helper function *filter*. Given a set of instances I, an argument tuple a and the argument position i to filter on, it removes all instances from the set of instances whose i-th pattern does not match the corresponding argument. This is expressed by the condition $p_i \subseteq \mathrm{dom(a)}$. Thus, all patterns that contain labels which are not present in corresponding argument are rejected.

The best match strategy is encoded using the *max* function in the definition of filter. The function $max_{|p_i|}$ maximises the number of labels contained in the patterns at argument position i such that the resulting set is non-empty. Thus, the function *filter* yields only those instances that match the most labels of a given argument.

In the fourth step, the set of matching instances is further reduced by filtering out all those instances that are shadowed by another instance with respect to the order on patterns $\overset{\rightarrow}{\prec}$. Analogously to the definition of the function *pattern*, I construct the result of the function *order* inductively to model filtering from left to right. For each pattern p of an instance, the instance is filtered out if at least one instance exists whose corresponding pattern p' precedes it in $\overset{\rightarrow}{\prec}$, *i.e.*, a pattern p' for which $p' \overset{\rightarrow}{\prec} p$ holds..

The function *order* yields the final set of instances, alongside the function environment, the order on labels and the argument tuple. As the final step in the matching process, the function *bind* constructs the new variable environments for each instance and glues the components together to a set of instance tuples as required in rule AP in Figure 3.8 on page 52. The actual environment construction is performed by the function *bind′*. Given the patterns of an instance, encoded as a tuple of (identifier,pattern) tuples, and the argument tuple, it produces a set of (identifier,value) pairs encoding the binding of each argument value to the corresponding identifier of the formal parameter of the given instance.

It is worth noting here that the function *match* yields a set of matching instances. If the best match is not uniquely defined, the resulting set contains multiple instances. Therefore, in rule AP of Figure 3.8 on page 52, I require that the result of the function *match* is a set with only a single element, *i.e.*, I require that the best match is uniquely defined. Another design choice would be to choose any of the resulting instances. For the example of matrix addition, as presented in Chapter 2, this would be advantageous. As all instances compute the same value, choosing any instance would suffice. However, in case of non-homomorphic function instances, this would lead to non-deterministic results.

This concludes the discussion of the semantics of LREC_C in this section. The semantics of the applied extensions of LREC used in the examples in Chapter 2 is provided for reference in Appendix B. I will extend the semantics by guard expressions in the next chapter. First, I close this chapter with some conclusions.

3.4. Conclusions

In this chapter, I have defined the syntax of LREC by means of a formal description in extended Backus Naur form. Next, I have described the lowering of expressions in LREC to a core language LREC_C. In Section 3.2, I have presented a set of source-to-source transformation that define equivalent expressions in LREC_C for the syntactic sugar of LREC. In particular, these transformations resolve

- tags in records (Section 3.2.1),
- implicit equality constraints and implicit labels in patterns (Section 3.2.2),
- implicit labels for values (Section 3.2.3).

Using this de-sugared core of LREC, I have provided a formal semantics for LREC_C without guards in Section 3.3.

In the next Chapter, I discuss an encoding of pattern guards and the corresponding extension to the syntax and semantics of LREC_C.

4. Checking Constraints

In the previous chapter, I have presented a range of lowering steps to reduce the syntactic sugar of LRec to a language core LRec$_C$. The corresponding transformations are all aware of guard expressions annotated at the signatures of function instances. The description of the semantics, however, has left these out as of yet. In this chapter, I will close this gap.

A straight forward solution would be to simply define the semantics of guard expressions and explicitly support guards in applied languages with auxiliary computations. However, one of my goals in this thesis is to allow for an easy implementation of my approach as an extension to existing languages. Requiring explicit support for guard expressions may interfere with this goal.

To gain an insight into the challenges that arise when extending a language by guards, I have investigated an extension of SaC [Scholz, 2003], a first order functional language geared at numerical computations, by guard expressions [Herhut et al., 2008]. The approach presented in this chapter is based on that work.

Apart from allowing for an easy implementation in compilers for existing languages, the extension of SaC by guard expressions has identified three further goals:

Feedback on the verification process Expressing constraints on arguments as explicit guard expressions allows me to use existing optimisations, *e.g.*, partial evaluation and code specialisation techniques, to discharge guard expressions statically. As an example, consider the matrix addition function `add` presented in Chapter 2. If the `shape` component of its arguments is statically known, we can statically decide the guard that checks for equality of these components. Such optimisations can have a dramatic impact on runtime performance [Bernecky, 1998]. Therefore, it is desirable for a programmer to specify his programs in such a way that the optimisation techniques available to him can decide as many guard expressions statically as possible. However, to make this decision, it is important that the guard expressions remain easily identifiable in intermediate versions of his code during program optimisation. This allows for an informed choice of where a program needs to be amended by further statical knowledge to aid the optimisation techniques in discharging guard expressions.

Once all means of optimisation have been exhausted, still unsafe regions of the program, *i.e.*, parts of the code that yet contain guard expressions, may remain. To assess the safety of a program with respect to the constraints expressed by guards, the programmer needs to be able to clearly identify where guard expressions remain.

Efficient runtime checking Not all guard expressions can be decided statically. One goal of my approach is to allow the programmer to express constraints that are not decidable in general. Thus, inevitably, some guard expressions will remain. To still make the use of guard expressions feasible, especially in the realm of numerical applications, their impact on the runtime behaviour needs to be kept as small as possible.

Firstly, this requires an efficient implementation of runtime checks. However, a discussion of implementation techniques for efficient runtime checks is beyond the scope of this thesis. Yet, this point needs to be kept in mind when implementing my approach in a compiler or interpreter.

Secondly, even if a guard can not be eliminated, the number of checks that are performed at runtime should be kept as small as possible. This requires an efficient means to propagate knowledge gained by a guard to similar guards later in the control flow. For example, again consider the matrix addition presented in Chapter 2. If we add the same matrix twice to some other matrix, a naïve approach would check at each invocation that the shapes of the arguments match, *i.e.*, the corresponding guard would be evaluated twice. However, if we statically know that a check has been performed earlier in the control flow, we can remove the corresponding guard at later stages.

Guard unaware optimisations The encoding used for guards needs to ensure that existing optimisations do not accidentally remove guards or propagate guarded expressions in the control flow before the evaluation of the actual guard expression. Given the number of optimisations that modern compilers apply – in the case of the SaC compiler this are currently more than 50[1] - checking the validity of each optimisation in the presence of guards and potentially adapting some of them to the new scenario is a major engineering task.

In the next section, I will evaluate different encodings for guard expressions with respect to the above criteria and present the solution I have chosen. I then discuss a lowering scheme that rewrites guard expressions to the chosen representation in Section 4.2. Finally, I describe the extensions to the syntax and semantics of LRec$_C$ required by the lowered representation of guards before I close this chapter in Section 4.4 with some conclusions.

4.1. An Encoding for Guards

Throughout this section, I will use the definition of matrix addition in LRec$_C$ as running example. To simplify the presentation, I provide a shorter version of the matrix addition defined in Chapter 2 that only uses the `shape` attribute below and has only a single

[1]The source tree of the SaC research compiler `sac2c` contains implementations of more than 50 optimisations as of revision 16529.

instance. However, the arguments given in the remainder of this section apply to the full version, as well.

```
1   fun add A{ shape} B{ shape}
      | (A.shape = B.shape)
3     = (vect_add A! B!){ shape=any(A.shape B.shape)}
```

In Line 1 above I define a single instance of the function `add` that expects two arguments which both carry the `shape` attribute. The value of the result is then computed by element-wise adding the data vectors of the two arguments. Furthermore, the `shape` component of the result is computed as any of the shape components of the arguments. This definition, of course, is only valid and it only yields a deterministic result if the `shape` components of both arguments equate. This is ensured by the guard expression in Line 2.

4.1.1. Guards by Conditionals

A naïve encoding of the above guard with only the language constructs contained in LRec$_\mathrm{C}$ would be to express guards as conditionals in the code. A simple rewriting could have the following form:

```
1   fun add A{ shape} B{ shape}
      = if (A.shape = B.shape)
3         (vect_add A! B!){ shape=any(A.shape B.shape)}
          fail
```

Above, I have replaced the guard by a conditional in the body of the function. The body expression of the original definition is only evaluated if the guard evaluates to true, *i.e.*, if the expression `(sA == sB)` evaluates to true. Otherwise, the expression in Line 4 is evaluated. I use the special function `fail` to encode that evaluation has to be halted due to a failed guard.

The above rewriting nicely captures the expected semantics of a guard expression. It is ensured that the guarded function body is only evaluated if the guards are true. Furthermore, the encoding fulfils the third design criterion, *i.e.*, using the above encoding for guards, existing optimisations need not to be checked nor modified. Even without guards, propagating code outside of a conditional is invalid. Moreover, as I only use the conditional construct, which should already exist in any language, existing optimisations might already discharge the guard statically if such an optimisation exists for conditionals in general. However, with respect to the other design criteria, the above encoding is far from optimal.

Reusing the existing conditional construct reduces the implementation effort. On the other hand, however, it disguises guard expressions. Differentiating between regular conditionals that stem from the original program text and those that result from guards is rather difficult. As a solution one could annotate the conditional as special. This, then again, would increase the implementation effort: Existing code transformations would need to be amended such that they preserve the information that a conditional was inserted due to a guard expression.

More importantly, the above encoding makes it hard to reach the second goal, *i.e.*, to produce efficient runtime checking code. As an example for this, consider the following application of matrix addition:

```
   let
2    fun add A{ shape} B{ shape}
       = if (A.shape = B.shape)
4          (vect_add A! B!){ shape=any(A.shape B.shape)}
           fail
6    val C = (add A B)
     val D = (add A C)
8  in
     D
10 end.
```

In the example above, A and B refer to some matrix values, *i.e.*, record values with a data vector as value and a shape vector as shape component. I first add A to B in Line 6 and then the result to A again in Line 7. Applying a standard function inlining optimisation might lead to the following, semantically equivalent code:

```
   let
2    val C = if (A.shape = B.shape)
                (vect_add A! B!){ shape=any(A.shape B.shape)}
4               fail
     val D = if (A.shape = C.shape)
6               (vect_add A! C!){ shape=any(A.shape C.shape)}
                fail
8  in
     D
10 end.
```

I have replaced the function applications by the corresponding body expression of the function add. The resulting code now contains two conditionals. The first in Line 2 asserts that the shape components of A and B are equal. Similarly, in Line 5 the same property is asserted for the shape components of A and C. However, one of the conditionals above is superfluous and thus imposes an unnecessary runtime overhead.

To see why, consider the evaluation of the above expression. Once the value of C has been computed, A and B must have identical shape components, as otherwise the computation would have failed. Thus, C has the same shape component, as well. It follows that the conditional in Line 5 will always evaluate to true.

Situations like the one above are common. It has been shown for SaC that propagating symbolic knowledge gained from conditionals and built-in operations like the shape equalities above can have a major impact on runtime behaviour [Bernecky, 2007; Trojahner et al., 2007]. However, the above encoding makes such an analysis difficult. When looking just at the conditionals, the shape equality property checked in Line 2 above is only valid within the *then* branch of the conditional. Thus, for the conditional in Line 5 no static knowledge about the value A is known. Even more, the information that C carries either the shape element of A or B is lost, as well. This information cannot be propagated out of the conditional, as it is not statically known which branch will be chosen at runtime. For the *else* branch in Line 4 of the above example, no information about the resulting value is known.

Only if the termination behaviour of the function `fail` is taken into account, it is valid to propagate the static knowledge about the equality of the `shape` components. However, this is neither conceptually trivial nor easy to implement.

4.1.2. Weaving Guards into the Dataflow

As the previous section has shown, an encoding for guards needs to allow for easy propagation of the validity of the encoded property. One means to achieve this is to weave the guards into the dataflow, *i.e.*, to encode the guards such that the guard expression conceptually returns a new copy of the data it asserts a property on, which is then statically known to have the asserted property. As an example for such an encoding, consider the following rewritten version of the matrix addition example.

```
    fun add A{ shape} B{ shape}
2     = let
          val A = guard(A (A.shape = B.shape))
4         val B = guard(B (A.shape = B.shape))
        in
6         (vect_add A! B!){ shape=any(A.shape B.shape)}
        end.
```

In lines 3 and 4 above, I use a special `guard` operation to enforce the guard expression annotated at the instance signature. The function `guard` is the identity on the first argument, if and only if the second argument evaluates to true. Otherwise, the `guard` operation cannot be evaluated and thus the program halts.

Semantically, this corresponds to the previous solution using conditionals. However, instead of wrapping the body expression of the function into such a conditional, I assert the property encoded by the guard expression for each of the values the guard expression enforces a property for. As the equality of the `shape` component encoded by the guard expression in my example is enforced for the `shape` component of the parameters `A` and `B`, I use two `guard` operations. The first in Line 3 checks the property for the first parameter whereas the second `guard` operation does so for the second parameter. In general, for each guard expression, one `guard` operation per free variable contained in the guard expression is needed. The idea here is that a guard expression always asserts a property for data that is defined outside of the guard expression itself.

Implementing the above solution still comes at a relatively low cost: Only the new `guard` operation needs to be added. The experiences made with an implementation for SaC support this. As the `guard` operation closely resembles a functional conditional or the ϕ function used in static single assignment form [Cytron et al., 1988], an implementation in languages that support either is straight forward.

Furthermore, the above solution fulfils the first requirement outlined in the introduction to this section, as well. As I use a special operation to encode guards, the guards in the resulting code can easily be identified. All `guard` operations correspond to a guard expression in the original program text. Even more, the data a property is asserted on is clearly visible as the result of the `guard` operation. Thus, to enhance the static correctness of a program, the programmer can now focus on remaining `guard` operations and means to enhance the static knowledge about the data needed to resolve these operations statically.

With respect to the second requirement, the above solution might seem to be even worse. For each data a guard asserts a property on, the guard expression is checked once. However, this can easily be rectified by applying a standard optimisation like common-sub-expression elimination: The guard expression used in each `guard` operation could be lifted in front of the `guard` operations and thus only be computed once. This optimisation could even be incorporated into the rewriting scheme. However, to simplify the presentation here, I will use one copy of the guard expression in each `guard` operation.

Another benefit of the above approach in the context of efficient runtime checks is the simplified propagation of knowledge gained from guard expressions. As an example, consider an inlined version of the application of `add` presented in the previous section. Using the encoding for guards presented above, the resulting code would look as follows.

```
1   val A = guard(A (A.shape = B.shape))
    val B = guard(B (A.shape = B.shape))
3   val C = (vect_add A! B!){ shape=any(A.shape B.shape)}
    val A = guard(A (A.shape = C.shape))
5   val C = guard(C (A.shape = C.shape))
    val D = (vect_add A! B!){ shape=any(A.shape C.shape)}
```

The first two lines in the above example assert the shape equality for the values `A` and `B`. Thus, it is statically known that the new values `A` and `B` as returned by the `guard` operations in lines 1 and 2 have identical `shape` components. From this, it directly follows that `C` has the same `shape` component, as well. Using this knowledge, the two `guard` operations in lines 4 and 5 can be statically evaluated to their respective first argument. Therefore, in the above example, only one guard expression needs to be checked at runtime.

Overall, the above solution allows for a more efficient implementation of the checks at runtime. However, with respect to the third criterion, *i.e.*, the support for guard unaware optimisations, the above solution is worse than the solution using conditionals. As an example, once more consider the implementation of matrix addition.

```
fun add A{ shape} B{ shape}
  = let
      val A = guard(A (A.shape = B.shape))
      val B = guard(B (A.shape = B.shape))
    in
      (vect_add A! B!){ shape=any(A.shape B.shape)}
    end.
```

In the above version of `add`, the data dependencies between the computation of the result and the results of the `guard` operations enforce that the guard expression is checked before the data, *e.g.*, the arguments `A` and `B`, are first used. Thus, existing optimisations should never propagate code that uses data for which a property is asserted by a guard in front of the evaluation of the actual guard. However, still in some cases the guard expression might not be checked.

The problem arises if a function only uses some but not all arguments that are used in guard expressions. As an example, consider the function `choose` as defined below:

```
1   fun choose N{} A{ shape} B{ shape} C{ shape}
      | (A.shape = B.shape) (B.shape = C.shape)
3     = if (!N = 1) A
          if (!N = 2) B
5           C
```

The function expects 4 arguments: A number `N` and three arrays `A`, `B` and `C`. Depending on the value of `N`, in lines 3ff. one of the arguments is chosen as result. Furthermore, the guards in Line 2 assert that all matrix arguments have the same `shape` component. However, I only use two guard expressions. One that asserts that `A` and `B` have equal `shape` components and a second one that checks this property for the arguments `B` and `C`. The third property that `A` and `C` have identical `shape` components, as well, then implicitly follows if the two guards are true.

Using the current approach, the above example would be lowered to the following representation:

```
1   fun choose N{} A{ shape} B{ shape} C{ shape}
      = let
3         val A = guard(A (A.shape = B.shape))
          val B = guard(B (A.shape = B.shape))
5         val B = guard(B (B.shape = C.shape))
          val C = guard(C (B.shape = C.shape))
7       in
          if (!N = 1) A
9           if (!N = 2) B
              C
11      end.
```

Above, the guards annotated at the function signature have been replaced by a sequence of `guard` operations in lines 3 and following. Note that, in order to introduce the new bindings, I have furthermore wrapped the function body into a `let` construct. The result of that `let` construct in lines 3ff. is then defined using the body of the original instance. However, due to the scoping rules of the `let` construct, the values `A`, `B` and `C` now refer to the results of the `guard` operations. Similarly, the second `guard` operation for the value `B` in Line 5 uses the result of the previous guard on `B` as its argument. Thus, all `guard` operations are properly tied into the dataflow and therefore are all evaluated before the result is computed.

Now consider the situation where `N` is statically known to have the value `1`. A compiler might inline the function `choose` or create a specialised version for just this specific value of `N`. In both cases, this enables the propagation of the statically known value of `N` to the conditional in Line 8. This, in turn, would allow an optimisation to statically decide the conditional and choose the *then* branch as the result of the function. Then, the residual program text for that instance would look as follows:

```
1   fun choose N{} A{ shape} B{ shape} C{ shape}
      = let
3         val A = guard(A (A.shape = B.shape))
          val B = guard(B (A.shape = B.shape))
5         val B = guard(B (B.shape = C.shape))
          val C = guard(C (B.shape = C.shape))
7       in
          A
9       end.
```

As can be seen, the result of the `let` expression is now defined as the value `A`. This optimisation is sound with respect to the result of the function `choose` where `N` has the value `1`. However, the body of the `let` construct now no longer contains any references to the values `B` and `C`. Thus, in a non-strict setting, these values might no longer be computed. Unfortunately, this implies that only the first `guard` operation in Line 3 would be checked and all other `guard` operations would be ignored.

As this example shows, the current encoding might invalidate optimisations in the setting with guards that used to be valid before. Using the above encoding thus would require all optimisations to be checked and adapted to the new setting. A clear violation of the third requirement.

4.1.3. Using Explicit Evidence

The underlying problem in the previous example is that the `guard` operation ensures that a guard expression is checked before the corresponding data is used. However, the `guard` operation does not enforce that the annotated guard expression is checked if the data is never used. To preserve the soundness of optimisations, I furthermore need to ensure that all guard expressions have been checked before the result of the function is used.

The idea is to provide explicit evidence that the result of the application of a guarded function is correct with respect to the guards. The notion of evidence in program optimisations is borrowed from [Menon et al., 2006]. Menon et al. use explicit evidence to ensure that assertions inserted by a compiler to verify properties of a low-level byte code are retained across optimisations. However, their encoding does not allow for user defined guards nor are they concerned with the propagation of static knowledge.

As an example for guards with explicit evidence, consider the following rewritten version of the function `choose`:

```
fun choose N{} A{ shape} B{ shape} C{ shape}
  = let
      val A = guard(A (A.shape = B.shape))
      val B = guard(B (A.shape = B.shape))
      val B = guard(B (B.shape = C.shape))
      val C = guard(C (B.shape = C.shape))
    in
      witness(if (!N = 1) A
                   if (!N = 2) B
```

```
                     C
11                   (A.shape = B.shape)
                     (B.shape = C.shape))
13          end.
```

In the code above, I use a further new operation `witness` in Line 8 to assert that the guards annotated at the function instance hold for the result of the function, as well. Semantically, the `witness` operation is similar to the `guard` operation: It is the identity on its first arguments, if the guard expressions evaluate to true. Otherwise, the `witness` operation cannot be evaluated. However, in contrast to the `guard` operation, the `witness` operation allows for multiple guard expressions as arguments. It is important to note here that the guard expressions need not to be evaluated again. Similar to the use of multiple `guard` operations, the guard expressions can be lifted out of the `witness` operation and standard optimisations like common-sub-expression elimination can be used to ensure that each guard expression is only evaluated once. Even the rewriting could already ensure this. To simplify the presentation, I have chosen not to do so here.

Specialising the above instance for a statically known argument `N` with value 1 now yields the following residual expression:

```
1     fun choose N{} A{ shape} B{ shape} C{ shape}
        = let
3           val A = guard(A (A.shape = B.shape))
            val B = guard(B (A.shape = B.shape))
5           val B = guard(B (B.shape = C.shape))
            val C = guard(C (B.shape = C.shape))
7         in
            witness(A{code:choose-43}*)
9                   (A.shape = B.shape)
                    (B.shape = C.shape))
11        end.
```

Again, the conditional has been reduced to the *then* case. However, as the conditional is wrapped into a `witness` operation, now even in a non-strict setting, all guards are evaluated. Furthermore, note that the guard expressions are still checked before the corresponding data is used for the first time. In the above example, the guard for `A` in Line 3 is checked before the result is passed to the `witness` operation. The guard on `B` and `C`, on the other hand, is checked by the `witness` operation just before the result is returned. This check is still early enough, as neither `B` nor `C` is ever used. In particular, properties of `B` and `C` cannot influence the value of the result.

This solution inherits all advantages of the previous dataflow encoding. Thus, it fulfils the first and second design criterion. Furthermore, by using explicit evidence, existing optimisations need not to be checked nor modified.

Using the above encoding for function guards, I can now formally define the corresponding lowering transformation for LRec.

4.2. A Lowering Transformation for Guards

In this section, I provide a formal definition of a lowering scheme $\mathscr{L}_c$ for lowering expressions in LREC with pattern guards to corresponding expressions in LREC$_\text{C}$ that use the encoding for guards as motivated in the previous section.

As can be seen in rule INSTANCE in Figure 4.1, an instance definition with guards is rewritten to an instance definition without guards by applying the rewriting function $\mathscr{R}_c$ to the set of lowered guard expressions and the lowered body expression of the function instance. Both need to be transformed, as well, as they may contain further function definitions with guards.

For all other expressions, the lowering scheme $\mathscr{L}_c$ is driven recursively into all subexpressions. For reference, I provide the corresponding rules in Figure C.4 on page 207 in the appendix.

The actual rewriting of a function body is performed by the rewriting function $\mathscr{R}_c$ as defined in Figure 4.2 on the facing page. Given a set of guard expressions $\{g_1, \ldots, g_n\}$ and the body expression of an instance definition e, it returns a new `let` construct that, for each guard expression g_i, introduces new bindings $\mathit{definitions}_i$ for all free variables contained in the guard expression g_i. The bindings for a guard g_i contain, for each free variable α of g_i, a new `val` construct that binds the result of an application of the **`guard`** operation with the free variable and guard expression as arguments to the free variable. Note that the order of these bindings has no impact on the semantics of the resulting expression as the **`guard`** operation does not alter the value. It is only important that each guard expression is properly woven into the dataflow. This is ensured by the scoping rules of the `let` construct.

Apart from introducing these new bindings, the rewriting scheme furthermore inserts explicit evidence in the form of a **`witness`** operation: The result of the created `let` construct is defined as the **`witness`** operation applied to the body of the instance and all guard expressions.

The helper function $\mathscr{F}\mathscr{V}$ used in Figure 4.2 on the next page extracts the set of free variables from an expression in LREC$_\text{C}$ using the usual way [Pierce, 2002]: An identifier is free with respect to an expression, if no corresponding `val` construct exists within that expression.

By applying the lowering scheme $\mathscr{L}_c$ as defined above, pattern guards in LREC can be rewritten to guards using **`guard`** and **`witness`** operations. In the next section, I will formally define their syntax and semantics.

(INSTANCE) $\mathscr{L}_\text{c}[\![\; a_1 \; p_1 \; \cdots \; a_n \; p_n \;\mid\; g_1 \; \cdots \; g_m \; = \; e \;]\!]$

$$\rightsquigarrow a_1 \; p_1 \; \cdots \; a_n \; p_n \; = \; \mathscr{R}_\text{c}[\![\{\mathscr{L}_\text{c}[\![\; g_1 \;]\!], \ldots, \mathscr{L}_\text{c}[\![\; g_m \;]\!]\},\; \mathscr{L}_\text{c}[\![\; e \;]\!]\;]\!]$$

Figure 4.1.. Transformation scheme $\mathscr{L}_c$ to resolve pattern guards.

$$\mathscr{R}_c \left[\!\left[\{g_1, \ldots, g_n\},\ e \right]\!\right] := \begin{array}{l} \texttt{let} \\ \quad \textit{definitions}_1 \\ \qquad \vdots \\ \quad \textit{definitions}_n \\ \texttt{in} \\ \quad \texttt{witness(}e\ g_1\ \cdots\ g_n\texttt{)} \\ \texttt{end.} \end{array}$$

where the $\textit{definition}_i$, $i \in \{1, \ldots, n\}$ are defined as

$$\textit{definitions}_i := \{\texttt{val}\ \alpha\ \texttt{=}\ \texttt{guard(}\alpha\ g_i\texttt{)} \mid \alpha \in \mathscr{FV}(g_i)\}.$$

Figure 4.2.. Transformation scheme $\mathscr{R}_c$ for rewriting guards as explicit `guard` and `witness` operations.

4.3. Formal Definition of LRec$_C$ with Guards

To formally describe the extensions to LREC$_C$ required by the lowering scheme $\mathscr{L}_c$, *i.e.*, to define the `guard` and `witness` operations, I in this section provide a definition of their syntax and semantics. These definitions are to be read as extensions to the syntax and semantics of LREC$_C$ without guards as described in Sections 3.2.4 and 3.3, respectively.

First, I give a definition of the extended syntax of LREC$_C$ with lowered guards that captures the additional two operations `guard` and `witness` and the modified syntax for instances in Figure 4.3 on the following page. As can be seen, the production rule *expression* now contains two more choices.

The first, the `guard` operation, is defined by the corresponding production rule *guard.* A `guard` operation syntactically consists of the keyword `guard` followed by two expressions in parentheses. The syntax of the second addition, *i.e.*, the `witness` operation, is defined by the production rule *witness.* A `witness` operation is expressed syntactically by the keyword `witness`, followed by two or more expressions surrounded by parentheses.

Furthermore, I have redefined the production rule *instance* in Figure 4.3 on the next page. As all guard expressions are rewritten to an explicit encoding using `guard` and `witness` operations, a function instance may no longer contain pattern guards.

Lastly, before I close the discussion of LREC$_C$, I fill the semantic gap left in the previous chapter by formally defining the semantics of the `guard` and `witness` operations. The corresponding rules are given in Figure 4.4 on the following page.

The first rule, *i.e.*, rule GUARD, defines the semantics of the `guard` operation. A `guard` operation is evaluated to the value of the first argument, if that argument can be evaluated and if the second argument can be evaluated to the Boolean value *true.* Similarly, the rule WITNESS defines the semantics for the `witness` operation. Here, however, only if all arguments apart from the first can be evaluated to the Boolean value *true*, the entire expression can be evaluated to the value of the first argument, if such value exists.

expression	$\Rightarrow$	... | **guard** | **witness**
guard	$\Rightarrow$	`guard` (**expression** **expression**)
witness	$\Rightarrow$	`witness` (**expression** *[* **expression** *]*$^+$)
instance	$\Rightarrow$	*[* **pattern** *]*$^+$ `=` **expression**

Figure 4.3.. Extension of the syntax of LREC$_C$ as presented in Figure 3.6 on page 50 by lowered guards.

$$\textsc{Guard} \;:\; \frac{(\mathscr{F}, \prec, \mathscr{E}) : e_g \Downarrow \mathit{true} \quad (\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{guard(}e\ \ e_g\texttt{)} \Downarrow v}$$

$$\textsc{Witness} \;:\; \frac{\forall i \in \{1, \ldots, n\} : (\mathscr{F}, \prec, \mathscr{E}) \ :\ e_i \Downarrow \mathit{true} \quad (\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{witness(}e\ \ e_1\ \cdots\ e_n\texttt{)} \Downarrow v}$$

Figure 4.4.. Extension of the operational semantics of LREC$_C$ given in Figure 3.8 on page 52 by guards.

With these extensions, the operational semantics for LREC$_C$ is complete. As with all other lowering steps, the semantics of the pattern guards in LREC are defined by the semantics of their corresponding lowered counterparts using `guard` and `witness` operations in LREC$_C$.

4.4. Conclusions

In this chapter, I have motivated the implementation of guards in LREC by an explicit dataflow encoding as part of the function body. As an experiment made in the context of the language SAC shows, such an encoding can be added to an existing language with relatively little effort.

Furthermore, the chosen encoding offers three advantages:

- Guard expressions can easily be discriminated from regular program code. In particular, those checks that cannot be optimised out are clearly visible and thus allow the programmer to further enrich a program by static knowledge to increase the static correctness of that program, if desired.

- The chosen encoding eases program analysis. This allows for efficient runtime code to be generated by eliminating duplicate guards using standard optimisations and symbolic analyses.

- Optimisations need not to be made aware of guards. The encoding is designed such that it does not invalidate assumptions made by existing optimisations.

I use a special `guard` operation to tie the check of guard expressions into the dataflow. This ensures that each guard expression has been checked before the corresponding data is used the first time. Furthermore, by using explicit evidence in form of the `witness` operation, I ensure that each guard expression is checked, regardless of whether the corresponding data is required to compute the result.

This completes the description of LREC$_\text{C}$. In the next chapter, I will discuss the partial evaluation of expressions in LREC$_\text{C}$.

5. Partial Evaluation

In this chapter I discuss the inference of static knowledge by partial evaluation of programs in LRec. First, I provide some background on the basic concepts of partial evaluation. In the next Section, I then revisit the examples presented in Chapter 2 and demonstrate how to infer static properties by evaluating programs to partial results. In particular, I motivate the use of a two-step process that first infers the information required to compute a partial result and then performs the actual partial evaluation. Section 5.3 formalises the second step by providing a semantics for partial evaluation of expressions in LRec$_C$. An in-depth discussion of the first step is given in the next chapter.

5.1. Some Background

Partial evaluation of a program or an expression usually refers to a source-to-source transformation that rewrites the program or expression to a specialised form. As an example, consider the following expression in LRec:

```
1  let
     val A = 1
3    val B = 2
   in
5    (A = B)
   end
```

The above expression can be rewritten in many ways without changing its value. For instance, I can propagate the values of `A` or `B` into the expression `(A = B)` in Line 5, yielding the following new overall expression:

```
   let
2    val A = 1
     val B = 2
4  in
     (1 = 2)
6  end
```

In the example above, I have performed one reduction step as defined in the evaluation rules in Chapter 3. Thus, to evaluate the above expression, less computation is required. In this sense, partial evaluation is an optimisation technique. I can, of course, reduce the above example further by rewriting the body of the `let` construct to `false` and ultimately replacing the entire `let` construct with the expression `false`. Such a result of partial evaluation is commonly referred to as the *residue* of an expression.

In the above example, partial evaluation is a simple top-down, single-pass process. This style of partial evaluation is referred to as *on-line*, as the decision what to evaluate is made whilst the program is rewritten. However, effectively deciding what parts to rewrite is not always possible. As an example consider the following expression:

```
  let
2   fun snd X{} Y{} = Y
    val A = 1
4   val C = 2
  in
6   ((snd B A) = C)
  end
```

To rewrite the body expression in Line 6, I would have to inline the definition of `snd` in Line 2 into the body expression or specialise the function `snd` for the special case where the second argument is known to be 1. However, it is not at all clear, whether either of these rewritings would allow me to further evaluate the comparison operation. In an on-line approach, I would have to speculatively perform the rewriting or I might miss the opportunity to partially evaluate the expression in Line 6.

Off-line partial evaluators circumvent this problem by staging partial evaluation in two phases. The first phase, referred to as *binding-time analysis*, abstractly evaluates the expression with respect to static knowledge: Given information on which free variables of an expression are statically known to be constant, the binding-time analysis computes which parts of the expression can be statically evaluated. In a second phase, the program is then rewritten using this information as a guide.

In the above example, the binding-time analysis would find that `A` and `C` are statically known and that, given that the second argument is known, the result of the function `snd` can be statically computed, as well. Thus, it is indeed beneficial to specialise `snd` or inline its definition into the expression in Line 6.

Even though an off-line approach allows a well-founded decision in the above example, using a two-step process with a binding-time analysis to drive partial evaluation does not always yield better results. A binding-time analysis can only infer a worst-case approximation of the static information that will be available during partial evaluation. As an example, consider the following expression:

```
if (1 = 1) 1 A
```

In the above example, the predicate and the then clause of the conditional are statically known, whereas the else clause is not. Without actually evaluating the predicate to *true*, an analysis cannot decide whether the result of the conditional will be statically known. Thus, a binding-time analysis would have to assume the worst case and treat the entire expression as statically unknown. An on-line approach, on the other hand, after rewriting the predicate, can directly derive that only the then clause is required.

The above discussion of partial evaluation only introduces key concepts. In its brevity, it can only give a very broad overview. For an in depth discussion, I refer the interested reader to the book by Jones et al. [1993].

Moreover, I use partial evaluation in a different context in this thesis. Instead of rewriting a program to a specialised residue, I use partial evaluation to compute partial results of a program. Thus, in the context of this thesis, partial evaluation is a rewriting from expressions to values as opposed to the usual source-to-source rewriting. However, this is merely a different way to present partial evaluation: The concepts remain the same and the approach developed here can be used in a setting that specialises programs, as well.

5.2. Partial Evaluation of Auxiliary Computations

The idea of auxiliary computations is to model additional properties of data in the programming language itself. This allows the programmer to exploit a rich set of properties in his algorithms. In Section 2, I have shown for the example of matrix addition, how to use additional properties to improve the runtime complexity of a program. For special kinds of matrices, *e.g.*, for the presented unit and lower diagonal matrices, adding specialised instances decreases the number of elements that need to be computed. The semantics presented in the previous chapter support this programming style. An interpreter or compiler that adheres to these semantics will use the implementation of matrix addition that has the best (available) runtime complexity.

The information encoded in auxiliary computations can be exploited further. For instance, in the evaluator example in Section 2.4, I have used auxiliary computations to model the kind of an expression. Using the semantics presented in the previous section, invalid expressions will already be rejected at construction time of the expression during the evaluation of the program.

So far, all these computations are performed at runtime. As a next step, I in this chapter discuss how to compute some or all auxiliary computations statically, *i.e.*, at compile time or in case of an interpreter at least before computing the actual value.

Computing the auxiliary computations without computing the actual value result of a program has two main benefits. Firstly, it allows the programmer to check whether an auxiliary computation can actually be computed, *i.e.*, whether the program is correct with respect to that property. Checking auxiliary computations in isolation is usually faster than evaluating the entire program. For instance, in the setting of array languages, computing only the rank and shape properties of arrays is in most cases less complex than performing the actual computation of the result.

Secondly, evaluating auxiliary computations separately allows the programmer to derive general properties of an algorithm, independently from the actual values it is applied to. Looking at array languages again, in many cases it suffices to know the shape and rank of the program inputs to compute the shape and rank of the results. For the example of matrix addition this is trivially the case. In general, only if the shape and rank depend on the value of the inputs, such a partial evaluation on only auxiliary computations will fail. In these cases, by evaluating the auxiliary computations and only those actual values that are required, it is still possible to derive the shape and rank for a given set of inputs.

These benefits apply to the example using algebraic data types, as well. By evaluating the expression kind property separately, malformed expressions, *i.e.*, those expressions that cannot be evaluated to a value, can be detected even before constructing the actual data or evaluating the program.

As an example, I return to the simple matrix addition presented first in Chapter 2:

```
1  let
     fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
3      = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
     val A        = [ 1, 2, 3, 4]{ shape=[ 2, 2], rank=2}
5    val B        = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
   in
7    (add A B)
   end
```

If, for example, in the application of the function `add` in Line 7 above we are only interested in the shape of the result, *i.e.*, the `shape` auxiliary computation, it suffices to compute only that element of the result record. In the above example, this is straight forward. As the function `add` has only one instance, it suffices to evaluate the `shape` component of the record in the defining expression of that instance. This component is defined in Line 3 above as the expression `any(A.shape B.shape)`. Thus, to compute the shape of the result, we need to compute the `shape` component of either the record corresponding to argument `A` or the one corresponding to argument `B`. As both are defined as `[ 2, 2]` in lines 4 and 5, respectively, the shape of the result can be computed by evaluating the expression `[ 2, 2]`. The same approach can be applied to compute the rank of the result.

Partial evaluation becomes more complex in the presence of multiple instances. As an example, reconsider the definition of `add` with support for lower diagonal matrices as presented first in Section 2.2. There, addition on these kinds of matrices is defined as follows:

```
   let
2    fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
4            A{ shape=sA, rank=rA, ldiag}
             B{ shape=sB, rank=rB, ldiag}
6      = (ldiag_add( any(sA sB) !A !B){ shape=any(sA sB),
                                        rank=any(rA rB),
8                                       ldiag}
     val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, ldiag}
10   val B = [ 9, 0, 8, 7]{ shape=[ 2, 2], rank=2, ldiag}
   in
12   (add A B)
   end
```

In Line 4 above, I have defined a second instance for adding two matrices that each carry the additional `ldiag` tag. Corresponding matrices carrying this tag are defined in lines 9 and 10. Each is a 2×2 lower diagonal matrix. Finally, in Line 12 above, the function `add` is applied to both matrices.

One interesting question to ask in the example above is whether the result is a lower diagonal matrix, *i.e.*, whether the result carries the `ldiag` property. By looking at the code, it is immediately clear that it does. As both arguments to the application of `add` carry the `ldiag` property, the instance in Line 10 matches. Thus, the result of the addition computed in Line 6 contains a `ldiag` label.

Actually computing this partial result is more complex though. The partial values for `A` and `B` need to contain sufficient information to decide the pattern match in the application of `add`. To decide the pattern match in the above example, we need to know that both arguments carry the `ldiag` property. In this case, only the second instance can match. Thus, we can compute the `ldiag` component of the result of the application of the function `add`. Otherwise, the first instance will match and the result does not carry the `ldiag` property.

It is worth noting here that the `ldiag` property only suffices to decide that only one instance can match. The `ldiag` property alone does not suffice to decide whether the instance actually will match when the program is fully evaluated. To decide this, we would additionally need to know that the arguments both carry the `shape` and `rank` properties. Deriving this information would require to evaluate all arguments at least up to their domain.

However, even if the domain of each argument is known, it is still not guaranteed that the full result will carry the `ldiag` property. The evaluation might get stuck due to any of the other auxiliary computations or during the evaluation of the value component of the result. Thus, in general, partial evaluation will only yield results that are conditional on the full evaluation to succeed. In this setting, identifying the only instance that can match suffices. If the full evaluation succeeds, the only matching instance indeed does match. Thus, the partial result will be correct.

As a further example, which makes use of pattern guards, reconsider the data constructors for the data type `Expr` presented in Section 2.4. I repeat their definition below:

```
let
  fun ENum I{} = (!I){ Expr, ENum, Kind=Int}
  fun EBool B{} = (!B){ Expr, EBool, Kind=Bool}
  fun ECond P{ Expr, Kind=kp} T{Expr, Kind=kt} E{Expr, Kind=ke}
    | (kp != Int)
    = (P, T, E){ Expr, ECond, Kind=if (kt = ke) any(kt ke) Any}
  fun EIsZero E{ Expr, Kind=k}
    | (k != Bool)
    = (E){ Expr, EIsZero, Kind=Bool}
  fun EDiv A{ Expr, Kind=ka} B{ Expr, Kind=kb}
    | (ka != Bool) (kb != Bool)
    = (A, B){ Expr, EDiv, Kind=Int}
in
  (EIsZero (ECond (EBool true) (ENum 0) (EBool false)))
end
```

For the algebraic data type defined above, an interesting property is whether the expression in Line 14 has kind `Int`, `Bool` or `Any`. To derive this information, I can partially evaluate the expression with respect to this auxiliary computation. For this example, the pattern match is uniquely defined, as all functions only have a single instance. Thus, I concentrate on deriving which sub-expressions need to be evaluated in order to compute the kind of the overall expression.

I start out with the application of `ECond`. The kind of the result depends on the kind of the then and else clause of the conditional, *i.e.*, it depends on the two arguments `T` and `E`. Thus, I need to partially evaluate the second and third parameter up to their `Kind` property. As the first argument is not involved in the computation of the `Kind` property of the result, I do not need to evaluate it at all.

For the second argument, I need to partially evaluate the expression `(ENum 0)`. In this case, the result always carries the value `Int` for the property `Kind`. Thus, I do not need to evaluate its argument to compute the required partial result.

Finally, the third argument is computed by a further application of `EBool`, this time to the argument `false`. As for the second argument, I do not need to evaluate this argument to compute the required partial result.

Thus, to compute the `Kind` property of the overall expression, I need not to compute the actual value component of the arguments to the `ECond` data constructor. Nor do I need to compute the arguments to the applications of `EBool` and `ENum`.

Using this information, I can now compute the value of the `Kind` component of the result of `ECond`. The kind of the result is computed using the expression `if (kt = ke) any(kt ke) Any`. As the kinds of the then and else branches differ, this expression evaluates to *Any*. Thus, the overall expression has kind `Any`, as well.

Another property one could be interested in is the well-formedness of an expression. In particular, an expression where the `Kind` properties do not match should be rejected. As an example, consider the following ill-formed expression

```
(EIsZero (EBool true))
```

with the data constructors `EIsZero` and `EBool` as defined above. To partially evaluate the application of `EIsZero` with respect to the `Kind` property, we first need to know which properties of the arguments of `EIsZero` need to be computed. As the definition of `EIsZero` in lines 7ff. shows, the `Kind` property always has the value `Bool`. Thus, to compute the `Kind` property of the above expression, no information is needed about the argument. Nonetheless, the above expression would fail under full evaluation: The `Kind` property of the argument has the value `Bool`. Therefore, the guard in Line 8 would fail during evaluation. To catch this during partial evaluation, as well, the guard expression needs to be evaluated. However, the guard expression requires that the `Kind` property of the argument has any value but `Bool`. Hence, to decide whether the function will evaluate and thus yield the `Kind` property `Bool`, we need to evaluate the `Kind` property of the argument. For the result of the function `EBool`, this property always has the value `Bool`. Accordingly, the argument to `EBool` needs not to be evaluated at all.

Using this information, we can now compute the required partial values. However, as the `Kind` property of the argument to `EIsZero` is `Bool`, this evaluation will fail, as the pattern guard evaluates to *false*. Therefore, the expression will be rejected.

In the examples above, inferring static knowledge for a given program in LRec is a two-step process. First, I have inferred for each sub-expression which properties of the corresponding value are required to compute a desired property of the overall result. This includes the values of components that are required, *e.g.*, the value of the `shape` component in the first example. Furthermore, I have derived for which properties their existence in a value is required to decide the pattern match in function applications. Given this information, as a second step, I have then partially evaluated the program accordingly.

In the remainder of this chapter and the next chapter, I will formalise this approach. First, the next section presents a formal semantics for partially evaluating expressions in LRec$_\mathrm{C}$ to partial values. Next, after some conclusions for this chapter, Chapter 6 presents an analysis that infers for each sub-expression the set of properties that need to be evaluated to compute a desired partial result.

5.3. A Partial Semantics of LRec$_\mathrm{C}$

In this section, I provide an operational semantics for partial evaluation of expressions in LRec$_\mathrm{C}$. The semantics for partial evaluation of the syntactic sugar of LRec that is missing in LRec$_\mathrm{C}$ is defined as the semantics of the corresponding de-sugared expressions as defined in Section 3.2.

Similarly to the semantics of LRec$_\mathrm{C}$, I define the partial semantics using a relation $\downarrow$ between expressions in LRec$_\mathrm{C}$ and partial values. The relation $\downarrow$ is not a function as an expression can potentially be evaluated to more than one partial result. To begin with, I first provide a formal definition of the set of partial values. The corresponding production rules in extended Backus Naur form are given in Figure 5.1.

As the examples in the previous section have shown, a partial value needs to encode two kinds of information. Firstly, the values for a subset of the range of the corresponding full record value. As I use sets of (label,value) pairs to encode record values, this information could already be encoded using the set of record values $\mathscr{R}$ as defined for full values in Figure 3.7 on page 51. All that is required is to leave out some of the (label,value) pairs.

value	$\Rightarrow$	**boolean** | **record** | `?` | `!`
boolean	$\Rightarrow$	`true` | `false`
record	$\Rightarrow$	`{` *[* **element** *[* `,` **element** *]** *]* `}`
element	$\Rightarrow$	`(` **label** `,` **value** `)`

Figure 5.1.. Set of legal partial values used as range of $\downarrow$.

However, secondly, partial values need to encode the presence or absence of labels within a full result. To cater for this information, I have introduced two special values in the production rule *value* in Figure 5.1 on the previous page. The value *?* encodes that the corresponding full value has some value or, if used in a record element, that the corresponding label is in the domain of that record. Dually, the value *!* encodes that a value does not exist. If used in a record element, *!* encodes that the corresponding label is not in the domain of the full value.

In the following, I will refer to the set of partial values that can be produced using the production rule *value* in Figure 5.1 on the preceding page as $\mathscr{V}_p$. The set $\mathscr{R}_p$ contains all partial record values, *i.e.*, those values that can be produced using the rule *record*. As in Section 3.3, $\mathscr{L}$ is the set of all labels. This set is identical for both full and partial values.

As a prerequisite to the discussion of the semantics of partial evaluation below, I first introduce some functions on partial values. Similar to the domain of a record value dom I define the domain of a partial record value dom_p as follows.

Definition 5.3.1 (Domain of a Partial Record). *The function* $dom_p : \mathscr{R}_p \to \mathscr{L}$ *is defined as*

$$dom_p(R) := \{l \mid \exists v \in \mathscr{V}_p \setminus \{!\} : (l, v) \in R\}.$$

Thus, the domain of a partial record value is the set of all labels contained in the partial record that are not marked as absent with respect to the corresponding full record value by the special *!* value.

Next, I define the anti-domain of a partial record value $\overline{\text{dom}}_p$. For full record values, to decide whether a label is not in their domain, it suffices to check whether the label is not contained in the set computed by the corresponding domain function dom. However, for the partial domain dom_p, if a label is not contained in the corresponding set this only means that it is not necessarily contained in the corresponding full value. To decide whether a label is guaranteed not to be in the full value, we have to inspect the labels within the partial record value that are explicitly excluded using the *!* value.

Definition 5.3.2 (Anti-Domain of a Partial Record). *The function* $\overline{dom}_p : \mathscr{V}_p \to \mathscr{L}$ *is defined as*

$$\overline{dom}_p(R) := \{l \mid (l, !) \in R\}.$$

Furthermore, I define the range of a partial record range_p as all partial values contained in a given record.

Definition 5.3.3 (Range of a Partial Record). *The function* $range_p : \mathscr{V}_p \to \mathscr{V}_p$ *is defined as*

$$range_p(R) := \{v \mid \exists l \in \mathscr{L} : (l, v) \in R\}.$$

It is worth noting here that the range includes the special values *?* and *!*, as well.

As a last function on partial record values, I define the counterpart to the elem function on full record values for partial record values.

Definition 5.3.4 (Element of a Partial Record)**.** *The function $elem_p : \mathscr{R}_p \times \mathscr{L} \to \mathscr{V}_p$ is defined as*

$$elem_p(R, l) := \begin{cases} v & \text{if } (l, v) \in R \wedge v \neq \,!, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Thus, $\text{elem}_p(R, l)$ yields the value bound to the label l in R if such a value exists and it is not the special *!* value. Note, however, that the special value *?* is considered an element of a record. The intuition is that *?* encodes the existence of a value and thus selecting such value should be possible whereas *!* encodes that the label is not contained in R and therefore the selection should fail.

Figure 5.2 on the next page shows a big-step operational semantics for partial evaluation of LRec_C. I use the same notation as for the semantics for full evaluation presented in Section 3.3. The statement $(\mathscr{F}, \prec, \mathscr{E}) : e \downarrow v$ is to be read as: Given an environment $(\mathscr{F}, \prec, \mathscr{E})$, the term e can be evaluated to the partial value v. The evaluation environment has the same structure as in the definition of full evaluation. $\mathscr{F}$ denotes the function environment, $\prec$ is the partial order defined by the `rel` construct of LRec and $\mathscr{E}$ is the environment for variable bindings.

The rules that are concerned with evaluating non-record values and expressions thereon, *i.e.*, the rules True, False, EqualTrue and EqualFalse, remain unchanged compared to the corresponding rules for full evaluation. This is a general property of all applied extensions of LRec. The partial evaluation presented here is only partial with respect to record values. Expressions that compute on non-record values and yield non-record values are fully evaluated under both full and partial evaluation.

For the special ~ expression, I have modified the corresponding rule Unit to reflect the change in meaning of an empty record value. In full evaluation, the empty record value $\emptyset$ represents a record value that has no labels. In the case of a partial record value, $\emptyset$ represents the record value where no information about its domain is known. To encode that a partial record value does not contain any labels, we would have to make this explicit by associating each valid label with the special value *!*. However, this would require a potentially infinite set to represent the value of the ~ expression. Fortunately, we do not need to know that the value of the `unit` expression is an empty record.

The motivation to represent `unit` as the empty record was to ensure that selections on its value would fail and that for all pattern of a program in LRec the corresponding match would fail, as well. The former is already ensured by using the empty set as partial value, as only those elements from a partial record can be selected whose labels are within the domain of that record. For the latter, it suffices to ensure that enough knowledge is encoded in the partial value for `unit` to make a particular match fail. I postpone a discussion of which labels these are to Chapter 6. For the definition of the semantics of partial evaluation, I allow as result for the ~ expression all partial record values R whose domain is empty, expressed by $R \subseteq \mathscr{L} \times \{!\}$ in rule Unit.

The next rule in Figure 5.2 on the following page, rule Var, remains unchanged compared to the semantics of full evaluation. An identifier can be evaluated to a value if a corresponding (identifier,value) pair is contained in the variable environment $\mathscr{E}$. This look-up operation can of course yield a partial result if the expression in the correspond-

$$\textsc{True} \quad : \quad \frac{}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{true} \downarrow \mathit{true}}$$

$$\textsc{False} \quad : \quad \frac{}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{false} \downarrow \mathit{false}}$$

$$\textsc{Unit} \quad : \quad \frac{R \subseteq \mathscr{L} \times \{\mathit{!}\}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{\textasciitilde} \downarrow R}$$

$$\textsc{Var} \quad : \quad \frac{(i, v) \in \mathscr{E}}{(\mathscr{F},\prec,\mathscr{E}) : i \downarrow v}$$

$$\textsc{EqualTrue} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_1 \downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \downarrow v_2 \quad v_1, v_2 \notin \mathscr{R}_p \quad v_1 \overset{v}{=} v_2}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1\ \texttt{=}\ e_2\texttt{)} \downarrow \mathit{true}}$$

$$\textsc{EqualFalse} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_1 \downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \downarrow v_2 \quad v_1, v_2 \notin \mathscr{R}_p \quad v_1 \overset{v}{\neq} v_2}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1\ \texttt{=}\ e_2\texttt{)} \downarrow \mathit{false}}$$

$$\textsc{Record} \quad : \quad \frac{\begin{array}{c}\forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow l_i \neq l_j \\ \forall i \in \{i_0, \ldots, i_m\} \subseteq \{1, \ldots, n\} \ : \ (\mathscr{F},\prec,\mathscr{E}) : e_i \downarrow v_i \\ R \subseteq (\mathscr{L} \setminus \{l_1, \ldots, l_n\}) \times \{\mathit{!}\}\end{array}}{\begin{array}{c}(\mathscr{F},\prec,\mathscr{E}) : \texttt{\{}l_1\texttt{=}e_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n\texttt{\}} \\ \downarrow \{(l_{i_0}, v_{i_0}), \ldots, (l_{i_m}, v_{i_m})\} \cup R\end{array}}$$

$$\textsc{Selection} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e \downarrow v \quad v \in \mathscr{R}_p \quad l \in \mathrm{range}_p(v)}{(\mathscr{F},\prec,\mathscr{E}) : e\texttt{.}l \downarrow \mathrm{elem}_p(v, l)}$$

$$\textsc{Any} \quad : \quad \frac{\exists i \in \{1, \ldots, n\} \ : \ (\mathscr{F},\prec,\mathscr{E}) : e_i \downarrow v}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{any(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \downarrow v}$$

$$\textsc{CondThen} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_p \downarrow \mathit{true} \quad (\mathscr{F},\prec,\mathscr{E}) : e_t \downarrow v}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{if}\ e_p\ e_t\ e_e \downarrow v}$$

$$\textsc{CondElse} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_p \downarrow \mathit{false} \quad (\mathscr{F},\prec,\mathscr{E}) : e_e \downarrow v}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{if}\ e_p\ e_t\ e_e \downarrow v}$$

$$\textsc{Guard} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_g \downarrow \mathit{true} \quad (\mathscr{F},\prec,\mathscr{E}) : e \downarrow v}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{guard(}e\ e_g\texttt{)} \downarrow v}$$

$$\textsc{Witness} \quad : \quad \frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} : (\mathscr{F},\prec,\mathscr{E}) \ : \ e_i \downarrow \mathit{true} \\ (\mathscr{F},\prec,\mathscr{E}) : e \downarrow v\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{witness(}e\ e_1\ \cdots\ e_n\texttt{)} \downarrow v}$$

Figure 5.2.. An operational semantics for partial evaluation of LREC$_{\text{C}}$.

$$\textsc{Let} \quad : \quad \frac{(\mathscr{F}', \prec', \mathscr{E}') : e \downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \ \texttt{let}\ d_1 \cdots d_n\ \texttt{in}\ e\ \texttt{end} \downarrow v}$$

$$\text{where} \quad \begin{array}{rcl} \prec' & = & \mathrm{rel}(\prec, \{d_1, \ldots, d_n\}) \\ \mathscr{F}' & = & \mathrm{fun}(\mathscr{F}, \prec', \{d_1, \ldots, d_n\}) \\ \mathscr{E}' & = & \mathrm{val}^p(\mathscr{F}', \prec', \mathscr{E}, (d_1, \ldots, d_n)) \end{array}$$

$$\textsc{Ap} \quad : \quad \frac{\begin{array}{c} \forall i \in \{1, \ldots, n\} \ : \ (\mathscr{F}, \prec, \mathscr{E}) \ : \ e_i \downarrow v_i \in \mathscr{R} \\ \{(\mathscr{F}', \prec', \mathscr{E}', e_b)\} = \mathrm{match}^p(\mathscr{F}, f, (v_1, \ldots, v_n)) \\ (\mathscr{F}', \prec', \mathscr{E}') : e_b \downarrow v \end{array}}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{(}f \ \ e_1 \ \ldots \ e_n\texttt{)} \downarrow v}$$

$$\textsc{Partial} \quad : \quad \frac{}{(\mathscr{F}, \prec, \mathscr{E}) : e \downarrow \mathord{?}}$$

Figure 5.2.. An operational semantics for partial evaluation of LRec$_C$ (contd.).

ing binding operation, *i.e.*, a use of the `val` construct or a function application, has been evaluated only partially.

Partial evaluation of record values is expressed by rule Record. I allow a record to be evaluated with respect to certain labels only. If for a subset of the labels that occur in the record expression the corresponding expressions can be evaluated to partial values, the entire record can be evaluated to a partial record value whose range only includes this subset of the labels contained in the record expression. Furthermore, as in the rule for the ~ expression, I allow any anti-domain R to be attributed to the partial record value as long as it does not contain any labels that are part of the record expression itself. Again, the choice of the anti-domain only impacts pattern matching on this record but has no impact on the actual value as such.

The next rule in Figure 5.2 on the preceding page defines the semantics for selection. It is the direct translation of the rule Selection for full evaluation to the domain of partial values. A selection expression can be evaluated to a partial value if the expression argument can be evaluated to a partial record value whose domain contains the label to be selected. Thus, a selection can yield any partial value apart from *!*.

The next three rules, *i.e.*, Any, CondThen and CondElse remain unchanged, as well. Partial evaluation for these constructs has the same semantics as in the full evaluation case. In particular, the conditional still requires the value of the predicate to be known. This is a design choice. Instead, I could have chosen to introduce a third rule of the form

$$\textsc{CondAny} \quad : \quad \frac{(\mathscr{F}, \prec, \mathscr{E}) : e_t \downarrow v \quad (\mathscr{F}, \prec, \mathscr{E}) : e_e \downarrow v}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{if}\ e_p\ \ e_t\ \ e_e \downarrow v}$$

Using the above rule, a conditional can be evaluated to a partial value if both branches evaluate to the same partial value. This rule corresponds more closely to the usual type derivation rule for conditionals [Pierce, 2002]. In type systems, the type of a conditional

$$\mathrm{val}^p(\mathscr{F}, \prec, \mathscr{E}, (d_1, \dots, d_n)) := \mathscr{E}_n$$

where

$$\begin{aligned}
\mathscr{E}_0 &:= \mathscr{E} \\
\mathscr{E}_{i+1} &:= \begin{cases}
\mathscr{E}_i \leftarrow (l, v) & \begin{array}{l}\text{if } d_{i+1} \equiv \texttt{val}\ l\texttt{=}e \\ \text{and } (\mathscr{F}, \prec, \mathscr{E}_i) : e \downarrow v,\end{array} \\
\mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{fun}\ f\ i_1\ \cdots\ i_m, \\
\mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{rel}\ l_1\ \texttt{<:}\ \ l_2, \\
\text{undefined} & \text{otherwise.}
\end{cases}
\end{aligned}$$

Figure 5.3.. Definition of function *val^p* as used in Figure 5.2 on page 86.

is the type of both branches, if such a common type exists, *i.e.*, both have the same type or, in the setting of subtyping, a least upper bound can be found. However, using this rule alone would impact the expressiveness of my approach. It would require that, in order to compute the value of a property of a conditional expression, both branches evaluate to partial record values that carry the same value for the corresponding label. Using all three rules, on the other hand, would make an implementation of the above semantics more difficult. It is not clear which rule to use when. For example, if both branches evaluate to different partial values, rule CondAny cannot be used. To decide this, however, first both branches need to be evaluated. Such an evaluation might not terminate. For instance, in case of a termination conditional of a recursive function, one branch is an unbound recursion.

The next two rules define the semantics for the operations that encode guards, *i.e.*, the semantics for the `guard` and `witness` operations. Like the rules for conditionals, the rules for guards remain unchanged: In order to evaluate either guard operation, the corresponding guard expressions need to evaluate to *true*. The idea here is that even a partial value can only be correct if the corresponding guard expressions hold.

Next in Figure 3.8 on page 52 is the rule Let for the `let` construct of LRec$_\mathrm{C}$. It remains largely unchanged, as well. The only difference is the use of the function *val^p* to process value bindings. Its definition is given in Figure 5.3. Overall, the function *val^p* has the same structure as its counterpart for full evaluation *val* (cf. Figure 3.11 on page 58): The new variable environment is computed by processing all definitions top down. To cater for partial evaluation, instead of evaluating expressions fully using $\Downarrow$, I use $\downarrow$ to only partially evaluate the expression before binding the result to an identifier by adding the corresponding partial value to the new variable environment. Thus, the resulting environment contains the same bindings as the corresponding environment under full evaluation. The only difference is that the bound values are partial values.

The penultimate rule in Figure 5.2 on page 86 defines the semantics of function applications. It only differs in the use of a different pattern matching function *match$_p$*.

I have chosen to allow only those partial matches that yield a single instance. As with conditionals, this is a design choice. A further possibility would be to allow multiple matching instances if all can be evaluated to the same partial value. A corresponding rule would have the form

$$\textsc{Ap} \ : \ \frac{\begin{array}{c}\forall i \in \{1,\dots,n\} \ : \ (\mathscr{F},\prec,\mathscr{E}) \ : \ e_i \downarrow v_i \in \mathscr{R} \\ \forall(\mathscr{F}',\prec',\mathscr{E}',e_b) \in \text{match}^p(\mathscr{F},f,(v_1,\dots,v_n)) : \\ (\mathscr{F}',\prec',\mathscr{E}') : e_b \downarrow v\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : (f \ e_1 \ \dots \ e_n) \downarrow v}$$

However, the arguments given for conditionals apply here, as well. By using an instance that might not be used during full evaluation, the termination behaviour under partial evaluation compared to full evaluation might change. In particular, partial evaluation might diverge where full evaluation does not.

Before I discuss this pattern matching process in detail, I first complete the discussion of the semantic rules with the last remaining rule. The rule PARTIAL caters for partial evaluation of general expressions. Any expression can be evaluated to the special value *?*, which denotes that a corresponding value might exist. This rule, in combination with the rule RECORD for record expressions, allows for evaluating a record expression such that it is known that a certain label is in the domain without needing to evaluate the corresponding value.

To complete the discussion of the partial semantics of LREC$_C$, the last part missing is the partial pattern matching function *match^p*. Its definition is given in Figure 5.4 on the next page. As in the semantics for full evaluation, the pattern matching process consists of five stages. The first two stages, *i.e.*, *lookup* and *arity*, are identical to the full pattern matching process as defined in Figure 3.12 on page 60. The former extracts the function instances corresponding to a given function name and the latter selects those instances whose arity matches the number of arguments. As partial evaluation does not impact function names nor the number of arguments for a function application, these two need not to be adapted to the different meaning of partial values.

The next two stages, *i.e.*, *pattern^p* and *order^p*, filter out non-matching instances and those matching instances that are shadowed with respect to the partial order on labels $\prec$, respectively. As both depend on the domain of their arguments, I have adapted them to the setting of partial values.

In the definition of the full pattern matching *match*, the function *pattern* removes all instances that contain labels that are not included in the domain of the argument values. In the setting of partial values, this would rule out instances that could still match if the arguments would be evaluated further. As an example, consider the following code fragment:

```
1 let
    fun foo A{} = ···
3           A{ X} = ···
  in
5   (foo { X=true})
  end
```

In the above code, I define two instances for the function `foo`. The instance in Line 2 matches any argument pattern, whereas the second instance in Line 3 requires at least a label `X` to be present. I then apply the function `foo` to an argument carrying the label `X`

$$\text{match}^p := \text{bind} \circ \text{order}^p \circ \text{pattern}^p \circ \text{arity} \circ \text{lookup}$$

with *lookup*, *arity* and *bind* as defined in Figure 3.12 on page 60. The function pattern^p is defined as

$$\text{pattern}^p(\mathscr{F}, \prec, I, (a_1, \ldots, a_n)) := (\mathscr{F}, \prec, I_n, (a_1, \ldots, a_n))$$

where

$$\begin{aligned} I_0 &:= I \\ I_{i+1} &:= \text{filter}_2^p(\text{filter}_1^p(I_i, (a_1, \ldots, a_n), i), i) \end{aligned}$$

with

$$\text{filter}_1^p(I, a, i) := \{(((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I \mid p_i \cap \overline{\text{dom}}_p(\text{a}) = \emptyset\}$$

and

$$\text{filter}_2^p(\prec, I, (a_1, \ldots, a_n), i) :=$$

$$\left\{ (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I \;\middle|\; \begin{array}{l} \forall(((\alpha'_1, p'_1), \ldots, (\alpha'_n, p'_n)), e') \in I : \\ \quad (\exists j \in \{1, \ldots, i\} \;: \\ \qquad p_j \setminus \text{dom}_p(a_j) \neq p'_j \setminus \text{dom}_p(a_j)) \vee \\ \quad |p_i \cap \text{dom}_p(a_i)| \geq |p'_i \cap \text{dom}_p(a_i)| \end{array} \right\}.$$

The function *order^p* is defined as

$$\text{order}^p(\mathscr{F}, \prec, I, (a_1, \ldots, a_n)) := (\mathscr{F}, \prec, I_n, (a_1, \ldots, a_n))$$

where

$$\begin{aligned} I_0 &:= I \\ I_{i+1} &:= \text{filter}'^p(\prec, I_i, (a_1, \ldots, a_n), i) \end{aligned}$$

with

$$\text{filter}'^p(\prec, I, (a_1, \ldots, a_n), i) :=$$

$$\left\{ (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I \;\middle|\; \begin{array}{l} \forall(((\alpha'_1, p'_1), \ldots, (\alpha'_n, p'_n)), e') \in I : \\ \quad (\exists j \in \{1, \ldots, n\} : \\ \qquad p_j \setminus \text{dom}_p(a_j) \neq p'_j \setminus \text{dom}_p(a_j)) \vee \\ \quad (\exists j \in \{1, \ldots, i-1\} : \\ \qquad \exists(((\alpha''_1, p''_1), \ldots, (\alpha''_n, p''_n)), e'') \in I : \\ \qquad\quad p'_j \overset{\rightarrow}{\prec} p''_j \wedge p_j \overset{\rightarrow}{\not\prec} p''_j) \vee \\ \quad (\exists j \in \{1, \ldots, i-1\} : \\ \qquad \exists(((\alpha''_1, p''_1), \ldots, (\alpha''_n, p''_n)), e'') \in I : \\ \qquad\quad p''_j \overset{\rightarrow}{\prec} p_j) \vee \\ \quad p_i \overset{\rightarrow}{\not\prec} p'_i \end{array} \right\}.$$

Figure 5.4.. Pattern matching function *match^p* for partial best match as used in Figure 5.2 on page 86.

in Line 5. I have left out the actual definition of the two instances, as it is not required for the example.

If fully evaluated, the second instance will be chosen as the argument matches the pattern of that instance best. Now consider a partial evaluation of the argument to a partial record value *{}*. The domain of that value is empty. Thus, if I would rule out all instances that contain labels that are not in the domain of the partial argument value, the first instance would be chosen. Ultimately, this might lead to a different result than under full evaluation. Even worse, if I decide to evaluate the argument further to the partial value *{(X,?)}*, the second instance would be chosen instead. Consequently, using domain based matching as in the full pattern matching process, the result of partial evaluation would depend on the grade of partiality of the function arguments. This is clearly not desirable.

To rule out only those instances that are guaranteed not to match, I instead remove all instances that contain labels that are included in the anti-domain $\overline{\mathrm{dom}}_p$ of the argument. As a reminder, the anti-domain of a partial value only contains those labels that are not included in the corresponding full value. This filtering step is performed by the helper function filter^p_1 in Figure 5.4 on the preceding page. In the above example, this would filter out neither instance. In general, the result of the filtering process contains at least those instances that the corresponding full matching process would yield.

Apart from filtering non-matching instances, the function *pattern* in the definition of full pattern matching only includes those matching instances with a maximal number of required labels. In the setting of partial values, this criterion does not suffice. By definition of the partial semantics of LRec, the anti-domain of a partial record value is not required to contain all labels that are not part of the corresponding value under full evaluation. Therefore, removing only those instances that do not match with respect to the anti-domain of a partial argument might leave instances that would not match the corresponding argument under full evaluation. If such an instance contains more labels than all other instances, it would be selected as the best match. Thus, the best match under partial evaluation might not include all instances that are selected as best match under full evaluation. This, ultimately, would again potentially yield different results under partial evaluation than under full evaluation.

A naïve approach to map the best match property to partial values would be to use the number of actually matched labels instead of the number of contained labels, *i.e.*, to maximise for a pattern p and argument a the value $|p \cap \mathrm{dom}_p(\mathrm{a})|$. However, this criterion does not suffice either. As an example consider the following code fragment:

```
  let
2   fun foo A{ X} = ···
            A{ Y, Z} = ···
4 in
    (foo {X=true, Y=true, Z=true})
6 end
```

Again, I have defined two instances for a function `foo`. The first in Line 2 matches arguments that contain at least the label `X`. For the second instance in Line 3, at least the labels `Y` and `Z` are required for a match.

Under full evaluation, in the expression in Line 5 the second instance would match, as it matches the most labels of the argument. If I, however, evaluate the record expression in Line 5 only partially to the partial record value *{(X,?)}*, using the above criterion for best match would choose the first instance.

The underlying problem is that for all labels that are neither in the domain, nor in the anti-domain of a partial record value, their influence on the matching process is not decidable. In the above example, the labels `Y` and `Z` are of this kind. Thus, it would be invalid to compare the two patterns, as it is not known how well the two labels in the second pattern match.

However, not everything is lost. As a further example, consider a third instance with an argument pattern `{X, Y, Z}`. Given the same partial value as above, *i.e.*, *{(X,?)}*, we still cannot decide whether the first or the new third instance matches better, as we do not know how well the two additional labels of the third pattern match. However, regardless of how well labels `Y` and `Z` match, the third instance always matches better as the second instance, if either of them matches at all. Thus, in this setting, it would be safe to remove the instance corresponding to the second pattern from the set of best matching instances.

In general, for instances with only one argument, it is safe to remove an instance if at least one other instance exists that contains the same set of labels with unknown state and that has more matching labels, *i.e.*, it has more labels that are in the domain of the partial record value. The set of labels with unknown state thereby is the set of all those labels that are neither in the domain nor in the anti-domain of the partial record value of the argument.

For multiple arguments, the situation becomes yet a little more complex. To show the problem, I give another example below.

```
  let
2   fun foo A{ Y} B{ X, Y} = ···
            A{}   B{ X} = ···
4 in
    (foo {} { X=true, Y=true})
6 end
```

Above, I define two instance for a function `foo` that both require two arguments. The first instance in Line 2 requires a label `A` for the first argument and labels `X` and `Y` for the second argument in order to match. For the second instance in Line 3, no labels are required for the first argument, whereas for the second argument the label `X` is required. Under full evaluation, in the expression in Line 5 the second instance will be chosen as only the second instance matches the first argument.

Now consider a partial evaluation of the two argument expression in Line 3 to the partial record values *{}* and *{(Y,?)}*. Using the matching process as described above, the filtering on the first argument will remove neither instance, as both still can match. However, the filtering on the second argument would remove the second instance, as, if either instance matches, the other instance will match as well and the first instance always matches better. Again, this would yield two different results under partial evaluation than under full evaluation.

The problem here is that I so far do not take into account that an instance that is guaranteed to match better might be removed under full evaluation in a previous filtering step. To make the filtering process work correctly in these circumstances, as well, I must only compare those instances where all pattern up to the one currently being filtered contain the same undetermined labels, *i.e.*, labels that are neither in the domain nor in the anti-domain of their corresponding argument.

This leads to the definition of the second stage of pattern filtering filter_2^p in Figure 5.4 on page 90. The restriction to only compare those instances that share a common set of undetermined labels is expressed by the condition $\exists j \in \{1, \ldots, i\} \ : \ p_j \setminus \mathrm{dom}_p(a_j) \neq p'_j \setminus \mathrm{dom}_p(a_j)$. If for any pattern previously compared or the pattern under comparison two instances have differing sets of undetermined labels, they are considered to match equally well. Otherwise, an instance is filtered if it contains less matching labels, *i.e.*, if $|p_i \cap \mathrm{dom}_p(a_i)| \geq |p'_i \cap \mathrm{dom}_p(a_i)|$ does not hold.

The next stage of the matching process is the removal of instances that are shadowed by another instance with respect to the partial order on labels $\prec$. For full pattern matching as defined in Figure 3.12 on page 60, this filtering step is performed regardless of the actual arguments. To decide whether an instance is shadowed by another instance, it suffices to compare the corresponding patterns using the partial order on patterns $\overset{\rightarrow}{\prec}$. However, in the setting of partial evaluation, the decision needs to take the partial nature of the arguments into account. As an example, consider the following program fragment:

```
  let
2   rel X <: Y
    fun foo A{ X, Z} = ···
4           A{ Y, Z} = ···
  in
6   (foo { X=true, Z=true})
  end
```

In Line 2 above, I declare that the label `Y` shadows the label `X` in pattern matching. Next, I define two instances for a function `foo` in lines 3 and 4. The first instance matches the labels `X` and `Z`, whereas the second instance matches the labels `Y` and `Z`. As the actual definition of the two instances is not of interest here, I have left it out in the code above. Finally, in Line 6 I apply the function `foo` to a record expression containing labels `X` and `Z`.

In the above example, only the first instance matches under full evaluation. Thus, the filtering of instances based on the order on labels $\prec$ does not impact the result set of instances. Now, if we assume a partial evaluation in the above setting that evaluates the argument of `foo` to the partial value *{(Z,?)}*, the set of matching instances after the third step of the matching process still contains both instances. In this case, using the order based filtering of the matching process under full evaluation would yield only the second instance, as the label `Y` shadows the label `X`. Thus, the set of matching instances under partial evaluation would not contain all instances that are contained in the set of matching instances under full evaluation.

The underlying cause in the above example is that I have removed an instance because of another instance that would not match under full evaluation. In this respect, order filtering suffers from the same problem as the best match filtering step. Therefore, to prevent cases like the one above, I use the same technique as in the third step of the matching process. I only compare those pattern whose unknown components are identical. Thus, if an instance leads to the exclusion of further instances, this exclusion is valid even if the instance is not part of the set of matching instances under full evaluation, as in both cases neither are the instances excluded by it.

In the context of functions with multiple arguments, the same restrictions as in the matching step *pattern* apply. Comparing two patterns is only valid, if for each pair of corresponding pattern the same labels have an unknown state. Note here, that for the *order* stage this is required for all pattern of the compared instances, as the order filtering happens after the pattern filtering stage and I thus have to ensure that I only compare instances where either both will be in the final result of the pattern filtering, or neither is contained in the result.

This restriction ensures that instances are not accidentally filtered out. However, it introduces a new problem. Now, at each step of the filtering process, instances that would be filtered in the full order filtering might remain in the set of matching instances. Thus, during consecutive filtering steps, these additional instances might trigger the filtering of instances that would otherwise have remained in the set of matching instances. As an example, consider the following code fragment:

```
1 let
    rel X <: Y
3   rel V <: W
    fun foo A{ V} B{ Y, W} = ···
5           A{ X} B{ X, W} = ···
            A{ W} B{ X, V} = ···
7 in
    (foo { X=true, V=true, W=true}
9        { X=true, Y=true, W=true, V=true})
  end
```

In the above example, I first declare in Line 2 that the label `Y` shadows the label `X` with respect to pattern matching. Furthermore, as declared in Line 3, the label `W` shadows the label `V`. Next, in lines 4 to 6 above, I define three instances of a function `foo`. The first matches the label `V` for the first argument and the labels `Y` and `W` in the second argument. For the second instance, the label `X` is required in the first argument and the labels `X` and `W` are required in the second argument. Lastly, for the third instance the first argument needs to carry the `W` label, whereas the second argument needs to carry the labels `X` and `V`. Finally, I apply the function `foo` to two arguments that carry the labels `X`, `V` and `W`, and the labels `X`, `Y`, `W` and `V`, respectively.

Under full evaluation, above the second instance would be chosen. All three instances match equally many labels for both arguments. However, as the label `W` shadows the label `V`, the first step of the order filtering process would filter out the first instance. In the second step of the order filtering process, the third instance is filtered out, again as the label `W` shadows the label `V`.

Now consider a partial evaluation of the arguments to the partial values *{(V,?),(X,?)}*

and *{(X,?),(Y,?)}*. In the pattern filtering step, all instances will remain as all instances partially match the arguments. In the first step of the *order* filtering process, no instance is filtered either. Even though the third instance shadows the first, the first is not filtered out as it cannot be decided whether the third instance would be in the set of instances under full evaluation. In the second step, however, the second instance would be filtered out, as it is shadowed by the first instance and both have the same set of labels with unknown state. Thus, under partial evaluation, the second instance, which would be chosen under full evaluation, is not in the set of matching instances.

To handle cases like the one above correctly, I add a second restriction on the order filtering process. I only compare two instances with respect to $\overrightarrow{\prec}$, if both are shadowed by the same instances in previous matching steps. This ensures that if such an undecided shadowing yields to the exclusion of the filtering instance, the filtered instance is excluded, as well. In the filtering process for the fourth stage in Figure 5.4 on page 90 this is expressed by the additional condition $\forall j \in \{1,\ldots,i-1\} : \forall(((\alpha''_1, p''_1),\ldots,(\alpha''_n, p''_n)), e'') \in I : p'_j \overrightarrow{\prec} p''_j \Rightarrow p_j \overrightarrow{\prec} p''_j$.

The above condition suffices to prevent filtering out the wrong instances for the example above. However, there are still cases where the above condition does not suffice. The problem arises, if an instance that would prevent the filtering by the above condition is itself filtered out. In this case, I lose the information than an instance might possibly be filtered out in previous steps. To prevent this, I add a last condition on order filtering: I only filter those instances that do not shadow any instances with respect to previously filtered pattern. In the definition of *filter'^{p}*, this is expressed by the condition $\forall j \in \{1,\ldots,i-1\} : \forall(((\alpha''_1, p''_1),\ldots,(\alpha''_n, p''_n)), e'') \in I : p''_j \overrightarrow{\not\prec} p_j)$.

The last step in the pattern matching process, the function *bind*, is again identical to the pattern matching process under full evaluation as defined in Figure 3.12 on page 60. Given a set of instances, it computes the corresponding environments required for the evaluation of the function bodies. These environments as such have not changed. However, during partial evaluation they of course can contain partial values if the arguments of a function have only been evaluated partially.

This closes the description of the partial semantics of LREC$_C$. Next, I will show the important property that partial evaluation as defined above indeed yields a partial result compared to full evaluation. To be able to formally show this, I first need to define what a partial result actually is.

Definition 5.3.5 (Partial Value)**.** *The relation $\sqsubseteq \subset \mathscr{V} \times \mathscr{V}_p$ is defined as*

$$\sqsubseteq \;:=\; \bigcup_{i\in\mathbb{N}} \sqsubseteq_i$$

where

$$\begin{aligned}
\sqsubseteq_0 &:= \{(\mathit{true},\mathit{true}),(\mathit{false},\mathit{false})\} \cup \mathscr{V} \times \{?\} \\
\sqsubseteq_{i+1} &:= \left\{ (r,r') \;\middle|\; \begin{array}{l} \mathit{dom}(r) \supseteq \overline{\mathit{dom}_p}(r') \wedge \\ \mathit{dom}(r) \cap \overline{\mathit{dom}_p}(r') = \emptyset \wedge \\ \forall l \in \mathit{dom}_p(r') : \mathit{elem}(r,l) \sqsubseteq_i \mathit{elem}_p(r',l) \end{array} \right\} \cup \sqsubseteq_i .
\end{aligned}$$

Above, I define the relation $\sqsubseteq$ that relates each full record value with all corresponding partial record values. As in previous definitions, I use an inductive definition over the nesting depth of partial record values. Initially, $\sqsubseteq_0$ defines the relation for Boolean values and for the special value *?*. Each Boolean value is a partial value of itself. Furthermore, *?* is a partial value of each value in $\mathscr{V}$.

In the inductive step of the above definition, I then define for a value $r \in \mathscr{V}$ and a partial value $r' \in \mathscr{V}_p$ that $r \sqsubseteq r'$ if

- the partial domain of the partial value r' is a subset of the domain of r,
- the anti-domain of the partial value r' and the domain of r are disjoint and
- for all labels l in the partial domain of r' the corresponding element value needs to be a partial value with respect to the element value for the label l in r.

A note on notation: It might seem counter intuitive to use $r \sqsubseteq r'$ to denote that r' is a partial value of r, as $r \sqsubseteq r'$ insinuates that r is in someway smaller than, or even a subset of r'. However, one might argue that indeed r' is smaller than r as r' is only partial and thus has a smaller domain than r. The motivation to use the notation $\sqsubseteq$ in the way I do in this thesis stems from the use of the same symbol to describe subtype relationships in type theory. One can see the partial record value r' as a descriptor (or even type) for many different actual record values and in particular the single record value r. Thus, the set of full record values described by r' has more members than the singleton set of record values described by the full record value r. In that sense, r is a subset of r'.

I extend the notion of partial value to variable environments as follows:

Definition 5.3.6 (Partial Variable Environment)**.** *The relation on variable environments* $\vec{\sqsubseteq} \subset \mathscr{P}(\mathscr{I} \times \mathscr{V}) \times \mathscr{P}(\mathscr{I} \times \mathscr{V}_p)$ *is defined as*

$$\vec{\sqsubseteq} :=$$

$$\left\{ (\mathscr{E}, \mathscr{E}') \in \mathscr{P}(\mathscr{I} \times \mathscr{V}) \times \mathscr{P}(\mathscr{I} \times \mathscr{V}_p) \;\middle|\; \begin{array}{l} \forall (l, v) \in \mathscr{E} \exists (l, v') \in \mathscr{E}' : v \sqsubseteq v' \wedge \\ \forall (l, v') \in \mathscr{E}' \exists (l, v) \in \mathscr{E} : v \sqsubseteq v' \end{array} \right\}$$

Thus, a variable environment $\prec$' is partial with respect to an environment $\prec$ if both contain the same labels and if each partial value bound to a label in $\prec$' is partial with respect to the corresponding value bound to the same label in $\prec$.

Using the definitions of $\sqsubseteq$ and $\vec{\sqsubseteq}$ above, I now show an important property of the pattern matching process of full and partial evaluation. If used in the same context, the partial match on a set of partial arguments yields at least the instances a full match on the corresponding full arguments yields. To show this property, I will first show in two lemmata that the corresponding property is true for the third and fourth step of the matching process. To differentiate between the results of the full and partial matching process, I will use the superscripts f and p, respectively.

Lemma 5.3.1 (Partial Match: Step 3)**.** *Given a function environment $\mathscr{F}$, a partial order on labels $\prec$, a set of function instances I, and arguments $a^f = (a_1^f, \ldots, a_n^f) \in \mathscr{V}^n$*

and $a^p = (a_1^p, \ldots, a_n^p) \in \mathscr{V}_p^n$*. Let* $(\mathscr{F}^f, \prec^f, I^f, a'^f) := pattern(\mathscr{F}, \prec, I, a)$ *and* $(\mathscr{F}^p, \prec^p, I^p, a'^p) := pattern^p(\mathscr{F}, \prec, I, a^p)$*. Then the following statement holds:*

$$(\forall i \in \{1, \ldots, n\} : a_i^f \sqsubseteq a_i^p) \Rightarrow \mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge I^f \subseteq I^p \wedge a^f = a'^f \wedge a^p = a'^p$$

Proof. Assume that for all $i \in \{1, \ldots, n\}$ the statement $a_i^f \sqsubseteq a_i^p$ holds. I will show that then the statement $\mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge I^f \subseteq I^p \wedge a^f = a'^f \wedge a^p = a'^p$ holds, as well.

The equalities in the above statement follow directly from the definition of *pattern* and *pattern*^{p} in Figures 3.12 and 5.4, respectively. It remains to be shown that $I^f \subseteq I^p$ holds.

The sets of matching instances I^f and I^p are defined inductively over the arguments a^f and a^p, respectively. I will show that in each step of the induction the resulting set of the filtering process in *pattern* is a subset of the result of the filtering process in *pattern*^{p}.

For the initial step this is obviously the case.

For the inductive step, assume I_i^f as the resulting set of instances of the previous filtering step in the function *pattern* as defined in Figure 3.12 on page 60 and I_i^p as the resulting set of instances of the previous filtering step in the function *pattern*^{p} as defined in Figure 5.4 on page 90. Furthermore, assume that $I_i^f \subseteq I_i^p$ holds. I will show the inductive step in two parts.

First, I show that *filter*$_1^p$ only filters instances that *filter* filters, as well. Let $I_{i+1}^f := \text{filter}(I_i^f, a_{i+1}^f, i+1)$ and $I_{i+1}^{p,1} := \text{filter}_1^p(I_i^p, a_{i+1}^p, i+1)$. I will show by contradiction that $I_{i+1}^f \subseteq I_{i+1}^{p,1}$ holds.

Assume that there exists an instance $f := (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e)$ with $f \in I_{i+1}^f$ but $f \notin I_{i+1}^{p,1}$. Then, by the definition of *filter*, we know that $p_{i+1} \subseteq \text{dom}(a_{i+1}^f)$. Furthermore, we know that $f \in I_i^{p,1}$, as $I_{i+1}^f \subseteq I_i^f$ by the definition of *filter* and thus $I_{i+1}^f \subseteq I_i^p$ by the inductive assumption. As $f \notin I_{i+1}^{p,1}$, the instance f is filtered out in the $i+1$-th step and thus we know that $p_{i+1} \cap \overline{\text{dom}}_p(a_{i+1}^p) \neq \emptyset$ by the definition of *filter*$_1^p$. Thus, $\overline{\text{dom}}_p(a_{i+1}^p) \cap \text{dom}(a_{i+1}^f) \neq \emptyset$. This contradicts that $a_{i+1}^f \sqsubseteq a_{i+1}^p$.

Second, I show that *filter*$_2^p$ only filters instances that *filter* filters, as well. Let $I_{i+1}^{p,2} := \text{filter}_2^p(I_{i+1}^{p,1}, a_{i+1}^p, i+1)$. I will show by contradiction that $I_{i+1}^f \subseteq I_{i+1}^{p,2}$ holds.

Assume that there exists an instance $f := (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e)$ with $f \in I_{i+1}^f$ but $f \notin I_{i+1}^{p,2}$. As I have shown above, $f \in I_{i+1}^{p,1}$ holds. Thus, by definition of *filter*$_2^p$, there exists at least one instance $f' := (((g_1', p_1'), \ldots, (g_n', p_n')), e')$ with $f' \in I_{i+1}^{p,1}$ such that for all $j \in \{1, \ldots, i+1\}$ the statement $p_j \setminus \text{dom}_p(a_j^p) = p_j' \setminus \text{dom}_p(a_j^p)$ holds, and such that $|p_{i+1} \cap \text{dom}_p(a_{i+1}^p)| < |p_{i+1}' \cap \text{dom}_p(a_{i+1}^p)|$ holds, as well.

First I will show that $f' \in I_{i+1}^f$. As $f \in I_{i+1}^f$, we know that for all $j \in \{1, \ldots, i+1\}$ the statement $p_j \subseteq \text{dom}(a_j^f)$ holds by the definition of *filter*. It follows that $p_j \setminus \text{dom}_p(a_j^p) \subseteq \text{dom}(a_j^f)$ holds, as well. Thus, $p_j' \setminus \text{dom}_p(a_j^p) \subseteq \text{dom}(a_j^f)$. From $a_j^f \sqsubseteq a_j^p$, we know that $\text{dom}_p(a_j^p) \subseteq \text{dom}(a_j^f)$. It follows that $p_j' \subseteq \text{dom}(a_j^f)$. Thus f' matches the first $i+1$ arguments with respect to *filter*.

We know that for all $j \in \{1, \ldots, i+1\}$ the statement $p_j \setminus \text{dom}_p(a_j^p) = p_j' \setminus \text{dom}_p(a_j^p)$ holds. As $f' \in I_{i+1}^{p,1}$ and $f \in I_{1+1}^{p,1}$, for all $j \in \{1, \ldots, i+1\}$ the statement $|p_j' \cap \text{dom}_p(a_j^p)| =$

$|p_j \cap \mathrm{dom}_p(a^p_j)|$ must hold, as well, as otherwise either instance would have been filtered out previously. As we furthermore know that $p'_j \setminus \mathrm{dom}_p(a^p_j) = p_j \setminus \mathrm{dom}_p(a^p_j)$, it follows that $|p'_j| = |p_j|$. Thus, for the first $i+1$ arguments, both instances match equally well. It follows that $f' \in I^f_{i+1}$.

Next I will show that $f \in I^f_{i+1}$ cannot hold. As $f \in I^f_{i+1}$, it follows that $p_{i+1} \subseteq \mathrm{dom}(a^f_{i+1})$ by the definition of *filter*. Thus, $p_{i+1} \setminus \mathrm{dom}_p(a^p_{i+1}) \subseteq \mathrm{dom}(a^f_{i+1})$. Consequently, $p'_{i+1} \setminus \mathrm{dom}_p(a^p_{i+1}) \subseteq \mathrm{dom}(a^f_{i+1})$. From $a^f_i \sqsubseteq a^p_i$, it follows that $\mathrm{dom}_p(a^p_i) \subseteq \mathrm{dom}(a_i)$. Thus, overall, it follows that $p'_{i+1} \subseteq \mathrm{dom}(a^f_{i+1})$.

Furthermore, we have that $|p_{i+1} \cap \mathrm{dom}_p(a^p_{i+1})| < |p'_{i+1} \cap \mathrm{dom}_p(a^p_{i+1})|$ holds. Thus, as $p_{i+1} \setminus \mathrm{dom}_p(a^p_{i+1}) = p'_{i+1} \setminus \mathrm{dom}_p(a^p_{i+1})$ holds, we can follow that $|p_i| < |p'_i|$. This contradicts that $f \in I_{i+1}$ as both f and f' match but f' contains more labels.

Overall, it follows that $I^f \subseteq I^p$. □

Lemma 5.3.2 (Partial Match: Step 4)**.** *Given a function environment $\mathscr{F}$, a partial order on labels $\prec$, a set of function instances I and arguments $a^f = (a^f_1, \dots, a^f_n) \in \mathscr{V}^n$ and $a^p = (a^p_1, \dots, a^p_n) \in \mathscr{V}^n_p$. Let $(\mathscr{F}, \prec, I^f, a^f) :=$ pattern$(\mathscr{F}, \prec, I, a^f)$ and $(\mathscr{F}, \prec, I^p, a^p) :=$ pattern$^p(\mathscr{F}, \prec, I, a^p)$. Furthermore, let $(\mathscr{F}^f, \prec^f, I'^f, a'^f) :=$ order$(\mathscr{F}, \prec, I^f, a^f)$ and $(\mathscr{F}^p, \prec^p, I'^p, a'^p) :=$ order$^p(\mathscr{F}, \prec, I^p, a^p)$. Then the following statement holds:*

$$(\forall i \in \{1, \dots, n\} : a^f_i \sqsubseteq a^p_i) \Rightarrow \mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge I'^f \subseteq I'^p \wedge a^f = a'^f \wedge a^p = a'^p$$

Proof. Assume that for all $i \in \{1, \dots, n\}$ the statement $a^f_i \sqsubseteq a^p_i$ holds. I will show that then the statement $\mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge I'^f \subseteq I'^p \wedge a^f = a'^f \wedge a^p = a'^p$ holds, as well.

The equalities in the above statement follow directly from the definition of *order* and *order^p* in Figures 3.12 and 5.4, respectively. It remains to be shown that $I'^f \subseteq I'^p$ holds.

As for the functions *pattern* and *pattern^p*, the set of resulting instances in the functions *order* and *order^p* is defined inductively over the arguments. Therefore, I will use a similar proof strategy as in Lemma 5.3.1 on page 96. I will show that in each step of the inductive construction of the result, the result of the filtering process in the function *filter$'$* for the full matching process is a subset of the result of the function *filter$'^p$* in the partial matching process.

For the initial step, this follows from the fact that I^f and I^p are the results of the functions *pattern* and *pattern^p*, respectively, and the result of Lemma 5.3.1 on page 96.

For the inductive step, let I'^f_i and I'^p_i be the result of the previous filtering step of *order* and *order^p*, respectively. Furthermore, assume that $I'^f_i \subseteq I'^p_i$ holds. I will show $I'^f_{i+1} \subseteq I'^p_{i+1}$ by contradiction.

Assume there exists an instance $f := (((\alpha_1, p_1), \dots, (\alpha_n, p_n)), e)$ which is in I'^f_{i+1} but not in I'^p_{i+1}. We know that $f \in I'^p_i$ holds, as $I'^f_i \subseteq I'^p_i$ by the inductive assumption and $I'^f_{i+1} \subseteq I'^f_i$ by the definition of *filter$'^p$* in Figure 3.12 on page 60. Thus, as f is filtered out in the current filtering step, there needs to exists an instance $f' := (((g'_1, p'_1), \dots, (g'_n, p'_n)), e') \in I'^p_i$ such that for all $j \in \{1, \dots, n\}$ the statement $p_j \setminus \mathrm{dom}_p(a^p_j) = p'_j \setminus \mathrm{dom}_p(a^p_j)$ holds and such that $p_{i+1} \overset{\rightarrow}{\prec} p'_{i+1}$ holds, as well.

First, I will show that $f' \in I^f$. From $f' \in I'^p_{i+1}$ directly follows by the definition of *filter*$'^p$ that $f' \in I^p$. As both instances are contained in I^p, it follows from the definition of *pattern*^{p} that for each $j \in \{i, \ldots, n\}$ the statement $|p_j \cap \mathrm{dom}_p(a^p_j)| = |p'_j \cap \mathrm{dom}_p(a^p_j)|$ holds. As we know that $p_j \setminus \mathrm{dom}_p(a^p_j) = p'_j \setminus \mathrm{dom}_p(a^p_j)$ holds, it follows that $|p_j| = |p'_j|$.

From $f \in I^f$, it follows that for all $j \in \{1, \ldots, n\}$ we know that $p_j \subseteq \mathrm{dom}(a^f_j)$. Consequently, $p_j \setminus \mathrm{dom}_p(a^p_j) \subseteq \mathrm{dom}(a^f_j)$. From $p_j \setminus \mathrm{dom}_p(a^p_j) = p'_j \setminus \mathrm{dom}_p(a^p_j)$ it then follows that $p'_j \setminus \mathrm{dom}_p(a^p_j) \subseteq \mathrm{dom}(a^f_j)$. As $a^f_j \sqsubseteq a^p_j$, it follows that $p'_j \subseteq \mathrm{dom}(a^f_j)$. Thus, from the definition of *pattern* it then follows that $f' \in I^f$.

Next, I will show that $f' \in I'^f_i$. $f' \in I'^f_0$ is obviously true by the definition of *order*. Assume f' is filtered out in any step $k \in \{1, \ldots, i\}$. Then an instance $f'' := (((a''_1, p''_1), \ldots, (a''_n, p''_n)), e'') \in I'^f_{k-1}$ such that $p'_k \overset{\rightarrow}{\prec} p''_k$ holds needs to exist by definition of the function *filter*$'$ in Figure 3.12 on page 60. Furthermore, as neither of the three instance has been filtered out in the *pattern* step, we know that $|p_k| = |p'_k| = |p''_k|$.

As $I'^f_{k-1} \subseteq I'^p_{k-1}$ by the inductive assumption, it follows that $f'' \in I'^p_{k-1}$. From $p'_k \overset{\rightarrow}{\prec} p''_k$ and $f' \in I'^p_i$, we can further follow that $f'' \in I'^p_i$, as f'' cannot be filtered by any instance as long as f' is not filtered. Furthermore, we can follow from $f \in I'^f_{i+1}$ that $p_k \overset{\rightarrow}{\not\prec} p''_k$, as otherwise f would have been filtered by f''. This contradicts that f' filters f, as $p_k \overset{\rightarrow}{\not\prec} p''_k$ but $p'_k \overset{\rightarrow}{\prec} p''_k$. Thus $f' \in I'^f_k$.

Overall, f' therefore cannot be filtered at any stage $k \in \{1, \ldots, i\}$. It follows that $f' \in I'^f_i$.

Finally, as we have $p_{i+1} \overset{\rightarrow}{\prec} p'_{i+1}$, $f' \in I'^f_{i+1}$ contradicts $f \in I'^f_{i+1}$. Thus, the statement $I'^f_{i+1} \subseteq I'^p_{i+1}$ must hold. □

Using the above two lemmata, I can now show the desired result for the matching process under full and partial evaluation.

Theorem 5.3.1 (Partial Match). *Given a function environment $\mathscr{F}$, an identifier $f \in \mathscr{I}$ and arguments $a^f = (a^f_1, \ldots, a^f_n) \in \mathscr{V}^n$ and $a^p = (a^p_1, \ldots, a^p_n) \in \mathscr{V}^n_p$. Let $I^f = match(\mathscr{F}, f, a^f)$ and $I^p = match^p(\mathscr{F}, f, a^p)$. Then the following statement holds:*

$$(\forall_{i \in \{1,\ldots,n\}} : a^f_i \sqsubseteq a^p_i) \Rightarrow (\forall (\mathscr{F}, \prec, \mathscr{E}^f, e) \in I \exists (\mathscr{F}, \prec, \mathscr{E}^p, e) \in I' : \mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}')$$

Proof. To show the above, I first show that the first four steps in the partial matching process (cf. Figure 5.4 on page 90) at least select those instances that the corresponding steps of the full matching process (cf. Figure 3.12 on page 60) select and that they produce identical function environments $\mathscr{F}$ and orders on labels $\prec$.

For the first step, the function *lookup*, this is obviously the case. Both matching processes use an identical first step. Furthermore, this step does not take the function arguments into account. It only operates on the function environment $\mathscr{F}$ and the function identifier f, which are assumed to be identical.

In the second step, again both matching processes use the same filtering function *arity*. The function *arity* reduces the set of instances based on the number of arguments. As for both, the partial and full match, the number of arguments is assumed to be n, the function *arity* filters the same instances.

For the third step, I have shown above property in Lemma 5.3.1 on page 96. Similarly, I have shown in Lemma 5.3.2 on page 98 the above property for the fourth step.

So after step four in the pattern matching process, we have $(\mathscr{F}^f, \prec^f, I^f, a^f)$ as result of the pattern matching process on full pattern and $(\mathscr{F}^p, \prec^p, I^p, a^p)$ as the corresponding result of the partial matching process. Furthermore, we know that $\mathscr{F}^f = \mathscr{F}^p$, $\prec^f = \prec^p$ and $I^f \subseteq I^p$. From the antecedent, we further know that for all $i \in \{1, \dots, n\}$ the statement $a_i^f \sqsubseteq a_i^p$ holds. Thus, we can conclude the consequent from the definition of *bind* in Figure 3.12 on page 60. □

Having shown that the partial pattern match under partial evaluation always yields at least the instances the full pattern match yields in a corresponding full evaluation context, I can now show the final result for this section: Under the assumption that the choice, which argument of the `any` operator is evaluated, is deterministic and identical for full and partial evaluation, for any expression in LRec$_C$, if the expression can be evaluated under both full and partial evaluation, the result of partial evaluation is a partial value of the result of full evaluation.

The restriction that the `any` operator behaves deterministically and identically under full and partial evaluation is required, as otherwise even two evaluations of the same expression using either $\Downarrow$ or $\downarrow$ could yield different results.

As a prerequisite, I first prove the following lemma for the function *val*^{p}:

Lemma 5.3.3 (Partial Bindings). *Given a function environment $\mathscr{F}$, a partial order on labels $\prec$, two variable environments $\mathscr{E}^f$ and $\mathscr{E}^p$ with $\mathscr{E}^f \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^p$ and definitions $d_1, \dots, d_n$ of a* `let` *construct. Furthermore, assume that for every expression e in* LRec$_C$ *with a given nesting depth $i \in \mathbb{N}$ the statement $(\mathscr{F}, \prec, \mathscr{E}) \ : \ e \Downarrow v \wedge (\mathscr{F}, \prec, \mathscr{E}') \ : \ e \downarrow v' \wedge \mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}' \Rightarrow v \sqsubseteq v'$ holds. Let $\mathscr{E}^{f'} := val(\mathscr{F}, \prec, \mathscr{E}^f, (d_1, \dots, d_n))$ and $\mathscr{E}^{p'} := val^p(\mathscr{F}, \prec, \mathscr{E}^p, (d_1, \dots, d_n))$ with val as defined in Figure 3.11 on page 58 and val^p as defined in Figure 5.3 on page 88. Then the statement $\mathscr{E}^{f'} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'}$ holds.*

Proof. Both functions *val* and *val*^{p} compute the new environment inductively starting out with the old environment and then step-wise amending the environment by the bindings produced by each definition. To prove the above statement, I will show by induction that during each processing step, the resulting environment in *val*^{p} is a partial environment of the corresponding environment in *val*.

For the initial step, this is trivially the case as $\mathscr{E}^f \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^p$ holds.

Let ${\mathscr{E}^{f}}'_j$ be the result of j-th processing step in the function *val* and ${\mathscr{E}^{p}}'_j$ be the corresponding result of the j-th processing step in the function *val*^{p}. Assume ${\mathscr{E}^{f}}'_j \overset{\rightarrow}{\sqsubseteq} {\mathscr{E}^{p}}'_j$ holds.

The definition d_{j+1} can be one of three different kinds: A function definition using the `fun` keyword, an order definition using the `rel` keyword or a value binding using the `val` keyword. I will show for each case that ${\mathscr{E}^{f}}'_{j+1} \overset{\rightarrow}{\sqsubseteq} {\mathscr{E}^{p}}'j + 1j$ holds.

Case 1: `fun` definition Assume d_{j+1} is a definition of the from `fun` $l\ i_1 \cdots i_k$ where the $i_1, \ldots, i_k$ are instance definitions. Then from the definition of *val* and *val*^{p} it follows that $\mathscr{E}^{f'}_{j+1} = \mathscr{E}^{f'}_{j}$ and $\mathscr{E}^{p'}_{j+1} = \mathscr{E}^{p'}_{j}$, respectively. Thus, from the inductive assumption, it follows that $\mathscr{E}^{f'}_{j+1} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'} j + 1j$ holds.

Case 2: `rel` definition $\mathscr{E}^{f'}_{j+1} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'} j + 1j$ follows analogously to case 1.

Case 3: `val` definition Assume d_{j+1} is a definition of the form `val` l=e. From the statements $(\mathscr{F}, \prec, \mathscr{E}^{f'}_{j}) : e \Downarrow v$ and $(\mathscr{F}, \prec, \mathscr{E}^{p'}_{j}) : e \downarrow v'$, we know that $v \sqsubseteq v'$. Thus, $\mathscr{E}^{f'}_{j} \leftarrow (l, v) \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'}_{j} \leftarrow (l, v')$ holds. Therefore, $\mathscr{E}^{f'}_{j+1} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'} j + 1j$, holds as well.

Overall, it follows that in the $i + 1$-th step the statement $\mathscr{E}^{f'}_{j+1} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'}_{j+1}$ holds. □

Using this lemma, I can now show the final result for this section.

Theorem 5.3.2 (Correctness of Partial Evaluation)**.** *Given that the choice of which argument expression to evaluate for the* `any` *operator is deterministic and identical for full and partial evaluation, for every expression e in* LREC$_C$*, the statement*

$$(\mathscr{F}, \prec, \mathscr{E}) \ : \ e \Downarrow v \wedge (\mathscr{F}, \prec, \mathscr{E}') \ : \ e \downarrow v' \wedge \mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}' \Rightarrow v \sqsubseteq v'$$

where $\mathscr{F}$ is a function environment, $\prec$ is an order on labels and $\mathscr{E}$ and $\mathscr{E}'$ are a variable and partial variable environment, holds.

Proof. I will show the above by induction of the nesting depth of expressions.

To begin the inductive proof, assume e has a nesting depth of 0, *i.e.*, e is a non-nested expression. Then, by the definition of LREC$_C$ in Figure 3.6 on page 50, e is either a Boolean expression, the special ˜ expression, an empty record expression or an identifier.

Assume that $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$, $(\mathscr{F}, \prec, \mathscr{E}') : e \downarrow v'$ and $\mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}'$ hold. I will show that for each e then $v \sqsubseteq v'$ holds.

For any expression in LREC$_C$, under partial evaluation the inference rule PARTIAL can always be used. However, as in this case $v' = ?$, the statement $v \sqsubseteq v'$ holds for all legal values $v \in \mathscr{V}$. Therefore, I will not handle this case explicitly for each expression below.

Case 1: `true` Under full evaluation, the expression `true` can only be evaluated by rule TRUE in Figure 3.8 on page 52. Thus, $v = true$. For the partial evaluation case, apart from the rule PARTIAL, only the rule TRUE in Figure 5.2 on page 86 can be used, yielding $v' = true$. Thus $v \sqsubseteq v'$ holds.

Case 2: `false` The result follows analogously to case 1 using the rule FALSE.

Case 3: ˜ For the unit expression, under full evaluation only the rule UNIT can be used. It follows that $v = \{\}$. Under partial evaluation, apart from the rule PARTIAL only the rule UNIT matches. From that rule it follows that $v' = R$ where $R \subseteq \mathscr{L} \times \{!\}$. Thus, $\mathrm{dom}(v) \cap \overline{\mathrm{dom}_p}(v') = \emptyset$ and $\mathrm{dom}_p(v') = \mathrm{dom}(v) = \emptyset$ hold. It follows that $v \sqsubseteq v'$.

Case 4: Empty Record Under full evaluation, the expression `{}` can only be evaluated using the rule Record. Thus, $v = \{\}$. Under partial evaluation, the rules Record and Partial can be used. From the rule Record we can follow that $v' = R$ where $R \subseteq \mathscr{L} \times \{!\}$. Analogously to case 3, it follows that $v \sqsubseteq v'$.

Case 5: Identifier In the case of full evaluation, only the rule Var applies. As e can be evaluated to a value v, $(e, v) \in \mathscr{E}$ must hold. Under partial evaluation, the rules Var and Partial match. I only consider the former rule here. As e can be evaluated to v', we know that $(e, v') \in \mathscr{E}'$. From $\mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}'$ it then follows that $v \sqsubseteq v'$ holds. Therefore, $v \sqsubseteq v'$ holds in this case, as well.

Overall, it follows that for all expressions with nesting depth 0, $v \sqsubseteq v'$ holds.

For the inductive step, assume that for all expressions e with a given nesting depth of $i \in \mathbb{N}$ the statement $(\mathscr{F}, \prec, \mathscr{E})\ :\ e \Downarrow v \wedge (\mathscr{F}, \prec, \mathscr{E}')\ :\ e \downarrow v' \wedge \mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}' \Rightarrow v \sqsubseteq v'$ holds. I will show that it then holds for expressions with nesting depth $i + 1$, as well. Given an expression e with nesting depth $i{+}1$. By the definition of LRec$_{\mathrm{C}}$ in Figure 3.6 on page 50, e can be a record expression, a selection operation, the `any` operation, a conditional, a `guard` operation, a `witness` operation, the equality operation on non-record values, a `let` expression or a function application.

Assume that $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$, $(\mathscr{F}, \prec, \mathscr{E}') : e \downarrow v'$ and $\mathscr{E} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}'$ hold. I will show for each kind of expression listed above that then $v \sqsubseteq v'$ holds, as well. As we know that e can be evaluated under both evaluation schemes, it is safe to assume in the following that all premises of semantics rules used in derivations for e are satisfied. Furthermore, I will ignore the rule Partial here, as its result always satisfies the partial value property.

Case 1: Record Expression Assume e is a record expression of the form `{`l_1 `=` e_1`, ...,` l_n `=` e_n`}`. As e can be evaluated, we know that each e_j for $j \in \{1, \ldots, n\}$ can be evaluated to a value v_j, as well. Thus, $v = \{(l_1,v_1),\ldots,(l_n,v_n)\}$. In case of partial evaluation, we know that a subset of the e_j are evaluated to v'_j. From the inductive assumption, we furthermore know that $v_j \sqsubseteq v'_j$ for all $j \in \{1, \ldots, n\}$ if such a v'_j exists. Furthermore, from the construction of v' as the result of rule Record in the partial evaluation case, it follows that $\mathrm{dom}_p(v') \subseteq \mathrm{dom}(v)$ and $\overline{\mathrm{dom}_p}(v') \cap \mathrm{dom}(v) = \emptyset$. Consequently, $v \sqsubseteq v'$ holds.

Case 2: Selection Operation Assume e is a selection operation of the form e_b`.`l. As e can be evaluated to values v and v' under full and partial evaluation, respectively, the expression e_b must evaluate to values v_b and v'_b under full and partial evaluation. From the inductive assumption we know that $v_b \sqsubseteq v'_b$ holds. It follows that then $\mathrm{elem}(v_b, l) \sqsubseteq \mathrm{elem}_p(v'_b, l)$ holds, as well. Consequently, $v \sqsubseteq v'$ holds.

Case 3: `any` Operation Assume e has the form `any(`e_1`, ...,` e_n`)`. From the statement $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ we know that for at least one e_j with $j \in \{1, \ldots, n\}$ the statement $(\mathscr{F}, \prec, \mathscr{E})\ :\ e_j \Downarrow v_j$ holds. Similarly, $(\mathscr{F}, \prec, \mathscr{E}')\ :\ e \downarrow v'$ implies that a e'_j with $j' \in \{1, \ldots, n\}$ can be evaluated to v'_j under partial evaluation. From the condition that the `any` operator is deterministic and that it chooses the same argument for evaluation under full and partial evaluation, it furthermore follows that $j = j'$. With the inductive assumption, we can derive that $v_j \sqsubseteq v'_j$. Therefore, $v \sqsubseteq v'$ holds.

Case 4: Conditional Assume e is a conditional expression of the form `if` e_p e_t e_e. From $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ it follows that either $(\mathscr{F}, \prec, \mathscr{E}) : e_p \Downarrow \mathit{true}$ or $(\mathscr{F}, \prec, \mathscr{E}) : e_p \Downarrow \mathit{false}$. I will assume the first case. For the second case, the result follows analogously.

From $(\mathscr{F}, \prec, \mathscr{E}') : e \downarrow v'$ we can derive $(\mathscr{F}, \prec, \mathscr{E}') : e_p \downarrow v'_p$ with $v'_p \in \{\mathit{true}, \mathit{false}\}$. With the inductive assumption, it follows that $\mathit{true} \sqsubseteq v'_p$ and thus $v'_p = \mathit{true}$. Thus, only the rule CondThen applies. We have that $(\mathscr{F}, \prec, \mathscr{E}) : e_t \Downarrow v_t$ and $(\mathscr{F}, \prec, \mathscr{E}) : e_t \downarrow v'_t$, as e can be evaluated under full and partial evaluation. Furthermore, from the inductive assumption we know that $v_t \sqsubseteq v'_t$. Thus $v \sqsubseteq v'$.

Case 5: `guard` operation Assume e has the form `guard(`e_v e_g`)`. From $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ it follows that $(\mathscr{F}, \prec, \mathscr{E}) : e_v \Downarrow v$. Similarly, we can follow from $(\mathscr{F}, \prec, \mathscr{E}) : e \downarrow v'$ that $(\mathscr{F}, \prec, \mathscr{E}) : e_v \downarrow v'$. From the inductive assumption we then know that $v \sqsubseteq v'$ holds.

Case 6: `witness` operation Then e has the form `witness(`e_v e_1 $\cdots$ e_n`)`. We know from $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$. Similarly, from $(\mathscr{F}, \prec, \mathscr{E}) : e \downarrow v'$ we can derive that $(\mathscr{F}, \prec, \mathscr{E}) : e_v \downarrow v'$. With the inductive assumption it then follows that $v \sqsubseteq v'$ holds.

Case 7: Equality operation Assume e has the form `(`e_1 `=` e_2`)`. Thus, either the rule EqualTrue or the rule EqualFalse applies. To show that $v \sqsubseteq v'$ holds, it suffices to show that e_1 and e_2 evaluate to the same value under full and partial evaluation. From $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ it follows that $(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow v_1$ and $(\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow v_2$ with $v_1, v_2 \notin \mathscr{R}$. Furthermore, from $(\mathscr{F}, \prec, \mathscr{E}') : e \downarrow v'$ it follows that $(\mathscr{F}, \prec, \mathscr{E}') : e_1 \downarrow v'_1$ and $(\mathscr{F}, \prec, \mathscr{E}') : e_2 \downarrow v'_2$ with $v'_1, v'_2 \notin \mathscr{R}_p$. From the inductive assumption, we know that $v_1 \sqsubseteq v'_1$ and $v_2 \sqsubseteq v'_2$ hold. Thus, by the definition of $\sqsubseteq$, it follows that $v_1 = v'_1$ and $v_2 = v'_2$.

Case 8: `let` Expression For a `let` expression, it suffices to show that the computed new function environment and order on labels are identical under full and partial evaluation, and that the new variable environment computed under partial evaluation is a partial environment of the new environment computed under full evaluation.

The former two equalities follow directly from the definition of the two helper functions *fun* and *rel* in Figures 3.10 and 3.9, respectively. Both functions only collect information independently of the variable environment and do not evaluate any expressions.

The latter follows with Lemma 5.3.3 on page 100.

Case 9: Function Application Assume the expression e is a function application of the form `(`f e_1 `...` e_n`)`. We know that $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ and $(\mathscr{F}, \prec, \mathscr{E}') : e \downarrow v'$. It follows that for all $j \in \{1, \ldots, n\}$ $(\mathscr{F}, \prec, \mathscr{E}) : e_j \Downarrow v_j$ and $(\mathscr{F}, \prec, \mathscr{E}') : e_j \downarrow v'_j$. From the inductive assumption, we furthermore know that $v_j \sqsubseteq v'_j$ holds. Furthermore, as the application can be evaluated, the functions *match* and *matchp* both yield a single instance. With Theorem 5.3.1 on page 99 it follows that even more both select the same instance and that the new environment under partial evaluation is a partial environment of the environment under full evaluation. Thus, we can follow from the inductive assumption that $v \sqsubseteq v'$.

□

From the above proposition, we can directly follow the desired property:

Corollary 5.3.1 (Correctness of Partial Evaluation)**.** *Given that the choice of the argument expression to evaluate for the* `any` *operator is deterministic and identical for full and partial evaluation, for every expression e in* LRec_C*, the statement*

$$(\bot, \emptyset, \emptyset) \ : \ e \Downarrow v \wedge (\bot, \emptyset, \emptyset) \ : \ e \downarrow v' \Rightarrow v \sqsubseteq v'$$

holds.

Proof. The above follows directly form Theorem 5.3.2 on page 101 and $\emptyset \overset{\rightarrow}{\sqsubseteq} \emptyset$ by definition of $\overset{\rightarrow}{\sqsubseteq}$ in Definition 5.3.6 on page 96. □

It is worth noting here that the inverse statement that if partial evaluation succeeds, full evaluation succeeds as well and the result of partial evaluation is a partial value of the result of full evaluation does not hold. The partial rules Record and Partial in Figure 5.2 on page 86 have an impact on termination behaviour in comparison to full evaluation. Even if the full evaluation diverges, partial evaluation might still terminate if the diverging expression is never evaluated. This is a general effect of non-strict evaluation and not a result of the design of the partial evaluator presented here. The other direction, *i.e.*, that full evaluation terminates and partial evaluation diverges, can not occur. By design, my partial evaluation rules never evaluate an expression that the full evaluator would not evaluate. However, as noted before, depending on the choice of the anti-domain R in rules Unit and Record in Figure 5.2 on page 86, partial evaluation might fail where full evaluation succeeds.

5.4. Conclusions

In this chapter, I have motivated the use of partial evaluation to compute certain auxiliary computations statically. I have shown for my running examples from Chapter 2, how such an approach can provide additional static safety. The computation of static information thereby consists of two steps. A first step that infers which expressions are to be evaluated and to what extend they need to be evaluated and a second step that actually evaluates a program to a corresponding partial result. I have focused on the second aspect in this chapter, leading to an operational semantics for partial evaluation as presented in Section 5.3. As I have shown, these semantics are correct in that partial evaluation yields results that are partial values of the result of full evaluation. However, the semantics allow a program to be evaluated to a range of partial results. Even more, partial evaluation of an expression might get stuck if insufficient information was computed for one of its sub-expressions.

In the next Chapter, I will investigate the first step and constrain the partial evaluation further such that it yields a well defined result and does not get stuck where the corresponding full evaluation would succeed.

6. Guiding Partial Evaluation

In the previous chapter, I have presented an operational semantics for partially evaluating expressions in LRec_C with respect to a given set of auxiliary computations only. The result of partial evaluation using those semantics is not uniquely defined. The main rules that introduce partial results, *i.e.*, the rules Record, Unit and Partial in Figure 5.2 on page 86 allow for two degrees of freedom.

Firstly, it is not defined which labels of a record value need to be evaluated to produce the results in rule Record. Even more, the rule Partial permits to stop evaluation for any expression and yield the undetermined partial result *?*. These two rules cater for the production of partial values.

Secondly, in the rules Record and Unit, the anti-domain of a partial result is not fully fixed. Any combination of labels that is not part of the domain of the corresponding full result is allowed. Even though the choice of which labels to include in the anti-domain does not impact the partial value as such, it has an impact on pattern matching in function applications. If insufficient information about the anti-domain of function arguments is present, partial evaluation might stall.

In this chapter, I present a form of binding-time analysis that, given any expression in LRec_C, determines the partial domain and anti-domain for all sub-expressions of such expression that need to be known in order to compute a desired set of properties for the given expression. This analysis differs from classical binding-time analyses in that it does not infer the binding-time of expressions, *i.e.*, whether an expression can be statically computed. Instead, the analysis computes a binding-time for an expression, *i.e.*, when and to what extent an expression should be computed.

I develop this analysis in two steps. First, I present a version of the analysis that is restricted to non-nested record expressions in LRec_C, *e.g.*, expressions like the examples on matrices presented in Chapter 2. Building on the general principles developed for the non-nested case, I then extend the analysis to full LRec_C with nested record expressions.

6.1. A Binding-Time Analysis for Non-Nested LRec_C

To motivate the design, I first give a couple of examples and discuss how the requirements for the domain and anti-domain of sub-expressions can be computed.

6.1.1. Binding-Time Analysis By Example

I start out with the simple example of matrix addition previously discussed in Chapter 2 and Section 5.2.

```
let
  fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
    = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
          A{ shape=sA, rank=rA, ldiag}
          B{ shape=sB, rank=rB, ldiag}
    = (ldiag_add( any(sA sB) !A !B){ shape=any(sA sB),
                                     rank=any(rA rB),
                                     ldiag}
  val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2, ldiag}
  val B = [ 9, 0, 8, 7]{ shape=[ 2, 2], rank=2, ldiag}
in
  (add A B)
end
```

Assume I want to know the `shape` component of the above expression. The partial semantics presented in Section 5.3 allows me to evaluate the above `let` expression to that respect only. However, in the above example, to compute the `shape` component of the entire expression, the shape component of the body expression of the `let` construct needs to be known. In other words, computing the `shape` component of the entire expression generates a demand for the `shape` component of the body of the `let` expression.

This process of deriving demands for certain components of a value continues for all sub-expressions in the above example. To compute the `shape` component of the body expression of the `let` construct, we have to compute the `shape` component of the result of the function application. In order to do so, we need first to derive which components of the two arguments to the function `add` are required.

To compute the demand on the arguments of a function application, we have to derive two kinds of information. Firstly, we need to infer which components of the arguments are required to compute the desired component of the result of the function. As we do not a priori know which instance of the function `add` will apply during evaluation of the program, we have to derive this demand separately for all instances. To ensure that enough information is available during partial evaluation regardless of which instance is chosen, the demand for the function arguments is the union of the separate demands for all instances of the function.

Secondly, we have to ensure that enough information is available to decide which instance to use during partial evaluation. As I have shown in the previous section, the partial match always yields at least the instances the full matching process would yield. However, if insufficient information is available, it may yield more instances and consequently partial evaluation may stall. To ensure that the pattern match under partial evaluation is decidable if it is decidable under full evaluation, we have to compute sufficient domain and anti-domain information for the arguments during partial evaluation.

Comparing the matching process of full and partial evaluation as defined in Figures 3.12 and 5.4, respectively, the main differences can be found in the *pattern* and *order* stages.

In case of the partial pattern match, both stages only compare two instances if they share a common set of labels with statically unknown state, *i.e.*, those labels that are neither in the domain nor anti-domain of the corresponding argument. I have put this restriction into place to ensure that the partial match does not filter out instances that the full match does not filter. However, as a side effect, the partial match, due to the above restriction, does not filter out instances the full match would filter.

To ensure that both matching processes filter exactly the same instances, it suffices to ensure that the above restriction never inhibits filtering out an instance, *i.e.*, it suffices to ensure that for every two instances the set of labels with statically unknown state is identical. Or, put in another way, it suffices to ensure that for every two instances all labels in their patterns that are not common for both instances have a statically known state. I will prove this property in Section 6.1.3.

To derive the first kind of demand in the above example, *i.e.*, to derive the demand that stems from the demand on the result of the function application, we have to infer what information is required of the arguments to compute the `shape` component of the result. For the first instance defined in Line 2 above, we need to compute the `shape` component of the record expression in Line 3. It is defined by means of an `any` operation. Thus, to compute the `shape` component of the result, we need to compute the result of the `any` operation. As the `any` operation may yield any of its arguments as its result, we need to know the value of its arguments. The first argument is the `shape` component of the parameter `A` of the function `add`. This gives us the first demand on one of the function arguments. Analogously, the second argument to the `any` operation produces a demand for the `shape` component of the parameter `B`. This completes the derivation of demands for the first instance. Overall, we get a demand for the value of the `shape` component for both arguments.

For the second instance in Line 4, the demand for its arguments can be derived similarly. Again, we get that we need to know the value of the `shape` components for both arguments to compute the `shape` component of the result.

To ensure that the pattern match succeeds under partial evaluation, we secondly need to derive for which labels their presence needs to be known in an application of the function `add`. As can be seen in the pattern for the first parameter of the two instances defined in lines 2 and 4, both instances use the `shape` and `rank` labels. The only discriminating label, *i.e.*, the only label that is not used by both instances, is the label `ldiag`. Thus, to decide the pattern match, we furthermore need to know whether the argument carries the label `ldiag`.

As can be seen in this example, the derived information does indeed not guarantee that any instance would match under full evaluation as the first argument to the function `add` might still lack any of the other labels `shape` and `rank`. However, if the example can be fully evaluated, the derived demands will ensure that the correct instance is chosen. Therefore, it is guaranteed that the result of partial evaluation is a partial value of the result of full evaluation, if such a result exists.

For the second parameter, we have the exact same set of patterns. Consequently, we need to know the same information to decide the pattern match. Overall, we get the following demands for the two function arguments:

- We need to know the value of the `shape` component and
- we need to know whether the `ldiag` label is present.

Next, we can propagate this demand to the actual function arguments `A` and `B` as defined in lines 9 and 10 above. To compute the `shape` component of the value `A`, we need to compute the corresponding element of the record expression. Furthermore, we need to compute whether the label `ldiag` is present. Similarly, we have to compute these components for the value `B`.

This completes the derivation of demands in the above example. Below is an annotated version of the source code including demands.

```
let
  fun add A{ shape=sA, rank=rA}^{(shape,↑{}),(ldiag,∅)}
          B{ shape=sB, rank=rB}^{(shape,↑{}),(ldiag,∅)}
    = (vect_add !A !B){ shape=any(sA^↑{} sB^↑{})^↑{},
                        rank=any(rA rB)}^{(shape,↑{})}
          A{ shape=sA, rank=rA, ldiag}^{(shape,↑{}),(ldiag,∅)}
          B{ shape=sB, rank=rB, ldiag}^{(shape,↑{}),(ldiag,∅)}
    = (ldiag_add( any(sA sB) !A !B){ shape=any(sA^↑{} sB^↑{})^↑{},
                                     rank=any(rA rB),
                                     ldiag}^{(shape,↑{})}
  val A = [ 1, 0, 0, 1]{ shape=[ 2, 2]^↑{},
                         rank=2, ldiag^∅}^{(shape,↑{}),(ldiag,∅)}
  val B = [ 9, 0, 8, 7]{ shape=[ 2, 2]^↑{},
                         rank=2, ldiag^∅}^{(shape,↑{}),(ldiag,∅)}
in
  (add A^{(shape,↑{}),(ldiag,∅)} B^{(shape,↑{}),(ldiag,∅)})^{(shape,↑{})}
end^{(shape,↑{})}
```

As a representation for demands, I use sets of (label,demand) pairs annotated to expressions. In such a pair, the second element is a set encoding the demand for that component of a record value. I use the empty set $\emptyset$ to encode that a value corresponding to a label needs not to be evaluated.

To annotate that a value needs to be fully evaluated, I use the demand $\uparrow \{\}$. The demand $\uparrow$ denotes that all labels contained in the domain of the full value are part of the domain of the demand in that value, as well. Thus, $\uparrow$ defines the domain of a demand as maximal with respect to any given value. The additional set of demands given after $\uparrow$ allows to further extend this set by labels that may not be contained in the domain of the full value. This set is important to force labels to be in the range of a partial value even if they are not in the domain. For example, in the above definition of `add`, knowledge about the label `ldiag` is required to decide the pattern match. Assume we get a demand on the first argument that requires it to be fully evaluated. If we only encode that requirement as the corresponding demand, the evaluation of a value that does not contain a `ldiag` component would not contain the `ldiag` label in the domain, nor in the anti-domain, as the corresponding demand has been lost. Consequently, the pattern match for the function `add` would fail and partial evaluation would get stuck. If we, however, use a demand of $\uparrow \{(\mathit{ldiag}, \{\})\}$, this additional demand information is preserved and the label will be part of the anti-domain of the partial value. Consequently, the pattern match can succeed.

In Line 17 above, the demand for the entire `let` expression is given as $\{(\mathit{shape}, \uparrow \{\})\}$, *i.e.*, we want to know the full value of the `shape` component of the result of evaluating the `let` expression. This, using the inference described above, leads ultimately to a demand of $\uparrow \{\}$ for the `shape` components of the values `A` and `B` in lines 11 and 13. Furthermore, in order to decide the pattern match, we need to compute the presence, but not the value, of the `ldiag` label for both values. This is encoded by the demand $\emptyset$ for the corresponding expressions in the definition of the two values in lines 12 and 14.

The function `add` in the above example is uniform with respect to demand derivation. To compute the `shape` component of the result, the `shape` component of the arguments is required. The same is true for the other auxiliary computations in the above example. However, my approach is not limited to uniform functions. As a demonstration, consider the function `shape` that computes the shape vector of an array and the corresponding demand derivations.

```
1 let
    fun shape A{ shape , rank}^{(rank,↑{})}
3     = (A.shape){ shape=[A^{(rank,↑{})}.rank]^↑{} ,
                   rank=1}^{(shape,↑{})}
5   val A = [ 1, 0, 0, 1]{ shape=[ 2, 2] , rank=2^↑{}}^{(rank,↑{})}
  in
7   (shape A^{(rank,↑{})})^{(shape,↑{})}
  end^{(shape,↑{})}
```

In the above code, I define a function `shape` in Line 2 that, given an array as argument, yields a new array which encodes the shape vector of the argument. In general, the shape vector of an array is a vector of the length of the array's shape. Thus, the result is an array with a rank of 1 and a one element shape component containing the rank of the argument as its only value. In the definition of the function `shape`, this is expressed by the expression `[A.rank]`.

In Line 7 above, I apply the function `shape` to the value `A` as defined in Line 5. As the array `A` has the shape `[ 2, 2]` and the rank 2, the result of evaluating the overall `let` expression using the semantics presented in Chapter 3 is the value *{(val,[2,2]), (shape,[2]), (rank,1)}*.

Now assume we want to compute only the `shape` component of the `let` expression. I have already annotated the corresponding demands in the above example. To compute the `shape` component of the `let` expression, we have to compute the `shape` component of the result of the application of `shape` to the value `A` in the body of the `let` expression.

As the function `shape` has only one instance, we do not get any demands due to the pattern matching, as all labels are included in all pattern. However, we have to propagate the demand $\{(\mathit{shape}, \uparrow \{\})\}$ to the defining expression of the function `shape` in Line 3. As we need to know the full value of the `shape` component, the corresponding element of the record expression in Line 3 gets a demand of $\uparrow\{\}$. This demand then propagates though the selection expression `A.rank` to the parameter `A`. As we select the `rank` component of `A`, the parameter `A` itself has a demand of $\{(\mathit{rank}, \uparrow\{\})\}$. This is then propagated to the definition of the argument `A` in the function application. As can be seen in Line 5, the corresponding record expression carries the demand $\{(\mathit{rank}, \uparrow\{\})\}$, as well. Thus, the expression of the `rank` component of the record expression in Line 5 has a demand of $\uparrow\{\}$.

As this example demonstrates, the inference of demands works for non-uniform functions, as well. In the above example, to compute the `shape` component of the result of an application of the function `shape`, we need to compute the `rank` component of the argument.

So far, my examples only contained a single application of each function. In the presence of multiple applications of the same function in different contexts, the derivation of demands becomes more complex. For a single function, different demands on the results may arise and thus different demands on the arguments may be inferred, depending on the context of the function application. As an example, consider applying the function `shape` twice.

```
  let
2   fun shape A{ shape , rank}
      = (A.shape){ shape=[A.rank],
4                  rank=1}
    val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
6 in
    (shape (shape A))
8 end
```

In Line 7 above, I first compute the shape vector for the array `A`. As in the previous example, evaluating the inner application of the function `shape` yields the value *{(val,[2,2]), (shape,[2]), (rank,1)}*. Next, I apply the function `shape` to this result, which yields the final result *{(val,[2]), (shape,[1]), (rank,1)}* for the entire `let` expression.

Now assume we only want to compute the `shape` component of the `let` expression. As the previous example has shown, the function `shape` is non-uniform. To compute the `shape` component of its result, the `rank` component of its argument is required. Thus, if we apply the function `shape` to the result of an application of the same function, we need to compute the `rank` component of the result of the inner application of `shape`. This, however, leads to a different set of demands for the sub-expressions of the definition of the function `shape`.

To encode different sets of demands depending on the calling context, I introduce a demand context to my annotations. Below are the derivations for the example above.

```
  let
2   fun shape A{ shape , rank}^{(shape,↑{})}⊢{(rank,↑{})},{(rank,↑{})}⊢∅
      = (A.shape){ shape=[A^{(shape,↑{})}⊢{(rank,↑{})}.rank]^{(shape,↑{})}⊢↑{},
4                  rank=1^{(rank,↑{})}⊢↑{}}^{(shape,↑{})}⊢{(shape,↑{})},{(rank,↑{})}⊢{(rank,↑{})}
    val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
6 in
```

$\texttt{(shape (shape A}^{\emptyset}\texttt{)}^{\{\}\vdash\{(rank,\uparrow\{\})\}}\texttt{)}^{\{\}\vdash\{(shape,\uparrow\{\})\}}$

8 $\texttt{end}^{\{\}\vdash\{(shape,\uparrow\{\})\}}$

A demand annotation of the form $e^{C\vdash D}$ is to be read as: Given a demand C on the result of the current function, the expression e needs to be evaluated up to the demand D. For top-level expressions, I use the empty context demand $\{\}$. Thus, if we are interested in the `shape` component of the above `let` expression, we get a demand of $\{\} \vdash \{(shape, \uparrow \{\})\}$ as annotated in Line 8. This demand is then propagated as before. The only difference is that whenever I enter a function definition, I update the derivation context to the current demand on the result of the application of the function that is analysed.

Above, I first infer the demand for the outer application of the function `shape`. This leads to the demand annotation $\{(shape, \uparrow \{\})\} \vdash \{(shape, \uparrow \{\})\}$ in Line 4. Analogously to the previous example, this demand is then propagated to the parameter `A`, leading to a demand annotation of $\{(shape, \uparrow \{\})\} \vdash \{(rank, \uparrow \{\})\}$ in Line 2. Thus, we get a demand of $\{\} \vdash \{(rank, \uparrow \{\})\}$ on the result of the inner application of the function `shape`.

This demand triggers a second round of demand analysis for the function `shape`. During this round of analysis, I use the demand context $\{(rank, \uparrow \{\})\} \vdash \{(rank, \uparrow \{\})\}$. As can be seen in Line 4, the `rank` component of the result of the function `shape` is defined as the constant `1`. Consequently, to compute the `rank` component of the result of an application of `shape`, no information about its argument is required. This is annotated accordingly in Line 2 by the demand $\{(rank, \uparrow \{\})\} \vdash \emptyset$. This empty demand then propagates to the argument `A` of the function application in Line 7.

As these demand annotations show, in order to compute the `shape` component of the above `let` expression, the expression in Line 5 needs not to be evaluated. This is a generic result. In general, the shape of the shape of an array is independent of the array itself.

For non-recursive functions like the two examples shown so far, this demand analysis suffices and always terminates. As the set of labels used in any LREC$_C$ expression is finite, so is the set of demands that can arise. Thus, for each function, only a finite set of different demands needs to be inferred. However, for recursive functions, the simple demand analysis as motivated so far does not suffice. As an example, consider the following expression that defines a function `addn` that adds a matrix n-times to itself.

```
   let
2    fun add A{ shape=sA, rank=rA} B{ shape=sB, rank=rB}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}
4    fun addn A{shape, rank} n{}
       = if (n = 0)
6          A
           (addn (add A A) (n - 1))
8    val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
   in
10   (addn A 3)
   end
```

In Line 2 above, I define a function `add` for adding two matrices in the usual way. I have removed the additional instance for lower diagonal matrices to keep the example shorter. Next, in Line 4, I define a function `addn` that expects a matrix as first argument and an integer value as second argument. If `n` is zero, the function `addn` returns its first argument. Otherwise, in Line 7, the function `addn` is applied recursively to the result of adding the matrix to itself and the value of `n` decremented by one. Finally, the function `addn` is applied in Line 10 to a matrix `A` as defined in Line 8 and the integer value `3`.

Using the semantics for full evaluation presented in Section 3.3, the above program evaluates to the value *{(val,[4,0,0,4]),(shape,[2,2]),(rank,2)}*. Now, assume we are only interested in the `shape` component of the result. To partially evaluate the above example to this respect only, we first have to derive the corresponding demands on all sub-expressions. Below, I give an accordingly annotated version of the above example.

```
1  let
     fun add A{ shape=sA, rank=rA}{}⊢{}
3             B{ shape=sB, rank=rB}{}⊢{}
       = (vect_add !A !B){ shape=any(sA sB), rank=any(rA rB)}{}⊢{}
5    fun addn A{shape, rank}{(shape,↑{})}⊢{(shape,↑{})}
                n{}}{(shape,↑{})}⊢↑{}
7      = if (n = 0){(shape,↑{})}⊢↑{}
           A{(shape,↑{})}⊢{(shape,↑{})}
9          (addn (add A A){(shape,↑{})}⊢{}
                 (n - 1){(shape,↑{})}⊢{}
11         ){(shape,↑{})}⊢{(shape,↑{})}
     val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}
13 in
     (addn A 3){}⊢{(shape,↑{})}
15 end{}⊢{(shape,↑{})}
```

To start off, I have annotated the corresponding demand $\{\} \vdash \{(\textit{shape}, \uparrow \{\})\}$ for the entire `let` expression. This demand is then propagated as before. Thus, for the body expression of the single instance of the function `addn` in Line 11 we get the demand $\{(\textit{shape}, \uparrow \{\})\} \vdash \{(\textit{shape}, \uparrow \{\})\}$. For the conditional expression, this demand is then propagated to the then and else branches. Furthermore, as the demand on the conditional is non-empty, we have to ensure that the value of the predicate expression is available during partial evaluation. This yields a demand of $\{(\textit{shape}, \uparrow \{\})\} \vdash \uparrow \{\}$ for the sub-expression `(n = 0)`.

For the then branch, the demand directly propagates to the first argument of the function `addn`. The derivation for the else branch, however, is more complex. The else branch is defined as a recursive application of the function `addn` with a demand of $\{(\textit{shape}, \uparrow \{\})\} \vdash \{(\textit{shape}, \uparrow \{\})\}$. However, we are currently inferring just this demand. Thus, propagating this demand to the function definition would yield a non-terminating loop. Instead, as no other information is available, I propagate the empty demand $\{(\textit{shape}, \uparrow \{\})\} \vdash \{\}$ to both arguments and continue the analysis with this information.

After propagating the demands through the non-recursive function `add`, this yields an overall demand of $\{(\textit{shape}, \uparrow \{\})\} \vdash \{(\textit{shape}, \uparrow \{\})\}$ for the first argument of the function `addn` and a demand of $\{(\textit{shape}, \uparrow \{\})\} \vdash \uparrow \{\}$ for the second argument. Thus, the assumed demand of $\{(shape, \uparrow \{\})\} \vdash \{\}$ for the recursive application was actually to weak. Using this information, we can now start a second round of demand inference for the function addn. The corresponding demands are given below.

```
1   let
      fun add A{ shape=sA, rank=rA}^{{(shape,↑{})}⊢{(shape,↑{})}}
3             B{ shape=sB, rank=rB}^{{(shape,↑{})}⊢{(shape,↑{})}}
        = (vect_add !A !B){ shape=any(sA sB),
5                           rank=any(rA rB)}^{{(shape,↑{})}⊢{(shape,↑{})}}
      fun addn A{shape, rank}^{{(shape,↑{})}⊢{(shape,↑{})}}
7              n{}}^{{(shape,↑{})}⊢↑{}}
        = if (n = 0)^{{(shape,↑{})}⊢↑{}}
9            A^{{(shape,↑{})}⊢{(shape,↑{})}}
             (addn (add A A)^{{(shape,↑{})}⊢{(shape,↑{})}}
11                 (n - 1)^{{(shape,↑{})}⊢↑{}}
             )^{{(shape,↑{})}⊢{(shape,↑{})}}
13    val A = [ 1, 0, 0, 1]{ shape=[ 2, 2], rank=2}^{{}⊢{(shape,↑{})}}
    in
15    (addn A 3^{{}⊢↑{}})^{{}⊢{(shape,↑{})}}
    end^{{}⊢{(shape,↑{})}}
```

As can be seen in Line 10 above, I have now used the new demands for the function arguments of the function addn in the recursive application. This leads to a new demand to the function add, as well. After propagating the demand through the function add, we get a demand of $\{(shape, \uparrow \{\})\} \vdash \{(shape, \uparrow \{\})\}$ for the first argument of addn. Thus, after the second round of analysis, the overall demands of both arguments have not changed. Consequently, we have reached a fix-point.

Such a fix-point for the demand analysis for recursive functions always exists. Firstly, demand analysis is monotonically increasing with respect to the derived demands by nature of the demand analysis. I always propagate the entire demand or parts of it into sub-expressions. Thus, if a stronger demand is propagated, the corresponding demand for the sub-expressions cannot be weaker. The only exception to this is the demand for the predicate of conditionals. In this case, I use the demand $\uparrow \{\}$ if the demand on the overall conditional is non-empty. However, even in this case the propagated demand for a stronger demand cannot be weaker.

Secondly, the demand that can arise for function arguments is bounded. The maximum demand that can arise is a demand on all labels that occur in the corresponding LRec$_C$ expression. As this set of labels is finite for any LRec$_C$ expression, such a maximum exists.

Using the final demand inferred for the arguments of the function addn, we can now compute the demands for the arguments in Line 15 above. This yields a final demand of $\{\} \vdash \uparrow \{\}$ for the sub-expression 3 and a demand of $\{\} \vdash \{(shape, \uparrow \{\})\}$ for the matrix A. Thus, to compute the shape component of the let expression, we need to compute the shape component of the matrix A and the value of the expression 3.

$$
\begin{array}{lcl}
\textit{context} & \Rightarrow & \textbf{demand} \vdash \textbf{demand} \\[1em]
\textit{demand} & \Rightarrow & \textbf{record} \\
 & | & \uparrow \{\ \textbf{record}\ \} \\[1em]
\textit{record} & \Rightarrow & \{\ \mathit{[}\ (\ \textbf{label}\ ,\ \textbf{demand}\ ) \\
 & & \mathit{[}\ ,\ (\ \textbf{label}\ ,\ \textbf{demand}\ )\ \mathit{]}^{*}\ \mathit{]}\ \}
\end{array}
$$

Figure 6.1.. Syntax of demand annotations in extended Backus Naur form.

The result of demand analysis demonstrates a main difference between my approach and classical type systems. In contrast with type systems, my approach does not approximate the type of a conditional expression by using the common type or a least upper bound in the setting of subtyping but computes which branch is actually chosen at runtime. For the above example, this would not be necessary as both branches yield the same `shape` component. However, in general this needs not to be the case. In these cases, classical type systems would fail whereas my approach still yields the desired information.

This completes the examples for demand analysis. The demand analysis with fixpoint iteration for recursive functions as motivated so far suffices to derive demands for all LREC$_{\mathrm{C}}$ expressions without nested records. Next, I will give a formal definition of demand analysis.

6.1.2. A Formal Definition of Demand Inference

I start off by formally specifying the syntax for demand annotations in Figure 6.1. The production rule *context* defines the syntax for demand contexts as annotated to subexpressions. A demand context consists of a demand, followed by a $\vdash$-symbol and a further demand. The syntax for demands in turn is defined by the production rule *demand*. A demand can either be a record demand or the demand for full evaluation $\uparrow \{\mathit{record}\}$ where *record* denotes a record demand. Lastly, a record demand is defined by the production rule *record* as a set of (label,demand) pairs. I will refer to the set of demands that can be produced using the production rule *demand* in Figure 6.1 as $\mathscr{D}$ in the following. Furthermore, I will use $\mathscr{D}_r$ to refer to the set of record demands that can be produced by using the production rule *record*.

To simplify further definitions, I first introduce the notion of a sub-demand as follows.

Definition 6.1.1 (Subdemand)**.** *The sub-demand* $\mathrm{elem}_d : \mathscr{D} \times \mathscr{L} \to \mathscr{D}$ *is defined as*

$$
\mathrm{elem}_d(\delta, l) := \begin{cases} \{\delta'\} & \text{if } \delta \in \mathscr{D}_r \text{ and } (l, \delta') \in \delta, \\ \mathrm{elem}_d(\delta', l) & \text{if } \delta \equiv \uparrow \{\delta'\}, \\ \{\} & \text{otherwise.} \end{cases}
$$

Thus, the sub-demand of a record demand is the demand of the corresponding element. For the full-evaluation demand $\uparrow \{\delta\}$, the sub-demand is the corresponding sub-demand of the contained demand δ.

Similarly to record values, I define the notion of the domain of a demand.

Definition 6.1.2 (Domain of a Demand)**.** *The domain of a demand* $\mathrm{dom}_d : \mathscr{D} \to \mathscr{L}$ *is defined as*

$$\mathrm{dom}_d(\delta) := \begin{cases} \{l_1, \ldots, l_n\} & \textit{if } \delta \equiv \{(l_1, \delta_1), \ldots, (l_n, \delta_n)\}, \\ \mathrm{dom}_d(\delta') & \textit{if } d \equiv \uparrow \{\delta'\}. \end{cases}$$

Using this language for demand annotations, I can now define the notion of *demand satisfaction*, *i.e.*, I can define which partial values satisfy a given demand with respect to the corresponding full value.

Definition 6.1.3 (Demand Satisfaction)**.** *Given a value $v \in \mathscr{V}$ and a partial value $v' \in \mathscr{V}_p$ with $v \sqsubseteq v'$. For any demand $\delta \in \mathscr{D}$, we say that v' satisfies the demand δ with respect to v, or $v \overset{\delta}{\sqsubseteq} v'$ for short, if one of the following statements holds:*

1. $v' = ?$ *and* $\delta = \{\}$
2. $v \notin \mathscr{R}$ *and* $v' \neq ?$
3. $v \in \mathscr{R}$ *and* $v' \in \mathscr{R}_p$ *and* $d \in \mathscr{D}_r$ *and* $dom(v) \cap \mathrm{dom}_d(\delta) \setminus dom_p(v') = \emptyset$ *and* $\mathrm{dom}_d(\delta) \subseteq dom_p(v') \cup \overline{dom}_p(v')$ *and* $\forall l \in dom_p(v') : elem(v, l) \overset{\mathrm{elem}_d(\delta,\, l)}{\sqsubseteq} elem_p(v', l)$
4. $v \in \mathscr{R}$ *and* $v' \in \mathscr{R}_p$ *and* $\delta = \uparrow \{\delta'\}$ *for some* $\delta' \in \mathscr{D}$ *and* $dom(v) = dom_p(v')$ *and* $\mathrm{dom}_d(\delta) \setminus dom(v) \subseteq \overline{dom}_p(v')$ *and* $\forall l \in dom(v) : elem(v, l) \overset{\mathrm{elem}_d(\delta,\, l)}{\sqsubseteq} elem_p(v', l)$.

In the above definition, the first statement covers demand satisfaction for the special value *?*. It only satisfies the empty demand.

The second statement defines demand satisfaction for non-record values. A non-record partial value that is not the special *?*-value satisfies all demands.

Next, the third statement defines demand satisfaction for record values. A partial record value satisfies a record demand if the domain of the partial value contains all labels that are part of both the record demand and the domain of the full value. Furthermore, the anti-domain of the partial record value needs to contain all labels that are part of the demand but not in the domain of the full value. Lastly, the elements of the partial value need to satisfy the corresponding sub-demands of the record demand.

Finally, the last statement defines demand satisfaction for the demand for full evaluation $\uparrow$. Such a demand is satisfied if the partial value contains all labels that the full value contains and if each element of the partial value satisfies the corresponding element of the demand with respect to the same element of the full value. Additionally, I require that the anti-domain of the partial value covers all labels required by the demand that are not part of the domain of the partial value. Note that, by definition of sub-demands, this does not automatically require all elements of the partial value to be fully evaluated, as well. If a full evaluation of components of a record value is desired, this needs to be explicitly encoded in the record component of the full-evaluation demand.

A further important operation on demands that will be required for the demand analysis is the union of two demands.

Definition 6.1.4 (Union of Demands). *The union on demands $\uplus : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ is defined as follows:*

$$\delta_1 \uplus \delta_2 := \begin{cases} \uparrow \{\delta_1' \uplus \delta_2'\} & \text{if } \delta_1 = \uparrow \{\delta_1'\} \text{ and } \delta_2 = \uparrow \{\delta_2'\}, \\ \uparrow \{\delta_1' \uplus \delta_2\} & \text{if } \delta_1 = \uparrow \{\delta_1'\} \text{ and } \delta_2 \in \mathscr{D}_r, \\ \uparrow \{\delta_1 \uplus \delta_2'\} & \text{if } \delta_1 \in \mathscr{D}_r \text{ and } \delta_2 = \uparrow \{\delta_2'\}, \\ \left\{ \begin{array}{l} (l, \mathrm{elem}_d(\delta_1, l) \uplus \mathrm{elem}_d(\delta_2, l)) \\ \mid l \in \mathrm{dom}_d(\delta_1) \cup \mathrm{dom}_d(\delta_2) \end{array} \right\} & \text{if } \delta_1, \delta_2 \in \mathscr{D}_r. \end{cases}$$

Next, I will give some important properties of the union of demand. Firstly, the union of demands is commutative.

Theorem 6.1.1 (Commutativity of Union of Demands). *For two demands $\delta_1, \delta_2 \in \mathscr{D}$ the following statement holds*

$$\delta_1 \uplus \delta_2 = \delta_2 \uplus \delta_1$$

Proof. The above property follows directly from the symmetry of the cases in the definition of the union of demands in Definition 6.1.4. □

The next two theorems explore the relationship between the union on demands, and the domain and satisfaction of demands, respectively. First, I will show the following relation between the domain of a demand and the domain of the union of demands.

Theorem 6.1.2 (Distributivity of Domain over Union of Demands). *Given the two demands $\delta_1, \delta_2 \in \mathscr{D}$. Then the following statement holds:*

$$\mathrm{dom}_d(\delta_1) \cup \mathrm{dom}_d(\delta_2) = \mathrm{dom}_d(\delta_1 \uplus \delta_2)$$

Proof. I will show that the above statement holds by case analysis for the demands δ_1 and δ_2. We have to consider four cases.

$\delta_1 \in \mathscr{D}_r$ **and** $\delta_2 \in \mathscr{D}_r$. Then $\mathrm{dom}_d(\delta_1) \cup \mathrm{dom}_d(\delta_2) = \mathrm{dom}_d(\delta_1 \uplus \delta_2)$ follows directly from the definition of the union of demands.

$\delta_1 \notin \mathscr{D}_r$ **and** $\delta_2 \notin \mathscr{D}_r$. Then $\delta_1 = \uparrow \{\delta_1'\}$ and $\delta_2 = \uparrow \{\delta_2'\}$ for some $\delta_1', \delta_2' \in \mathscr{D}_r$. Furthermore, by definition of the union of demands, we know that $\delta_1 \uplus \delta_2 = \uparrow \{\delta_1' \uplus \delta_2'\}$. Consequently, from the definition of the domain of a demand, it follows that $\mathrm{dom}_d(\delta_1 \uplus \delta_2) = \mathrm{dom}_d(\delta_1' \uplus \delta_2')$. This is another instance of case one above.

$\delta_1 \notin \mathscr{D}_r$ **and** $\delta_2 \in \mathscr{D}_r$. Then $\delta_1 = \uparrow \{\delta_1'\}$ for some $\delta_1' \in \mathscr{D}_r$. It follows that $\delta_1 \uplus \delta_2 = \uparrow \{\delta_1' \uplus \delta_2\}$. Thus, $\mathrm{dom}_d(\delta_1 \uplus \delta_2) = \mathrm{dom}_d(\delta_1' \uplus \delta_2)$. This is another instance of case one above.

$\delta_1 \in \mathscr{D}_r$ **and** $\delta_2 \notin \mathscr{D}_r$. This case follows from the commutativity of the union on demands and the case above.

Overall, it follows that $\mathrm{dom}_d(\delta_1) \cup \mathrm{dom}_d(\delta_2) = \mathrm{dom}_d(\delta_1 \uplus \delta_2)$ holds. □

A further important property of the union of demands is that it always yields a demand that is not weaker than any of its argument demands.

Theorem 6.1.3 (Monotonicity of Union of Demands). *Given two demands $\delta_1, \delta_2 \in \mathscr{D}$. Then for all $v \in \mathscr{V}$ and $v' \in \mathscr{V}_p$ with $v \sqsubseteq v'$ the following statement holds:*

$$v \overset{\delta_1 \uplus \delta_2}{\sqsubseteq} v' \Rightarrow v \overset{\delta_1}{\sqsubseteq} v' \wedge v \overset{\delta_2}{\sqsubseteq} v'$$

Proof. Given $v \in \mathscr{V}$ and $v' \in \mathscr{V}_p$. Assume that $v \overset{\delta_1 \uplus \delta_2}{\sqsubseteq} v'$ holds. With Theorem 6.1.1 on the preceding page, it suffices to show that then $v \overset{\delta_1}{\sqsubseteq} v'$ holds, as well. I will show this by induction on the nesting depth of the demand $\delta_1 \uplus \delta_2$.

To start off, assume $\delta_1 \uplus \delta_2$ is a non-nested demand. Then $\delta_1 \uplus \delta_2 = \{\}$. Consequently, by definition of the union of demands, $\delta_1 = \{\}$ and $\delta_2 = \{\}$. Thus, given that $v \overset{\delta_1 \uplus \delta_2}{\sqsubseteq} v'$, the statement $v \overset{\delta_1}{\sqsubseteq} v'$ holds.

For the inductive step, assume that the above statement holds for all demands with a nesting depth smaller than n for some $n \in \mathbb{N}$. Let $\delta_1, \delta_2 \in \mathscr{D}$ such that $\delta_1 \uplus \delta_2$ has a nesting depth of $n + 1$.

For $v \notin \mathscr{R}$ and $v' \notin \mathscr{R}_p$, $v \overset{\delta_1}{\sqsubseteq} v'$ follows trivially from the definition of demand satisfaction. For all $v \notin \mathscr{R}$ all $v' \in \mathscr{V}_p$ with $v \sqsubseteq v'$ satisfy any demand. In case that $v' \notin \mathscr{R}_p$, no non-empty demand can be satisfied.

Now assume $v \in \mathscr{R}$ and $v' \in \mathscr{R}_p$. We have to consider two cases. Firstly, assume $\delta_1 \uplus \delta_2 \in \mathscr{D}_r$. Then by definition of the union of demands, $\delta_1 \in \mathscr{D}_r$ and $\delta_2 \in \mathscr{D}_r$, as well. Furthermore, from $v \overset{\delta_1 \uplus \delta_2}{\sqsubseteq} v'$, we know that $\mathrm{dom}(v) \cap \mathrm{dom}_d(\delta_1 \uplus \delta_2) \setminus \mathrm{dom}_p(v') = \emptyset$ and $\mathrm{dom}_d(\delta_1 \uplus \delta_2) \subseteq \mathrm{dom}_p(v') \cup \overline{\mathrm{dom}}_p(v')$ and $\forall l \in \mathrm{dom}_p(v') : \mathrm{elem}(v, l) \overset{\mathrm{elem}_d(\delta_1 \uplus \delta_2,\, l)}{\sqsubseteq} \mathrm{elem}_p(v', l)$ hold. From Theorem 6.1.2 on the facing page, we know that $\mathrm{dom}_d(\delta_1) \subseteq \mathrm{dom}_d(\delta_1 \uplus \delta_2)$. Thus, it follows that $\mathrm{dom}(v) \cap \mathrm{dom}_d(\delta_1) \setminus \mathrm{dom}_p(v') = \emptyset$ and $\mathrm{dom}_d(\delta_1) \subseteq \mathrm{dom}_p(v') \cup \overline{\mathrm{dom}}_p(v')$. Next, from the definition of sub-demands (cf. Definition 6.1.1 on page 116), we know that for all $l \in \mathscr{L}$ the demand $\mathrm{elem}_d(\delta_1, l)$ has a nesting depth smaller than $n + 1$. Consequently, we can follow with the inductive assumption from $\forall l \in \mathrm{dom}_p(v') : \mathrm{elem}(v, l) \overset{\mathrm{elem}_d(\delta_1 \uplus \delta_2,\, l)}{\sqsubseteq} \mathrm{elem}_p(v', l)$ that $\forall l \in \mathrm{dom}_p(v') : \mathrm{elem}(v, l) \overset{\mathrm{elem}_d(\delta_1,\, l)}{\sqsubseteq} \mathrm{elem}_p(v', l)$ holds. Thus, overall $v \overset{\delta_1}{\sqsubseteq} v'$ holds.

Secondly, we have to consider the case that $\delta_1 \uplus \delta_2 = \uparrow \{\delta'\}$ with $\delta' \in \mathscr{D}$. Then, by the definition of the union of demands, either $\delta_1 = \uparrow \{\delta_1'\}$ for some $\delta_1' \in \mathscr{D}$, or $\delta_1 \in \mathscr{D}_r$. We know from $v \overset{\delta_1 \uplus \delta_2}{\sqsubseteq} v'$ that $\mathrm{dom}(v) = \mathrm{dom}_p(v')$ holds and that for all $l \in \mathrm{dom}(v)$ the statement $\mathrm{elem}(v, l) \overset{\mathrm{elem}_d(\delta_1 \uplus \delta_2,\, l)}{\sqsubseteq} \mathrm{elem}_p(v', l)$ holds and that $\mathrm{dom}_d(\delta) \setminus \mathrm{dom}(v) \subseteq \overline{\mathrm{dom}}_p(v')$ holds. Again, we know from the definition of sub-demands that for all $l \in \mathscr{L}$ the demand

$\mathrm{elem}_d(\delta_1 \uplus \delta_2, l)$ has a nesting depth smaller than $n+1$. Consequently, it follows with the inductive assumption that for all $l \in \mathrm{dom}(v)$ the statement $\mathrm{elem}(v,l) \overset{\mathrm{elem}_d(\delta_1,l)}{\sqsubseteq} \mathrm{elem}_p(v',l)$ holds. For the case that $\delta_1 =\uparrow \{\delta_1'\}$, it remains to be shown that $\mathrm{dom}_d(\delta_1) \setminus \mathrm{dom}(v) \subseteq \overline{\mathrm{dom}}_p(v')$ holds. This follows directly with Theorem 6.1.2 on page 118.

Consider the other case that $\delta_1 \in \mathscr{D}_r$. Here, it remains to be shown that $\mathrm{dom}(v) \cap \mathrm{dom}_d(\delta_1) \setminus \mathrm{dom}_p(v') = \emptyset$ and $\mathrm{dom}_d(\delta_1) \subseteq \mathrm{dom}_p(v') \cup \overline{\mathrm{dom}}_p(v')$ hold. The former follows from $\mathrm{dom}(v) = \mathrm{dom}_p(v')$. The latter follows with Theorem 6.1.2 on page 118 from $\mathrm{dom}_d(\delta) \setminus \mathrm{dom}(v) \subseteq \overline{\mathrm{dom}}_p(v')$. $\square$

With the syntax for demand annotations and the notion of demand satisfaction in place, I can now formally define the demand analysis for LREC$_\mathrm{C}$ without nested record expressions. I define the analysis as a code rewriting scheme from LREC$_\mathrm{C}$ to LREC$_\mathrm{D}$, an extension of LREC$_\mathrm{C}$ with demand annotations. First, I give the syntax of LREC$_\mathrm{D}$ in Figure 6.2 on the facing page.

As can be seen, the production rules for all expressions in LREC$_\mathrm{D}$, *i.e.*, all tokens in the rule *expression*, carry an additional token *contexts* compared to the rules for LREC$_\mathrm{C}$ presented in Figure 3.6 on page 50. The token *contexts* is defined in the corresponding production rule as a potentially empty set of demand contexts as defined by the production rule *context* in Figure 6.1 on page 116. Furthermore, I have extended the pattern for function arguments by a set of demand contexts, as well.

Using this extended syntax, I can now define demand inference. The corresponding rewriting scheme $\mathscr{A}$ is given in Figure 6.3 on page 122. For convenience of specification, I will in the following treat context annotations as sets where a missing context annotation corresponds to an empty set.

Given an expression in LREC$_\mathrm{D}$ with empty sets of contexts, the rewriting scheme $\mathscr{A}$ yields a corresponding expression in LREC$_\mathrm{D}$ with annotated demands. It is worth noting here that every expression in LREC$_\mathrm{C}$ is syntactically identical to a semantically equivalent expression in LREC$_\mathrm{D}$ with empty demand annotations. The rewriting is parametrised, apart from the expression to rewrite, by a function environment $\mathscr{F}$, a variable environment $\mathscr{E}_A$ and the current context and demand for the overall expression $\delta_c \vdash \delta$.

The function environment $\mathscr{F}$ is similar in structure to the function environment $\mathscr{F}$ used in the definition of the semantics and partial semantics for LREC$_\mathrm{C}$ in Sections 3.3 and 5.3. It, as well, is nested to support the scoping of function definitions by nested `let` constructs. It consists of a tuple of the outer, potentially nested, function environment and a set of local instance definitions. I will discuss the structure of the set of local instances in more detail when describing the demand derivation for function definitions and function applications.

The variable environment $\mathscr{E}_A$ is a set of (identifier,contexts) pairs, where the identifier corresponds to a variable used in the current expression and the contexts give the demands that arise for this variable in different demand contexts.

Overall, the demand analysis presented in Figure 6.3 on page 122 propagates demands starting from the top-level into each sub-expression. Contrary to the rules for evaluation, the rules for demand analysis progress bottom-up, *i.e.*, demands are propagated from the

```
program       ⇒  expression

expression    ⇒  record | selection | unit | any
              |  identifier contexts | boolean | conditional
              |  guard | witness | equal | let | application

record        ⇒  { [ element [ , element ]* ] } contexts

element       ⇒  label = expression

selection     ⇒  expression . label contexts

unit          ⇒  ~ contexts

any           ⇒  any ( [ expression ]+ ) contexts

boolean       ⇒  true contexts | false contexts

conditional   ⇒  if expression expression expression contexts

guard         ⇒  guard ( expression expression ) contexts

witness       ⇒  witness ( expression [ expression ]+ ) contexts

equal         ⇒  ( expression = expression ) contexts

let           ⇒  let [ definition ]* in expression end contexts

definition    ⇒  relation | value | function

relation      ⇒  rel label <: label

value         ⇒  val identifier = expression

function      ⇒  fun [ instance ]+

instance      ⇒  [ pattern ]+ = expression

pattern       ⇒  identifier { [ label [ , label ]* ] } contexts

application   ⇒  ( identifier [ expression ]+ ) contexts

contexts      ⇒  [ context [ , context ]* ]
```

Figure 6.2.. Syntax of LREC$_D$ in extended Backus Naur form.

(True) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{true}\ \mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}, \mathscr{E}_A,\ \texttt{true}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

(False) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{false}\ \mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}, \mathscr{E}_A,\ \texttt{false}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

(Unit) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{~}\mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}, \mathscr{E}_A,\ \texttt{~}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

(Var) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \alpha\ \mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}, \mathscr{E}_A \xleftarrow{\mathscr{D}} (\alpha, \delta_c \vdash \delta),\ \alpha\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

(Equal) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{(}e_1\ \mathscr{S}_1\ \texttt{=}\ e_2\ \mathscr{S}_n\texttt{)}\ \mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}'', {\mathscr{E}_A}'',\ \texttt{(}e'_1\ \mathscr{S}'_1\ \texttt{=}\ e'_2\ \mathscr{S}'_2\texttt{)}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}', {\mathscr{E}_A}', e'_1\ \mathscr{S}'_1 & := & \mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \Uparrow(\delta),\ e_1\ \mathscr{S}_1\ \right\rrbracket \\ \mathscr{F}'', {\mathscr{E}_A}'', e'_2\ \mathscr{S}'_2 & := & \mathscr{A}\left\llbracket \mathscr{F}', {\mathscr{E}_A}', \delta_c \vdash \Uparrow(\delta),\ e_2\ \mathscr{S}_2\ \right\rrbracket \end{array}$$

(Record) $\mathscr{A}\left\llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{\{}l_1\texttt{=}e_1\ \mathscr{S}_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n\ \mathscr{S}_n\texttt{\}}\ \mathscr{S}\ \right\rrbracket$

$\leadsto\ (\mathscr{F}^n, {\mathscr{E}_A}^n,\ \texttt{\{}l_1\texttt{=}e'_1\ \mathscr{S}'_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e'_n\ \mathscr{S}'_n\texttt{\}}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}^0 & := & \mathscr{F} \\ {\mathscr{E}_A}^0 & := & \mathscr{E}_A \\ \mathscr{F}^{i+1}, {\mathscr{E}_A}^{i+1}, e'_{i+1}\ \mathscr{S} + i + 1' & := & \mathscr{A}\left\llbracket \mathscr{F}^i, {\mathscr{E}_A}^i, \delta_c \vdash \mathrm{elem}_d(\delta, l_{i+1}),\ e_{i+1}\ \mathscr{S}_{i+1}\ \right\rrbracket \end{array}$$

Figure 6.3.. Demand analysis scheme $\mathscr{A}$ for translating expressions in LRec$_\mathrm{C}$ without nested records into expressions in LRec$_\mathrm{D}$ with demand annotations.

(Selection) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ e_b\ \mathscr{S}_b\texttt{.}l\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}', {\mathscr{E}_A}',\ e_b'\ \mathscr{S}_b'\texttt{.}l\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where
$\mathscr{F}', {\mathscr{E}_A}', e_b'\ \mathscr{S}_b' \ := \ \mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \mathrm{nest}(l, \delta),\ e_b\ \mathscr{S}_b\ ]\!]$

(Any) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{any(}e_1\ \mathscr{S}_1\ \cdots\ e_n\ \mathscr{S}_n\texttt{)}\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}^n, {\mathscr{E}_A}^n,\ \texttt{any(}e_1'\ \mathscr{S}_1'\ \cdots\ e_n'\ \mathscr{S}_n'\texttt{)}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where
$\mathscr{F}^0 \ := \ \mathscr{F}$
${\mathscr{E}_A}^0 \ := \ \mathscr{E}_A$
$\mathscr{F}^{i+1}, {\mathscr{E}_A}^{i+1}, e_{i+1}'\ \mathscr{S}_{i+1} \ := \ \mathscr{A}\ [\![\mathscr{F}^i, {\mathscr{E}_A}^i, \delta_c \vdash \delta,\ e_{i+1}\ \mathscr{S}_{i+1}\ ]\!]$

(Cond) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{if}\ e_p\ \mathscr{S}_p\ e_t\ \mathscr{S}_t\ e_e\ \mathscr{S}_e\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}^3, {\mathscr{E}_A}^3,\ \texttt{if}\ e_p'\ \mathscr{S}_p'\ e_t'\ \mathscr{S}_t'\ e_e'\ \mathscr{S}_e'\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where
$\mathscr{F}', {\mathscr{E}_A}', e_p'\ \mathscr{S}_p' \ := \ \mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \Uparrow(\delta),\ e_p\ \mathscr{S}_p\ ]\!]$
$\mathscr{F}'', {\mathscr{E}_A}'', e_t'\ \mathscr{S}_t' \ := \ \mathscr{A}\ [\![\mathscr{F}\text{'}, \mathscr{E}_A\text{'}, \delta_c \vdash \delta,\ e_t\ \mathscr{S}_t\ ]\!]$
$\mathscr{F}^3, {\mathscr{E}_A}^3, e_e'\ \mathscr{S}_e' \ := \ \mathscr{A}\ [\![\mathscr{F}\text{''}, \mathscr{E}_A\text{''}, \delta_c \vdash \delta,\ e_e\ \mathscr{S}_e\ ]\!]$

(Guard) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{guard(}e_1\ \mathscr{S}_1\ e_2\ \mathscr{S}_2\texttt{)}\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}'', {\mathscr{E}_A}'',\ \texttt{guard(}e_1'\ \mathscr{S}_1'\ e_2'\ \mathscr{S}_2'\texttt{)}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where
$\mathscr{F}', {\mathscr{E}_A}', e_1'\ \mathscr{S}_1' \ := \ \mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ e_1\ \mathscr{S}_1\ ]\!]$
$\mathscr{F}'', {\mathscr{E}_A}'', e_2'\ \mathscr{S}_2' \ := \ \mathscr{A}\ [\![\mathscr{F}\text{'}, \mathscr{E}_A\text{'}, \delta_c \vdash \Uparrow(\delta),\ e_2\ \mathscr{S}_2\ ]\!]$

(Witness) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{witness(}e_v\ \mathscr{S}_v\ e_1\ \mathscr{S}_1\ \cdots\ e_n\ \mathscr{S}_n\texttt{)}\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}^n, {\mathscr{E}_A}^n,\ \texttt{witness(}e_v'\ \mathscr{S}_v'\ e_1'\ \mathscr{S}_1'\ \cdots\ e_n'\ \mathscr{S}_n'\texttt{)}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where
$\mathscr{F}', {\mathscr{E}_A}', e_v'\ \mathscr{S}_v' \ := \ \mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ e_v\ \mathscr{S}_v\ ]\!]$
$\mathscr{F}^{i+1}, {\mathscr{E}_A}^{i+1}, e_{i+1}'\ \mathscr{S}_{i+1}' \ := \ \mathscr{A}\ [\![\mathscr{F}^i, {\mathscr{E}_A}^i, \delta_c \vdash \Uparrow(\delta),\ e_{i+1}\ \mathscr{S}_{i+1}\ ]\!]$

Figure 6.3.. Demand analysis scheme $\mathscr{A}$ for translating expressions in LRec$_C$ without nested records into expressions in LRec$_D$ with demand annotations (contd.).

(LET) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{let}\ d_1\ \cdots\ d_n\ \texttt{in}\ e_b\ \mathscr{S}_b\ \texttt{end}\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}^4, \mathscr{E}_A{}'',\ \texttt{let}\ d'_1\ \cdots\ d'_n\ \texttt{in}\ e'\ \mathscr{S}'_b\ \texttt{end}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}', D & = & \mathrm{fun}^d(\mathscr{F}, \emptyset, (d_1, \dots, d_n)) \\ \mathscr{F}'', \mathscr{E}_A{}', e'\ \mathscr{S}'_b & = & \mathscr{A}\ [\![\mathscr{F}', \mathscr{E}_A, \delta_c \vdash \delta,\ e\ \mathscr{S}_b\ ]\!] \\ \mathscr{F}^3, \mathscr{E}_A{}'', D' & = & \mathrm{propvar}(\mathscr{F}'', \mathscr{E}_A{}', \delta_c, D) \\ \mathscr{F}^4, \{d'_1, \dots, d'_n\} & = & \mathrm{disperse}(\mathscr{F}^3, D') \end{array}$$

(AP) $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \texttt{(}f\ e_1\ \mathscr{S}_1\ \cdots\ e_n\ \mathscr{S}_n\texttt{)}\ \mathscr{S}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}^n, \mathscr{E}_A{}^n,\ \texttt{(}f\ e'_1\ \mathscr{S}'_1\ \cdots\ e'_n\ \mathscr{S}'_n\texttt{)}\ \mathscr{S} \lhd \delta_c \vdash \delta\ )$

where

$$\begin{array}{lcl} \mathscr{E}_A{}^0 & := & \mathscr{E}_A \\ \mathscr{F}^0, (\delta_1, \dots, \delta_n) & := & \mathrm{apply}(\mathscr{F}, f, n, \delta) \\ \mathscr{F}^{i+1}, \mathscr{E}_A{}^{i+1}, e'_{i+1}\ \mathscr{S}'_{i+1} & := & \mathscr{A}\ [\![\mathscr{F}^i, \mathscr{E}_A{}^i, \delta_c \vdash \delta_{i+1},\ e_{i+1}\ \mathscr{S}_{i+1}\ ]\!] \end{array}$$

Figure 6.3.. Demand analysis scheme $\mathscr{A}$ for translating expressions in LREC$_\mathrm{C}$ without nested records into expressions in LREC$_\mathrm{D}$ with demand annotations (contd.).

result of a function to its arguments and from the body expression of a `let` construct to the bindings of the `let` construct in inverse evaluation order.

The first rule in Figure 6.3 on page 122, the rule TRUE, describes the demand derivation for the `true` expression. Given an expression `true` with an existing set of demands $\mathscr{S}$, I add the new demand $\delta_c \vdash \delta$ to the set of demands using the join operation $\mathscr{S} \lhd \delta_c \vdash \delta$. This operation is defined as follows:

Definition 6.1.5 (Join of Demand Contexts)**.** *Given a set of demand contexts $\mathscr{S}$ and a context with demand $\delta_c \vdash \delta$, the* join of the demand contexts, *$\mathscr{S} \lhd \delta_c \vdash \delta$, is defined as*

$$\mathscr{S} \lhd \delta_c \vdash \delta := \begin{cases} (\mathscr{S} \setminus \{\delta_c \vdash \delta'\}) \cup \{\delta_c \vdash \delta\} & \textit{if some } \delta_c \vdash \delta' \in \mathscr{S} \textit{ exists,} \\ \mathscr{S} \cup \{\delta_c \vdash \delta\} & \textit{otherwise.} \end{cases}$$

As can be seen, the join of a demand $\mathscr{S} \lhd \delta_c \vdash \delta$ replaces any existing demand for the context δ_c in $\mathscr{S}$ with the new demand δ. Another way to define the join would be to compute the union of the existing demand and the new demand. However, this is not strictly necessary, as in multiple rounds of demand annotation, the demand for a given expression under any context can only become stronger.

Analogously, the demand is propagated in rules FALSE and UNIT in Figure 6.3 on page 122 for expressions of kind `false` and `~`. In all three cases, the function environment

$\mathscr{F}$ and variable environment $\mathscr{E}_A$ remain unchanged.

The rule VAR handles demand derivation for identifiers that occur in applied positions. As in the other rules discussed so far, the demand is added to the existing demand annotations for that variable. Furthermore, I update the variable environment by inserting the new demand and context $\delta_c \vdash \delta$ for the identifier α. This is facilitated by means of the insertion function $\mathscr{E}_A \overset{\mathscr{D}}{\leftarrow} (\alpha, \delta_c \vdash \delta)$, which is defined as follows:

Definition 6.1.6 (Variable Demand Insertion). *Given a variable environment $\mathscr{E}_A$, an identifier α and a demand with context $\delta_c \vdash \delta$. The insertion of the demand with context $\delta_c \vdash \delta$ into the variable environment $\mathscr{E}_A$ for the identifier α, $\mathscr{E}_A \overset{\mathscr{D}}{\leftarrow} (\alpha, \delta_c \vdash \delta)$, is then defined as*

$$\mathscr{E}_A \overset{\mathscr{D}}{\leftarrow} (\alpha, \delta_c \vdash \delta) := \begin{cases} (\mathscr{E}_A \setminus \{(\alpha, \mathscr{S})\}) \cup \{(\alpha, \mathscr{S}')\} & \textit{if some } (\alpha, \mathscr{S}) \in \mathscr{E}_A \textit{ exists,} \\ (\mathscr{E}_A \cup \{(\alpha, \{\delta_c \vdash \delta\})\} & \textit{otherwise,} \end{cases}$$

where

$$\mathscr{S}' := \begin{cases} (\mathscr{S} \setminus \{\delta_c \vdash \delta'\}) \cup \{\delta_c \vdash \delta' \uplus \delta\} & \textit{if some } \delta_c \vdash \delta' \in \mathscr{S} \textit{ exists,} \\ \mathscr{S} \cup \{\delta_c \vdash \delta\} & \textit{otherwise.} \end{cases}$$

Thus, the demand insertion into the variable environment $\mathscr{E}_A$ adds the demand to the corresponding set of demands and, if such a demand already exist, computes the union of the existing and new demand. Here, contrary to the join of demand contexts, computing the union of the new and existing demand is strictly necessary: As variables might be used in different expression positions, different demands might arise. However, as I have shown in Theorem 6.1.3 on page 119, the union of two demands satisfies both demands. Thus, using the demand insertion defined above, the resulting demand on the bound expression for a variable suffices to satisfy the demand at each applied occurrence.

Next in Figure 6.3 on page 122 is the rule EQUAL for demand analysis on equality expressions. Again, I annotate the new demand to the expression. Furthermore, as the equality operation contains sub-expressions, I rewrite these, as well. The order in which the demands for the two expressions are derived is not of importance. However, it is important that the updated function and variable environments are propagated properly. This is achieved in rule EQUAL by using the updated environment returned by the demand analysis of the first expression during the demand analysis of the second expression.

Instead of directly propagating the demand on the overall equality expression to the sub-expressions, I use the demand $\delta_c \vdash \Uparrow(\delta)$, which is defined as:

Definition 6.1.7 (Lifting of Demands). *Given a demand δ. Then the* lifting of the demand *δ, $\Uparrow(\delta)$, is defined as*

$$\Uparrow(\delta) := \begin{cases} \{\} & \textit{if } \delta = \{\}, \\ \uparrow\{\delta\} & \textit{otherwise.} \end{cases}$$

The idea is that we need to know the full values of both sub-expressions to compute any partial result of the equality operation. Only for the empty demand, *i.e.*, if we do not need the value of the equality operation at all, we do not need to compute the values of the sub-expressions either.

For LREC$_\text{C}$, it would suffice to propagate the demand on the equality operation to its sub-expressions, as in LREC$_\text{C}$ only Boolean values can be compared and Boolean values are either fully evaluated or not at all. However, when more complex data structures are added, this might no longer be the case. To cater for these cases and allow for an easier extension of the demand derivation, I use the explicit lifting operation $\Uparrow(\delta)$ to lift all but the empty demand to the full evaluation demand $\uparrow\{\delta\}$. Note that using $\uparrow\{\delta\}$ directly would not suffice, as in this case the arguments would always be fully evaluated.

The rule RECORD in Figure 6.3 on page 122 describes demand analysis for record expressions. Again, I annotate the demand at the expression. Additionally, for each component of the record the corresponding sub-expressions are rewritten using the sub-demand elem_d, if the corresponding label is part of the domain of the demand. Otherwise, no demand is propagated. To ensure proper propagation of function demands and variable demands, the two corresponding environments $\mathscr{F}$ and $\mathscr{E}_A$ are threaded though all demand derivations for the components of the record. Formally, this is specified by the inductive definition of the result environments $\mathscr{F}^n$ and ${\mathscr{E}_A}^n$.

For the dual expression to record construction, the selection of a component of a record, demand derivation is defined by rule SELECTION. Here, I rewrite the expression that the selection is performed on using the new demand $\delta_c \vdash \text{nest}(l, \delta)$, which is defined as:

Definition 6.1.8 (Nesting of Demands)**.** *Given a demand δ and a label l. Then the* nesting of the demand *δ with label l, $nest(l, \delta)$, is defined as*

$$nest(l,\delta) := \begin{cases} \{\} & \text{if } \delta = \{\}, \\ \{(l,\delta)\} & \text{otherwise.} \end{cases}$$

This nesting of demands reflects that, if no information about a component of a record is required, no information about the record itself is required either. In case that the demand on the component is non-empty, the corresponding demand on the corresponding record is a record demand where the selected label is associated with the demand of the component that is selected.

The demand derivation for the `any` operation is defined by rule ANY in Figure 6.3 on page 122. As the `any` operation may yield any of its argument as its result, I propagate the demand for the `any` expression to all the argument expressions. Again, to ensure the propagation of demands on functions and variables, the corresponding environments $\mathscr{F}$ and $\mathscr{E}_A$ are threaded through the demand analysis for the sub-expressions.

The rule COND describes demand derivation for conditionals. For evaluating a conditional partially, the full value of the predicate expression and the partial value of either the then or else branch is required. However, if the demand for the conditional is empty, none of its sub-expressions needs to be evaluated. To express this, I use $\Uparrow(\delta)$ as the demand for the predicate expression, similar to the rule EQUAL for equality operations.

$$\mathrm{fun}^d(\mathscr{F}_\delta, \prec, D) := (\mathscr{F}_\delta, \prec, \bigcup_{d \in D} \mathrm{fun}'^d(d))$$

where

$$\mathrm{fun}'^d(d) := \begin{cases} \{f\} \times \mathrm{inst}^d(\{i_1, \ldots, i_n\}) & \text{if } d \equiv \texttt{fun}\ f\ i_1\ \cdots\ i_n, \\ \emptyset & \text{otherwise.} \end{cases}$$

with

$$\mathrm{inst}^d(I) := \bigcup_{i \in I} \mathrm{inst}'^d(i)$$

where

$$\begin{array}{l} \mathrm{inst}'^d(\alpha_1\texttt{\{}\ l^1_1\texttt{,}\ \ldots\texttt{,}\ l^1_{n_1}\texttt{\}}\ \mathscr{S}_1\ \cdots\ \alpha_m\texttt{\{}\ l^m_1\texttt{,}\ \ldots\texttt{,}\ l^m_{n_m}\texttt{\}}\ \mathscr{S}_m\ \texttt{=}\ e\ \mathscr{S}) \\ \quad := \{(((\alpha_1, \{l^1_1, \ldots, l^1_{n_1}\}, \mathscr{S}_1), \ldots, (\alpha_m, \{l^m_1, \ldots, l^m_{n_m}\}, \mathscr{S}_m)), e\ \mathscr{S})\} \end{array}$$

Figure 6.4.. Definition of function *fun*^{d} as used in Figure 6.8 on page 136.

For the then and else expressions, the demand on the overall conditional expression is propagated as we do not know during demand analysis which expression will be evaluated.

Next in Figure 6.3 on page 122 is the rule GUARD for `guard` operations. Similar to conditionals, to evaluate a `guard` operation partially, the full value of the second argument is required. Thus, I propagate the demand $\Uparrow(\delta)$ to that expression. For the first argument, the demand on the `guard` operation itself is used. As the `guard` operation, if it can be evaluated, is the identity on the first argument, the demand on that argument is equal to the demand on the operation itself.

The demand inference for the second guard operation `witness` is defined by rule WITNESS. It is similar to the rule GUARD except that for `witness` operations the lifted demand $\Uparrow(\delta)$ is propagated to all but the first argument expression.

The penultimate rule in Figure 6.3 on page 122, the rule LET, defines demand propagation for `let` constructs. Demand propagation in this case is performed in four steps. First, the function environment $\mathscr{F}$ is extended by the functions defined by the `let` construct. This is expressed by means of the function *fun*^{d} as defined in Figure 6.4. The function *fun*^{d} mostly resembles its counterpart for full and partial evaluation *fun*. The resulting function environment only differs in the additional encoding of the sets of demand contexts that are annotated at the pattern of each function instance. To keep the structure of the function environment for demand inference and the corresponding function environment for partial evaluation similar and thus to allow for a later reuse, the function environment for demand inference contains an order on labels $\prec$, as well. This order, however, is not used during demand inference and thus remains empty.

Using the extended function environment, I next derive the demands that arise from the body expression of the `let` construct. This might yield a modified function environment, if the function definitions contained in the environment have been extended due

$$\text{propvar}(\mathscr{F}, \mathscr{E}_A, \delta_c, (d_1, \dots, d_n)) := (\mathscr{F}_n, \mathscr{E}_{An}, D_n)$$

with

$$\begin{aligned}
\mathscr{F}_0, \mathscr{E}_{A0}, D_0 &:= (\mathscr{F}, \mathscr{E}_A, ()) \\
\mathscr{F}_{i+1}, \mathscr{E}_{Ai+1}, D_{i+1} &:= \begin{cases} (\mathscr{F}_i', \mathscr{E}_{Ai}', (\texttt{var}\ \alpha\ \texttt{=}\ e'\ \mathscr{S}') + D_i) & \text{if } d_{n-i} \equiv \texttt{var}\ \alpha\ \texttt{=}\ e\ \mathscr{S} \\ (\mathscr{F}_i, \mathscr{E}_{Ai}, (d_{i+1}) + D_i) & \text{otherwise.} \end{cases}
\end{aligned}$$

where

$$\mathscr{F}_i', \mathscr{E}_{Ai}', e'\ \mathscr{S}' := \text{propvar}'(\mathscr{F}_i, \mathscr{E}_{Ai}, \delta_c, \alpha, e\ \mathscr{S})$$

The function *propvar′* is defined as

$$\text{propvar}'(\mathscr{F}, \mathscr{E}_A, \delta_c, \alpha, e\ \mathscr{S}) := (\mathscr{F}', \mathscr{E}_A{}'', e'\ \mathscr{S}')$$

with

$$\begin{aligned}
\mathscr{E}_A{}' &:= \{(\beta, \delta_c' \vdash \delta') \in \mathscr{E}_A \mid \alpha \neq \beta \lor \delta_c' \neq \delta_c\} \\
\mathscr{F}', \mathscr{E}_A{}'', e'\ \mathscr{S}' &:= \begin{cases} \mathscr{A}\left[\!\!\left[\mathscr{F}, \mathscr{E}_A{}', \delta_c \vdash \delta,\ \ e\ \mathscr{S} \right]\!\!\right] & \text{if } (\alpha, \delta_c \vdash \delta) \in \mathscr{E}_A, \\ \mathscr{F}, \mathscr{E}_A{}', e\ \mathscr{S} & \text{otherwise.} \end{cases}
\end{aligned}$$

Figure 6.5.. Definition of function *propvar* as used in Figure 6.3 on page 122.

to function applications in the `let` body. Furthermore, alongside the extended body expression, an updated variable environment $\mathscr{E}_A{}'$ is returned, which contains the demands for all variables used in the `let` body.

Using this extended variable environment, I next propagate the demands on variables to the variable bindings of the `let` construct. This step is expressed by the function *propvar* as defined in Figure 6.5.

Given the current function environment $\mathscr{F}''$, the updated variable environment $\mathscr{E}_A{}'$, the current context demand δ_c and the sequence of definitions, the function *propvar* computes an updated function environment, a variable environment where the local bindings have been removed and an annotated sequence of definitions. This is achieved by applying the function *propvar′* to each `var` definition in the sequence of definitions D. The order definitions using the `rel` construct are ignored.

Note that the demand analysis is performed bottom-up: In each step $i + 1$, the definition d_{n-i} is analysed. As in previous demand inference rules, I thread the function environment and variable environment through the sequence of demand derivations to ensure the correct propagation of demands on functions and variables.

Finally, the function *propvar′* propagates the demand for a single variable binding. If the environment contains a demand for the current identifier α under the current derivation context δ_c, this demand is propagated into the right hand side. As the identifier is no longer in scope, I use an updated variable environment where the demand for the

$$\text{disperse}((\mathscr{F}, \prec, \{\iota_1, \ldots, \iota_n\}), D) := (\mathscr{F}, D^n)$$

with

$$\begin{aligned} D_0 &:= \text{erase}(D) \\ D_{i+1} &:= D + \text{disperse}'(\iota_{i+1}) \end{aligned}$$

where the function *erase* is defined as

$$\text{erase}(D) := (d \in D \mid d \not\equiv \texttt{fun} \ \cdots)$$

and where the function *disperse′* is defined as

$$\begin{aligned} &\text{disperse}'(f, (((\alpha_1, p_1, \mathscr{S}_1), \ldots (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S})) \\ &:= \texttt{fun}\ f\ \alpha_1\ p_1\ \mathscr{S}_1\ \cdots\ \alpha_n\ p_n\ \mathscr{S}_n\ \texttt{=}\ e\ \mathscr{S}. \end{aligned}$$

Figure 6.6.. Definition of function *disperse* as used in Figure 6.3 on page 122.

identifier under the current context has been removed. It is important here that only the demand for the current context is removed from the environment, as the demands for other contexts are still in scope and might be needed later.

The last step in the demand analysis for `let` constructs is the function *disperse* as defined in Figure 6.6. It is the dual operation to the function *fun*^{d} defined in Figure 6.4 on page 127. Given a nested function environment and a sequence of definitions, the function *disperse* returns a new function environment and an updated sequence of definitions. The new function environment is computed by stripping off the top-level of the old environment. The sequence of definitions is updated by first erasing all function definitions using the function *erase* and then adding a corresponding function definition for each instance in the top-level set of instances of the old function environment.

As a last step, the updated definitions and the extended body expression are combined again into a `let` construct.

The final rule in Figure 6.3 on page 122, the rule AP, covers function applications. To derive the demands on the arguments of a function application, I first propagate the demand into the function definition using the function *apply*. It yields a modified function environment $\mathscr{F}'$ and a sequence of demands that contains the demand for each function argument. These demands are then propagated into the corresponding argument expressions.

The function *apply* is defined in Figure 6.7 on the following page. First, the function *apply* traverses the nested function environment until it either hits the initial empty environment $\perp$ or finds a nesting level where some instance for the function f is defined. In the former case, no function f is defined in the current scope. To nonetheless continue demand analysis, the function *apply* in this case returns empty demands for all parameters. An alternative choice would be to fail the demand analysis, as the corresponding expression cannot be evaluated. In the latter case, *i.e.*, if at least one instance has been found, the demands for the parameters are derived using the function *lookup*.

The function *apply* is defined as

$$\mathrm{apply}(\mathscr{F}, f, n, \delta) := \begin{cases} \mathrm{lookup}((\mathscr{F}', \prec, I), f, n, \delta_c) & \text{if } \mathscr{F} = (\mathscr{F}', \prec, I) \text{ and some} \\ & \ (f, (s, e)) \in I \text{ exists,} \\ ((\mathscr{F}'', \prec, I), (\delta_1, \dots, \delta_n)) & \text{if } \mathscr{F} = (\mathscr{F}', \prec, I) \text{ and no} \\ & \ (f, (s, e)) \in I \text{ exists,} \\ (\mathscr{F}, (\{\}, \dots, \{\})) & \text{if } \mathscr{F} = \perp, \end{cases}$$

where

$$\mathscr{F}'', (\delta_1, \dots, \delta_n) := \mathrm{apply}(\mathscr{F}', f, n, \delta_c).$$

Furthermore, the function *lookup* is defined as

$$\mathrm{lookup}((\mathscr{F}, \prec, I), f, n, \delta_c)$$

$$:= \begin{cases} ((\mathscr{F}, \prec, I), (\delta_1, \dots, \delta_n)) & \begin{array}{l}\text{if } \delta_1, \dots, \delta_n \in \mathscr{D} \text{ exists and} \\ (f, (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S}) \in I \\ \ \text{exists such that } \delta_c \vdash \delta_i \in \mathscr{S}_i \\ \ \text{for all } i \in \{1, \dots, n\},\end{array} \\ \mathrm{fix}(\mathscr{F}, \prec, I', f, n, \delta_c) & \text{otherwise.} \end{cases}$$

with

$$I' := \bigcup_{\iota \in I} \mathrm{amend}(\iota, f, n, \delta_c)$$

where *amend* is defined as

$$\mathrm{amend}((f', (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_m, p_m, \mathscr{S}_m)), e\ \mathscr{S})), f, n, \delta_c)$$

$$:= \begin{cases} (f', (((\alpha_1, p_1, \mathscr{S}_1 \lhd \delta_c \vdash \emptyset), \dots, (\alpha_m, p_m, \mathscr{S}_m \lhd \delta_c \vdash \emptyset)), e\ \mathscr{S})) & \begin{array}{l}\text{if } f = f' \\ \text{and } m = n,\end{array} \\ (f', (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_m, p_m, \mathscr{S}_m)), e\ \mathscr{S})) & \text{otherwise.} \end{cases}$$

and *fix* denotes the fix-point iteration on the result of *derive*, defined as

$$\mathrm{fix}(\mathscr{F}, \prec, I, f, n, \delta_c) := \begin{cases} ((\mathscr{F}, \prec, I), (\delta_1, \dots, \delta_n)) & \text{if } \mathscr{F} = \mathscr{F}' \text{ and } I = I', \\ \mathrm{fix}(\mathscr{F}', \prec, I', f, n, \delta_c) & \text{otherwise,} \end{cases}$$

where

$$\mathscr{F}', I', (\delta_1, \dots, \delta_n) := \mathrm{derive}(\mathscr{F}, I, f, n, \delta_c).$$

The function *derive* is defined as

$$\mathrm{derive}(\mathscr{F}, \{\iota_1, \dots, \iota_m\}, f, n, \delta_c) := ((\mathscr{F}_m, \{\iota'_1, \dots, \iota'_m\}), (\delta_1, \dots, \delta_n))$$

Figure 6.7.. Definition of function *apply* as used in Figure 6.3 on page 122.

with

$$\begin{array}{lcl}
\mathscr{F}_0 & := & \mathscr{F} \\
\delta_i^0 & := & \text{pattern}(I_0, f, n, i) \\
\mathscr{F}_{j+1}, \iota_{j+1}, (\delta_1^{j+1}, \dots, \delta_n^{j+1}) & := & \text{derive}'(\mathscr{F}_j, f, n, \iota_{j+1}, \delta_c) \\
\delta_k & := & \biguplus_{j \in \{0,\dots,m\}} \delta_i^j \text{ for all } k \in \{1, \dots, n\} \\
\iota_k' & := & \text{propdem}(\iota_k, f, n, \delta_c, (\delta_1, \dots, \delta_n)) \text{ for all} \\
 & & \quad k \in \{1, \dots, m\}
\end{array}$$

where the function *pattern* is defined as

$$\begin{array}{l}
\text{pattern}(\{\iota_0, \dots, \iota_m\}, f, n, i) := \\
\quad (\bigcup_{j \in \{1,\dots,m\}} \text{pattern}'(\iota_j, f, n, i) \setminus \bigcap_{j \in \{1,\dots,m\}} \text{pattern}'(\iota_j, f, b, i)) \times \{\emptyset\}
\end{array}$$

with

$$\text{pattern}'(\iota, f, n, i) := \begin{cases} p_i & \text{if } \iota \equiv (f, (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S})), \\ \emptyset & \text{otherwise.} \end{cases}$$

Furthermore, the function *derive′* is defined as

$$\begin{array}{l}
\text{derive}'(\mathscr{F}, f, n, \iota, \delta_c) \\
\quad := \begin{cases} (\mathscr{F}', \iota', (\delta_1, \dots, \delta_n)) & \text{if } \iota \equiv (f, (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S})), \\ (\mathscr{F}, \iota, (\{\}, \dots, \{\})) & \text{otherwise,} \end{cases}
\end{array}$$

where

$$\begin{array}{lcl}
\mathscr{F}', \mathscr{E}_A, e'\ \mathscr{S}' & := & \mathscr{A}\ [\![\mathscr{F}, \{\}, \delta_c \vdash \delta_c,\ \ e\ \mathscr{S}\ ]\!] \\
\iota' & := & (f, (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e'\ \mathscr{S}')) \\
\delta_i & := & \begin{cases} \delta_{\alpha_i} & \text{if } (\alpha_i, \delta_{\alpha_i}) \in \mathscr{E}_A, \\ \{\} & \text{otherwise,} \end{cases} \text{ for all } i \in \{1, \dots, n\}.
\end{array}$$

Lastly, the function *propdem* is defined as

$$\begin{array}{l}
\text{propdem}(\iota, f, n, \delta_c, (\delta_1, \dots, \delta_n)) := \\
\quad \begin{cases} \iota' & \text{if } \iota \equiv (f, (((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S})), \\ \iota & \text{otherwise,} \end{cases}
\end{array}$$

where

$$\iota' := (f, (((\alpha_1, p_1, \mathscr{S}_1 \lhd \delta_c \vdash \delta_1), \dots, (\alpha_n, p_n, \mathscr{S}_n \lhd \delta_c \vdash \delta_n)), e\ \mathscr{S})).$$

Figure 6.7.. Definition of function *apply* as used in Figure 6.3 on page 122 (contd.).

The traversal of the nested function environment as performed by the function *apply* corresponds to the *lookup* stage of the instance matching process during evaluation as defined in Figures 3.12 and 5.4 for full and partial evaluation, respectively. There, as well, the first environment that contains some instance definition for a given function name is used. In both cases, this models the scoping rules for LREC, *i.e.*, it encodes that a function definition at a certain nesting level shadows all other definitions using the same name in surrounding nesting levels, regardless of the arity of the function.

Furthermore, it is important to note here that throughout the function *apply* and its sub-functions, the function environment is only extended but no instances are ever removed. In the recursive descend of the function *apply*, the original environment is reconstructed by adding the outer nesting levels of the function environment that contained no matching instances back to the updated lower nesting levels. Moreover, all sub-functions operate on the full set of instances defined at a given nesting level, leaving non-matching functions unchanged. This, as well, ensures that the structure of the function environment remains unchanged.

If some instances for the function name f have been found, the function *lookup* in Figure 6.7 on page 130 is used to derive the demands on the function parameters that result from these instances. The function *lookup* expects as arguments the function environment for the nesting level the function is defined in, the function name, the arity to lookup and the demand to propagate. If for the given function name, arity and demand an instance that has already been annotated can be found, the corresponding demands on the function parameters are returned. It suffices to find a single instance that has been annotated, as always all instances of a given arity are annotated with the same demands.

If no such instance can be found, the demands are freshly derived. This is encoded by the function *fix*. It expects the outer function environment and the set of instances, the function name and the arity to derive demands for, and the demand on the result of the function application. It is important here that the set of instances given to the function *fix* is first extended by empty demands for the current context using the function *amend*. This is done to ensure that recursive calls to a function, for which demands are currently derived, do not trigger a further identical derivation process. Instead, the function *lookup* will yield the empty demands in this case.

The function *fix* derives the demands for all matching function instances using the function *derive* until no new demands in the function environment $(\mathscr{F}, I)$ arise. As noted before, such a fix-point needs to exist, as demand derivation is monotonically increasing and the maximum demand that can arise is the demand on all labels that are used in an expression.

In the function *derive*, the new demand on the i-th parameter is computed by first computing the demand that arises due to the pattern matching δ_i^0 using the function *pattern*. Furthermore, for each instance ι_j, the corresponding demand is inferred using the function *derive*$'$. Finally, all individual demands for each parameter are combined using the union on demands $\uplus$ and annotated at the function instances using the function *propdem*.

To ensure that no demands on other functions that arise during the demand derivation for the matching instances get lost, the function environment $\mathscr{F}$ and the set of instances I is threaded through the multiple demand derivation steps. These demands are then propagated back up alongside the demands that have been inferred for the matching instances.

The function *pattern* first computes the set of labels used in the pattern for a given argument in each matching instance, *i.e.*, in each instance for the function f with the correct arity. The filtering of non-matching instances is performed using the function *pattern*$'$. The demand that arises from the pattern match is then computed as the Cartesian product of set of differentiating labels, *i.e.*, the set of labels that are used in the pattern of some but not all instances, with the empty set. The resulting demand thus contains the empty demand for each label that is used in some but not all instances.

The function *derive*$'$ derives the demand that arises from an instance. If the instance does not match, *i.e.*, if it uses a different function name or has the wrong arity, the resulting demand is empty. For matching instances, the demand is derived by applying the demand analysis to the body expression of the instance, using an empty variable environment and the current demand as context demand. As all functions are closed in LREC, the resulting environment contains only the demands to the function parameters. These are then extracted and combined to the sequence of demands for the arguments. Furthermore, the instance is updated with the amended body expression.

However, the inferred demands are not yet annotated at the function parameters. To ensure that each instance is annotated with the same demands, this is performed during the next stage of *derive*, once all demands have been derived. The corresponding rewriting of the function instances is performed by applying the function *propdem* to each instance at the current nesting level. Given an instance, a function name and arity, the context demand and the demands for the parameters, the function *propdem* joins the demands into the existing demands for the parameters of an instance, if the instance matches the given function name and arity. Otherwise, the instance remains unchanged.

Once the demands on the argument expressions in rule LET have been computed using the function *apply*, the last step is to continue the demand derivation for each argument expressions. Again, the function and variable environments are threaded through the demand derivations for the argument expressions to ensure that all demands are properly propagated back up.

This completes the description of the demand derivation for expressions in LREC$_C$ without nested records. Using the above demand analysis, an expression in LREC$_C$ can be annotated with demands such that the result of partial evaluation satisfies a given demand. I will show that the annotated demands indeed suffice to guide partial evaluation to a desired partial result later in this section. However, first I formalise the notion of using the demand annotations to guide partial evaluation.

6.1.3. Demand Driven Partial Evaluation

As mentioned in the introduction to this chapter, the partial evaluation as presented in Section 5.3 does not yield a uniquely defined result. Even worse, it indeed might get stuck although full evaluation succeeds. The underlying cause is that the rules for partial evaluation as presented in Figure 5.2 on page 86 allow for two degrees of freedom. Firstly, it is not defined when to use the rule PARTIAL and thus stop further evaluating an expression and when to actually evaluate an expression at least to a partial result using the specific rule for that kind of expression. Furthermore, in the rule RECORD for record expressions, it is not defined which components of such a record expression shall be evaluated and which not.

Secondly, the computation of the anti-domain of record values in rules RECORD and UNIT is largely unconstrained. The sole condition is that only labels that are not contained in the domain of the full value may be part of the partial value, *i.e.*, only those labels that are not part of the record expression may be in the anti-domain.

Using the results of the demand analysis, I can now further constrain the evaluation rules to enforce a certain level of partial evaluation. In particular, I will constrain the rules such that for every expression with a demand annotation at most one rule matches. Thus, the resulting guided partial evaluation is uniquely defined up to the non-determinism introduced by the `any` operation.

To ease the specification of the constrained semantics, I first define two helper predicates *empty* and $\overline{\mathit{empty}}$ for empty and non-empty demand annotations, respectively.

Definition 6.1.9 (Empty Demand)**.** *Given a set of demand contexts $\mathscr{S}$ and a context demand δ_c, the predicate empty$(\delta_c, \mathscr{S})$ is defined as follows:*

$$\mathit{empty}(\delta_c, \mathscr{S}) := \delta_c \vdash \{\} \in \mathscr{S}$$

Thus, empty$(\delta_c, \mathscr{S})$ is true if the set of demand annotations $\mathscr{S}$ contains the empty demand context $\delta_c \vdash \{\}$ for the demand δ. Dually, I define a predicate for non-empty demand annotations.

Definition 6.1.10 (Non-Empty Demand)**.** *Given a set of demand contexts $\mathscr{S}$ and a context demand δ_c, the predicate $\overline{\mathit{empty}}(\delta_c, \mathscr{S})$ is defined as follows:*

$$\overline{\mathit{empty}}(\delta_c, \mathscr{S}) := \exists \delta \in \mathscr{D} \setminus \{\{\}\} : \delta_c \vdash \delta \in \mathscr{S}$$

The predicate $\overline{\text{empty}}(\delta_c, \mathscr{S})$ is true if the set of demand contexts $\mathscr{S}$ contains a context $\delta_c \vdash \delta$ for the context demand δ_c, where δ is not the empty demand.

Apart from these predicates, I define a shorthand notation for the extraction of a demand for a given context from a set of demand annotations:

Definition 6.1.11 (Demand Extraction)**.** *Given a context demand δ_c and a set of demand annotations $\mathscr{S}$. Then the extracted demand from $\mathscr{S}$ in the context δ_c is defined as*

$$\mathit{extract}(\delta_c, \mathscr{S}) := \begin{cases} \delta & \text{if some } \delta \in \mathscr{D} \text{ exists with } \delta_c \vdash \delta \in \mathscr{S}, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

Using these definitions, I can now define a constrained semantics for partial evaluation with demand annotations that yields a uniquely defined result. For the rules of this big-step operational semantics as presented in Figure 6.8 on page 136, I use the same notation as for the semantics for unguided partial evaluation presented in Figure 5.2 on page 86. As before, $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e \downarrow v$ is to be read as: Given an environment $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c)$, the term e can be evaluated to a partial value v. Note that, compared to the previous semantics for partial evaluation, the environment $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c)$ additionally contains the current context demand δ_c. This information is used to chose the right demand annotations depending on the current evaluation context. Furthermore, the variable environment $\mathscr{E}_\delta$, apart from the identifier and its bound value, additionally contains the demand that this value satisfies. This demand is not necessary for evaluation but will be used to define the notion of a satisfying environment in later proofs. Thus, an implementation may not actually store this additional demand in the environment. Similarly, the function environment $\mathscr{F}_\delta$ additionally contains the demand annotations for function parameters. I use the same function environment for the evaluation as during demand inference. However, in contrast with demand inference, during evaluation the order on labels contained in the function environment may be non empty. As with the variable environment, the additional demand information is not needed for evaluation and only serves for later proofs. The remaining component of the environment, *i.e.*, the order on labels $\prec$, remains unchanged.

The first striking difference between the unconstrained evaluation rules in Figure 5.2 on page 86 and the corresponding constrained rules in Figure 6.8 on the following page is the addition of a further premise to each rule. For example, the first two rules, *i.e.*, the rules True and False for Boolean constants, now require that the demand annotations of the expression to evaluate are non-empty. Note that, in order to ensure that the demand is non-empty in the current evaluation context, the premise makes use of the context demand δ_c that is part of the evaluation environment. Similarly, all other rules for specific expressions now require non-empty demand annotations. Thus, using these extended semantics, an expression can only be evaluated if its value is actually required by the demand annotations.

The only rule that does not require non-empty demand annotations is the last rule in Figure 6.8 on the next page, *i.e.*, the rule Partial. Contrary to all other rules, the rule Partial requires that the demand annotation of the expression to evaluate in the current demand context is empty. Therefore, the rule Partial can only be used to evaluate expression whose value is not flagged as needed by the demand annotations.

These additional premises resolve the first degree of freedom. It is now well defined which expressions are evaluated and where evaluation is stopped by application of the special rule Partial.

A further change in the rules of Figure 6.8 on the following page compared to those in Figure 5.2 on page 86 is the construction of the anti-domain in the rules for unit expressions and record expressions. The unconstrained versions of these rules as presented in Figure 5.2 on page 86 do not restrict the anti-domain of a record value apart from requiring that each label that is part of the anti-domain of a partial value may not be in the domain of the corresponding full value. For the rule Unit, this means that the anti-domain can contain any label as the domain of the unit value is always empty.

This degree of freedom may lead to situations where a pattern match cannot be decided for the partial evaluation whereas it is decidable in the full evaluation case. To circumvent

$$\textsc{True} \quad : \quad \frac{\overline{\text{empty}}(\delta_c, \mathscr{S})}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{true}\ \ \mathscr{S} \downarrow \mathit{true}}$$

$$\textsc{False} \quad : \quad \frac{\overline{\text{empty}}(\delta_c, \mathscr{S})}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{false}\ \ \mathscr{S} \downarrow \mathit{false}}$$

$$\textsc{Unit} \quad : \quad \frac{\overline{\text{empty}}(\delta_c, \mathscr{S}) \quad \exists\delta := \text{extract}(\delta_c, \mathscr{S})}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{\~{}}\mathscr{S} \downarrow \text{dom}_d(\delta) \times \{\mathit{!}\}}$$

$$\textsc{Var} \quad : \quad \frac{(i, \mathscr{S}', v) \in \mathscr{E} \quad \overline{\text{empty}}(\delta_c, \mathscr{S})}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : i\ \ \mathscr{S} \downarrow v}$$

$$\textsc{EqualTrue} \quad : \quad \frac{\begin{array}{c}(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_1\ \ \mathscr{S}_1 \downarrow v_1 \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_2\ \ \mathscr{S}_2 \downarrow v_2 \quad v_1, v_2 \notin \mathscr{R}_p \\ v_1 \overset{v}{=} v_2 \quad \overline{\text{empty}}(\delta_c, \mathscr{S})\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{(}e_1\ \ \mathscr{S}_1\ \texttt{=}\ e_2\ \ \mathscr{S}_2\texttt{)}\ \ \mathscr{S} \downarrow \mathit{true}}$$

$$\textsc{EqualFalse} \quad : \quad \frac{\begin{array}{c}(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_1\ \ \mathscr{S}_1 \downarrow v_1\ \mathscr{S}_2 \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_2\ \ \mathscr{S}_1 \downarrow v_2\ \mathscr{S}_2 \quad v_1, v_2 \notin \mathscr{R}_p \\ v_1 \overset{v}{\neq} v_2 \quad \overline{\text{empty}}(\delta_c, \mathscr{S})\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{(}e_1\ \ \mathscr{S}_1\ \texttt{=}\ e_2\ \ \mathscr{S}_2\texttt{)}\ \ \mathscr{S} \downarrow \mathit{false}}$$

$$\textsc{Record} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \quad \exists\delta := \text{extract}(\delta_c, \mathscr{S}) \\ \forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow l_i \neq l_j \\ \forall l_i \in \text{dom}_d(\delta) \cap \{l_1, \ldots, l_n\} \ : \ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_i\ \ \mathscr{S}_i \downarrow v_i\end{array}}{\begin{array}{c}(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{\{}l_1\texttt{=}e_1\ \ \mathscr{S}_1\texttt{,}\ \ \ldots\texttt{,}\ \ l_n\texttt{=}e_n\ \ \mathscr{S}_2\texttt{\}}\ \ \mathscr{S} \\ \downarrow \{(l_i, v_i) \mid l_i \in \text{dom}_d(\delta) \cap \{l_1, \ldots, l_n\}\} \cup R\end{array}}$$

where $R := (\text{dom}_d(\delta) \setminus \{l_1, \ldots, l_n\}) \times \{\mathit{!}\}$

$$\textsc{RecordFull} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \quad \exists\delta \in \mathscr{D} :\uparrow \{\delta\} = \text{extract}(\delta_c, \mathscr{S}) \\ \forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow l_i \neq l_j \\ \forall l_i \in \{l_1, \ldots, l_n\} \ : \ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_i\ \ \mathscr{S}_i \downarrow v_i\end{array}}{\begin{array}{c}(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{\{}l_1\texttt{=}e_1\ \ \mathscr{S}_1\texttt{,}\ \ \ldots\texttt{,}\ \ l_n\texttt{=}e_n\ \ \mathscr{S}_n\texttt{\}}\ \ \mathscr{S} \\ \downarrow \{(l_1, v_1), \ldots, (l_n, v_n)\} \cup R\end{array}}$$

where $R := (\text{dom}_d(\delta) \setminus \{l_1, \ldots, l_n\}) \times \{\mathit{!}\}$

$$\textsc{Selection} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_b\ \ \mathscr{S}_b \downarrow v \quad v \in \mathscr{R}_p \quad l \in \text{range}_p(v)\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_b\ \ \mathscr{S}_b\texttt{.}l\ \ \mathscr{S} \downarrow \text{elem}_p(v, l)}$$

Figure 6.8.. An operational semantics for partial evaluation of LREC$_{\text{D}}$.

$$\textsc{Any} \quad : \quad \frac{\overline{\text{empty}}(\delta_c, \mathscr{S}) \;\; \exists i \in \{1, \ldots, n\} \; : \; (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_i \;\; \mathscr{S}_i \downarrow v}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{any(}e_1 \;\; \mathscr{S}_1\texttt{,} \;\; \ldots\texttt{,} \;\; e_n \;\; \mathscr{S}_n\texttt{)} \;\; \mathscr{S} \downarrow v}$$

$$\textsc{CondThen} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_p \;\; \mathscr{S}_p \downarrow \mathit{true} \;\; (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_t \;\; \mathscr{S}_t \downarrow v\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{if} \;\; e_p \;\; \mathscr{S}_p \;\; e_t \;\; \mathscr{S}_t \;\; e_e \;\; \mathscr{S}_e \;\; \mathscr{S} \downarrow v}$$

$$\textsc{CondElse} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_p \;\; \mathscr{S}_p \downarrow \mathit{false} \;\; (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_e \;\; \mathscr{S}_e \downarrow v\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{if} \;\; e_p \;\; \mathscr{S}_p \;\; e_t \;\; \mathscr{S}_t \;\; e_e \;\; \mathscr{S}_e \;\; \mathscr{S} \downarrow v}$$

$$\textsc{Guard} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \;\; (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_g \;\; \mathscr{S}_g \downarrow \mathit{true} \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_v \;\; \mathscr{S}_v \downarrow v\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{guard(}e_v \;\; \mathscr{S}_v \;\; e_g \;\; \mathscr{S}_g\texttt{)} \;\; \mathscr{S} \downarrow v}$$

$$\textsc{Witness} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \\ \forall i \in \{1, \ldots, n\} : (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \; : \; e_i \;\; \mathscr{S}_i \downarrow \mathit{true} \\ (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e_v \;\; \mathscr{S}_v \downarrow v\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{witness(}e_v \;\; \mathscr{S}_v \;\; e_1 \;\; \mathscr{S}_1 \;\; \cdots \;\; e_n \;\; \mathscr{S}_n\texttt{)} \;\; \mathscr{S} \downarrow v}$$

$$\textsc{Let} \quad : \quad \frac{\overline{\text{empty}}(\delta_c, \mathscr{S}) \;\; (\mathscr{F}'_\delta, \prec', \mathscr{E}'_\delta, \delta_c) : e_b \mathscr{S}_b \downarrow v}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \;\; \texttt{let} \;\; d_1 \cdots d_n \;\; \texttt{in} \;\; e_b \;\; \mathscr{S}_b \;\; \texttt{end} \;\; \mathscr{S} \downarrow v}$$

$$\begin{array}{rrcl} & \prec' & = & \text{rel}(\prec, \{d_1, \ldots, d_n\}) \\ \text{where} & \mathscr{F}'_\delta & = & \text{fun}^d(\mathscr{F}_\delta, \prec', \{d_1, \ldots, d_n\}) \\ & \mathscr{E}' & = & \text{val}^d(\mathscr{F}'_\delta, \prec', \mathscr{E}, \delta_c, (d_1, \ldots, d_n)) \end{array}$$

$$\textsc{Ap} \quad : \quad \frac{\begin{array}{c}\overline{\text{empty}}(\delta_c, \mathscr{S}) \;\; \exists \delta := \text{extract}(\delta_c, \mathscr{S}) \\ \forall i \in \{1, \ldots, n\} \; : \; (\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \; : \; e_i \;\; \mathscr{S}_i \downarrow v_i \in \mathscr{R} \\ \{(\mathscr{F}'_\delta, \prec', \mathscr{E}', e_b \;\; \mathscr{S}_b)\} = \text{match}^d(\mathscr{F}_\delta, f, (v_1, \ldots, v_n)) \\ (\mathscr{F}'_\delta, \prec', \mathscr{E}'_\delta, \delta) : e_b \;\; \mathscr{S}_b \downarrow v\end{array}}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : \texttt{(}f \;\; e_1 \;\; \mathscr{S}_1 \;\; \ldots \;\; e_n \;\; \mathscr{S}_n\texttt{)} \;\; \mathscr{S} \downarrow v}$$

$$\textsc{Partial} \quad : \quad \frac{\text{empty}(\delta_c, \mathscr{S})}{(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) : e \;\; \mathscr{S} \downarrow \mathit{?}}$$

Figure 6.8.. An operational semantics for partial evaluation of LREC$_\text{D}$ (contd.).

$$\mathrm{val}^d(\mathscr{F}_\delta, \prec, \mathscr{E}, \delta_c, (d_1, \ldots, d_n)) := \mathscr{E}_n$$

where

$$\begin{aligned}
\mathscr{E}_0 &:= \mathscr{E} \\
\mathscr{E}_{i+1} &:= \begin{cases}
\mathscr{E}_i \overset{d}{\leftarrow} (l, \mathscr{S}, v) & \text{if } d_{i+1} \equiv \texttt{val } l\texttt{=}e\mathscr{S} \\
 & \text{and } (\mathscr{F}_\delta, \prec, \mathscr{E}_i, \delta_c) : e\mathscr{S} \downarrow v, \\
\mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{fun } f \;\; i_1 \;\cdots\; i_m, \\
\mathscr{E}_i & \text{if } d_{i+1} \equiv \texttt{rel } l_1 \texttt{ <: } \; l_2, \\
\text{undefined} & \text{otherwise.}
\end{cases}
\end{aligned}$$

Figure 6.9.. Definition of function *val*^{d} as used in Figure 6.8 on page 136.

these situations, the demand analysis annotates all discriminating labels of a pattern match as required. Therefore, I use this information in rule Unit to construct the anti-domain of the partial result. As can be seen in Figure 6.8 on page 136, the anti-domain of the partial result of evaluating a unit expression is the domain of the demand annotation. Formally, this is expressed by computing the value of a unit expression as the Cartesian product of the domain of the demand $\mathrm{dom}_d(\delta)$ with the special partial value `!`.

For record expressions the situation is slightly more complex. Record expressions are handled by two rules in Figure 6.8 on page 136. The first rule, the rule Record, handles the evaluation of records where the annotated demand is a non-empty record demand. The special case of the demand for full evaluation $\uparrow \{\cdot\}$ is handled by the second rule RecordFull.

In both rules, similarly to the rule Unit, the anti-domain of the partial result is computed using the domain of the demand $\mathrm{dom}_d(\delta)$. However, contrary to unit expressions, the domain of record expressions is not necessarily empty. Thus, to enforce the invariant that the anti-domain of a partial value may not contain any labels that are part of the domain of the corresponding full value, these are excluded from the anti domain R.

For records that need not to be evaluated fully, the domain of the partial result is computed in rule Record as the intersection of the domain of the demand with the set of labels that are contained in the record expression. Thus, all components of the record that are marked as required are computed. In case that the demand for full evaluation is annotated, the rule RecordFull evaluates all components of the record expression.

Using these modified evaluation rules, all results of partial evaluation always contain all labels that are flagged as required to decide consecutive pattern matches by the demand annotations.

The last two differences between the semantics for guided partial evaluation in Figure 6.8 on page 136 and the unconstrained semantics in Figure 5.2 on page 86 are of rather technical nature. Firstly, the rule Let uses functions *val*^{d} and *fun*^{d} instead of the functions *val*^{p} and *fun* defined in Figure 5.3 on page 88 and 3.10, respectively. Figure 6.9 gives the definition of val^d. As can be seen, the function mainly differs in the evaluation relation and evaluation environment they use. The function val^d uses the additional context demand δ_c. Furthermore, it stores the set of demand contexts $\mathscr{S}$ that the bound value satisfies alongside the identifier and the bound value in the environment. This is achieved by means of an extended variable insertion function $\overset{d}{\leftarrow}$ which is defined as follows:

Definition 6.1.12 (Variable-Demand Insertion)**.** *Given a set $\mathscr{E} \subset \mathscr{I} \times \mathscr{D} \times \mathscr{V}_p$, the set $\mathscr{E} \overset{d}{\leftarrow} (i, \mathscr{S}, v)$ where $i \in \mathscr{I}$, $\mathscr{S} \in \mathscr{D} \times \mathscr{D}$ and $v \in \mathscr{V}$ is defined as*

$$\mathscr{E} \overset{d}{\leftarrow} (i, \mathscr{S}, v) := \{(i', \mathscr{S}', v') \mid (i', \mathscr{S}', v') \in \mathscr{E} \wedge i \neq i'\} \cup \{(i, \mathscr{S}, v)\}$$

Similar to $\leftarrow$ as defined in Definition 3.3.4 on page 53, the variable insertion with demands $\overset{d}{\leftarrow}$ overwrites existing bindings in the environment to model the scoping of the `let` construct.

Likewise, the function *fun*^{d} as defined in Figure 6.4 on page 127 only differs in the way the resulting function environment is constructed. Analogously to the construction of the function environment during demand inference, the function environment during evaluation now contains the inferred demands for all function parameters, as well.

This modification of the function environment requires a modification to the rule Ap, as well. Instead of using the partial matching function *match*^{p}, I use a partial matching function match^d that can handle the additional demand annotations in the function environment $\mathscr{F}_\delta$. Its definition is given in Figure 6.10 on the following page. As can be seen, it is largely identical to the partial matching function *match*^{p} as defined in Figure 5.4 on page 90. The first two stages, *i.e.*, the *lookup* and *arity* stages, are identical to those used in the functions *match* and *match*^{p}. The third and fourth stage, *i.e.*, the *pattern*^{d} and *order*^{d} stages, have been modified to handle the additional demand annotations. However, the matching process as such remains unchanged. In particular, the Lemmata 5.3.1 and 5.3.2 apply to these stages, as well. The last stage, *i.e.*, the *bind*^{d} stage, has been adapted such that it produces a variable environment with demands $\mathscr{E}_\delta$ by extracting the corresponding set of demand contexts from the set of matching instances.

Besides the different matching function, I have modified the rule Ap to appropriately switch the demand context, as well. For each function application, the function body is evaluated using a new context demand. This context demand, analogously to the demand analysis, is the demand on the result of the function application.

This completes the discussion of the semantics of guided partial evaluation. Before I show the final result of this section, *i.e.*, that the guided partial evaluation always succeeds if the full evaluation does and furthermore, that the result of such partial evaluation satisfies the annotated demand, I first give the promised proof that the demand annotations inferred by the demand analysis suffice to decide the partial pattern match. In particular, I will show that knowledge of the existence of all discriminating labels of the parameter pattern suffices to decide a partial pattern match if the corresponding pattern match under full evaluation is decidable, *i.e.*, if the pattern match under full evaluation yields any instances. All discriminating labels here refers to those labels that are part of the pattern of corresponding parameters of some instances but not all instances. In Theorem 5.3.1 on page 99 I have already shown that the partial matching

$$\mathrm{match}^d := \mathrm{bind}^d \circ \mathrm{order}^d \circ \mathrm{pattern}^d \circ \mathrm{arity} \circ \mathrm{lookup}$$

with *lookup* and *arity* as defined in Figure 3.12 on page 60. The function pattern^d is defined as

$$\mathrm{pattern}^d(\mathscr{F}_\delta, \prec, I, (a_1, \dots, a_n)) := (\mathscr{F}_\delta, \prec, I_n, (a_1, \dots, a_n))$$

where

$$\begin{aligned} I_0 &:= I \\ I_{i+1} &:= \mathrm{filter}_2^d(\mathrm{filter}_1^d(I_i, (a_1, \dots, a_n), i), i) \end{aligned}$$

with

$$\mathrm{filter}_1^d(I, a, i) := \{(((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S}) \in I \mid p_i \cap \overline{\mathrm{dom}}_p(\mathrm{a}) = \emptyset\}$$

and

$$\mathrm{filter}_2^d(\prec, I, (a_1, \dots, a_n), i) :=$$
$$\left\{ \begin{array}{l|l} \begin{array}{l} (((\alpha_1, p_1, \mathscr{S}_1), \dots, \\ \qquad (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S}) \in I \end{array} & \begin{array}{l} \forall(((\alpha'_1, p'_1, \mathscr{S}'_1), \dots, (\alpha'_n, p'_n, \mathscr{S}'_n)), e'\ \mathscr{S}') \in I : \\ \quad (\exists j \in \{1, \dots, i\}\ : \\ \qquad p_j \setminus \mathrm{dom}_p(a_j) \neq p'_j \setminus \mathrm{dom}_p(a_j)) \vee \\ \quad |p_i \cap \mathrm{dom}_p(a_i)| \not< |p'_i \cap \mathrm{dom}_p(a_i)| \end{array} \end{array} \right\}.$$

The function *order^d* is defined as

$$\mathrm{order}^d(\mathscr{F}_\delta, \prec, I, (a_1, \dots, a_n)) := (\mathscr{F}_\delta, \prec, I_n, (a_1, \dots, a_n))$$

where

$$\begin{aligned} I_0 &:= I \\ I_{i+1} &:= \mathrm{filter}'^d(\prec, I_i, (a_1, \dots, a_n), i) \end{aligned}$$

with

$$\mathrm{filter}'^d(\prec, I, (a_1, \dots, a_n), i) :=$$
$$\left\{ \begin{array}{l|l} \begin{array}{l} (((\alpha_1, p_1, \mathscr{S}_1), \dots, \\ \qquad (\alpha_n, p_n, \mathscr{S}_n)), e\ \mathscr{S}) \in I \end{array} & \begin{array}{l} \forall(((\alpha'_1, p'_1, \mathscr{S}'_1), \dots, (\alpha'_n, p'_n, \mathscr{S}'_n)), e') \in I : \\ \quad (\exists j \in \{1, \dots, n\} : \\ \qquad p_j \setminus \mathrm{dom}_p(a_j) \neq p'_j \setminus \mathrm{dom}_p(a_j)) \vee \\ \quad (\exists j \in \{1, \dots, i-1\} : \\ \qquad \exists(((\alpha''_1, p''_1, \mathscr{S}''_1), \dots, (\alpha''_n, p''_n, \mathscr{S}''_n)), \\ \qquad\quad e''\ \mathscr{S}'') \in I : p'_j \overset{\rightarrow}{\prec} p''_j \wedge p_j \overset{\rightarrow}{\not\prec} p''_j) \vee \\ \quad (\exists j \in \{1, \dots, i-1\} : \\ \qquad \exists(((\alpha''_1, p''_1, \mathscr{S}''_1), \dots, (\alpha''_n, p''_n, \mathscr{S}''_n)), \\ \qquad\quad e''\ \mathscr{S}'') \in I : p''_j \overset{\rightarrow}{\prec} p_j) \vee \\ \quad p_i \overset{\rightarrow}{\not\prec} p'_i \end{array} \end{array} \right\}.$$

Figure 6.10.. Pattern matching function for partial best match with demands *match^d* as used in Figure 6.8 on page 136.

Lastly, the function *bind*^{d} is defined as

$$\text{bind}^d(\mathscr{F}_\delta, \prec, I, a) := \{(\mathscr{F}_\delta, \prec, \text{bind}'^d(P, a), e\ \mathscr{S}) \mid (P, e\ \mathscr{S}) \in I\}$$

where

$$\text{bind}'^d(((\alpha_1, p_1, \mathscr{S}_1), \dots, (\alpha_n, p_n, \mathscr{S}_n)), (a_1, \dots, a_n)) := \{(\alpha_1, \mathscr{S}_1, a_1), \dots, (\alpha_n, \mathscr{S}_n, a_n)\}.$$

Figure 6.10.. Pattern matching function for partial best match with demands *match*^{d} as used in Figure 6.8 on page 136 (contd.).

function *match*^{p} always yields at least those instances that the matching function for full evaluation *match* does. Thus, it remains to be shown that if all discriminating labels are known, the partial matching functions returns only instances that are returned by the full matching function, as well, if the full matching function returns any instances at all.

I will use the same proof strategy as in the proof of Theorem 5.3.1 on page 99. As both matching functions match and match^p use exactly the same two first steps, it suffices to show that the third and fourth step, *i.e.*, the *pattern* and *order* steps, yield the same set of matching instances. To differentiate between the results of the full and partial matching process, I will use the superscripts f and p, respectively, in the following. First, I show the identity of the result sets for the pattern step.

Lemma 6.1.1 (Guided Partial Match: Step 3)**.** *Given a function environment $\mathscr{F}$, a complete partial order on labels $\prec$, a set of function instances I, where I is of the form $\{(((\alpha_1^1, p_1^1), \dots, (\alpha_n^1, p_n^1)), e^1), \dots, (((\alpha_1^m, p_1^m), \dots, (\alpha_n^m, p_n^m)), e^m)\}$, and two tuple of arguments $a^f = (a_1^f, \dots, a_n^f) \in \mathscr{V}^n$ and $a^p = (a_1^p, \dots, a_n^p) \in \mathscr{V}_p^n$. Let $(\mathscr{F}^f, \prec^f, I^f, a'^f) :=$ pattern$(\mathscr{F}, \prec, I, a)$ and $(\mathscr{F}^p, \prec^p, I^p, a'^p) :=$ pattern$^p(\mathscr{F}, \prec, I, a^p)$. Then the following statement holds:*

$$\begin{aligned}(\forall i \in \{1, \dots, n\} : a_i^f \sqsubseteq a_i^p \wedge \textstyle\bigcup_{j \in \{1,\dots,m\}} p_i^j \setminus \bigcap_{j \in \{1,\dots,m\}} p_i^j \subseteq \mathit{range}_p(a_i^p)) \wedge I^f \neq \emptyset \\ \Rightarrow \mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge a^f = a'^f \wedge a^p = a'^p \wedge I^f = I^p\end{aligned}$$

Proof. Given that for all $i \in \{1, \dots, n\}$ the statements $a_i^f \sqsubseteq a_i^p$ and $\bigcup_{j \in \{1,\dots,m\}} p_i^j \setminus \bigcap_{j \in \{1,\dots,m\}} p_i^j \subseteq \text{range}_p(a_i^p)$ hold. Furthermore, assume that $I^f \neq \emptyset$. I will show that then the equalities $\mathscr{F}^f = \mathscr{F}^p$, $\prec^f = \prec^p$, $a^f = a'^f$, $a^p = a'^p$ and $I^f = I^p$ are true, as well.

The first four equalities in the statement to be shown follow directly from the definition of *pattern* and *pattern*^{p} in Figures 3.12 and 5.4, respectively. Remains to be shown that $I^f = I^p$ holds, as well.

From Lemma 5.3.1 on page 96, we already know that $I^f \subseteq I^p$ holds. Thus, it suffices to show that $I^f \supseteq I^p$ holds. The result of the order matching functions order and order^p is defined inductively over the arguments. I will therefore show the above property by co-induction over the computation steps in function *pattern* and *pattern*^{p}.

In the initial step, the statement $I_0^f \supseteq I_0^p$ is obviously true.

For the inductive step, given $i \in \{1, \ldots, n+1\}$, and I_i^f and I_i^p as the resulting sets of matching instances from the previous filtering step, assume that $I_i^f \supseteq I_i^p$ holds. I will show by contradiction that then $I_{i+1}^f \supseteq I_{i+1}^p$ holds, as well.

First, I will show some properties of the instances contained in I_i^p. Assume $I_i^p := \{(((\alpha_1^1, p_1^1), \ldots, (\alpha_n^1, p_n^1)), e^1), \ldots, (((\alpha_1^m, p_1^m), \ldots, (\alpha_n^m, p_n^m)), e^m)\}$. We know from the assumptions that the partial arguments $a_1^p, \ldots, a_n^p$ contain all discriminating labels, *i.e.*, we know that the condition $\bigcup_{j\in\{1,\ldots,m\}} p_k^j \setminus \bigcap_{j\in\{1,\ldots,m\}} p_k^j \subseteq \mathrm{range}_p(a_k^p)$ is true for all $k \in \{1, \ldots, n\}$. Furthermore, we know that for all $j \in \{1, \ldots, m\}$,$k \in \{1, \ldots, i-1\}$ the statement $p_k^j \cap \overline{\mathrm{dom}}_p(a_k^p) = \emptyset$ holds, as otherwise the corresponding instance would have already been filtered out by the function *filter*$_1^p$. Let $R_k := \bigcup_{j\in\{1,\ldots,m\}} p_k^j \setminus \mathrm{range}_p(a_k^p)$ for all $k \in \{1, \ldots, i-1\}$. It follows that for all $f := (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I_i^p$ the statement $p_k \setminus \mathrm{dom}_p(a_k^p) = R_k$ holds for all $k \in \{1, \ldots, i-1\}$.

Now assume it exists $f := (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I_{i+1}^p$ with $f \notin I_{i+1}^f$. As f is not filtered out in the $i+1$-th step, we know that $p_i \cap \overline{\mathrm{dom}}_p(a_i^p) = \emptyset$. Furthermore, we know that for all other instances $f' := (((\alpha_1', p_1'), \ldots, (\alpha_n', p_n')), e') \in I_i^f$ either there exists $k \in \{1, \ldots, i\}$ such that $p_j \setminus \mathrm{dom}_p(a_k^p) \neq p_j' \setminus \mathrm{dom}_p(a_k^p)$ or $|p_i \cap \mathrm{dom}_p(a_i^p)| \not< |p_i' \cap \mathrm{dom}_p(a_i^p)|$ per definition of *filter*$_2^p$. However, for all $k \in \{1, \ldots, i-1\}$ we know that $p_k \setminus \mathrm{dom}_p(a_k^p) = R_k = p_k' \setminus \mathrm{dom}_p(a_k^p)$ holds. Therefore, only the two conditions $p_i \setminus \mathrm{dom}_p(a_i^p) \neq p_i' \setminus \mathrm{dom}_p(a_i^p)$ and $|p_i \cap \mathrm{dom}_p(a_i^p)| \not< |p_i' \cap \mathrm{dom}_p(a_i^p)|$ remain. I will show that both cannot be true.

From the assumptions, we know that $I^f \neq \emptyset$. It follows that $I_{i+1}^f \neq \emptyset$ by definition of the filtering process *filter*. With Lemma 5.3.1 on page 96, it follows that $I_{i+1}^f \subseteq I_{i+1}^p$. Thus, there exists at least one instance $f' := (((\alpha_1', p_1'), \ldots, (\alpha_n', p_n')), e') \in I_{i+1}^f$ with $f' \in I_{i+1}^p$ and thus $f' \in I_i^p$. As f' in I_{i+1}^p, we know that $p_i' \cap \overline{\mathrm{dom}}_p(a_i^p) = \emptyset$. Combined with the assumption that a_i^p contains all discriminating labels, it follows that $p_i \setminus \mathrm{dom}_p(a_i^p) = p_i' \setminus \mathrm{dom}_p(a_i^p)$. Thus, the first remaining condition cannot be true for all instances in I_i^p.

Remains to be shown that $|p_i \cap \mathrm{dom}_p(a_i^p)| \not< |p_i' \cap \mathrm{dom}_p(a_i^p)|$ cannot be true. From $f' \in I_{i+1}^f$, it follows that $|p_i' \cap \mathrm{dom}(a_i^f)|$ is maximal with respect to all instances contained in I_i^f by definition of the matching function *pattern*.. In particular, as $f \notin I_{i+1}^f$, it follows that $|p_i' \cap \mathrm{dom}(a_i^f)| > |p_i \cap \mathrm{dom}(a_i^f)|$. From $a_i^f \sqsubseteq a_i^p$ we can follow that $\mathrm{dom}(a_i^f) \supseteq \mathrm{dom}_p(a_i^p)$. As $\mathrm{range}_p(a_i^p)$ contains all discriminating labels and with $p_i \cap \overline{\mathrm{dom}}_p(a_i^p) = \emptyset = p_i' \cap \overline{\mathrm{dom}}_p(a_i^p)$, it follows that $p_i \cup p_i' \setminus p_i \cap p_i' \subseteq \mathrm{dom}_p(a_i^p)$. Overall, it follows that $|p_i \cap \mathrm{dom}_p(a_i^p)| < |p_i' \cap \mathrm{dom}_p(a_i^p)|$.

This contradicts $f \in I_{i+1}^p$. □

With this lemma in place, for the pattern match I lastly show that a similar property holds for the fourth matching step, *i.e.*, the matching based on the order of labels.

Lemma 6.1.2 (Guided Partial Match: Step 4). *Given a function environment $\mathscr{F}$, a complete partial order on labels $\prec$, a set of function instances I and arguments $a^f = (a_1^f, \ldots, a_n^f) \in \mathscr{V}^n$ and $a^p = (a_1^p, \ldots, a_n^p) \in \mathscr{V}_p^n$. Let $(\mathscr{F}, \prec, I^f, a^f) :=$ pattern$(\mathscr{F}, \prec, I, a^f)$ and $(\mathscr{F}, \prec, I^p, a^p) :=$ pattern$^p(\mathscr{F}, \prec, I, a^p)$. Furthermore, let $(\mathscr{F}^f, \prec^f, I'^f, a'^f) :=$*

order$(\mathscr{F}, \prec, I^f, a^f)$ *and* $(\mathscr{F}^p, \prec^p, I'^p, a'^p) :=$ *order*$^p(\mathscr{F}, \prec, I^p, a^p)$. *Then the following statement holds:*

$$\begin{gathered}(\forall i \in \{1, \ldots, n\} : a_i^f \sqsubseteq a_i^p \wedge \bigcup_{j\in\{1,\ldots,m\}} p_i^j \setminus \bigcap_{j\in\{1,\ldots,m\}} p_i^j \subseteq \mathit{range}_p(a_i^p)) \wedge I \neq \emptyset \\ \Rightarrow \mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge a^f = a'^f \wedge a^p = a'^p \wedge I'^f = I'^p\end{gathered}$$

Proof. Given that for all $i \in \{1, \ldots, n\}$ the statements $a_i^f \sqsubseteq a_i^p$ and $\bigcup_{j\in\{1,\ldots,m\}} p_i^j \setminus \bigcap_{j\in\{1,\ldots,m\}} p_i^j \subseteq \mathrm{range}_p(a_i^p)$ hold and that $I \neq \emptyset$ is true. I will show that then statement $\mathscr{F}^f = \mathscr{F}^p \wedge \prec^f = \prec^p \wedge a^f = a'^f \wedge a^p = a'^p \wedge I'^f = I'^p$ holds, as well.

The first four equalities in the above statement follow directly from the definition of the pattern and order matching steps *pattern* and *order*, and *pattern*^{p} and *order*^{p} as defined in Figures 3.12 and 5.4, respectively. From Lemma 5.3.1 on page 96 we already know that $I^f = I^p$ holds. Remains to be shown that then $I'^f = I'^p$ holds, as well.

As the result of the functions *order* and *order*^{p} is defined inductively over the number of arguments, I will show this property by co-induction over the single filtering steps.

In the initial step, the statement $I_0'^f = I_0'^p$ follows from the definition of I^f and I^p, and the result of Lemma 6.1.1 on page 141.

For the inductive step, assume that $I_i^f = I_i^p$ holds for some $i \in \{1, \ldots, n\}$. From the proof of Lemma 5.3.2 on page 98 we already know that $I_{i+1}'^f \subseteq I_{i+1}'^p$ holds. It remains to be shown that $I'^f \supseteq I'^p$ holds, as well. I will show that then $I_{i+1}^f \supseteq I_{i+1}^p$ holds, as well, by contradiction.

Assume an instance $f := (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e) \in I_{i+1}'^p$ exists with $f \notin I_{i+1}'^f$. From the inductive assumption and the monotonicity of the order filtering process, it follows that $f \in I_i'^p$. Then, by definition of the order matching function *order*^{p} we know that for all instances $f' := (((\alpha_1', p_1'), \ldots, (\alpha_n', p_n')), e') \in I_i'^p$ one of the following conditions must be true, as otherwise f would not be filtered in the $i+1$-th step.

1. $\exists j \in \{1, \ldots, n\} : p_j \setminus \mathrm{dom}_p(a_j) \neq p_j' \setminus \mathrm{dom}_p(a_j)$
2. $\exists j \in \{1, \ldots, i\} : \exists(((\alpha_1'', p_1''), \ldots, (\alpha_n'', p_n'')), e'') \in I_i'^p : p_j' \overset{\rightarrow}{\prec} p_j'' \wedge p_j \overset{\rightarrow}{\not\prec} p_j''$
3. $\exists j \in \{1, \ldots, i\} : \exists(((\alpha_1'', p_1''), \ldots, (\alpha_n'', p_n'')), e'') \in I_i'^p : p_j'' \overset{\rightarrow}{\prec} p_j$
4. $p_i \overset{\rightarrow}{\not\prec} p_i'$

I will show for each condition that it cannot be true. As neither f nor f' has been filtered during the *pattern*^{p} filtering step, we know that for all pattern p_j and p_j' with $j \in \{1, \ldots, n\}$ the statement $p_j \cap \overline{\mathrm{dom}}_p(a_j^p) = \emptyset$ and $p_j' \cap \overline{\mathrm{dom}}_p(a_j^p) = \emptyset$ hold. Furthermore, as for all $j \in \{1, \ldots, n\}$ the partial value a_j^p contains all discriminating pattern of the instances in I_i^p, it follows that $p_j \setminus \mathrm{dom}_p(a_j^p) = p_j' \setminus \mathrm{dom}_p(a_j^p)$. This invalidates the first condition.

The second and third conditions cannot be true either. From the inductive assumption, we know that $I_i'^p = I_i'^f$ holds. Assume there exists an instance f'' with $f'' := (((\alpha_1'', p_1''), \ldots, (\alpha_n'', p_n'')), e'') \in I_i'^p$ such that for some $j \in \{1, \ldots, i\}$ the statement $p_j' \overset{\rightarrow}{\prec}$

$p''_j \wedge p_j \overset{\rightarrow}{\not\prec} p''_j$ holds. As $f'' \in I'^p_i$, it follows that $f'' \in I'^f_i$ is true, as well. Thus, $f' \notin I'^f_i$ due to $p'_j \overset{\rightarrow}{\prec} p''_j$. This contradicts that $f' \in I'^f_{i+1}$.

Similarly, assume that for some $j \in \{0, \ldots, i\}$ the statement $p''_j \overset{\rightarrow}{\prec} p_j$ holds. Then $f'' \notin I'^f_i$, as $f \in I'^f_i$. Combined with the inductive assumption this contradicts that $f'' \in I'^p_i$.

Lastly, assume the fourth condition holds, *i.e.*, assume that $p_i \overset{\rightarrow}{\not\prec} p'_i$ is true for all instances $f' \in I'^p_i$. From the inductive assumption we know that $I'^f_i = I'^p_i$. It follows that the instance f cannot be filtered out in the $i+1$-th step of the *order* function, as no other instance shadows it. This contradicts $f \notin I'^f_{i+1}$.

Overall, it follows that $I'^f = I'^p$. □

Thus, it indeed suffices to annotate a demand for all discriminating labels of the pattern of a set of instances to ensure that the partial match is decidable if the full match would be. It is worth noting here that even if all discriminating labels are known, the partial pattern match might still yield an instance where the full pattern match would not yield any instances. The above two lemmata are conditional on the full pattern match to succeed, *i.e.*, they are conditional on the full pattern match to yield at least one instance. Thus, by design, the result of partial evaluation in my approach is always conditional on the full evaluation to succeed. However, as stated earlier, this is a general property of any kind of partial evaluation. Even without the dependency of partial pattern matching on the success of the corresponding full pattern match, the result of partial evaluation would still be dependant on full evaluation to succeed. For instance, full evaluation might get stuck evaluating a component of a record expression that is not evaluated under partial evaluation.

Given these lemmata, I can now show the final result of this section, *i.e.*, that the guided partial evaluation always succeeds if the full evaluation does and furthermore, that the result of such partial evaluation satisfies the annotated demand. As a prerequisite to the proof of this property, I first extend the notion of demand satisfaction to environments:

Definition 6.1.13 (Satisfying Environment). *Given a variable environment $\mathscr{E}$, a partial variable environment with demands $\mathscr{E}_\delta$ and a context demand $\delta_c \in \mathscr{D}$. We say that $\mathscr{E}$ is satisfied by $\mathscr{E}_\delta$ in the context δ_c, or $\mathscr{E} \overset{\overset{\delta_c}{\rightarrow}}{\sqsubseteq} \mathscr{E}_\delta$ for short, if the following condition holds:*

$$\forall (i, v) \in \mathscr{E} \ !\exists (i, \mathscr{S}, v') \in \mathscr{E}_\delta : \exists \delta \in \mathscr{D} : \delta_c \vdash \delta \in \mathscr{S} \wedge v \overset{\delta}{\sqsubseteq} v'$$

Thus, a partial variable environment with demands satisfies a full variable environment if for each binding in the latter exactly one corresponding binding in the former exists such that the former bound value satisfies the latter in the context δ_c.
Similarly, I define the notion of matching function environment with demands.

Definition 6.1.14 (Matching Function Environment With Demands). *Given a function environment $\mathscr{F}$ and a function environment with demands $\mathscr{F}_\delta$. We say that $\mathscr{F}$ is matched by $\mathscr{F}_\delta$, or $\mathscr{F} \overset{d}{\equiv} \mathscr{F}_\delta$ for short, if the following statement holds:*

- *If $\mathscr{F}$ is empty, then $\mathscr{F}_\delta$ is empty, as well, and*
- *if $\mathscr{F}$ has the form $(\mathscr{F}', \prec, D)$ for some order on labels $\prec$ and instances D then $\mathscr{F}_\delta$ has the form $(\mathscr{F}'_\delta, \prec, D')$ where*
    1. *$\mathscr{F}'$ is matched by $\mathscr{F}'_\delta$,*
    2. *D' contains the same instances as D modulo demand annotations,*
    3. *the demand annotations in D' can be computed from D using the demand inference for function instances as defined by function* apply *in Figure 6.7 on page 130.*

Furthermore, I extend the partial binding property of *val*^{p} as shown in Lemma 5.3.3 on page 100 to the function for evaluating value bindings with demands *val*^{d}.

Lemma 6.1.3 (Partial Bindings With Demands)**.** *Given a function environment $\mathscr{F}$ and a matching function environment with demands $\mathscr{F}_\delta$, a complete partial order on labels $\prec$, two variable environments $\mathscr{E}$ and $\mathscr{E}_\delta$ with $\mathscr{E} \overset{\overset{\delta_\subseteq}{\rightarrow}}{\sqsubseteq} \mathscr{E}_\delta$. Furthermore, let δ_c be a context demand and δ be a demand. Lastly, let $d_1, \ldots, d_n$ be definitions of a* `let` *construct and $d'_1, \ldots, d'_n$ be definitions of the corresponding* `let` *construct that results from a demand inference using $\mathscr{F}$, some variable environment $\mathscr{E}_A$ and $\delta_c \vdash \delta$.*

Assume that for every expression e in LRec$_C$ *with a given nesting depth $i \in \mathbb{N}$ and a corresponding expression $e'\mathscr{S}$ in* LRec$_D$*, where $e'\mathscr{S}$ is the result of the demand inference $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \ e\ ]\!]$, the statement $(\mathscr{F}, \prec, \mathscr{E})\ :\ e \Downarrow v \Rightarrow (\mathscr{F}, \prec, \mathscr{E}', \delta_c)\ :\ e\mathscr{S} \downarrow v' \wedge v \overset{\delta}{\sqsubseteq} v'$ holds.*

Let $\mathscr{E}^{f'} := val(\mathscr{F}, \prec, \mathscr{E}^f, (d_1, \ldots, d_n))$ and $\mathscr{E}^{d'} := val^d(\mathscr{F}, \prec, \mathscr{E}^d, \delta_c, (d_1, \ldots, d_n))$ with val as defined in Figure 3.11 on page 58 and val^d as defined in Figure 6.9 on page 138. Then the statement $\mathscr{E}^{f'} \overset{\overset{\delta_\subseteq}{\rightarrow}}{\sqsubseteq} \mathscr{E}^{p'}$ holds.

Proof. Both functions *val* and *val*^{d} compute the new environment inductively starting out with the old environment and then step-wise amending the environment by the bindings produced by each definition. To prove the above statement, I will show by induction that during each processing step, the resulting environment in *val*^{d} satisfies the corresponding environment in *val*. To simplify the presentation, I will assume that corresponding definitions occur in the same order.

For the initial step, this is trivially the case as $\mathscr{E}^f \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^d$ holds.

Let $\mathscr{E}^{f'}_j$ be the result of j-th processing step in the function *val* and $\mathscr{E}^{d'}_j$ be the corresponding result of the j-th processing step in the function *val*^{d}. Assume $\mathscr{E}^{f'}_j \overset{\overset{\delta_\subseteq}{\rightarrow}}{\sqsubseteq} \mathscr{E}^{d'}_j$ holds.

The definition d_{j+1} can be one of three different kinds: A function definition using the `fun` keyword, a order definition using the `rel` keyword or a value binding using the `val` keyword. I will show for each case that $\mathscr{E}^{f'}_{j+1} \overset{\rightarrow}{\sqsubseteq} \mathscr{E}^{p'}j + 1j$ holds.

Case 1: `fun` definition Assume d_{j+1} is a definition of the from `fun` $l\ i_1 \cdots i_k$ where the $i_1, \ldots, i_k$ are instance definitions. Then d'_{j+1} is a corresponding function definition, as well. From the definition of *val* and *val*^{d} it follows that $\mathscr{E}^{f'}_{j+1} = \mathscr{E}^{f'}_{j}$ and $\mathscr{E}^{d'}_{j+1} = \mathscr{E}^{d'}_{j}$, respectively. Thus, from the inductive assumption, it follows that $\mathscr{E}^{f'}_{j+1} \overset{\delta_{\varsigma}}{\overset{\rightarrow}{\sqsubseteq}} \mathscr{E}^{d'}j + 1j$ holds.

Case 2: `rel` definition $\mathscr{E}^{f'}_{j+1} \overset{\delta_{\varsigma}}{\overset{\rightarrow}{\sqsubseteq}} \mathscr{E}^{d'}j + 1j$ follows analogously to case 1.

Case 3: `val` definition Assume d_{j+1} is a definition of the form `val` l=e and d'_{j+1} is the corresponding definition of the form `val` l=$e'\mathscr{S}$. From $(\mathscr{F}, \prec, \mathscr{E}^{f'}_{j}) \ :\ e \Downarrow v$ we can then follow that $\delta_c \vdash \delta \in \mathscr{S}$ exists by definition of demand inference and $(\mathscr{F}, \prec, \mathscr{E}^{d'}_{j}, \delta_c) : e\mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. Thus, $\mathscr{E}^{f'}_{j} \leftarrow (l, v) \overset{\delta_{\varsigma}}{\overset{\rightarrow}{\sqsubseteq}} \mathscr{E}^{d'}_{j} \overset{d}{\leftarrow} (l, \delta, v')$ holds. Therefore, $\mathscr{E}^{f'}_{j+1} \overset{\delta_{\varsigma}}{\overset{\rightarrow}{\sqsubseteq}} \mathscr{E}^{d'}j + 1j$, holds as well.

Overall, it follows that in the $i + 1$-th step the statement $\mathscr{E}^{f'}_{j+1} \overset{\delta_{\varsigma}}{\overset{\rightarrow}{\sqsubseteq}} \mathscr{E}^{d'}_{j+1}$ holds. □

As a last prerequisite, I show that, given a function environment and a matching function environment with demands, the function *fun*^{d} produces a new function environment with demands that matches the function environment returned by the function *fun*.

Lemma 6.1.4 (Construction of Matching Function Environments)**.** *Given an order on labels $\prec$, a function environment $\mathscr{F}$ and a matching function environment with demands $\mathscr{F}_\delta$. Furthermore, let δ_c be a context demand and δ be a demand. Lastly, let $d_1, \ldots, d_n$ be definitions of a* `let` *construct and $d'_1, \ldots, d'_n$ be definitions of the corresponding* `let` *construct that results from a demand inference using $\mathscr{F}$, some variable environment $\mathscr{E}_A$ and $\delta_c \vdash \delta$. Then the following statement holds:*

$$fun(\mathscr{F}, \prec, \{d_1, \ldots, d_n\}) \overset{d}{\equiv} fun^d(\mathscr{F}_\delta, \prec, \{d'_1, \ldots, d'_n\})$$

Proof. The above follows trivially from the definition of *fun* and *fun*^{d}. □

Using the above definitions and lemmata, I can now show the final result for this section.

Theorem 6.1.4 (Partial Evaluation with Demands)**.** *Given that the choice of the argument expression to evaluate for the* `any` *operator is deterministic and identical for full and partial evaluation. Let e be an expression in* LRec$_C$ *and $\delta \in \mathscr{D}$. Furthermore, let $e'\mathscr{S}$ be the expression that results from the demand analysis $\mathscr{A}\ [\![\bot, \emptyset, \bot \vdash \delta,\ e\ ]\!]$. Then the following statement holds:*

$$(\bot, \emptyset, \emptyset)\ :\ e \Downarrow v \Rightarrow (\bot, \emptyset, \emptyset, \delta)\ :\ e'\mathscr{S} \downarrow v' \wedge v \overset{\delta}{\sqsubseteq} v'$$

Proof. To prove the above statement, I will show the following, more general statement:

Let $\mathscr{F}$ be a function environment, $\mathscr{E}$ be a variable environment, $\mathscr{E}'$ be a partial variable environment with $\mathscr{E} \overset{\overset{\delta_c}{\rightarrow}}{\sqsubseteq} \mathscr{E}'$ and $\prec$ be an order on labels. Furthermore, let $\mathscr{F}$ be a function environment with demands that matches $\mathscr{F}$ and $\mathscr{E}_A$ be some variable-demand environment. Lastly, let e be an expression in LREC$_C$, $\delta_c \in \mathscr{D}$ be the current context demand and $\delta \in \mathscr{D}$ be the current demand. Then for the expression $e'\mathscr{S}$ that results from the demand inference $\mathscr{A} \llbracket \mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ \ e\ \rrbracket$ the following statement

$$(\mathscr{F}, \prec, \mathscr{E})\ :\ e \Downarrow v \Rightarrow (\mathscr{F}_\delta, \prec, \mathscr{E}\text{'}, \delta_c)\ :\ e'\mathscr{S} \downarrow v' \wedge v \overset{\delta}{\sqsubseteq} v'$$

holds.

For the proof, assume that $(\mathscr{F}, \prec, \mathscr{E})\ :\ e \Downarrow v$ holds. I will show that then the statements $(\mathscr{F}_\delta, \prec, \mathscr{E}\text{'}, \delta_c)\ :\ e'\mathscr{S} \downarrow v'$ and $v \overset{\delta}{\sqsubseteq} v'$ hold, as well, by induction over the nesting depth of the expression e. Note that e' by definition of the demand inference has the same nesting depth as e: The demand inference does not modify the structure or kind of an expression but merely annotates demands.

To start the induction, assume that e is an expression with a nesting depth of 0. Then e by definition of LREC$_C$ in Figure 3.6 on page 50 can either be a Boolean expression, the special ˜ expression, an empty record expression or an identifier.

We know from the definition of the demand inference, that $\delta_c \vdash \delta \in \mathscr{S}$. I will handle the special case of $\delta = \emptyset$ separately. In this case, regardless of e, the rule PARTIAL in Figure 6.8 on page 136 and only that rule applies for e'. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e' \downarrow v'$ holds with $v' = ?$. It directly follows that $v \overset{\delta}{\sqsubseteq} v'$.

Now assume that $\delta \neq \emptyset$. We have to consider four cases:

Case 1: `true` Then $v = \mathit{true}$. As $\delta \neq \emptyset$, the single rule TRUE in Figure 6.8 on page 136 matches. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e' \downarrow v'$ with $v' = \mathit{true}$, as well. It follows that $v \overset{\delta}{\sqsubseteq} v'$.

Case 2: `false` This case is analogous to case 1 using rule FALSE.

Case 3: ˜ We know that $v = \{\}$ by definition of rule UNIT in Figure 3.8 on page 52. From $\delta \neq \emptyset$ it furthermore follows that the single rule UNIT in Figure 6.8 on page 136 matches. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e' \downarrow v'$ with $v' = \mathrm{dom}_d(\delta) \times \{!\}$. It follows directly from the definition of demand satisfaction that then $v \overset{\delta}{\sqsubseteq} v'$.

Case 4: Empty Record Then $v = \{\}$. As $e' =$ `{}`, both rules RECORD and RECORD-FULL yield the same result $v' = \mathrm{dom}_d(\delta)$. Furthermore, as $\delta \neq \emptyset$, only either of these two rules can match but one does match. It follows that the statement $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'\mathscr{S} \downarrow v'$ holds with $v' = \mathrm{dom}_d(\delta) \times \{\texttt{!}\}$ and therefore $v \overset{\delta}{\sqsubseteq} v'$.

Case 5: Identifier Thus $e = \alpha$ for some $\alpha \in \mathscr{I}$. From $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ it then follows that $(\alpha, v) \in \mathscr{E}$. As $\mathscr{E} \overset{\overset{\delta_c}{\rightarrow}}{\sqsubseteq} \mathscr{E}'$ holds, we know that some $\delta' \in \mathscr{D}$ exists with $(\alpha, \delta', v') \in$

$\mathscr{E}'$ and $v \overset{\delta'}{\sqsubseteq} v'$. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'\mathscr{S} \downarrow v'$ holds. By construction of the demand inference combined with Theorem 6.1.3 on page 119, we can furthermore follow that $v \overset{\delta}{\sqsubseteq} v'$ then holds, as well.

Overall, it follows that for expressions with nesting depth 0 the statement to be shown holds.

For the inductive step, assume that for all expressions $e \in \text{LRec}_\text{C}$ with some nesting depth $i \in \mathbb{N}$ the statement $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v \Rightarrow (\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'\mathscr{S} \downarrow v' \wedge v \overset{\delta}{\sqsubseteq} v'$ holds. I will show that the statement then holds for expressions with nesting depth $i+1$, as well.

For this, assume $e \in \text{LRec}_\text{C}$ to be such expression and $e'\mathscr{S}$ to be the expression resulting from the demand inference $\mathscr{A}\ [\![\mathscr{F}, \mathscr{E}_A, \delta_c \vdash \delta,\ e\ ]\!]$. From the definition of the demand inference, we then know that $\delta_c \vdash \delta \in \mathscr{S}$. Furthermore, assume that $(\mathscr{F}, \prec, \mathscr{E}) : e \Downarrow v$ holds, as well, for some $v \in \mathscr{V}$.

Again, I will handle the special case where $\delta = \emptyset$ separately. For this, assume that δ indeed is the empty demand. Then, regardless of e and e', the single matching rule in Figure 6.8 on page 136 is the rule Partial. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'\mathscr{S} \downarrow v'$ holds with $v' = ?$. It follows, that for all $v \in \mathscr{V}$ the statement $v \overset{\delta}{\sqsubseteq} v'$ is true.

For the situation that $\delta \neq \emptyset$, we have to consider 9 cases. As e has a nesting depth greater than 0, e can either be a non-empty record expression, a selection operation, an application of the `any` operation, a conditional, an application of the `guard` operation, an application of the `witness` operation, the equality operation on non-record values, a `let` expression or a function application.

Case 1: Record Expression Then e has the form $\{l_1 = e_1,\ \ldots, l_n = e_n\}$ and e' has the form $\{l_1 = e'_1\mathscr{S}_1,\ \ldots,\ l_n = e'_n\mathscr{S}_n\}$ where $n \in \mathbb{N}$, and $e_j \in \text{LRec}_\text{C}$, $e'_j\mathscr{S}_j \in \text{LRec}_\text{D}$ and $l_j \in \mathscr{L}$ for all $j \in \{1, \ldots, n\}$. As e can be evaluated to v, we know that all e_j can be evaluated to some $v_j \in \mathscr{V}$. By definition of the demand inference, we know that for all l_j with $j \in \{1 \ldots, n\}$ the context $\delta_c \vdash \text{elem}_d(\delta, l_j)$ is in $\mathscr{S}_j$. Thus, using the inductive assumption, we can follow that for all $j \in \{1, \ldots, n\}$ some $v'_j \in \mathscr{V}_p$ exists with $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'_j \downarrow v'_j$ and $v_j \overset{\text{elem}_d(\delta, l_j)}{\sqsubseteq} v'_j$.

We now have to differentiate two situations: Either $\delta \in \mathscr{D}_r$. Then the rule Record matches. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e' \downarrow v'$ with $v' = \{(l_j, v_j) \mid l_j \in \text{dom}_d(\delta) \cap \{l_1, \ldots, l_n\}\} \cup (\text{dom}_d(\delta) \setminus \{l_1, \ldots, l_n\}) \times \{!\}$. It follows from the definition of demand satisfaction that $v \overset{\delta}{\sqsubseteq} v'$.

Otherwise exists some $\delta' \in \mathscr{D}_r$ with $\delta = \uparrow \{\delta'\}$. In this case the rule RecordFull matches. Therefore, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) : e'\mathscr{S} \downarrow v'$ with $v' = \{(l_1, v_1), \ldots, (l_n, v_n)\} \cup (\text{dom}_d(\delta) \setminus \{l_1, \ldots, l_n\}) \times \{!\}$. Again, by definition of demand satisfaction, $v \overset{\delta}{\sqsubseteq} v'$ holds.

Case 2: Selection Operation Then e has the form $e_b.l$ and e' has the form $e'_b\mathscr{S}_b.l$. From the definition of demand inference, we can follow that $\delta_c \vdash \{(l, \delta)\} \in \mathscr{S}_b$. As

e can be evaluated to v, we know that e_b can be evaluated to a value $v_b \in \mathscr{R}$ with $l \in \text{dom}(v_b)$. With the inductive assumption it follows that $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'_b\mathscr{S}_b \downarrow v'_b$ with $v_b \overset{\{(l,\delta)\}}{\sqsubseteq} v'_b$. Thus, $l \in \text{dom}_p(v'_b)$ and $\text{elem}(v_b, l) \overset{\delta}{\sqsubseteq} \text{elem}_p(v'_b, l)$. It follows that $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'\mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$.

Case 3: `any` Operation Then e has the form `any(` $e_1 \;\cdots\; e_n$`)` and e' has the form `any(` $e'_1\mathscr{S}_1 \;\cdots\; e'_n\mathscr{S}_n$`)`. From the definition of demand inference, it follows that $\delta_c \vdash \delta \in \mathscr{S}_j$ for all $j \in \{1, \ldots, n\}$. As $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$ holds, we can follow that for some $j \in \{1, \ldots, n\}$ the statement Senvs $:\; e_j \Downarrow v$ holds, as well. From the inductive assumption directly follows that then $(\mathscr{F}_\delta, \prec, \mathscr{E}, \delta_c) \;:\; e'_j\mathscr{S}_j \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. As we assume the `any` operation to be deterministic, we have $(\mathscr{F}_\delta, \prec, \mathscr{E}, \delta_c) \;:\; e'\mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$.

Case 4: Conditional Here, e has the form `if` $e_p\; e_t\; e_e$ and e' has the form `if` $e'_p\mathscr{S}_p\; e'_t\mathscr{S}_t\; e'_e\mathscr{S}_e$. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$, we can follow that $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_p \Downarrow v_p$ with $v_p = \textit{true}$ or $v_p = \textit{false}$. I will assume the former case in the following. The latter case follows analogously. From the definition of demand inference, we know that $\delta_c \vdash \uparrow\{\delta\} \in \mathscr{S}_p$. It follows with the inductive assumption that then $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'_p \downarrow v'_p$ with $v_p \overset{\uparrow\{\delta\}}{\sqsubseteq} v'_p$. Thus, by definition of demand satisfaction, $v'_p = \textit{true}$ follows from $\delta \neq \emptyset$. Similarly, we know that $\delta_c \vdash \delta \in \mathscr{S}_t$ and that $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_t \Downarrow v$. Again we can follow from the inductive assumption that then $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'_t\mathscr{S}_t \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. It directly follows that then $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'\mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$ holds, as well.

Case 5: `guard` operation In this scenario, e has the form `guard(`$e_v\; e_g$`)` and e' has the form `guard(`$e'_v\; \mathscr{S}_v\; e'_g\; \mathscr{S}_g$`)`. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$ we can follow that $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_v \Downarrow v$ and $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_g \Downarrow \textit{true}$. As δ is non-empty, we know from the definition of demand inference that $\delta_c \vdash \uparrow\{\delta\} \in \mathscr{S}_g$. With the inductive assumption we then get $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e'_g\; \mathscr{S}_g \downarrow \textit{true}$, as *true* is the only value that satisfies the result of full evaluation *true* given the demand $\uparrow\{\delta\}$. Furthermore, we know from the definition of demand inference that $\delta_c \vdash \delta \in \mathscr{S}_v$. Thus, we get with the inductive assumption that $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e'_v\; \mathscr{S}_v \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. It then directly follows that $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e'\; \mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$, as well.

Case 6: `witness` operation Here, e has the form `witness(`$e_v\; e_1 \;\cdots\; e_n$`)` and e' has the form `witness(`$e'_v\; \mathscr{S}_v\; e'_1\; \mathscr{S}_1 \;\cdots\; e'_n\; \mathscr{S}_n$`)`. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$ we can follow that $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_v \Downarrow v$ and $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_j \Downarrow \textit{true}$ for all $j \in \{1, \ldots, n\}$. Furthermore, we know from the definition of demand inference that $\delta_c \vdash \delta \in \mathscr{S}_v$ and $\delta_c \vdash \uparrow\{\delta\} \in \mathscr{S}_j$ for all $j \in \{1, \ldots, n\}$. Thus, with the inductive assumption it follows that $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e'_v\; \mathscr{S}_v \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$ holds. Similarly, it follows that $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e'_j\; \mathscr{S}_j \downarrow v'_j$ with $\textit{true} \overset{\uparrow\{\delta\}}{\sqsubseteq} v'_j$ holds for all $j \in \{1, \ldots, n\}$.

It follows, that $v'_j = \mathit{true}$ for all $j \in \{1, \ldots, n\}$. From this, we can follow that $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e' \;\mathscr{S} \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$.

Case 7: Equality Operation Then e has the form `(` e_1 `=` e_2 `)` and e' has the form `(` $e'_1\mathscr{S}_1$ `=` $e'_2\mathscr{S}_2$ `)` $\mathscr{S}$. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$, we know that $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_1 \Downarrow v_1$ with $v_1 \notin \mathscr{R}$ and $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_2 \Downarrow v_2$ with $v_2 \notin \mathscr{R}$. Furthermore, from the definition of demand inference, we know that $\delta_c \vdash \Uparrow(\delta) \in \mathscr{S}_1$ and $\delta_c \vdash \Uparrow(\delta) \in \mathscr{S}_2$. As $\delta \neq \emptyset$, it follows that $\Uparrow(\delta) = \uparrow\{\delta\}$. With the inductive assumption, we can follow that for $j \in \{1, 2\}$ the statement $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e_j\mathscr{S}_j \downarrow v'_j$ with $v_j \overset{\uparrow\{\delta\}}{\sqsubseteq} v'_j$ holds. From $v_j \notin \mathscr{R}$, by definition of demand satisfaction, it follows that $v_j = v'_j$. Thus, $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'\mathscr{S} \downarrow v'$ with $v' = v$. Therefore, trivially $v \overset{\delta}{\sqsubseteq} v'$ holds.

Case 8: `let` Construct Then e is of the form `let` $d_1 \;\cdots\; d_n$ `in` e_b `end` and e' has the form `let` $d'_1 \;\cdots\; d'_n$ `in` $e'_b\mathscr{S}_b$ `end` $\mathscr{S}$ where for each definition d_j in e a corresponding definition d'_k in e' exists. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$, it follows that $(\mathscr{F}_l, \prec_l, \mathscr{E}_l) \;:\; e_b \Downarrow v$ for some $\mathscr{F}_l$, $\prec_l$ and $\mathscr{E}_l$ that are computed by the functions *fun*, *rel* and *val*, respectively, as defined in Figures 3.10, 3.9 and 3.11. Let $\mathscr{F}_{\delta l}$ and $\mathscr{E}_{\delta l}$ be the function and variable environments computed by the functions fun^d and val^d, respectively, as defined in Figures 6.4 and 6.9. From Lemma 6.1.3 on page 145 we know that $\mathscr{F}_l \overset{\overset{\delta_c}{\rightarrow}}{\sqsubseteq} \mathscr{F}_{\delta l}$ holds. Furthermore, with Lemma 6.1.4 on page 146 we know that $\mathscr{F}_l \overset{d}{\equiv} \mathscr{F}_{\delta l}$ holds, as well. Thus, with the inductive assumption, we can follow $(\mathscr{F}_{\delta l}, \prec_l, \mathscr{E}'_l, \delta_c) \;:\; e'_b\mathscr{S}_b \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. Finally, we get by definition of rule Let in Figure 6.8 on page 136 that $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'\mathscr{S} \downarrow v'$.

Case 9: Function Application Lastly, e can be of the form `(`f $e_1 \;\cdots\; e_n$`)` and e' then has the form `(`f $e'_1\mathscr{S}_1 \;\cdots\; e'_n\mathscr{S}_n$`)` $\mathscr{S}$. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$ it follows that the function *match* as used in rule AP in Figure 3.8 on page 52 yields a single instance and that for all $j \in \{1, \ldots, n\}$ we have $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e_j \Downarrow v_j$. Furthermore, from the inductive assumption it follows that for all $j \in \{1, \ldots, n\}$ some δ_j exists with $\delta_c \vdash \delta_j \in \mathscr{S}_j$ such that $(\mathscr{F}_\delta, \prec, \mathscr{E}', \delta_c) \;:\; e'_j \downarrow v'_j$ with $v_j \overset{\delta_j}{\sqsubseteq} v'_j$. By definition of the function *apply* in Figure 6.7 on page 130, we know that the set of instances selected after the *lookup* stage corresponds to the set of instances selected after the second stage of the function *match*. Thus, each demand δ_j contains at least all discriminating labels for the corresponding patterns. With Lemmata 6.1.1 and 6.1.2, it then follows that after the fourth stage, *i.e.*, the *order* stage, the sets of instances returned in the functions *match* as defined in Figure 3.12 on page 60 and match^p as defined in Figure 6.10 on page 140 contain both the same instance. Let $\mathscr{F}_b$, $\prec_b$, $\mathscr{E}_b$ and e_b be the function environment, order on labels, variable environment and body expression as returned by the function *match*. Furthermore, let $\mathscr{F}_{\delta b}$, $\prec'_b$, $\mathscr{E}_{\delta b}$ and e'_b $\mathscr{S}_b$ be the function environment with demands, order on labels, variable environment with demands and body expression as returned by the function match^d. From $\mathscr{F} \overset{d}{\equiv} \mathscr{F}_\delta$, we know that $\prec_b = \prec'_b$. Furthermore, as all partial arguments satisfy the full arguments, we know that $\mathscr{E}_b \overset{\overset{\delta_\varsigma}{\rightarrow}}{\sqsubseteq} \mathscr{E}_{\delta b}$, as well. As $\mathscr{F}_b$ and $\mathscr{F}_{\delta b}$ are computed form $\mathscr{F}$ and $\mathscr{F}_\delta$, respectively, by peeling off the same number of nesting levels, we know that $\mathscr{F}_b \overset{d}{\equiv} \mathscr{F}_{\delta b}$. From $(\mathscr{F}, \prec, \mathscr{E}) \;:\; e \Downarrow v$, it follows that $(\mathscr{F}_b, \prec_b, \mathscr{E}_b) \;:\; e_b \Downarrow v$, as well. Thus, with the inductive assumption, we can follow that $(\mathscr{F}_{\delta b}, \prec_b, \mathscr{E}_{\delta n}, \delta) \;:\; e_b \;\; \mathscr{S}_b \downarrow v'$ with $v \overset{\delta}{\sqsubseteq} v'$. It follows by definition of rule LET in Figure 6.8 on page 136 that then $(\mathscr{F}_\delta, \prec, \mathscr{E}_\delta, \delta_c) \;:\; e \;\; \mathscr{S} \downarrow v$.

□

As I have shown, using the demand inference, partial evaluation can be guided to evaluate specified auxiliary computations only. However, so far, this only applies to expressions in LREC$_C$ without nested records. In the next section, I will extend the demand analysis such that the above holds for arbitrary expressions in LREC$_C$.

6.2. Extending the Binding-Time Analysis to Full LRec$_C$

The demand inference and guided partial evaluation presented in the previous section might not terminate for expressions in LREC$_C$ that compute nested records. As an example for such a case, consider the following definition of a function `last` on non-empty lists of Boolean values.

```
   let
2    fun
       last L{ List , Next}
4        = (last L.Next)
           L{ List , Last}
6        = L
     val L = true{ List , Next=false{ List , Last}}
8  in
     (last L)
10 end
```

Similar to the encoding of the `Expr` algebraic data type in Section 2.4, I use three labels to model non-empty lists. The first label, the tag `List`, is used to identify data as non-empty lists in general. To encode the structure of a list, I decompose the list into a nesting of one-element lists. The tag `Last` thereby identifies the last one-element list in a chain of one-element lists. The chain itself is constructed using the `Next` label. The label `Next` carries as value the remainder of the list. For both list components, the value element of the record contains the actual element of the list.

Line 7 above shows the encoding of the list (*true*, *false*) using the record structure described above. The last element is represented by the record expression `false{ List, Last}`. To model the full list, I then wrap the expression for the last element into a further record using the `Next` tag.

To access the last element of a list, I define in Line 3 a function `last` that recursively descends through the nested structure until it reaches an element with a `Last` tag.

The first instance in Line 3 thereby performs the recursive descend whereas the second instance in Line 5 is used to terminate the recursion.

Now, consider we want to know whether the result of an application of `last` carries the `Last` tag, *i.e.*, whether the function `Last` for a given argument actually returns the last one-element list. To derive this demand, I start out with a demand of $^{\{(Last,\emptyset)\}}$ on the application of `last` in Line 5 above. Below, the result after propagating the demands into the instances of `last` after one round of the fix-point iteration is given. To ease the presentation, I have only annotated the demands that are of interest in the following discussion.

```
   let
2    fun
       last L{ List, Next}$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset),(Next,\emptyset)\}}$
4        = (last L$^{\{(Last,\emptyset)\}\vdash\emptyset}$.Next)$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset)\}}$
           L{ List, Last}$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset),(Next,\emptyset)\}}$
6        = L$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset)\}}$
     val L = true{ List, Next=false{ List, Last}}
8  in
     (last L)$^{\bot\vdash\{(Last,\emptyset)\}}$
10 end
```

As can be seen, the demand has been propagated from the application of `last` in Line 9 to the body expression of each instance. For the second instance in Line 5, the demand derivation is trivial, as it is the identity on its argument. However, for the first instance in Line 3, the demand inference is more complex. The instance is defined by a recursive call to the function `last` with the `Next` component of the argument as parameter. As we have not yet inferred a demand for `last`, the propagation yields the empty demand. By adding the demand that results from the pattern, we get a final demand of $\{(Last,\emptyset)\}\vdash\{(Last,\emptyset),(Next,\emptyset)\}$ for both instances after round one of the fix-point iteration.

To check whether the fix-point has been reached, we have to perform a second round of demand inference. The resulting demand annotations are given below. Again, I have only annotated those demands that are of interest here.

```
   let
2    fun
       last L{ List, Next}$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset),(Next,\{(Last,\emptyset),(Next,\emptyset)\})\}}$
4        = (last L$^{\{(Last,\emptyset)\}\vdash\{(Next,\{(Last,\emptyset),(Next,\emptyset)\})\}}$.Next)$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset)\}}$
           L{ List, Last}$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset),(Next,\{(Last,\emptyset),(Next,\emptyset)\})\}}$
6        = L$^{\{(Last,\emptyset)\}\vdash\{(Last,\emptyset)\}}$
     val L = true{ List, Next=false{ List, Last}}
8  in
     (last L)$^{\bot\vdash\{(Last,\emptyset)\}}$
10 end
```

The demand derivation for the second instance in Line 5 above remains unchanged. However, as we already have inferred a demand for the function `last`, we can now propagate this demand in the recursive application in the second instance in Line 3.

Thus, we get a demand of $\{(\mathit{Last}, \emptyset)\} \vdash \{(\mathit{Last}, \emptyset), (\mathit{Next}, \emptyset)\}$ for the parameter of the recursive call. After nesting the demand, we get an additional demand on the argument `L` of $\{(\mathit{Last}, \emptyset)\} \vdash \{(\mathit{Next}, \{(\mathit{Last}, \emptyset), (\mathit{Next}, \emptyset)\})\}$. Finally, after adding in the demand of the other instance and the demand that arises from the pattern match, we get a demand of $\{(\mathit{Last}, \emptyset)\} \vdash \{(\mathit{Last}, \emptyset), (\mathit{Next}, \{(\mathit{Last}, \emptyset), (\mathit{Next}, \emptyset)\})\}$ on the parameter of the function `last`.

As the demand annotation has changed, we have not reached a fix-point, yet. Moreover, we will obviously never reach such a fix-point as during each iteration on the demand analysis, we get a further nesting of the demand for the label `Next`. The demand inference as defined in the previous section will thus never terminate.

These infinite demands are a general result of recursive functions on the nesting structure of record values. As such functions descend through a nested record until some property holds, *e.g.* until the argument carries the `Last` label in the above example, we cannot decide without knowing the argument at what level such function will terminate and, thus, how far the argument needs to be evaluated. For the example above, the demands inferred so far would already suffice. The argument as defined in Line 7 above only has one nesting level. Thus, if we propagate the demands as inferred so far, we get the desired result.

```
   let
2    fun
       last L{ List , Next}^{(Last,∅)}⊢{(Last,∅),(Next,{(Last,∅),(Next,∅)})}
4        = (last L^{(Last,∅)}⊢{(Next,{(Last,∅),(Next,∅)})} .Next)^{(Last,∅)}⊢{(Last,∅)}
           L{ List , Last}^{(Last,∅)}⊢{(Last,∅),(Next,{(Last,∅),(Next,∅)})}
6        = L^{(Last,∅)}⊢{(Last,∅)}
     val L = true{ List ,
8                  Next=false{ List ,
                               Last^⊥⊢∅
10                            }^⊥⊢{(Last,∅),(Next,∅)}
                }^⊥⊢{(Last,∅),(Next,{(Last,∅),(Next,∅)})}
12 in
     (last L)^⊥⊢{(Last,∅)}
14 end
```

For the value bound to `L` in Line 7, we get a demand that only requires the presence of the labels `Next` and `Last` to be know at each level. Thus, using the partial evaluation with demand as defined in Figure 6.8 on page 136, the value bound to `L` would only be evaluated to $\{(\mathit{Next}, \{(\mathit{Last}, \mathit{?})\})\}$. This, however, suffices to partially evaluate the application of the function `last` to the value $\{(\mathit{Last}, \mathit{?})\}$. Thus, we have found out that the result of the above example indeed carries the `Last` tag.

Remains the challenge to decide how far to compute an infinite demand annotation. The key observation here is that during partial evaluation, at each step only the top-level demands are required. In the above example, to evaluate the outer record expression that is bound to `L`, we only need to know that we have to evaluate the `Next` component. When, in turn, evaluating that component, we only need to know that the `Last` label is required. Similarly, to decide demand satisfaction as defined in Definition 6.1.3 on page 117 for a given demand and value, we only need to know the demand at each level of the nested value. And the list goes on. All properties of demands as used in the previous section either only inspect the top level of a demand or only inspect a finite number of levels that is given by some record value.

Thus, to incorporate infinite demands that arise from recursive functions on nested record values, I postpone the actual demand computation until evaluation time. The demand inference, instead of computing the demand on some expression, then computes abstract demands, *i.e.*, expressions that can be evaluated to some actual demand. These abstract demands then again are finite and thus demand inference terminates. Furthermore, as partial evaluation at each step only requires the top-level demand to be known, the evaluation of the abstract demands to such a top-level demand is finite again, as well. The argument here is the same as for finite demands that arise from non-nested record expressions: As the set of labels that are used in any expression in $\text{LR}\textsc{ec}_\text{C}$ is finite, such are all demands that can arise for each level of a nested record expression.

My approach uses the same idea as used in *lazy evaluation* [Friedman and Wise, 1976; Henderson and Morris, 1976] for functional programming languages like HASKELL. The classical example in that setting is the partial evaluation of infinite lists. Similar to delaying the evaluation of infinite demands until some part is actually needed, in lazy evaluation the elements of a list, and the actual structure of the list itself, are computed only if they are required for the result to be computed. If, for instance, a function only iterates over a finite prefix of an infinite list, only that prefix will be computed. Like in the demand analysis presented here, this allows programs on infinite lists to terminate if only finite prefixes of the list are required to compute the result.

It is worth noting here that I in this thesis use the equivalent of a call-by-name [Plotkin, 1975] evaluation strategy: Instead of computing demands, I propagate unevaluated demand expressions. In particular, these demand expressions might be copied multiple times into different contexts. This leads to a potentially significant overhead when computing the actual demands for each sub-expression of a program. However, as with lazy evaluation, an actual implementation might use a call-by-need strategy [Ariola et al., 1995; Maraist et al., 1998; Wadsworth, 1971], which memoizes the result of evaluating common demand expressions and reuses these results if multiple copies of a demand expression exist.

To formalise the delayed evaluation of demand annotations, I first define an applied lambda calculus for demand expressions. The corresponding syntax is shown by Figure 6.11 on the facing page. The production rule *demand* defines all valid demand expressions. Apart from the record demands as used in the demand annotations for non-nested record expressions, I additionally allow lambda abstractions on demands and the corresponding application of such an abstraction to a demand. I use a single variable Δ for abstractions, as demand annotations will only be parametrised by a single demand context. Furthermore, demand annotations may contain lifting, nesting and selection operations that correspond to the respective computation on demands in the non-nested case. Finally, a demand can be a union of two demands or the identifier Δ as used in lambda abstractions.

```
demand   ⇒  record
         |  λ Δ . demand
         |  ( identifier demand )
         |  ⇑ demand
         |  nest ( label , demand )
         |  demand . label
         |  demand ⊎ demand
         |  Δ

record   ⇒  { [ label = demand [ , label = demand ]* ] }
```

Figure 6.11.. Demand language in extended Backus Naur form.

In the following, I will use $\mathscr{D}$ to refer to the set of demand expressions that can be produced using the rule *demand* above and $\mathscr{D}_r$ to refer to only those demands that can be produced using the rule *record*.

I provide a formal semantics for this demand language in Figure 6.16 on page 165. They correspond to the standard semantics of an applied lambda calculus known from textbooks like [Pierce, 2002]. Before I discuss the semantics in detail, I first introduce the modified demand inference that computes appropriate demand annotations using the above language.

The demand inference scheme $\mathscr{A}_{\mathscr{F}}$ as shown in Figure 6.12 on the following page is parametrised with a function-demand environment $\mathscr{F}_A$, a variable environment $\mathscr{E}_A$, a binding environment $\mathscr{B}_A$, the demand to propagate δ and the expression to annotate e. It yields an amended function environment, an extended variable environment and the annotated expression.

The function environment used in the demand inference for abstract demands differs from the corresponding environment used in demand inference presented in the last section. Instead of using a nested environment to model the scoping of the `let` construct, I employ an α-conversion scheme when storing the demands for function arguments in the environment. The corresponding mapping from a function's name and its arity to the α-converted name of the demand is stored in the binding environment $\mathscr{B}_A$. That environment contains tuples of the form (f, n, i, α) where f is the function name, n its arity and i the parameter starting from 1 for the left-most parameter. The identifier α then is the α-converted name of the corresponding demand in the function environment $\mathscr{F}_A$.

The variable environment $\mathscr{E}_A$ is structurally identical to the corresponding environment of the demand inference for non-nested record expressions. However, instead of actual demands, it now stores demand expressions as define in Figure 6.11.

Lastly, the demand to propagate is a demand expression, as well. However, instead of propagating a fixed demand, I initially propagate the abstract demand Δ. This later enables the partial evaluation of the annotated expression with respect to any actual

(True) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{true}\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A, \mathscr{E}_A,\ \texttt{true}\ \lambda\Delta\texttt{.}\delta\ )$

(False) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{false}\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A, \mathscr{E}_A,\ \texttt{false}\ \lambda\Delta\texttt{.}\delta\ )$

(Unit) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{\textasciitilde}\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A, \mathscr{E}_A,\ \texttt{\textasciitilde}\ \lambda\Delta\texttt{.}\delta\ )$

(Var) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \alpha\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A, \mathscr{E}_A \xleftarrow{\mathscr{D}} (\alpha, \delta),\ \alpha\ \lambda\Delta\texttt{.}\delta\ )$

(Equal) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{(}e_1\ \texttt{=}\ e_2\texttt{)}\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A{''}, \mathscr{E}_A{''},\ \texttt{(}e_1'\ \texttt{=}\ e_2'\texttt{)}\ \lambda\Delta\texttt{.}\delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}_A{'}, \mathscr{E}_A{'}, e_1' & := & \mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \Uparrow(\delta),\ e_1\ \right\rrbracket \\ \mathscr{F}_A{''}, \mathscr{E}_A{''}, e_2' & := & \mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A{'}, \mathscr{E}_A{'}, \mathscr{B}_A, \Uparrow(\delta),\ e_2\ \right\rrbracket \end{array}$$

(Record) $\mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{\{}l_1\texttt{=}e_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n\texttt{\}}\ \right\rrbracket$

$\rightsquigarrow\ (\mathscr{F}_A{}^n, \mathscr{E}_A{}^n,\ \texttt{\{}l_1\texttt{=}e_1'\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n'\texttt{\}}\ \lambda\Delta\texttt{.}\delta\ )$

where

$$\begin{array}{lcl} \mathscr{E}_A{}^0 & := & \mathscr{E}_A \\ \mathscr{F}_A{}^0 & := & \mathscr{F}_A \\ \mathscr{F}_A{}^{i+1}, \mathscr{E}_A{}^{i+1}, e_{i+1}' & := & \mathscr{A}_{\mathscr{F}}\left\llbracket \mathscr{F}_A{}^i, \mathscr{E}_A{}^i, \mathscr{B}_A{}^i, \delta\texttt{.}l_{i+1},\ e_{i+1}\ \right\rrbracket \end{array}$$

Figure 6.12.. Demand analysis scheme $\mathscr{A}_{\mathscr{F}}$ for translating expressions in full LRec$_\text{C}$ into expressions in LRec$_\text{D}$ with demand annotations.

(SELECTION) $\mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ e.l \ \rrbracket$

$\rightsquigarrow\ ({\mathscr{F}_A}', {\mathscr{E}_A}',\ e'.l\ \lambda\Delta.\delta\ )$

where
$${\mathscr{F}_A}', {\mathscr{E}_A}', e' \ := \ \mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{B}_A, \mathscr{E}_A, \mathrm{nest}(l,\delta),\ e\ \rrbracket$$

(ANY) $\mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{any(}e_1 \cdots e_n\texttt{)}\ \rrbracket$

$\rightsquigarrow\ ({\mathscr{F}_A}^n, {\mathscr{E}_A}^n,\ \texttt{any(}e'_1 \cdots e'_n\texttt{)}\ \lambda\Delta.\delta\ )$

where
$$\begin{aligned}
{\mathscr{E}_A}^0 &:= \mathscr{E}_A \\
{\mathscr{F}_A}^0 &:= \mathscr{F}_A \\
{\mathscr{F}_A}^{i+1}, {\mathscr{E}_A}^{i+1}, e'_{i+1} &:= \mathscr{A}_{\mathscr{F}} \llbracket {\mathscr{F}_A}^i, {\mathscr{E}_A}^i, \mathscr{B}_A, \delta,\ e_{i+1}\ \rrbracket
\end{aligned}$$

(COND) $\mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{if}\ e_p\ e_t\ e_e\ \rrbracket$

$\rightsquigarrow\ ({\mathscr{F}_A}^3, {\mathscr{E}_A}^3,\ \texttt{if}\ e'_p\ e'_t\ e'_e\ \lambda\Delta.\delta\ )$

where
$$\begin{aligned}
{\mathscr{F}_A}', {\mathscr{E}_A}', e'_p &:= \mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \Uparrow(\delta),\ e_p\ \rrbracket \\
{\mathscr{F}_A}'', {\mathscr{E}_A}'', e'_t &:= \mathscr{A}_{\mathscr{F}} \llbracket {\mathscr{F}_A}', {\mathscr{E}_A}', \mathscr{B}_A, \delta,\ e_t\ \rrbracket \\
{\mathscr{F}_A}^3, {\mathscr{E}_A}^3, e'_e &:= \mathscr{A}_{\mathscr{F}} \llbracket {\mathscr{F}_A}'', {\mathscr{E}_A}'', \mathscr{B}_A, \delta,\ e_e\ \rrbracket
\end{aligned}$$

(GUARD) $\mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{guard(}e_1\ e_2\texttt{)}\ \rrbracket$

$\rightsquigarrow\ ({\mathscr{F}_A}'', {\mathscr{E}_A}'',\ \texttt{guard(}e'_1\ e'_2\texttt{)}\ \lambda\Delta.\delta\ )$

where
$$\begin{aligned}
{\mathscr{F}_A}', {\mathscr{E}_A}', e'_1 &:= \mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ e_1\ \rrbracket \\
{\mathscr{F}_A}'', {\mathscr{E}_A}'', e'_2 &:= \mathscr{A}_{\mathscr{F}} \llbracket {\mathscr{F}_A}', {\mathscr{E}_A}', \mathscr{B}_A, \Uparrow(\delta),\ e_2\ \rrbracket
\end{aligned}$$

(WITNESS) $\mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{witness(}e_v\ e_1 \cdots e_n\texttt{)}\ \rrbracket$

$\rightsquigarrow\ ({\mathscr{F}_A}^n, {\mathscr{E}_A}^n,\ \texttt{witness(}e'_v\ e'_1 \cdots e'_n\texttt{)}\ \lambda\Delta.\delta\ )$

where
$$\begin{aligned}
{\mathscr{F}_A}', {\mathscr{E}_A}', e'_v &:= \mathscr{A}_{\mathscr{F}} \llbracket \mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ e_v\ \rrbracket \\
{\mathscr{F}_A}^{i+1}, {\mathscr{E}_A}^{i+1}, e'_{i+1} &:= \mathscr{A}_{\mathscr{F}} \llbracket {\mathscr{F}_A}^i, {\mathscr{E}_A}^i, \mathscr{B}_A, \Uparrow(\delta),\ e_{i+1}\ \rrbracket
\end{aligned}$$

Figure 6.12.. Demand analysis scheme $\mathscr{A}_{\mathscr{F}}$ for translating expressions in full LREC$_C$ into expressions in LREC$_D$ with demand annotations (contd.).

(LET) $\mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{let}\ d_1 \cdots d_n\ \texttt{in}\ e\ \texttt{end} \right]\!\right]$

$$\rightsquigarrow\ ({\mathscr{F}_A}^3, {\mathscr{E}_A}'',\ \texttt{let}\ d'_1 \cdots d'_n\ \texttt{in}\ e'\ \texttt{end}\ \lambda\Delta.\delta\ )$$

where

$$\begin{array}{lcl} {\mathscr{F}_A}', {\mathscr{B}_A}', D & = & \text{derivefun}(\mathscr{F}_A, \mathscr{B}_A, (d_1, \ldots, d_n)) \\ {\mathscr{F}_A}'', {\mathscr{E}_A}', e' & = & \mathscr{A}_{\mathscr{F}} \left[\!\left[{\mathscr{F}_A}', \mathscr{E}_A, {\mathscr{B}_A}', \delta,\ e \right]\!\right] \\ {\mathscr{F}_A}^3, {\mathscr{E}_A}'', (d'_1, \ldots, d'_n) & = & \text{derivevar}({\mathscr{F}_A}'', {\mathscr{E}_A}', {\mathscr{B}_A}', D) \end{array}$$

(AP) $\mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{(}f\ e_1 \cdots e_n\texttt{)} \right]\!\right]$

$$\rightsquigarrow\ ({\mathscr{F}_A}^n, {\mathscr{E}_A}^n,\ \texttt{(}f\ e'_1 \cdots e'_n\texttt{)}\ \lambda\Delta.\delta\ )$$

where

$$\begin{array}{lcl} {\mathscr{E}_A}^0 & := & \mathscr{E}_A \\ {\mathscr{F}_A}^0 & := & \mathscr{F}_A \\ {\mathscr{F}_A}^{i+1}, {\mathscr{E}_A}^{i+1}, e'_{i+1} & := & \mathscr{A}_{\mathscr{F}} \left[\!\left[{\mathscr{F}_A}^i, {\mathscr{E}_A}^i, \mathscr{B}_A, \texttt{(}\text{apply}(\mathscr{B}_A, f, n, i)\ \delta\texttt{)},\ e_{i+1} \right]\!\right] \end{array}$$

Figure 6.12.. Demand analysis scheme $\mathscr{A}_{\mathscr{F}}$ for translating expressions in full LREC$_\text{C}$ into expressions in LREC$_\text{D}$ with demand annotations (contd.).

demand. Similarly, the demands that are derived for functions are parametrised by an abstract demand, as well. This makes the demand context as used in the non-nested case superfluous: Instead of deriving and annotating different demands for different contexts, I use parametrised demands and compute the actual demand when it is required.

Most rules for $\mathscr{A}_{\mathscr{F}}$ as shown in Figure 6.12 on page 156 are largely identical to their counterparts for the non-nested case given in Figure 6.3 on page 122. However, instead of computing a demand, I compute a demand expression. For example, in rule EQUAL, the demand expression $\Uparrow(\delta)$ is annotated at both sub-expressions. As each sub-expression is only processed once instead of once for each context in the non-nested case, I do not need to store sets of demands and join demand annotations. Instead, I simply annotate one demand expression. However, the demand δ that is passed as argument to the demand inference contains free occurrences of the abstract demand Δ that provide the current context. Therefore, I first wrap the demand δ into a lambda abstraction $\lambda\Delta$. This wrapped demand can then later be specialised to a given context by applying it to the corresponding context demand.

Apart from these minor changes, the main differences are in the rules LET for `let` constructs and the rule AP for function applications. In the rule LET first a new function environment and amended function definitions are computed using the function *derivefun*. As I derive abstract demands that are parametrised by the context of function application, I no longer need to postpone the derivation of the demands on functions to the application time. Instead, the function *derivefun* directly infers the demands for all function instances and rewrites the function definitions. The definition of *derivefun* is is given in Figure 6.13 on the following page. To derive the demands for the functions, I first extract all function definitions from the definitions contained in a `let` construct using the function *inst*. The resulting set of instances is structurally identical to the set of instances in the function environment used in the demand inference for non-nested record expressions. Furthermore, I compute a set A that contains for each defined function f of arity n a tuple (f, n). This set is used twofold. Firstly, I construct a new binding environment $\mathscr{B}_A{}'$ that contains fresh names for each argument of the newly defined functions. This step basically performs the α-conversion of function names. It is important here that in each extension of the binding environment new variables are used to ensure that no parasitic demand bindings are introduced at later stages. Secondly, I will use this set to later compute the set of discriminating labels for each function.

Using this extended binding environment, I then derive the demands for each instance. This is achieved using the function *deriveinst*. Given a function environment, the extended binding environment and the set of instances, it computes a new function environment and an updated set of instances with demand annotations. The new function environment needs to be propagated, as the instances might contain function definitions themselves. Their corresponding demands need to be incorporated into the overall function environment.

The function *deriveinst* derives the demand for each instance by applying the function *deriveinst′* to each single instance. That function computes the actual demands by applying the demand inference to the function body using the abstract demand Δ. Thus, the annotated demands will be parametrised by Δ. Furthermore, I use an empty variable environment as all functions in LREC$_C$ are closed. Similar to the demand inference for non-nested records, the demands for the parameters of the instance are extracted from the variable environment returned by the demand inference for the body expression. If no demand is annotated for a parameter, the empty demand $\emptyset$ is used.

After this step, the set of instances now contains annotated body expressions and an additional tuple for each instance that contains the demands for the parameters of that instance. However, so far these demands do not yet contain the demands that result from pattern matching. This demand is added using the function *pattern*. The function expects a function environment $\mathscr{F}_A$, a binding environment $\mathscr{B}_A$, a set of instances I and the (name of function,arity) tuples computed earlier. The latter are then used by the function *pattern′* to compute the final demand for each function definition f with a given arity n. This demand is computed for each parameter by collecting the demands for the corresponding parameter of each instance and adding the demand that arises from the pattern match. Note here that in the definition of *pattern′* the symbol $\uplus$ refers to the corresponding expression in the demand language and not the $\uplus$ operation used in the demand inference for non-nested record expressions. The resulting demands are then added to the function environment $\mathscr{F}_A$. As the demands contain unbound occurrences of the abstract demand Δ, I first wrap each into a lambda abstraction $\lambda\Delta$. Thus, the demand for each parameter can later be specialised for a specific context by applying the demand to that context.

The function *derivefun* is defined as

$$\text{derivefun}(\mathscr{F}_A, \mathscr{B}_A, (d_1, \dots, d_n)) := \text{derivefun'}(\mathscr{F}_A, \mathscr{B}_A, I_n, D_n)$$

where

$$\begin{aligned} I_0 &:= \emptyset \\ D_0 &:= () \\ I_{i+1}, D_{i+1} &:= \begin{cases} (I_i \cup (\text{inst}(f, \{\iota_1, \dots, \iota_n\})), D_i) & \text{if } d_{i+1} \equiv \texttt{fun}\ f\ \ \iota_1\ \cdots\ \iota_n, \\ (I_i, D_i + (d_{i_1}) & \text{otherwise,} \end{cases} \end{aligned}$$

with

$$\text{inst}(f, I) := \bigcup_{\iota \in I} \text{inst}'(f, \iota)$$

where

$$\text{inst}'(f, \alpha_1\ p_1\ \cdots\ \alpha_n\ p_n\ \texttt{=}\ e) := (f, (((\alpha_1, p_1), \dots, (\alpha_n, p_n)), e)).$$

The function *derivefun*′ is then defined as

$$\text{derivefun}'(\mathscr{F}_A, \mathscr{B}_A, I, D) := (\mathscr{F}_A{}'', \mathscr{B}_A{}', D')$$

where $\mathscr{B}_A{}'$, $\mathscr{F}_A{}''$ and D' are defined as

$$\mathscr{B}_A{}' := \{(f', m, l, \beta) \in \mathscr{B}_A \mid f \neq f'\} \cup \bigcup_{(f,n) \in A} \{(f, n, 1, \alpha_1), \dots, (f, n, n, \alpha_n)\}$$

with all α being fresh demand variables,

$$\begin{aligned} \mathscr{F}_A{}', I' &:= \text{deriveinst}(\mathscr{F}_A, \mathscr{B}_A{}', I), \\ \mathscr{F}_A{}'', I'' &:= \text{pattern}(\mathscr{F}_A{}', \mathscr{B}_A{}', I, A), \text{ and} \\ D' &:= \text{disperse}(I'', \mathscr{B}_A, D). \end{aligned}$$

The set A above is defined as

$$A := \{(f, n) \mid (f, ((a_1, \dots, a_n), e)) \in I\}.$$

The function *deriveinst* is defined as

$$\text{deriveinst}(\mathscr{F}_A, \mathscr{B}_A, \{\iota_1, \dots, \iota_n\}) := (\mathscr{F}_A{}^n, I^n)$$

with

$$\begin{aligned} \mathscr{F}_A{}^0 &:= \mathscr{F}_A \\ \mathscr{F}_A{}^{i+1}, \iota'_{i+1} &:= \text{deriveinst}'(\mathscr{F}_A{}^i, \mathscr{B}_A, I_i, \iota_{i+1}) \end{aligned}$$

Figure 6.13.. Definition of function *derivefun* as used in Figure 6.12 on page 156.

where *deriveinst′* is defined as

$$\text{deriveinst}'(\mathscr{F}_A, \mathscr{B}_A, (f, (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e))) := \mathscr{F}_A{}', I'$$

with

$$\begin{aligned} \mathscr{F}_A{}', \mathscr{E}_A, e' :=&\ \mathscr{A}_{\mathscr{F}}\ [\![\mathscr{F}_A, \emptyset, \mathscr{B}_A, \Delta,\ \texttt{e}\ ]\!] \\ \delta_i :=&\ \begin{cases} \delta_{\alpha_i} & \text{if } (\alpha_i, \delta_{\alpha_i}) \in \mathscr{E}_A, \\ \emptyset & \text{otherwise,} \end{cases} \quad \text{for all } i \in \{1, \ldots, n\}, \end{aligned}$$

and

$$\iota' := (f, (((\alpha_1, p_1), \ldots, (\alpha_n, p_n,)), e'), (\delta_1, \ldots, \delta_n)).$$

The function *pattern* is defined as

$$\text{pattern}(\mathscr{F}_A, \mathscr{B}_A, I, (a_0, \ldots, a_n)) := \mathscr{F}_A{}'$$

where

$$\begin{aligned} \mathscr{F}_{A0} :=&\ \mathscr{F}_A \\ \mathscr{F}_{Ai+1} :=&\ \text{pattern}'(\mathscr{F}_{Ai}, \mathscr{B}_A, I, a_{i+1}) \end{aligned}.$$

The function *pattern′* is defined as

$$\text{pattern}'(\mathscr{F}_A, \mathscr{B}_A, I, (f, n)) := \mathscr{F}_A{}'$$

with

$$\begin{aligned} \delta_i^\iota :=&\ \biguplus_{(f,(s,e),(\delta_1',\ldots,\delta_n'))\in I} \delta_i' \\ \delta_i^p :=&\ (\bigcup_{(f,(((\alpha_1,p_1),\ldots,(\alpha_n,p_n)),e),\delta)\in I} p_i \setminus \bigcap_{(f,(((\alpha_1,p_1),\ldots,(\alpha_n,p_n)),e),\delta)\in I} p_i) \times \{\emptyset\} \\ \delta_i :=&\ \delta_i^\iota \uplus \delta_i^p \end{aligned}$$

for all $i \in \{1, \ldots, n\}$ and

$$\mathscr{F}_A{}' := \mathscr{F}_A \cup \{(\alpha, \lambda\Delta\,.\,\delta_i) \mid i \in \{1, \ldots, n\} \wedge (f, n, i, \alpha) \in \mathscr{B}_A\}.$$

Lastly, the function *disperse* is defined as

$$\text{disperse}(\{\iota_1, \ldots, \iota_n\}, D, \mathscr{B}_A) := D_n$$

where

$$\begin{aligned} D_0 :=&\ D \\ D_{i+1} :=&\ D_i + \text{disperse}'(\iota_i, \mathscr{B}_A) \end{aligned}$$

with

$$\text{disperse}'((f, (((\alpha_1, p_1), \ldots, (\alpha_n, p_n)), e)), \mathscr{B}_A) := (\texttt{fun}\ f\ \ \alpha_1\ \ p_1\ \ \beta_1\ \cdots\ \alpha_n\ \ p_n\ \ \beta_n\ \texttt{=}\ e)$$

where for all $i \in \{1, \ldots, n\}$ the tuple $(f, n, i, \beta_i) \in \mathscr{B}_A$.

Figure 6.13.. Definition of function *derivefun* as used in Figure 6.12 on page 156 (contd.).

$$\text{derivevar}(\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, D) := (\mathscr{F}_{An}, \mathscr{E}_{An}, D_n)$$

with

$$\begin{aligned}
\mathscr{F}_{A0}, \mathscr{E}_{A0}, D_0 &:= (\mathscr{F}_A, \mathscr{E}_A, ()) \\
\mathscr{F}_{Ai+1}, \mathscr{E}_{Ai+1}, D_{i+1} &:= \begin{cases} (\mathscr{F}_{Ai}', \mathscr{E}_{Ai}', D_i + (\texttt{var}\ \alpha\ \texttt{=}\ e')) & \text{if } d_{n-i} \equiv \texttt{var}\ \alpha\ \texttt{=}\ e \\ (\mathscr{F}_{Ai}, \mathscr{E}_{Ai}, D_i + (d_{i+1})) & \text{otherwise.} \end{cases}
\end{aligned}$$

where

$$\mathscr{F}_{Ai}', \mathscr{E}_{Ai}', e' := \text{derivevar}'(\mathscr{F}_{Ai}, \mathscr{E}_{Ai}, \mathscr{B}_A, \alpha, e).$$

The function derivevar$'$ is defined as

$$\text{derivevar}'(\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \alpha, e) := (\mathscr{F}_A{}', \mathscr{E}_A{}'', e')$$

with

$$\begin{aligned}
\mathscr{E}_A{}' &:= \{(\beta, \delta') \in \mathscr{E}_A \mid \alpha \neq \beta\} \\
\mathscr{F}_A{}', \mathscr{E}_A{}'', e' &:= \begin{cases} \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A{}', \mathscr{B}_A, \delta,\ e \right]\!\right] & \text{if } (\alpha, \delta) \in \mathscr{E}_A, \\ \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A{}', \mathscr{B}_A, \emptyset,\ e \right]\!\right] & \text{otherwise.} \end{cases}
\end{aligned}$$

Figure 6.14.. Definition of function *derivevar* as used in Figure 6.12 on page 156.

As the last step in the derivation of demands for functions, I transform the function environment $\mathscr{F}_A$ back to function definitions. This is done using the function *disperse*. For each instance ι in the set of instances I, I create a corresponding function definition. However, instead of using the demands that are annotated in ι, I annotate the identifiers that correspond to the demands for that instance in the function environment $\mathscr{F}_A$. This is done to keep the annotations consistent: The demands annotated in ι do not contain demands from other instances or those demands that arise due to the pattern match. However, the annotated demands are not used for evaluation and thus serve merely a documentation purpose.

After deriving demands for function definitions in rule Let, I next derive the demand for the body expression. The body expression needs to be handled first, as the demand inference is a bottom up process. Again, the body of the `let` expression may contain further function definitions, thus I propagate the resulting function environment.

Lastly, the demands for variable bindings are computed. This step is performed using the function *derivevar* as defined in Figure 6.14. The function closely corresponds to its counterpart *propvar* for demand inference in the non-nested case as defined in Figure 6.5 on page 128. The function *derivevar* annotates the set of definitions D by applying the helper function *derivevar$'$* bottom-up to each instance definition. This helper function rewrites a variable binding by first computing a new variable environment where the binding for the currently processed variable has been removed and then rewriting the body expression. The latter is done with the demand that is annotated for the variable, The function *apply* is defines as

$$\text{apply}(\mathscr{B}_A, f, n, i) := \begin{cases} \alpha & \text{if } (f, n, i, \alpha) \in \mathscr{B}_A), \\ \lambda\Delta\,.\,\emptyset & \text{otherwise.} \end{cases}$$

Figure 6.15.. Definition of function *apply* as used in Figure 6.12 on page 156.

if such demand exists, or the empty demand otherwise. All definitions that are not variable bindings are left as is.

Thus, after the *derivevar* stage, the entire `let` construct has been annotated with demands. It is important here that the modified binding environment $\mathscr{B}_A$ is used throughout the `let` construct but not passed back further up. This corresponds to the scoping rules for function definitions in LREC$_C$. Similarly, after the *derivevar* stage, all local bindings have been removed from the variable environment $\mathscr{E}_A$. Only the function environment $\mathscr{F}_A$ is passed back up. This is important, as demands inferred for variables that are defined outside the scope of the current `let` construct may contain references to the demand of the parameters of local functions. To later be able to evaluate these references, we need to keep the corresponding demands in the function environment.

The last rule in Figure 6.12 on page 156 is the rule for function applications AP. As can be seen, for the function application itself, the current demand wrapped in a lambda abstraction is annotated. Furthermore, the demand for all argument expressions is computed by applying the demand inference to the corresponding expressions. However, instead of computing a specialised demand for the current context as done in the non-nested case, I construct a demand expression that evaluates to this demand. As a first step, the demand expression for the corresponding function parameter is looked up in the function environment using the function *apply* as defined in Figure 6.15. The function *apply* expects the current binding environment $\mathscr{B}_A$, a function name f, the arity of the function n and the number i of the parameter counting left to right starting with 1. Using this information, the function *apply* looks up the corresponding identifier in the binding environment. This identifier refers to the demand expression for that function parameter in the function environment $\mathscr{F}_A$. If no such identifier can be found, *i.e.*, if the corresponding function does not exist in the current scope, the demand expression $\lambda\Delta\,.\,\emptyset$ is returned. It evaluates to the empty demand regardless of the context. An alternative design would be to fail the demand inference, as the corresponding expression cannot be evaluated. However, I have chosen the first approach to ensure that the demand inference succeeds for all expressions in LREC$_C$.

In either case, the demand for each parameter is a λ-abstraction. As a second step, to specialise each parameter to the context of the application at hands, I apply the parameters to the current demand. The resulting demand expressions are then used to infer the demands for the arguments.

This completes the definition of the extended demand inference for arbitrary expressions in LREC$_C$. Using the above definition, I can now derive demands for the example of the function `last` given at the beginning of this section. The corresponding annotated expression is given below. As before, I have only annotated those demands that are of interest for the discussion.

```
   let
2    fun
       last L{ List, Next}^{λΔ.α}
4        = (last L^{λΔ.nest(Next,(αΔ))}.Next^{λΔ.(αΔ)})^{λΔ.Δ}
            L{ List, Last}^{λΔ.α}
6        = L^{λΔ.Δ}
     val L = true{ List,
8                  Next=false{ List,
                               Last^{λΔ.α.Next.Last}
10                           }^{λΔ.α.Next}
                }^{λΔ.α}
12 in
     (last L)^{λΔ.Δ}
14 end
```

The corresponding function environment $\mathscr{F}_A$ returned by the demand inference contains a single entry for α with the demand $\lambda\Delta.\mathtt{nest}(\mathtt{Next},(\alpha\Delta))\uplus\Delta\uplus\{(\mathtt{Next},\emptyset),(\mathtt{Last},\emptyset)\}$. The first component of this demand stems from the first instance in Line 3 above. It captures the recursive nature of that instance with a recursive application of the overall demand α. Obviously, this component encodes an infinite demand. The second component of the demand corresponds to the second instance in Line 5. As that instance is the identity on its argument, the demand for that instance simply is the abstract demand Δ. Lastly, the third component of the demand arises from the pattern match. It contains the empty demand for each discriminating label, *i.e.*, for the labels `Next` and `Last`.

To make use of the above demand annotations for partial evaluation, I next define the evaluation of demand expressions to top-level normal form. As a reminder: To guide the partial evaluation, at each step only the top-level demands of a nested record demand are required. A corresponding small-step operational semantics that computes these top-level demands from demand expressions is given in Figure 6.16 on the next page. The statement $\mathscr{F}_A : e \rightarrow e'$ is to be read as: Given a function environment $\mathscr{F}_A$, the expression e can be reduced to the expression e'. The function environment $\mathscr{F}_A$ thereby is the function environment returned by the demand inference that has computed the expression e.

The first rule, *i.e.*, rule Apply, defines the reduction of function applications. A function application can be reduced if the function environment contains a corresponding lambda abstraction. The reduced expression is then the body of that lambda abstraction where all free occurrences of the abstract demand Δ have been substituted by the argument expression. The substitution function $\delta_b[\Delta/\delta]$ is defined in the usual way [Pierce, 2002] and does not substitute bound occurrences, *i.e.*, those occurrences that are protected by a surrounding lambda abstraction.

The next three rules define the reduction of the lifting of demands $\Uparrow$. Rule Lift defines the reduction in case the argument is not yet a record demand. Then the lift operation

$$\textsc{Apply} \quad : \quad \frac{(\alpha, \lambda\Delta\,.\,\delta_b) \in \mathscr{F}_A}{\mathscr{F}_A : (\alpha\ \ \delta) \to \delta_b[\Delta/\delta]}$$

$$\textsc{Lift} \quad : \quad \frac{\delta \notin \mathscr{D}_r \quad \mathscr{F}_A : \delta \to \delta'}{\mathscr{F}_A : \Uparrow(\delta) \to \Uparrow(\delta')}$$

$$\textsc{LiftEmpty} \quad : \quad \frac{\delta = \{\}}{\mathscr{F}_A : \Uparrow(\delta) \to \{\}}$$

$$\textsc{LiftNEmpty} \quad : \quad \frac{\delta \in \mathscr{D}_r \quad \delta \neq \{\}}{\mathscr{F}_A : \Uparrow(\delta) \to \uparrow\{\delta\}}$$

$$\textsc{Nest} \quad : \quad \frac{}{\mathscr{F}_A : \texttt{nest}(l, \delta) \to \{(l,\delta)\}}$$

$$\textsc{Selection} \quad : \quad \frac{\delta \in \mathscr{D}_r}{\mathscr{F}_A : \delta\,.\,l \to \mathrm{elem}_d(\delta, l)}$$

$$\textsc{SelArg} \quad : \quad \frac{\delta \notin \mathscr{D}_r \quad \mathscr{F}_A : \delta \to \delta'}{\mathscr{F}_A : \delta\,.\,l \to \delta'.l}$$

$$\textsc{Union} \quad : \quad \frac{\delta_1 \in \mathscr{D}_r \quad \delta_2 \in \mathscr{D}_r}{\mathscr{F}_A : \delta_1 \uplus \delta_2 \to \delta_1 \uplus \delta_2}$$

$$\textsc{UnionL} \quad : \quad \frac{\delta_1 \notin \mathscr{D}_r \quad \mathscr{F}_A : \delta_1 \to \delta_1'}{\mathscr{F}_A : \delta_1 \uplus \delta_2 \to \delta_1' \uplus \delta_2}$$

$$\textsc{UnionR} \quad : \quad \frac{\delta_2 \notin \mathscr{D}_r \quad \mathscr{F}_A : \delta_2 \to \delta_2'}{\mathscr{F}_A : \delta_1 \uplus \delta_2 \to \delta_1 \uplus \delta_2'}$$

Figure 6.16.. Small step operational semantics for the demand language defined in Figure 6.11 on page 155.

can be reduced if the argument can be reduced and the result is the lifting of the reduced argument. In case the argument is the empty demand, rule LiftEmpty allows to reduce the lifting of the demand to the empty demand, as well. For the other case where the argument is a non-empty record demand, we can reduce the demand to the corresponding demand for full evaluation. These two rules correspond to the definition of lifted demands in Definition 6.1.7 on page 125.

The next rule, *i.e.*, rule Nest, handles the nest operation on demands. The resulting demand is then a new record demand that contains the label given as argument and the demand given as argument as that label's demand. Note here that the argument of a nest operation is never reduced. As it is not part of the top-level, it need not be reduced for the top-level normal from.

For selection operations, the semantics are defined in rules Selection and SelArg. In case the demand argument of a selection operation already is a record demand,the rule Selection allows the reduction to the corresponding element. If, however, the demand argument is not yet a record demand, it can be reduced using rule SelArg.

Lastly in Figure 6.16 on the previous page, I define three rules for the reduction of the $\uplus$ operation. The first of these, the rule Union, defines the union of two record demands. Note that the $\uplus$ sign on the right hand side of the reduction rule refers to the union of demands as defined in Definition 6.1.4 on page 118. In case that either of the two arguments is not yet a record demand, the rules UnionL and UnionR allow for reduction of the left or right argument, respectively.

Deliberately, the semantics for top-level normal form reduction do not contain a rule for record demands. These need not be evaluated further as they already are in top-level normal form.

Using these semantics, I can define the notion of top-level normal form as follows:

Definition 6.2.1 (Top-Level Normal Form)**.** *Given a demand expression e and a corresponding function environment $\mathscr{F}_A$. We say e is in top-level normal form, if no demand expression e' exists with*

$$\mathscr{F}_A \ : \ e \rightarrow e'.$$

Thus, the top-level normal form of an expression is computed by applying the reduction relation $\rightarrow$ until no further reduction is possible. In the following, I will denote this reduction to top-level normal from using the reduction operation $\overset{*}{\rightarrow}$.

As an example, consider the previous example now annotated with reduced demands below. I have used the context demand $\{(\mathit{Last}, \emptyset)\}$, *i.e.*, I have applied all annotated abstractions to that demand.

```
let
  fun
    last L{ List, Next}$^{\alpha}$
      = (last L$^{\{(Next,(\alpha\{(\mathrm{Last},\emptyset)\}))\}}$.Next$^{\{(Next,(\alpha\{(Last,\emptyset)\})),(Last,\emptyset)\}}$
         L{ List, Last}$^{\alpha}$
      = L$^{\{(Last,\emptyset)\}}$
  val L = true{ List,
                 Next=false{ List,
                                              Last^∅
10                                        }^{(Next,(α{(Last,∅)})),(Last,∅)}
                          }^{(Next,(α{(Last,∅)})),(Last,∅)}
12  in
      (last L)^{(Last,∅)}
14  end
```

As can be seen, the reduced demands encode the required information for guided partial evaluation. In particular, the expression bound to the identifier `L` in Line 7 will only be evaluated up to the partial value $\{(Next, \{(Last, ?)\})\}$, which suffices to compute the final partial result $\{(Last, ?)\}$.

To incorporate the reduction of the annotated demand expressions to top-level normal form into the guided partial evaluation as defined in Figure 6.8 on page 136, it suffices to redefine the functions extract, empty and $\overline{\text{empty}}$ as follows.

Definition 6.2.2 (Demand Extraction for Demand Expressions)**.** *Given a function environment for demand expressions $\mathscr{F}_A$. Let $e\ \delta$ be an annotated expression in* LREC$_D$ *and δ_c be a context demand. Then the extraction of the demand from e in context δ_c is defined as*

$$extract(\delta_c, \delta) := \delta'$$

where $\mathscr{F}_A\ :\ (\delta\delta_c) \xrightarrow{*} \delta'$.

Thus, to extract a demand for a given context from a demand annotation, the context demand is applied to the annotated demand expression. The corresponding demand in top-level normal form is then the demand for that expression in the given context.

Similarly, the functions empty and $\overline{\text{empty}}$ can be redefined.

Definition 6.2.3 (Empty Demand Expression)**.** *Given a function environment for demand expressions $\mathscr{F}_A$. Let $e\ \delta$ be an annotated expression in* LREC$_D$ *and δ_c be a context demand. Then the predicate $empty(\delta_c, \delta)$ is defined as*

$$empty(\delta_c, \delta) := \mathscr{F}_A\ :\ (\delta\delta_c) \xrightarrow{*} \emptyset.$$

Definition 6.2.4 (Non-Empty Demand Expression)**.** *Given a function environment for demand expressions $\mathscr{F}_A$. Let $e\ \delta$ be an annotated expression in* LREC$_D$ *and δ_c be a context demand. Then the predicate $\overline{empty}(\delta_c, \delta)$ is defined as*

$$\overline{empty}(\delta_c, \delta) := \mathscr{F}_A\ :\ (\delta\delta_c) \xrightarrow{*} \delta' \wedge \delta' \neq \emptyset.$$

As can be seen in the above definitions, a demand annotation is empty if the application of the demand expression to the context demand yields the empty demand under top-level normal form reduction; it is not empty, if the application of the demand expression to the context demand yields any non-empty demand under top-level normal form reduction.

Using these definitions, the semantics for guided partial evaluation presented in Figure 6.8 on page 136 applies to expressions annotated using the demand inference for expressions in full LREC$_C$, as well. As a last definition, I extend the notion of demand satisfaction to demand expressions, as well.

Definition 6.2.5 (Demand Satisfaction for Demand Expressions). *Let $\mathscr{F}_A$ be a function environment for demand expressions. Given a value $v \in \mathscr{V}$ and a partial value $v' \in \mathscr{V}_p$ with $v \sqsubseteq v'$. For any demand expression $\delta \in \mathscr{D}$ and context demand $\delta_c \in \mathscr{D}_r$, we say that v' satisfies the demand $(\delta\delta_c)$ with respect to v, or $v \overset{(\delta\delta_c)}{\sqsubseteq} v'$ for short, if one of the following statements holds:*

1. *$v' = ?$ and $\mathscr{F}_A \;:\; (\delta\delta_c) \overset{*}{\to} \emptyset$*

2. *$v \notin \mathscr{R}$ and $v' \neq ?$*

3. *$v \in \mathscr{R}$ and $v' \in \mathscr{R}_p$ and $\mathscr{F}_A \;:\; (\delta\delta_c) \overset{*}{\to} \delta'$ for some $\delta' \in \mathscr{D}_r$ and $dom(v) \cap \mathrm{dom}_d(\delta') \setminus dom_p(v') = \emptyset$ and $\mathrm{dom}_d(\delta') \subseteq dom_p(v') \cup \overline{dom_p}(v')$ and $\forall l \in dom_p(v') : elem(v, l) \overset{\delta'.l}{\sqsubseteq} elem_p(v', l)$*

4. *$v \in \mathscr{R}$ and $v' \in \mathscr{R}_p$ and $\mathscr{F}_A \;:\; (\delta\delta_c) \overset{*}{\to} \uparrow\{\delta'\}$ for some $\delta' \in \mathscr{D}_r$ and $dom(v) = dom_p(v')$ and $\mathrm{dom}_d(\delta') \setminus dom(v) \subseteq \overline{dom_p}(v')$ and $\forall l \in dom(v) : elem(v, l) \overset{\delta'.l}{\sqsubseteq} elem_p(v', l)$.*

In the above definition, demand satisfaction is now defined by first evaluating the demand expression to top-level normal form and then applying the same rules as in the original Definition 6.1.3 on page 117. Using this adapted notion of demand satisfaction, the property shown in Theorem 6.1.4 on page 146 holds for expressions in full LREC$_\mathrm{C}$, as well. Even the proof remains largely unchanged. Thus, using the demand inference for full LREC$_\mathrm{C}$, any expression in LREC$_\mathrm{C}$ that can be evaluated can be partially evaluated to satisfy a given demand, as well.

Before I end this chapter with a final example, I first discuss two special cases of top-level normal form reduction of demand expressions, for which the corresponding semantics presented in Figure 6.16 on page 165 alone do not suffice. In particular, top-level normal form reduction may not terminate, even though the top-level normal form as such is finite. As a first example, consider the following expression in LREC$_\mathrm{C}$:

```
  let
2   fun id A{} = (id A)
  in
4   (id true{})
  end.
```

In Line 2 above, I define a recursive function `id` as the application of itself to its single argument. Obviously, the above function does not terminate. However, the demand inference does and yields the following demand annotations:

```
1 let
    fun id A{}^α = (id A^λΔ.(αΔ))^λΔ.Δ
3 in
    (id true{}^λΔ.α)^λΔ.Δ
5 end.
```

The corresponding function environment $\mathscr{F}_A$ contains a single definition for the identifier α as $\lambda\Delta$.($\alpha\Delta$). Thus, the demand for the argument to the application of `id` in Line 4 above is a recursive function, as well. However, it never produces any demand and only replicates itself during reduction using the rule AP in Figure 6.16 on page 165. As a consequence, the top-level normal form reduction never terminates.

This kind of empty cycle in demand expressions needs to be detected and the reduction stopped once no more top-level demands arise. A second kind of non-terminating demand expressions may arise in recursive record constructions. As an example, consider the LREC$_C$ expression below:

```
1  let
     fun wrap A{}
3          = (wrap true{ Wrapped=A})
       wrap A{ Wrapped}
5          = A
   in
7    (wrap true{})
   end
```

The function `wrap` defined in lines 2ff. above wraps an argument into a new record value with the component `Wrapped` that contains the original value. In case the argument already is wrapped, *i.e.*, in case it already has a `Wrapped` component, the argument is returned. To trigger a cyclic demand, I use a recursive application of `wrap` in the first instance. This application is the identity on the argument, as the newly constructed record always carries a `Wrapped` component. We can derive demands for the above example as follows:

```
   let
2    fun wrap A{}^α
           = (wrap true{ Wrapped=A^λΔ.(αΔ).Wrapped}^λΔ.(αΔ))^λΔ.Δ
4      wrap A{ Wrapped}^α
           = A^λΔ.Δ
6  in
     (wrap true{})^λΔ.Δ
8  end
```

In the above example, α is defined in $\mathscr{F}_A$ as $\lambda\Delta$.($\alpha\Delta$).Wrapped$\uplus\Delta\uplus${(Wrapped,$\emptyset$)}. The first component arises from the first instance in Line 2 above. As the second instance in Line 4 is the identity on its argument, we get Δ as the second component. Lastly, the this component is the demand due to the pattern match.

If we now start to reduce the above demand to top-level normal from using for example the context $\{$*Wrapped*$, \emptyset)\}$, we get after some reductions the demand {(Wrapped,$\emptyset$) $\uplus$(α{(Wrapped,$\emptyset$)}).Wrapped.Wrapped}. The second component of this demand, if further reduced, keeps producing nestings of selection operations using the label `Wrapped`. However, these demands refer to levels above the current top-level and thus are empty with respect to the current top-level. Like the first example for empty cycles, these demands on higher levels need to be detected and removed, as well.

6.3. Examples, Revisited

To conclude this chapter, I revisit the example of the algebraic expression data type first presented in Section 2.4. As promised there, I will show that by using the demand analysis to guide partial evaluation, it is indeed possible to statically infer the kind of an expression.

To start, I below give the partly de-sugared version of the definition of the data type constructors from Section 2.4. I have replaced implicit equality constraints, implicit bindings of labels and contracts by their lowered equivalent. However, for the sake of readability, I have kept the implicit label of the value component of records. When applying the demand inference and semantic rules for evaluation, these have to be rewritten on the fly. I will, however, use the explicit value label in the demand annotations.

For the tuples used in those examples, I use the demand annotations and demand derivation as described in Appendix B.2. I omit a detailed description here. For the following example, it suffices to know that tuples are rewritten into records with numerical indices as labels.

```
let
  fun ENum I{}^α1
    = (!I){ Expr, ENum, Kind=Int}
  fun EBool B{}^β1
    = (!B){ Expr, EBool, Kind=Bool}
  fun ECond P{ Expr, Kind}^γ1 T{ Expr, Kind}^γ2 E{ Expr, Kind}^γ3
    = let
        P = guard(P (P.Kind = Bool))
        T = guard(T (T.Kind = E.Kind))
        E = guard(E (T.Kind = E.Kind))
      in
        witness((P, T, E){ Expr, ECond, Kind=any(T.Kind E.Kind)}
                (P.Kind = Bool)
                (T.Kind = E.Kind))
      end
  fun EIsZero E{ Expr, Kind}^δ1
    = let
        E = guard(E (E.Kind = Int))
      in
        witness((E){ Expr, EIsZero, Kind=Bool}
                (E.Kind = Int))
      end
  fun EDiv A{ Expr, Kind}^ε1 B{ Expr, Kind}^ε2
    = let
        A = guard(A (A.Kind = Int))
        B = guard(B (B.Kind = Int))
      in
        witness((A, B){ Expr, EDiv, Kind=Int}
                (A.Kind = B.Kind)
                (A.Kind = Int)
                (B.Kind = Int))
```

```
32        end
   in
34   (EIsZero (EBool true))^Δ
   end
```

As can be seen in lines 6ff., 16ff. and 23ff. above, the pattern guards have been replaced by explicit data-flow annotations using the `guard` and `witness` operations. Furthermore, I have annotated the abstract demands at each function parameter. To preserve space and enhance readability, I have not annotated the full demand expressions but I use place holders instead. The actual demands are given in Figure 6.17 on the next page. I use a Greek letter to denote the demands for each function and a numerical subscript to specify which parameter of a function a demand belongs to. The first parameter thereby has subscript 1 and subscripts increment left to right.

For the first data constructor, *i.e.*, the `ENum` function, I use the Greek letter α. As can be seen in Figure 6.17 on the following page, the demand for the value component of the single argument of the function `ENum` is, rather straight forward, the demand on the first component of the function's result's value. The function `ENum` uses no pattern guards and thus no demands from `guard` or `witness` operations arise. Furthermore, its result is computed by wrapping the value of the argument into a tuple and then using that tuple as value component of the result.

Similarly, the demand for the `EBool` constructor can be inferred. As it computes its result in a similar manner, the result demand annotation β_1 is identical.

The demand annotation for the third data constructor is more complex. As can be seen in lines 6ff., the function `ECond` contains `guard` and `witness` operations. This shows in the resulting demand annotations, as well. As an example, consider the demand γ_1 for the first argument of `ECond`. The demand consists of three parts. The first part, *i.e.*, Δ`.val.#1`, results from the construction of the result record: The record contains the first argument as first element of the tuple used as the record's value component.

The second part of the demand annotation, *i.e.*, $\text{nest}(\mathit{Kind}, \Uparrow(\Uparrow(\Delta)))$, arises from the `witness` operation. The first guard expression in Line 13 requires the `Kind` component of the first argument to be known in order to be evaluated. Thus, if the overall result of the `witness` operation is needed at all, the `Kind` component needs to be evaluated. The double nesting of $\Uparrow$ results from the demand inference rules for `witness` and `=` operations. Each passes on the demand for full evaluation if the result of the respective operation is required at all. The surrounding nest then passes this demand, if it is non-empty, on to the `Kind` component.

Lastly, the third part of the demand, *i.e.*,

$$\text{nest}(\mathit{Kind}, \Uparrow(\Uparrow(\Delta\texttt{.val.\#1} \uplus \text{nest}(\mathit{Kind}, \Uparrow(\Uparrow(\Delta)))))),$$

is triggered by the `guard` operation in Line 8. To use the first argument, the corresponding guard expression needs to evaluate to true. Thus, if the demand on the first argument, which so far consists of the first two parts of the overall demand, is non-empty, the `Kind` component of the first argument's value needs to be known. Again, the double $\Uparrow$ artifact is caused by the use of the `=` operation inside of a `guard` operation.

$$
\begin{array}{lll}
\alpha_1 & \lambda\Delta. & \mathrm{nest}(\mathit{val}, \Delta\texttt{.val.\#1}) \\
\beta_1 & \lambda\Delta. & \mathrm{nest}(\mathit{val}, \Delta\texttt{.val.\#1}) \\
 & \lambda\Delta. & \Delta\texttt{.val.\#1} \uplus \\
\gamma_1 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta\texttt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta)))))) \\
 & \lambda\Delta. & \Delta\texttt{.val.\#2} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
\gamma_2 & & \mathrm{nest}(\mathit{Kind}, \Uparrow \left(\Uparrow \left(\begin{array}{l} \Delta\texttt{.val.\#2} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \end{array} \right) \right)) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow \left(\Uparrow \left(\begin{array}{l} \Delta\texttt{.val.\#3} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \end{array} \right) \right)) \uplus \\
 & \lambda\Delta. & \Delta\texttt{.val.\#3} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
\gamma_3 & & \mathrm{nest}(\mathit{Kind}, \Uparrow \left(\Uparrow \left(\begin{array}{l} \Delta\texttt{.val.\#2} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \end{array} \right) \right)) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow \left(\Uparrow \left(\begin{array}{l} \Delta\texttt{.val.\#3} \uplus \mathrm{nest}(\mathit{Kind}, \Delta\texttt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \end{array} \right) \right)) \uplus \\
\delta_1 & \lambda\Delta. & \Delta\texttt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta\texttt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta)))))) \\
\epsilon_1 & \lambda\Delta. & \Delta\texttt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta\texttt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta)))))) \\
\epsilon_2 & \lambda\Delta. & \Delta\texttt{.val.\#2} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))) \uplus \\
 & & \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta\texttt{.val.\#2} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\Delta))))))
\end{array}
$$

Figure 6.17.. Demand annotations for the data constructors for the algebraic expression data type.

The demands for the remaining two arguments are derived similarly. Note here the interdependency between the two arguments: If the second argument is required at all, the `guard` operation in Line 9 will trigger a demand on the `Kind` component of the third argument. Dually, the `guard` operation in Line 10 will trigger a demand on the `Kind` component of the second argument if the third argument is needed. Furthermore, as the `Kind` component of the result is derived using an `any` operation on the `Kind` component of the second and third argument, the demand on the `Kind` component is passed on to those two arguments.

The demands for the arguments of the remaining data constructors result from similar derivations. I omit a detailed discussion here. Using these demands, we can now apply the demand analysis to expressions that make use of the data constructors. As a first example, I have provided a simple construction of a Boolean expression in Line 34. Lets assume, I want to know the expression kind of that expression statically. Thus, I need to propagate the demand $\Delta = \{(\mathit{Kind}, \uparrow \{\})\}$. From the demand δ_1 for the first argument of the function `EIsZero`, we can directly follow the demand on the application of `EBool` as `(`δ_1 $\{(\mathit{Kind}, \uparrow \{\})\}$`)`. Using the evaluation rules for demand expressions defined in Figure 6.16 on page 165, this expression can be reduced to the top-level normal form $\{(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\})) \uplus \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\})))))))\}$. Thus, to evaluate the `Kind` component of the application of `EIsZero`, I need to compute the `Kind` component of its first argument.

Next, to compute the demand on the argument of `EBool`, I propagate that demand into the demand expression β_1. After evaluation to top-level normal form, this yields the empty demand, as the demand on the result of `EBool` does not contain a demand on the `val` component.

Using this demand knowledge, I can now apply the rules for partial evaluation with demands given in Figure 6.8 on page 136. The function `EBool` is evaluated up to its `Kind` component, only. Thus, evaluation yields the value $\{(\mathit{Kind}, \mathit{Bool})\}$. This is then passed as argument to the function `EIsZero`. Note that, as the `Kind` component of the argument is known, the `guard` and `witness` operations contained in `EIsZero` can be evaluated and succeed. Furthermore, due to the demand on the result of `EIsZero`, the `Kind` component of its result is computed, as well. Thus, ultimately, we get $\{(\mathit{Kind}, \mathit{Int})\}$ as the overall result.

As can be seen, using the demand inference and partial evaluation, I have computed the kind of the expression by only evaluating kind related sub-expressions. In particular, the actual value of the expression has never been constructed in the process.

Of course, the sample expression I have chosen was correct and thus evaluation had to succeed. As a final example, I will show the above process for an application of data constructors that constructs a malformed expression. For this, consider the LRec expression below:

```
(ECond (ENum 1) (EBool true) (EBool false))
```

Again, I start by propagating the overall demand $\{(\mathit{Kind}, \uparrow \{\})\}$ into the application of the function `ECond`. This gives the following demands after top-level evaluation for the three arguments:

$$\left\{\left(\mathit{Kind}, \begin{array}{l} \Uparrow (\Uparrow (\Delta)) \uplus \\ \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.val.\#1} \uplus \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}))))) \end{array}\right)\right\},$$

$$\left\{\left(\mathit{Kind}, \begin{array}{l} \{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.Kind} \uplus \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\})) \uplus \\ \Uparrow \left(\Uparrow \left(\begin{array}{l} \{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.val.\#2} \uplus \\ \mathrm{nest}(\mathit{Kind}, \{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}))) \end{array} \right) \right) \uplus \\ \Uparrow \left(\Uparrow \left(\begin{array}{l} \{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.val.\#3} \uplus \\ \mathrm{nest}(\mathit{Kind}, \{(\mathit{Kind}, \uparrow \{\})\}\mathtt{.Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}))) \end{array} \right) \right) \end{array}\right)\right\}, \text{ and}$$

$$\left\{\left(\mathit{Kind}, \begin{array}{c} \{(\mathit{Kind}, \uparrow \{\})\}.\texttt{Kind} \uplus \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\})) \uplus \\ \Uparrow \left(\Uparrow \begin{pmatrix} \{(\mathit{Kind}, \uparrow \{\})\}.\texttt{val.\#2} \uplus \\ \mathrm{nest}(\mathit{Kind}, \{(\mathit{Kind}, \uparrow \{\})\}.\texttt{Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}))) \end{pmatrix} \right) \uplus \\ \Uparrow \left(\Uparrow \begin{pmatrix} \{(\mathit{Kind}, \uparrow \{\})\}.\texttt{val.\#3} \uplus \\ \mathrm{nest}(\mathit{Kind}, \{(\mathit{Kind}, \uparrow \{\})\}.\texttt{Kind}) \uplus \\ \mathrm{nest}(\mathit{Kind}, \Uparrow (\Uparrow (\{(\mathit{Kind}, \uparrow \{\})\}))) \end{pmatrix} \right) \end{array}\right)\right\}.$$

As can be seen, to evaluate the `Kind` property of the overall expression, the `Kind` property of all arguments is required. Next, these demands can be propagated through the argument expressions. As in all cases the corresponding demand expressions do only refer to the value component of the result, the demands on the arguments of the applications of `EBool` and `ENum`, respectively, are empty.

Using these demands, I can now evaluate the expression accordingly. First, the arguments to `ECond` are evaluated up to the `Kind` property. This yields arguments $\{(\mathit{Kind}, \mathit{Int})\}$, $\{(\mathit{Kind}, \mathit{Bool})\}$ and $\{(\mathit{Kind}, \mathit{Bool})\}$, respectively. Given the first argument, the `guard` expression in Line 8 will fail as the first argument is not of Boolean kind. Thus, the expression is rejected. Again, note that the actual value component or structure of the expression was computed at no time.

As these examples show, demand annotation can quickly grow. However, I use a rather naïve approach here by strictly only evaluating to top-level form. By taking further rewriting rules into account, the above demands can be further reduced. Possible candidates in these examples would be

- the distributivity of $\uparrow$ with respect to $\uplus$,
- that $\uparrow$ and $\uplus$ are idempotent, and
- that it is always safe to reduce primitive selections like $\{(\mathit{Kind}, \uparrow \{\})\}$`.Kind`.

I leave a further exploration of rewriting rules to minimise demand annotations as future research. In particular, it would be interesting to show whether a principal demand, similar to principal types in type systems [Pierce, 2002], exists.

6.4. Conclusions

As motivated in the previous chapter, computing static information from auxiliary computations by means of partial evaluation consists of two steps: A first step that infers which sub-expressions need to be evaluated and a second step that evaluates the entire expression accordingly. In the previous chapter, I have discussed the second step. The corresponding semantics for partial evaluation, however, are under constrained: They allow a sub-expression to be evaluated to different degrees. Thus, the result of partial evaluation using only those semantics is not uniquely defined. Even more, partial evaluation might get stuck if sub-expressions have been evaluated to an insufficient degree.

In this chapter I have extended the partial evaluation semantics by the concept of demands. A demand formally defines to what degree a sub-expression needs to be evaluated. Thus, the result of partial evaluation with demand annotations is uniquely defined. However, using demand annotations alone, partial evaluation can still get stuck.

To ensure progress of evaluation, I have furthermore specified analyses that, given a demand on the overall result of an expression, infer sub-demands for all sub-expressions. As I have shown, an expression that has been annotated with demands using such analyses can always be evaluated partially if full evaluation succeeds and the result of partial evaluation satisfied the given demand.

I have specified the demand analysis in two steps. The first version specified in Section 6.1 only caters for expressions without nested records. In that version of the analysis, demands are non-nested records, as well. However, to accommodate expression with nested records and demand analysis in the case of recursively constructed records, the demand language needs to be more expressive. In particular, as I have shown, such expressions require the ability to express infinite demands.

To cater for full LRec, I have therefore extended my demand language in Section 6.2 to a small applied lambda calculus. The corresponding demand analysis then supports nested record expressions and full recursion, as well. Infinite demands that arise during demand analysis are handled by using lazy evaluation of demand expressions. As I have discussed, this ensures that both, demand analysis and demand evaluation during partial evaluation, always terminate.

Lastly, I have applied the demand analysis and partial evaluation to the example of an algebraic expression data type first presented in Section 2 to show that my approach is expressible enough to model and statically check the kind of an expression.

7. Conclusion and Future Work

I have shown in this thesis that static properties of data can be modelled, checked and exploited without the need for a sophisticated type system. My approach, in contrast to type based approaches, does not a priori differentiate between dynamic and static parts of a program. Instead, it allows to model properties in a dynamic fashion using auxiliary computations and uses partial evaluation to prove some if not all of these properties statically. Static and dynamic properties thereby are not distinguished in the program text itself: The decision, which properties to compute statically, is guided by instrumenting the evaluation of the program. Thus, the users of the language can decide at any time, which properties they want to be computed and when these should be computed, without modifying the program text. In particular, the program remains executable at all stages, even if a property cannot be statically computed.

The contribution of this thesis is two-fold. Firstly, I have designed a language that allows to enrich existing programs by properties of data in a non-intrusive way. This includes the concept of auxiliary computations and a pattern matching based programming approach thereon. Properties in my approach thereby may depend on other properties and even values from the actual program itself. Whilst drastically increasing the range of properties that can be modelled, such expressiveness significantly complicates the derivation of static knowledge by partial evaluation: It is no longer clear which parts of a program need to be evaluated to compute a desired property.

This challenge leads to my second contribution. I have shown that it is always possible to statically determine which parts of a program need to be evaluated to compute a desired property. Using the notion of evaluation demands, I have devised an analysis that, given a evaluation demand on an expression, computes corresponding evaluation demands for all sub-expressions of such expression. I have proven that partial evaluation instrumented by these demands will indeed yield the desired property.

7.1. Auxiliary Computations

To be able to enrich existing programs by additional properties, whether dynamic or static, I have developed the concept of auxiliary computations and the basic principles of a corresponding programming language. Auxiliary computations allow the programmer to encode additional properties of data alongside the main computation, in a language he knows. I have presented a core language LRec that demonstrates the language constructs required to program with auxiliary computations. The language features of LRec are chosen such that auxiliary computations are non-intrusive to existing programs: A program can be enriched by auxiliary computations only partially, if desired. At the interface with code that does not use the additional properties offered by auxiliary computations, these will automatically be discarded. Likewise, the enriched and original versions of functions can co-exist. Using a pattern matching based function overloading mechanism, I ensure that always the version that makes the most use of auxiliary computations is chosen.

To express general constraints on data, I have added support for contracts to LRec. Again, I have chosen an encoding for this language feature that can be easily added to existing languages. Instead of adding contracts as a core feature, I have presented a data-flow based encoding that ensures evaluation of contracts even in the presence of compiler optimisations. Using this encoding, it suffices to extend an existing language by two special, conditional-like constructs, without the need to check existing optimisations for their contract awareness and prove their correctness in the presence of contracts.

7.2. Demands and Partial Evaluation

By nature, auxiliary computations are dynamic, runtime computations much like the actual stream of computation. As such, they are only evaluated during program execution and thus provide no additional static information. However, separating auxiliary computations syntactically from the main stream of computation allows me to evaluate some or all auxiliary computations at compile time, if desired. To this effect, I have presented a partial evaluation semantics that formalise the evaluation of only certain auxiliary computations. As I have shown, the result of such partial evaluation always yields the same result for auxiliary computations as a full evaluation would, if both yield a result.

To allow the programmer to choose which auxiliary computations to evaluate statically and which of them to delay until runtime, I have introduced the notion of demands. A demand allows the programmer to specify to what extent he wants an expression to be evaluated. Starting out with such a programmer specified demand annotation, I then use a demand inference to propagate demands to the entire program. As I have shown, the resulting annotated program can always be evaluated to satisfy the desired demand, if the program could be fully evaluated.

Using both techniques, *i.e.*, the partial evaluation semantics and demand inference, auxiliary computations can be used to statically check properties of programs. I have shown how this applies to two scenarios: In the context of numerical applications on arrays, I have shown for the example of a element-wise matrix addition how auxiliary computations can be used to statically check the compliance of argument shapes. For a setting with algebraic data types, I have shown how an evaluator for a small expression language can be extended to identify a range of invalid expressions at their construction time.

7.3. Future Work

Jointly with Bernecky et al. [2010], I have studied the use of *symbiotic expressions* to facilitate advanced symbolic optimisations in the context of the array programming language SaC [Scholz, 2003]. Like in the approach presented here, we use existing symbolic optimisations to partially evaluate auxiliary computations of the minima and maxima of iteration spaces. The gained static knowledge is then used by Bernecky to perform sophisticated symbolic optimisations to improve the runtime performance even in cases where array boundaries and the corresponding iteration spaces are not statically known.

Symbiotic expressions extend the idea of using partial evaluation to gain static knowledge into the realm of refinement types [Freeman and Pfenning, 1991; Xi and Pfenning, 1999], also referred to as predicate subtyping [Rushby et al., 1998]. Similar to refinement types, symbiotic expressions annotate predicates at expressions, *i.e.*, the minimum and maximum value of an iteration index. However, in contrast with refinement types, these predicates are annotated by the compiler.

Symbiotic expressions differ in some key points from the concept of auxiliary computations as presented in this thesis. Firstly, symbiotic expressions are not specified by the programmer but automatically inserted by the compiler at compile time. Furthermore, purely to reduce the implementation effort, symbiotic expressions do not use records to glue the main computation and the computation of the minima and maxima together but instead use dedicated data-flow hooks in the style of the contracts presented in Chapter 4. Lastly, as the symbiotic expressions are only of interest to the compiler and only useful during compilation, they are stripped out before the actual runtime code is generated and thus are never evaluated at runtime.

Despite these differences, applying the demand analysis used for auxiliary computations in the setting of symbiotic expressions seems desirable and possible, as well. So far, we use an on-line partial evaluation approach to simplify symbiotic expressions. By switching to an off-line approach using demands, I would expect major improvement in the static knowledge gathered. Such an off-line approach would require the demand analysis and partial evaluation to be lifted to symbolic evaluation, as well. In particular, the demand annotations and inference would need to cater for different levels of evaluation, *i.e.*, the evaluation of an expression to a value or merely symbolic simplification of such expression.

Apart from type safety and program optimisation, type information is increasingly used for generic programming, as well. One interesting line of research in this context would be to explore how the additional properties encoded in auxiliary computations can be exploited for generic programming. In particular, approaches like the data type generic programming known from mainstream functional languages like Haskell [Hinze, 1999, 2000, 2006; Hinze et al., 2006; Lämmel and Peyton-Jones, 2003] or Clean [Alimarine and Plasmeijer, 2002; Alimarine and Smetsers, 2004, 2005] seem to be applicable to my setting, as well. In those approaches, structural information about algebraic data types is used to express algorithms generically for arbitrary algebraic data types. All approaches have in common that, as a prerequisite, some form of abstract view on the data type is required that decomposes a specific data type into common components, usually a sums-of-products view.

With auxiliary computations in LRec, this view comes nearly for free, as the record based encoding used in LRec already makes most of the structure of data explicit. I have used this for a limited form of generic programming in the matrix addition examples presented first in Chapter 2.2. There, I provide a generic instance for arbitrary matrices alongside more specialised instances for matrices that fulfil certain structural criteria. However, the current approach does not suffice to write generic functions that transform the structure of data in some way.

As an example for such a setting, I have studied the use of enhanced structural information in the context of nested arrays [Herhut et al., in press]. Nested arrays are common in numerical applications. For instance, in SaC an array of complex numbers in fact is an array of two-element double vectors. Similarly, a colour image can be seen as a matrix of three-element vectors. Although such abstractions allow for a clean design and are thus clearly desirable, they come with their own challenges: As nested arrays are structurally different from common, flat arrays, even basic operations like element-wise addition and multiplication do no longer apply. Instead, these need to be reimplemented for every new nested structure. The added boilerplate code often outweighs the advantages of nesting in the first place.

Generic programming in this setting enables code reuse and thus reduces the amount of boilerplate code. For instance, element-wise addition is similar for both matrices of complex numbers and images: To add either, it suffices to element-wise add the array with the nesting removed.

Ignoring the nesting structure of the arguments, given a suitable encoding, is already possible in LRec as presented in this thesis. However, reinstating the nesting structure in the result is not. The root cause here is that labels and thus the structure of data, are not first-class objects in LRec and thus cannot be passed from the arguments of a generic function to the result. In the limited setting presented in [Herhut et al., in press], I use two dedicated language constructs to remove and reinstate nesting of arrays. However, this approach does not scale to the general case. Adding labels as first-class object to LRec, on the other hand, would significantly complicate the demand analysis and best match pattern matching used in LRec.

Type information is furthermore increasingly used to reason about the resource usage of applications [Hughes et al., 1996; Pareto, 1998; Portillo et al., 2003; Shkaravska et al., 2007; Tamalet et al., 2009; Vasconcelos and Hammond, 2004]. This property is particularly of interest in the setting of embedded systems. When deploying an application to a chosen hardware platform, it is essential that the application, even in the worst case, can be executed within the resource limitations of that platform.

Type system based approaches typically use a specialised form of type annotation and type inference to approximate resource usage. However, size constraints to a certain degree can be modelled in general purpose dependently-typed languages, as well [Grobauer, 2001]. In this context, it would be interesting to see how resource bounds could be expressed as properties in my approach and to what extent such an encoding could be used to statically compute resource bounds.

Apart from spreading the use of the techniques discussed in this thesis to further application areas, open questions about the approach itself remain, as well. Firstly, the partial evaluation technique itself could be further optimised. A key area that comes to mind here is reducing the degree to which a program needs to be evaluated in order to show a certain property statically. As discussed in Section 5.3, conditionals have a key impact to this regard. Like in classical type systems, undecidability is looming here: Simply evaluating both branches of a conditional to infer properties of the overall result may not terminate if the branch that would not be computed during full evaluation does not terminate. This is commonly the case for the termination conditional in recursive functions. To overcome this, in classical type systems, both branches of a condition are required to have the same type or, in the setting of subtyping, a least upper bound needs to exist [Pierce, 2002]. Thus, if a type can be derived for one branch of a conditional, that type can be used for the overall expression regardless of whether the type inference would terminate for the other branch. In contrast, I evaluate the predicate expression of a conditional to decide during partial evaluation which branch to evaluate. This approach provides the additional freedom that the properties of both branches of a conditional need not to be related. However, this freedom comes at a price: Evaluating the predicate expression of a conditional commonly leads to large parts of a program being evaluated. These two design choices are a classical example for a trade-off between flexibility and efficiency.

A third approach would be to allow the programmer to decide which approach to use on a per-property basis. One could encode this by means of *monomorphic labels*, *i.e.*, labels for which both branches of a conditional are guaranteed to yield the same result. In such a setting, to compute a partial result containing only monomorphic labels, it would suffice to compute the partial result for one branch only. Thus, if both branches are evaluated concurrently, the evaluation can be stopped as soon as one result has been found. Ultimately, this would ensure that evaluation still terminates whilst reducing the degree to which a program needs to be evaluated.

Lastly, I would like to further investigate the boundaries between my approach and general type systems. I, for example, have shown that auxiliary computations can be used to encode static properties similar to generalised algebraic data types in HASKELL [Peyton-Jones et al., 2006]. However, such a comparison may seem unfair. Whereas my approach as presented in this thesis is limited to first order languages, HASKELL supports higher order functions, which significantly complicate the type theory involved. Adding higher order functions to my approach, on the other hand, would complicate the demand inference. Naïvely, to support higher order functions, a higher order demand language would be required. Whereas at first glance an extension of the demand language and corresponding inference in that direction seems possible, a formalisation of such an approach remains future work.

Apart from extending my approach, I am interested in the interplay between subtyping based type systems and my demand analysis. The record encoding used in LREC lends itself nicely to a subtyping based type theory. Records, quite naturally, form a subtype relationship. It would be interesting to see whether my approach can be formulated as a type system, as well.

In the context of the SAC programming language with its subtyping based array types, I am currently looking into reformulating the type system by means of auxiliary computations. However, this research is at an early stage.

7.4. Closing Remark

This dissertation has illuminated types and their inference from a new angle. I have put forward the idea to treat types and values uniformly, both in syntax and semantics. With auxiliary computations, demands and demand inference, I have presented the techniques that are required to develop languages that follow my idea. I have shown that it is feasible to use partial evaluation to compute static type information.

The foundation has been laid. Now, it is up to you to build on my work and put these ideas into action.

8. Bibliography

The citations in this bibliography are sorted alphabetically and thus do not appear in the order they are cited in the main body of this thesis. However, at the end of each bibliography entry, I have annotated the page numbers of all pages it appears on.

Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran-95 Handbook — Complete ANSI/ISO Reference.* MIT Press, Cambridge, Massachusetts, USA, 1997. 3

Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *IFL '02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 168–185, London, UK, 2002. Springer-Verlag, Berlin, Heidelberg, New York. ISBN 3-540-43537-9. 179

Artem Alimarine and Sjaak Smetsers. Efficient generic functional programming. Technical Report NIII-R0425, Nijmegen Institute for Computing and Information Sciences, June 2004. 179

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel Hermenegildo and Daniel Cabeza, editors, *Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages*, number 3350 in Lecture Notes in Computer Science, pages 203 – 218. Long Beach, CA, USA, Springer-Verlag, Berlin, Heidelberg, New York, January 2005. 179

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. 154

Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-024-4. 5

Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathmatics.* North-Holland, 1981. 55

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon

Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. Available online at `http://agilemanifesto.org/` as of March 2009, 2001. 2

Robert Bernecky. Reducing computational complexity with array predicates. In *APL '98: Proceedings of the APL98 Conference on Array Processing Language*, pages 39–43, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-181-X. 4, 63

Robert Bernecky. Shape cliques. *ACM SIGAPL Quote Quad*, 35(3):7–17, September 2007. 66

Robert Bernecky and Paul Berry. *SHARP APL Reference Manual.* Iverson Software Inc., 2nd edition, 1993. 10

Robert Bernecky, Stephan Herhut, and Sven-Bodo Scholz. Symbiotic expressions. In *Implementation and Application of Functional Languages, 21st International Symposium (IFL'09), South Orange, New Jersey, USA, Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 2010. 179

L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002. ISSN 0098-3500. 17

Guy E. Blelloch. Nesl: A nested data-parallel language. Technical report, Carnegie Mellon University, March 1994. 11

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994. ISSN 0743-7315. 11, 12

Guy E. BlellochBlelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8: 119–134, 1990. 12

F. Warren Burton. Nondeterminism with referential transparency in functional programming languages. *Computer Journal*, 31(3):243–247, 1988. ISSN 0010-4620. 55

David Cann, John Feo, Wim Bohm, and Rodney Oldehoeft. The SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991. 12

David C. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989. 12

Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, volume 39, pages 278–292, New York, NY, USA, 1991. ACM. 3

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003. 32

Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. ISSN 00029947. 55

William Clinger. Nondeterministic call by need is neither lazy nor by name. In *LFP '82: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 226–234, New York, NY, USA, 1982. ACM. ISBN 0-89791-082-6. 55

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. An efficient method of computing static single assignment form. Technical report, Brown University, Providence, RI, USA, 1988. 67

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988. ISSN 0098-3500. 17

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Ian S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16 (1):1–17, 1990. ISSN 0098-3500. 17

Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. 4

Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. 179

Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages, and Programming*, pages 257–281. Edinburgh University Press, Edinburgh, Scotland, 1976. 154

Gerhard Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39:176–210, 1934. 51

James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* Java series. Prentice Hall PTR, third edition, 2005. 1

Bernd Grobauer. Cost recurrences for DML programs. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 253–264, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. 180

Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, pages 95–103, New York, NY, USA, 1976. ACM. 154

Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. From contracts towards dependent types: Proofs by partial evaluation. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 254–273, Berlin, Heidelberg, 2008. Springer-Verlag, Berlin, Heidelberg, New York. ISBN 978-3-540-85372-5. 63

Stephan Herhut, Sven-Bodo Scholz, and Clemens Grelck. Generic programming on the nesting structure of arrays. In *Proceedings of the 2007 APL conference*, New York, NY, USA, in press. ACM Press. 180

High Performance Fortran language specification V1.1. High Performance Fortran Forum, 1994. 3

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986. 55

Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, number UU-CS-1999-28 in Technical report of Utrecht University, 1999. 179

Ralf Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-125-9. 179

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, 2006. ISSN 0956-7968. 179

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap your boilerplate" reloaded. In *Proceedings of the Eighth International Symposium on Functional and Logic Programming, FLOPS 2006*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 2006. 179

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. 180

Roger K.W. Hui and Ken E. Iverson. *J Introduction and Dictionary.* Jsoftware Inc., 2004. 10

International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993. 4, 10

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5. 79

Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective java library to support design by contract. In *Proceedings Reflection'99, The Second International Conference on Meta-Level Architectures and Reflection*, pages 19–21, Santa Barbara, CA, USA, 1999. University of California at Santa Barbara. 24

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709. 1

R. Kramer. iContract - The Java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8482-8. 24

Ralf Lämmel and Simon Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. 179

Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003. ISSN 0018-9162. 2

Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979. ISSN 0098-3500. 17

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998. ISSN 0956-7968. 154

Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445. 2

MathWorks. *Matlab User's Manual.* Natick, MA, 2009. URL `http://www.mathworks.com`. 2

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2004. 5

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. ISSN 0956-7968. 5

John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *IRE-AIEE-ACM '61 (Western): Papers presented at the May 9-11, 1961, western* *joint IRE-AIEE-ACM computer conference*, pages 225–238, New York, NY, USA, 1961. ACM. 13, 54

John McCarthy. A basis for a mathematical theory of computation. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1962. 13, 54

Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. 70

Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1990. ISBN 0-13-247925-7. 24

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. 24

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 1

Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised.* The MIT Press, May 1997. ISBN 0262631814. 2, 15

Lars Pareto. *Sized Types.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1998. 180

Simon Peyton-Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003. 1, 26

Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, September 2006. ACM SIGPLAN. 32, 181

Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. 51, 72, 87, 155, 164, 174, 181

Reinhold Ploesch. Design by contract for Python. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 213, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8271-X. 24

G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975. ISSN 03043975. 154

Alvaro J. Rebon Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Cost analysis using automatic size and time inference. In *Implementation of Functional Languages, 14th International Workshop, IFL 2002*, volume 2670 of *Lecture Notes* *in Computer Science*, pages 232–248. Springer-Verlag, Berlin, Heidelberg, New York, 2003. 180

John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, sep 1998. 179

Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, Cambridge, MA, USA, 1989. ISBN 0262192772. 12

S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003. 10, 63, 179

Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. 1

Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-071-X. 1

Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial size analysis of first-order functions. In *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, Berlin, Heidelberg, New York, 2007. 180

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006. 3

Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. Technical report, Carnegie Mellon University, March 1991. 11

Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen. Size analysis of algebraic data types. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Trends in Functional Programming*, volume 9 of *Trends in Functional Programming*, pages 33–48. Intellect, 2009. ISBN 978-1-84150-277-9. 180

Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *Journal of Logic and Algebraic Programming*, 78(7):643 – 664, 2009. ISSN 1567-8326. The 19th Nordic Workshop on Programming Theory (NWPT 2007). 4

Kai Trojahner, Clemens Grelck, and Sven-Bodo Scholz. On optimising shape-generic array programs using symbolic structural information. In Zoltan Horváth and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 18th International Symposium (IFL'06), Budapest, Hungary, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, Heidelberg, New York, 2007. 66

Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Implementation and Application of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Berlin, Heidelberg, New York, 2004. 180

Christopher P. Wadsworth. *Semantics and Pragmatics of the lambda-calculus.* PhD thesis, Programming Research Group, Oxford University, 1971. 154

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, New York, NY, USA, 1999. ACM. 179

Dana N. Xu. Extended static checking for Haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM. ISBN 1-59593-489-8. 24

Dana N. Xu, Simon Peyton-Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 41–52, New York, NY, USA, 2009. ACM. 24

Na Xu. *Static Contract Checking for Haskell.* PhD thesis, University of Cambridge, 2008. 24

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997. ISSN 0304-3975. 4

Christoph Zenger. *Indizierte Typen.* PhD thesis, Universität Karlsruhe, 1998. 4

Appendix A.

Source Code for Examples

A.1. Amended Evaluator for Haskell

A.1.1. Using an Additional Tag

```
data Kind = KInt | KBool
data Expr = ENum Kind Int
          | EBool Kind Boolean
          | ECond Kind Expr Expr Expr
          | EIsZero Kind Expr
          | EDiv Kind Expr Expr

eval (ENum _ v)       = ENum v
eval (EBool _ v)      = EBool v
eval (ECond _ p t e)  = case (eval p) of
                        | EBool v  -> if v (eval t) (eval e)
eval (EIsZero _ e)    = case (eval e) of
                        | (ENum v) -> EBool (v = 0)
eval (EDiv _ a b)     = case ( (eval a), (eval b)) of
                        | ( (ENum va), (ENum vb)) -> ENum (va/vb)

eval (EIsZero KBool (EDiv KInt (ENum KInt 5) (ENum KInt 2)))
```

A.1.2. Using a Stratified Approach

```
data IKind = ENum Int
           | EDiv IKind IKind
           | EICond BKind IKind IKind
data BKind = EBool Boolean
           | EBCond BKind BKind BKind
           | EIsZero IKind
data Expr = IExpr BKind
          | BExpr BKind

eval (IExpr e) = IExpr (evalI e)
eval (BExpr e) = BExpr (evalB e)
```

```
evalI (ENum v)        = ENum v
evalI (EDiv a b)      = case ( (evalI a), (evalI b)) of
                        | ( (ENum va), (ENum vb)) -> ENum (va/vb)
evalI (EICond p t e) = case (evalB p) of
                        | EBool v  -> if v (evalI t) (evalI e)

evalB (EBool v)       = EBool v
evalB (EIsZero e)     = case (evalI e) of
                        | ENum v -> EBool (ve = 0)
evalB (EBCond p t e) = case (evalB p) of
                        | EBool v  -> if v (evalB t) (evalB e)
```

Appendix B.

Language Extensions

In this appendix, I provide for reference the extensions to LRec$_\text{C}$ that are required for the examples given throughout this thesis. The next section describes the extensions needed for the examples that use integers and vectors as data. For those examples that use tuples to model algebraic data-types, the corresponding extensions are given in Section B.2.

B.1. Matrix Examples

First, I in Figure B.1 describe the syntax extensions to LRec$_\text{C}$. As can be seen, three additional token have been added to the production rule *expression*. The first, the production rule *integer* defines the syntax for integer values in the usual way. Using these integer values, I furthermore allow the construction of vectors as described in the production rule *vector*. A vector consists of an opening bracket `[`, a potentially empty list of integers and a closing bracket `]`.

Lastly, I have added three built-in functions as defined by the production rule *builtin*. In general, the application of a built-in function syntactically consists of the function name and a sequence of argument expressions enclosed in parentheses. For the `vect_add` function, this sequence needs to contain the two vectors to add. The two functions `diag_add` and `ldiag_add` additionally require a third argument.

Before I define the semantics of these extensions, I first extend the set of legal values as used as range of the evaluation relations $\Downarrow$ and $\downarrow$. As non-record values are always fully evaluated, these extensions apply to both the set of full values and the set of partial values. Figure B.2 shows the production rules for these additional values.

As can be seen, I have added two more kinds of values: integer values and vectors of integers. Their syntax is identical to the syntax of the corresponding expressions in the extend LRec$_\text{C}$. I will in the following use $\mathbb{I}$ and $\mathbb{I}^n$ to refer to the set of values that can be produced using only the rule *integer* and *vector*, respectively.

To allow for comparing integer values and vector values using the equality operation `=`, I furthermore extend the equality relation $\overset{v}{=}$ on non-record values as follows:

Definition B.1.1 (Extended Equivalence of Values)**.** *The equivalence relation* $\overset{v}{=} \subset \mathscr{V} \times \mathscr{V}$ *on values is defined as*

$$\overset{v}{=} \;:=\; \{(\mathit{true}, \mathit{true}), (\mathit{false}, \mathit{false})\} \cup \{(i, i) \mid i \in \mathbb{I}\} \cup \{(v, v) \mid v \in \mathbb{I}^n\}$$

expression $\Rightarrow$ $\cdots$
 | **integer**
 | **vector**
 | **builtins**

integer $\Rightarrow$ *[* - *][* **numeral** *]*$^+$

numeral $\Rightarrow$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

vector $\Rightarrow$ `[` *[* **integer** *[* `,` **integer** *]** *]* `]`

builtins $\Rightarrow$ `(` `vect_add` **expression** **expression** `)`
 | `(` `diag_add` **expression** **expression** **expression** `)`
 | `(` `ldiag_add` **expression** **expression** **expression** `)`

Figure B.1.. Syntax extensions for matrix examples in extended Backus Nauer form.

In the above definition, apart from the equality on Boolean values, I now define that each integer and vector equates to itself. The above extension suffices to extend the equality operation to integer and vector values, as well. As can be seen in rules EQUAL-TRUE and EQUALFALSE in Figures 3.8, 5.2 and 6.8, the existing rule only requires the arguments to evaluate to values that are not in $\mathscr{R}$. This, by definition of the set $\mathscr{R}$, is true for integer and vector values.

The semantics for integer and vector expressions are straight forward. Both reduce to themselves, as expressions and the corresponding values have the same form. For the built-in functions, I provide corresponding semantic rules in Figure B.3. However, I only provide rules for full evaluation. The corresponding rules for partial evaluation can easily be derived by adding a further condition that the annotated demand is not empty in the current context and by using the partial evaluation relation $\downarrow$ with the corresponding

value $\Rightarrow$ $\cdots$ | **integer** | **vector**

integer $\Rightarrow$ *[* - *][* **numeral** *]*$^+$

numeral $\Rightarrow$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

vector $\Rightarrow$ `[` *[* **integer** *[* `,` **integer** *]** *]* `]`

Figure B.2.. Extension of the set of values and partial values for matrix examples in extended Backus Nauer form.

$$\textsc{VectAdd} \quad : \quad \frac{(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow [v_1^1, \ldots, v_n^1] \quad (\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow [v_1^2, \ldots, v_n^2]}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{vect\_add(}e_1 \ e_2\texttt{)} \Downarrow [v_1^1 + v_1^2, \ldots, v_n^1 + v_n^2]}$$

$$\textsc{DiagAdd} \quad : \quad \frac{\begin{array}{c}(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow [s_1, s_2] \\ (\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow [v_1^1, \ldots, v_{s_1 s_2}^1] \quad (\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow [v_1^2, \ldots, v_{s_1 s_2}^2]\end{array}}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{diag\_add(}e_1 \ e_2 \ e_3\texttt{)} \Downarrow [v_1, \ldots, v_{s_1 s_2}]}$$

$$\text{where } v_i := \begin{cases} v_i^1 + v_i^2 & \text{if } i \div s_2 + 1 = i \bmod s_2, \\ v_i^1 & \text{otherwise.} \end{cases}$$

$$\textsc{LDiagAdd} \quad : \quad \frac{\begin{array}{c}(\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow [s_1, s_2] \\ (\mathscr{F}, \prec, \mathscr{E}) : e_1 \Downarrow [v_1^1, \ldots, v_{s_1 s_2}^1] \quad (\mathscr{F}, \prec, \mathscr{E}) : e_2 \Downarrow [v_1^2, \ldots, v_{s_1 s_2}^2]\end{array}}{(\mathscr{F}, \prec, \mathscr{E}) : \texttt{ldiag\_add(}e_1 \ e_2 \ e_3\texttt{)} \Downarrow [v_1, \ldots, v_{s_1 s_2}]}$$

$$\text{where } v_i := \begin{cases} v_i^1 + v_i^2 & \text{if } i \div s_2 + 1 \leq i \bmod s_2, \\ v_i^1 & \text{otherwise.} \end{cases}$$

Figure B.3.. Semantics extensions for matrix examples.

environments.

The first rule in Figure B.3, rule VectAdd, gives the semantics for the special `vect_add` operation. As can be seen, as a prerequisite I require both arguments of the `vect_add` operation to evaluate to integer vectors of equal lenght n. The `vect_add` operation itself is then evaluated to the n-element vector where each element equates to the sum of the corresponding two elements of the argument vectors.

Similarly, rule DiagAdd gives the semantics for the `diag_add` operation. As the `diag_add` operation only adds the elements along the diagonal of the second matrix argument, more structural information is required in the form of an additional shape argument. In the semantics this is reflected by the additional precondition that the shape-vector argument evaluates to a two-element integer-vector. Furthermore, both matrix arguments need to evaluate to integer vectors with a valid lenght for the given shape vector. In rule DiagAdd this is expressed by requiring that both vectors have $s_1 s_2$ many elements, where s_1 and s_2 are the number of rows and columns as specified by the shape argument. The result is then defined for each index depending on whether it lies on the diagonal of the matrix. If so, the result at that index is computed as the sum of the two corresponding elements of the matrix arguments. Otherwise, the result at that index equates to the corresponding value of the first matrix argument.

Lastly, the semantics for the `ldiag_add` operation are defined by rule LDiagAdd. It corresponds largely to the rule for the `diag_add` operation. The only difference is that now for all indices below the diagonal an actual sum is computed.

The demand inference for the additional non-record values and operations thereon is identical to that for the existing non-record values and corresponding operations. For integer values, the demand is simply annotated analogously to the rules True and False

(Integer) $\mathscr{A}_{\mathscr{F}}\,[\![\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ i\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}_A, \mathscr{E}_A,\ i\ \lambda\Delta.\delta\ )$

(Vector) $\mathscr{A}_{\mathscr{F}}\,[\![\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{]}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}_{An}, \mathscr{E}_{An},\ \texttt{[}e'_1\texttt{,}\ \ldots\texttt{,}\ e'_n\texttt{]}\ \lambda\Delta.\delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}_{A0} & := & \mathscr{F}_A \\ \mathscr{E}_{A0} & := & \mathscr{E}_A \\ \mathscr{F}_{Ai+1}, \mathscr{E}_{Ai+1}, e'_{i+1} & := & \mathscr{A}_{\mathscr{F}}\,[\![\mathscr{F}_{Ai}, \mathscr{E}_{Ai}, \mathscr{B}_A, \delta,\ e_{i+1}\ ]\!] \end{array}$$

(BuiltIn) $\mathscr{A}_{\mathscr{F}}\,[\![\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta,\ \texttt{op(}\ e_1\ \cdots\ e_n\texttt{)}\ ]\!]$

$\rightsquigarrow\ (\mathscr{F}_{An}, \mathscr{E}_{An},\ \texttt{op(}\ e'_1\ \cdots\ e'_n\texttt{)}\ \lambda\Delta.\delta\ )$

where

$$\begin{array}{lcl} \mathscr{F}_{A0} & := & \mathscr{F}_A \\ \mathscr{E}_{A0} & := & \mathscr{E}_A \\ \mathscr{F}_{Ai+1}, \mathscr{E}_{Ai+1}, e'_{i+1} & := & \mathscr{A}_{\mathscr{F}}\,[\![\mathscr{F}_{Ai}, \mathscr{E}_{Ai}, \mathscr{B}_A, \delta,\ e_{i+1}\ ]\!] \end{array}$$

Figure B.4.. Extensions to the demand analysis scheme $\mathscr{A}_{\mathscr{F}}$ as required for the matrix examples.

in Figures 6.3 and 6.12. I provide the corresponding rule Integer for the generalised demand analysis in Figure B.4. In case of integer vectors, additionally to annotating the demand, the demand needs to be propagated to the components of the vector. Note that other than for record expressions, no sub-demand is selected. This follows the notion that vectors as non-record values are either fully evaluated or not at all. Thus, if the initial demand is the empty demand, so is the demand on the components of the integer vector. In all other cases, *i.e.*, if the demand is non-empty, the full vector needs to be evaluated.

For the three operations on vectors, *i.e.*, `vect_add`, `diag_add` and `ldiag_add`, I have only specified one single rule BuiltIn. In all three cases, the demand is annotated at the overall expression and propagated to each argument. Analogously to the rule for integer vectors, I directly propagate the overall demand to each argument position. Again, this is motivated by the fact that integer vectors are either fully evaluated or not at all.

expression	⇒	···
	|	**tuple**
	|	**tupleselection**
	|	**integer**
	|	**builtins**
tuple	⇒	`(` [**expression** [`,` **expression**]*] `)`
tupleselection	⇒	**expression** `#` [**numeral**]+
integer	⇒	[`-`][**numeral**]+
numeral	⇒	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
builtins	⇒	`(` **expression** `!=` **expression** `)`
	|	`(` **expression** `/` **expression** `)`
	|	`(` **expression** `||` **expression** `)`

Figure B.5.. Syntax extensions for algebraic data-type examples in extended Backus Nauer form.

B.2. Algebraic Data-Type Examples

The extensions to the syntax of LRec required for the examples on algebraic data-types are given in Figure B.5. Like the extended syntax for vectors presented in the previous section, the syntax defined in Figure B.5 extends LRec by integer numbers. However, instead of vectors, the syntax of LRec now is extended by tuples. A tuple is represented by a, potentially empty, comma-separated list of expressions surrounded by parentheses. It is worth noting here that, other than vectors, tuples may contain arbitrary expressions. Thus, in particular, tuples may be nested.

A further addition to the syntax of LRec is the selection on tuples. To differentiate it from the existing record selection, I use the symbol `#` instead of the `.`-symbol. Like the record selection, the selection on tuples is an infix operation that expects the argument to select from on the left-hand and the index to select with on the right-hand. However, other than the record selection, the tuple selection uses integers as indices: Tuple elements are indexed from left to right, starting with the index 0 for the left-most element.

Lastly, I have extended the syntax of LRec by some additional infix operations: An inequality operation `!=` on values, a division operation `/` on integer values and the Boolean *or* operation `||`.

Having the extended syntax in place, I next define the semantics of the additions presented above. For tuples and the selection thereon, I do so by providing a lowering step from LRec with tuples to LRec_C. The idea is to represent tuples as records with

$$(\textsc{Tuple})\quad \mathscr{L}_{\mathrm{i}}[\![\ \texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)}\ ]\!]$$

$$\rightsquigarrow\quad \texttt{\{\#0=}e_1\texttt{,}\ \ldots\texttt{,}\ \texttt{\#}n-1\texttt{=}e_n\texttt{\}}$$

$$(\textsc{TupleSel})\quad \mathscr{L}_{\mathrm{i}}[\![\ \textit{expression}\texttt{\#}i\ ]\!]$$

$$\rightsquigarrow\quad \mathscr{L}_{\mathrm{i}}[\![\ \textit{expression}\,\texttt{.\#}i\ ]\!]$$

Figure B.6.. Transformation scheme $\mathscr{L}_{\mathrm{x}}$ to resolve tuples.

special labels. As the tuple selection does not allow for expressions as indices, *i.e.*, as the selection index is not a first class value, such a mapping is relatively easy to achieve. It suffices to rewrite all tuple creations into record creations and tuple selections into record selections. The rewriting rules for the corresponding lowering scheme $\mathscr{L}_{\mathrm{x}}$ are given in Figure B.6.

The first rule, rule Tuple, rewrites a tuple expression into a record expression. For each subexpression of the tuple expression, the newly created record expression contains a corresponding label-expression pair. The label thereby is derived from the index positon of the subexpression within the tuple expression by prepending the index with the #-symbol. I use a special symbol for readability reasons only. Instead, any other injective mapping from indices to labels could be used.

The second rule in Figure B.6 specifies the rewriting of tuple-selection expressions. As tuple expressions are rewritten into record expressions, tuple-selection expressions are rewritten into record-selection expressions. I use the same translation from indices to labels.

The further rules for the rewriting only drive the translation into all subexpressions of an expression. I omit those here.

As tuple expressions are rewritten to record expressions, the extension of LRec presented in this section does not require tuples on the value level. Thus, the extended set of values as given in Figure B.7 does not contain any production rules for tuple values. Instead, only the extensions integer values are given. The corresponding production rule *integer* is identical to the one presented in the previous section.

Using this extended set of values, I next define the semantics of the extensions to LRec$_{\mathrm{C}}$. As before, the semantics for rewritten expressions, *i.e.*, tuple and tuple-selection expressions in this case, is given by the semantics of the corresponding rewritten expression. For all other extensions, the corresponding rules are given in Figure B.8. As for the extensions to LRec to support the matrix examples, I only provide rules for full evaluation. The corresponding rules for partial evaluation can easily be derived by adding a further condition that the annotated demand is not empty in the current context and by using the partial evaluation relation $\downarrow$ with the corresponding environments.

The first two rules, *i.e.*, the rules NEqualTrue and NEqualFalse, define the semantics for the inequality operation `!=`. They are the inverse of the corresponding rules for the equality operation presented in Figure 3.8. For an application of the inequality operation to be evaluated, both operands must evaluate to non-record values. Note that $\overset{v}{=}$ referts to the extended version given in Definition B.1.1. Thus, integer values are comparable. However, tuples are not comparable, as they are represented by records in LRec_C.

Next, rule Or defines the semantics of the Boolean or operation. For such an operation to be evaluated, both arguments need to evaluate to Boolean values. The overall expression then evaluates to true if either value is true.

Lastly, the semantics for the division operation are given by rule Div. Here, both operand expression need to evaluate to integer values. In this case, the result is the integer division of the left-hand and right-hand operand. I use the ÷-symbol to denote integer division.

Finally, it remains to extend demand derivation by the extensions presented in this section. Again, as tuples are mapped to records, the only extensions that need to be supported by demand inference are integer values and the three additional infix operations. The corresponding rules are given in Figure B.9. The first rule, *i.e.*, rule Integer, described demand derivation for integer expressions. The rule is identical to the rule presented in the previous section; it annotates the current demand at the integer expression.

Lastly, the rule Infix defines demand derivation for infix operations. As all three infix operations operate on non-record values only, any non-empty demand encodes that they need to be fully evaluated. Thus, the operands need to be fully evaluated for all non-empty demands on the overall expression, as well. However, if the demand on the overall infix operation is empty, such is the demand on its operands. Consequently, it suffices to simply propagate the demand for the infix operation to both operands.

value	⇒	···
	|	**integer**
integer	⇒	*[* `-` *][* **numeral** *]*$^+$
numeral	⇒	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Figure B.7.. Extension of the set of values and partial values for algebraic data-type examples in extended Backus Nauer form.

$$\textsc{NEqualTrue} \quad : \quad \frac{\begin{array}{c}(\mathscr{F},\prec,\mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \Downarrow v_2 \\ v_1, v_2 \notin \mathscr{R} \quad (v_1, v_2) \notin \overset{v}{=}\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1 \ \texttt{!=} \ e_2\texttt{)} \Downarrow \mathit{true}}$$

$$\textsc{NEqualFalse} \quad : \quad \frac{\begin{array}{c}(\mathscr{F},\prec,\mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \Downarrow v_2 \\ v_1, v_2 \in \mathscr{R} \quad (v_1, v_2) \notin \overset{v}{=}\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1 \ \texttt{!=} \ e_2\texttt{)} \Downarrow \mathit{false}}$$

$$\textsc{Or} \quad : \quad \frac{\begin{array}{c}(\mathscr{F},\prec,\mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \Downarrow v_2 \\ v_1, v_2 \in \{\mathit{true}, \mathit{false}\}\end{array}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1 \ \texttt{||} \ e_2\texttt{)} \Downarrow v_1 \vee v_2}$$

$$\textsc{Div} \quad : \quad \frac{(\mathscr{F},\prec,\mathscr{E}) : e_1 \Downarrow v_1 \quad (\mathscr{F},\prec,\mathscr{E}) : e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z}}{(\mathscr{F},\prec,\mathscr{E}) : \texttt{(}e_1 \ \texttt{/} \ e_2\texttt{)} \Downarrow v_1 \div v_2}$$

Figure B.8.. Semantics extensions for algebraic data-type examples.

$$(\textsc{Integer}) \quad \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta, \ i \ \right]\!\right]$$

$$\rightsquigarrow \ (\mathscr{F}_A, \mathscr{E}_A, \ i \ \lambda\Delta\,.\,\delta \ )$$

$$(\textsc{Infix}) \quad \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta, \ \texttt{(}e_1 \ \mathtt{op} \ e_2\texttt{)} \ \right]\!\right]$$

$$\rightsquigarrow \ (\mathscr{F}_A{}'', \mathscr{E}_A{}'', \ \texttt{(}e_1' \ \mathtt{op} \ e_2'\texttt{)} \ \lambda\Delta\,.\,\delta \ )$$

where

$$\begin{array}{lcl} \mathscr{F}_A{}', \mathscr{E}_A{}', e_1' & := & \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A, \mathscr{E}_A, \mathscr{B}_A, \delta, \ e_1 \ \right]\!\right] \\ \mathscr{F}_A{}'', \mathscr{E}_A{}'', e_2' & := & \mathscr{A}_{\mathscr{F}} \left[\!\left[\mathscr{F}_A{}', \mathscr{E}_A{}', \mathscr{B}_A, \delta, \ e_2 \ \right]\!\right] \end{array}$$

Figure B.9.. Extensions to the demand analysis scheme $\mathscr{A}_{\mathscr{F}}$ as required for the matrix examples.

Appendix C. Transformation Schemes

C.1. Remaining Rules for $\mathscr{L}_t$

(Let) $\mathscr{L}_{\mathtt{t}}[\![\ \mathtt{let}\ d_1\ \cdots\ d_n\ \mathtt{in}\ \mathit{expression}\ \mathtt{end}\ ]\!]$

$\rightsquigarrow\ \mathtt{let}\ \mathscr{L}_{\mathtt{t}}[\![\ d_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathtt{t}}[\![\ d_n\ ]\!]\ \mathtt{in}\ \mathscr{L}_{\mathtt{t}}[\![\ \mathit{expression}\ ]\!]\ \mathtt{end}$

(Relation) $\mathscr{L}_{\mathtt{t}}[\![\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}\ ]\!]$

$\rightsquigarrow\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}$

(Function) $\mathscr{L}_{\mathtt{t}}[\![\ \mathtt{fun}\ \mathit{instance}_1\ \cdots\ \mathit{instance}_n\ ]\!]$

$\rightsquigarrow\ \mathtt{fun}\ \mathscr{L}_{\mathtt{t}}[\![\ \mathit{instance}_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathtt{t}}[\![\ \mathit{instance}_n\ ]\!]$

(Instance) $\mathscr{L}_{\mathtt{t}}[\![\ \mathit{quantifiers}\ \mathit{guards}\ \mathtt{=}\ \mathit{expression}\ ]\!]$

$\rightsquigarrow\ \mathit{quantifiers}\ \mathscr{L}_{\mathtt{t}}[\![\ \mathit{guards}\ ]\!]\ \mathtt{=}\ \mathscr{L}_{\mathtt{t}}[\![\ \mathit{expression}\ ]\!]$

Figure C.1.. Remaining rules for the transformation scheme $\mathscr{L}_t$ in Section 3.2.1.

(Value) $\mathscr{L}_{\texttt{t}}[\![$ `val` *identifier expression* $]\!]$

$\rightsquigarrow$ `val` *identifier* `=` $\mathscr{L}_{\texttt{t}}[\![$ *expression* $]\!]$

(Guards) $\mathscr{L}_{\texttt{t}}[\![$ `|` $g_1 \cdots g_n$ $]\!]$

$\rightsquigarrow$ `|` $\mathscr{L}_{\texttt{t}}[\![\ g_1\ ]\!] \cdots \mathscr{L}_{\texttt{t}}[\![\ g_n\ ]\!]$

(Selection) $\mathscr{L}_{\texttt{t}}[\![$ *expression*`.`*label* $]\!]$

$\rightsquigarrow$ $\mathscr{L}_{\texttt{t}}[\![$ *expression* $]\!]$`.`*label*

(Blink) $\mathscr{L}_{\texttt{t}}[\![$ `!`*expression* $]\!]$

$\rightsquigarrow$ `!`$\mathscr{L}_{\texttt{t}}[\![$ *expression* $]\!]$

(Any) $\mathscr{L}_{\texttt{t}}[\![$ `any(` $e_1 \cdots e_n$`)` $]\!]$

$\rightsquigarrow$ `any(` $\mathscr{L}_{\texttt{t}}[\![\ e_1\ ]\!] \cdots \mathscr{L}_{\texttt{t}}[\![\ e_n\ ]\!]$`)`

(Conditional) $\mathscr{L}_{\texttt{t}}[\![$ `if` *predicate then else* $]\!]$

$\rightsquigarrow$ `if` $\mathscr{L}_{\texttt{t}}[\![$ *predicate* $]\!]$ $\mathscr{L}_{\texttt{t}}[\![$ *then* $]\!]$ $\mathscr{L}_{\texttt{t}}[\![$ *else* $]\!]$

(Application) $\mathscr{L}_{\texttt{t}}[\![$ `(` *identifier* $e_1 \cdots e_n$`)` $]\!]$

$\rightsquigarrow$ `(` *identifier* $\mathscr{L}_{\texttt{t}}[\![\ e_1\ ]\!] \cdots \mathscr{L}_{\texttt{t}}[\![\ e_n\ ]\!]$`)`

Figure C.1.. Remaining rules for the transformation scheme $\mathscr{L}_t$ in Section 3.2.1 (contd.).

C.2. Remaining Rules for $\mathscr{L}_e$

(Let) $\mathscr{L}_\mathsf{e}[\![\ \texttt{let}\ d_1 \cdots d_n\ \texttt{in}\ \mathit{expression}\ \texttt{end}\ ]\!]$

$\rightsquigarrow\ \texttt{let}\ \mathscr{L}_\mathsf{e}[\![\ d_1\ ]\!] \cdots \mathscr{L}_\mathsf{e}[\![\ d_n\ ]\!]\ \texttt{in}\ \mathscr{L}_\mathsf{e}[\![\ \mathit{expression}\ ]\!]\ \texttt{end}$

(Relation) $\mathscr{L}_\mathsf{e}[\![\ \mathit{identifier}\ \texttt{<:}\ \mathit{identifier}\ ]\!]$

$\rightsquigarrow\ \mathit{identifier}\ \texttt{<:}\ \mathit{identifier}$

(Function) $\mathscr{L}_\mathsf{e}[\![\ \texttt{fun}\ \mathit{instance}_1 \cdots \mathit{instance}_n\ ]\!]$

$\rightsquigarrow\ \texttt{fun}\ \mathscr{L}_\mathsf{e}[\![\ \mathit{instance}_1\ ]\!] \cdots \mathscr{L}_\mathsf{e}[\![\ \mathit{instance}_n\ ]\!]$

(Value) $\mathscr{L}_\mathsf{e}[\![\ \texttt{val}\ \mathit{identifier}\ \mathit{expression}\ ]\!]$

$\rightsquigarrow\ \texttt{val}\ \mathit{identifier}\ \texttt{=}\ \mathscr{L}_\mathsf{e}[\![\ \mathit{expression}\ ]\!]$

(Guards) $\mathscr{L}_\mathsf{e}[\![\ \texttt{|}\ g_1 \cdots g_n\ ]\!]$

$\rightsquigarrow\ \texttt{|}\ \mathscr{L}_\mathsf{e}[\![\ g_1\ ]\!] \cdots \mathscr{L}_\mathsf{e}[\![\ g_n\ ]\!]$

(Record) $\mathscr{L}_\mathsf{e}[\![\ e_{val}\texttt{\{}\ l_1\texttt{=}e_1\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}e_n\texttt{\}}\ ]\!]$

$\rightsquigarrow\ \mathscr{L}_\mathsf{e}[\![\ e_{val}\ ]\!]\texttt{\{}\ l_1\texttt{=}\mathscr{L}_\mathsf{e}[\![\ e_1\ ]\!]\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{=}\mathscr{L}_\mathsf{e}[\![\ e_n\ ]\!]\texttt{\}}$

Figure C.2.. Remaining rules for the transformation scheme $\mathscr{L}_e$ in Section 3.2.2.

(Selection) $\mathscr{L}_{\mathsf{e}}[\![\ \textit{expression}\,.\,\textit{label}\ ]\!]$

$\rightsquigarrow\ \ \mathscr{L}_{\mathsf{e}}[\![\ \textit{expression}\ ]\!]\,.\,\textit{label}$

(Blink) $\mathscr{L}_{\mathsf{e}}[\![\ \texttt{!}\textit{expression}\ ]\!]$

$\rightsquigarrow\ \ \texttt{!}\mathscr{L}_{\mathsf{e}}[\![\ \textit{expression}\ ]\!]$

(Any) $\mathscr{L}_{\mathsf{e}}[\![\ \texttt{any(}\ e_1\ \cdots\ e_n\texttt{)}\ ]\!]$

$\rightsquigarrow\ \ \texttt{any(}\ \mathscr{L}_{\mathsf{e}}[\![\ e_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathsf{e}}[\![\ e_n\ ]\!]\texttt{)}$

(Conditional) $\mathscr{L}_{\mathsf{e}}[\![\ \texttt{if}\ \textit{predicate}\ \textit{then}\ \textit{else}\ ]\!]$

$\rightsquigarrow\ \ \texttt{if}\ \mathscr{L}_{\mathsf{e}}[\![\ \textit{predicate}\ ]\!]\ \mathscr{L}_{\mathsf{e}}[\![\ \textit{then}\ ]\!]\ \mathscr{L}_{\mathsf{e}}[\![\ \textit{else}\ ]\!]$

(Application) $\mathscr{L}_{\mathsf{e}}[\![\ \texttt{(}\ \textit{identifier}\ e_1\ \cdots\ e_n\texttt{)}\ ]\!]$

$\rightsquigarrow\ \ \texttt{(}\ \textit{identifier}\ \mathscr{L}_{\mathsf{e}}[\![\ e_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathsf{e}}[\![\ e_n\ ]\!]\texttt{)}$

Figure C.2.. Remaining rules for the transformation scheme $\mathscr{L}_e$ in Section 3.2.2 (contd.).

C.3. Remaining Rules for $\mathscr{L}_i$

(Let) $\mathscr{L}_{\mathtt{i}}[\![\ \mathtt{let}\ d_1 \cdots d_n\ \mathtt{in}\ \mathit{expression}\ \mathtt{end}\ ]\!]$

$\rightsquigarrow\ \mathtt{let}\ \mathscr{L}_{\mathtt{i}}[\![\ d_1\ ]\!] \cdots \mathscr{L}_{\mathtt{i}}[\![\ d_n\ ]\!]\ \mathtt{in}\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{expression}\ ]\!]\ \mathtt{end}$

(Relation) $\mathscr{L}_{\mathtt{i}}[\![\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}\ ]\!]$

$\rightsquigarrow\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}$

(Function) $\mathscr{L}_{\mathtt{i}}[\![\ \mathtt{fun}\ \mathit{instance}_1 \cdots \mathit{instance}_n\ ]\!]$

$\rightsquigarrow\ \mathtt{fun}\ \mathscr{L}_{\mathtt{q}}[\![\ \mathit{instance}_1\ ]\!] \cdots \mathscr{L}_{\mathtt{q}}[\![\ \mathit{instance}_n\ ]\!]$

(Instance) $\mathscr{L}_{\mathtt{i}}[\![\ \mathit{pattern}\ \mathit{guards}\ \mathtt{=}\ \mathit{expression}\ ]\!]$

$\rightsquigarrow\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{pattern}\ ]\!]\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{guards}\ ]\!]\ \mathtt{=}\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{expression}\ ]\!]$

(Value) $\mathscr{L}_{\mathtt{i}}[\![\ \mathtt{val}\ \mathit{identifier}\ \mathit{expression}\ ]\!]$

$\rightsquigarrow\ \mathtt{val}\ \mathit{identifier}\ \mathtt{=}\ \mathscr{L}_{\mathtt{q}}[\![\ \mathit{expression}\ ]\!]$

(Guards) $\mathscr{L}_{\mathtt{i}}[\![\ \mathtt{|}\ g_1 \cdots g_n\ ]\!]$

$\rightsquigarrow\ \mathtt{|}\ \mathscr{L}_{\mathtt{i}}[\![\ g_1\ ]\!] \cdots \mathscr{L}_{\mathtt{i}}[\![\ g_n\ ]\!]$

Figure C.3.. Remaining rules for the transformation scheme $\mathscr{L}_i$ presented in Section 3.2.3.

$$(\textsc{Selection}) \quad \mathscr{L}_{\mathrm{i}}[\![\ \mathit{expression}.\mathit{label}\ ]\!]$$

$$\rightsquigarrow \quad \mathscr{L}_{\mathtt{i}}[\![\ \mathit{expression}\ ]\!].\mathit{label}$$

$$(\textsc{Any}) \quad \mathscr{L}_{\mathrm{i}}[\![\ \mathtt{any(}\ e_1\ \cdots\ e_n\mathtt{)}\ ]\!]$$

$$\rightsquigarrow \quad \mathtt{any(}\ \mathscr{L}_{\mathtt{i}}[\![\ e_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathtt{i}}[\![\ e_n\ ]\!]\mathtt{)}$$

$$(\textsc{Conditional}) \quad \mathscr{L}_{\mathrm{i}}[\![\ \mathtt{if}\ \mathit{predicate}\ \mathit{then}\ \mathit{else}\ ]\!]$$

$$\rightsquigarrow \quad \mathtt{if}\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{predicate}\ ]\!]\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{then}\ ]\!]\ \mathscr{L}_{\mathtt{i}}[\![\ \mathit{else}\ ]\!]$$

$$(\textsc{Application}) \quad \mathscr{L}_{\mathrm{i}}[\![\ \mathtt{(}\ \mathit{identifier}\ e_1\ \cdots\ e_n\mathtt{)}\ ]\!]$$

$$\rightsquigarrow \quad \mathtt{(}\ \mathit{identifier}\ \mathscr{L}_{\mathtt{i}}[\![\ e_1\ ]\!]\ \cdots\ \mathscr{L}_{\mathtt{i}}[\![\ e_n\ ]\!]\mathtt{)}$$

Figure C.3.. Remaining rules for the transformation scheme $\mathscr{L}_i$ presented in Section 3.2.3 (contd.).

C.4. Remaining Rules for $\mathscr{L}_c$

(Let) $\mathscr{L}_{\mathtt{c}}[\![\ \mathtt{let}\ d_1 \cdots d_n\ \mathtt{in}\ \mathit{expression}\ \mathtt{end}\ ]\!]$

$\rightsquigarrow\ \mathtt{let}\ \mathscr{L}_{\mathtt{c}}[\![\ d_1\ ]\!] \cdots \mathscr{L}_{\mathtt{c}}[\![\ d_n\ ]\!]\ \mathtt{in}\ \mathscr{L}_{\mathtt{c}}[\![\ \mathit{expression}\ ]\!]\ \mathtt{end}$

(Relation) $\mathscr{L}_{\mathtt{c}}[\![\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}\ ]\!]$

$\rightsquigarrow\ \mathit{identifier}\ \mathtt{<:}\ \mathit{identifier}$

(Function) $\mathscr{L}_{\mathtt{c}}[\![\ \mathtt{fun}\ \mathit{instance}_1 \cdots \mathit{instance}_n\ ]\!]$

$\rightsquigarrow\ \mathtt{fun}\ \mathscr{L}_{\mathtt{c}}[\![\ \mathit{instance}_1\ ]\!] \cdots \mathscr{L}_{\mathtt{c}}[\![\ \mathit{instance}_n\ ]\!]$

(Value) $\mathscr{L}_{\mathtt{c}}[\![\ \mathtt{val}\ \mathit{identifier}\ \mathit{expression}\ ]\!]$

$\rightsquigarrow\ \mathtt{val}\ \mathit{identifier}\ \mathtt{=}\ \mathscr{L}_{\mathtt{c}}[\![\ \mathit{expression}\ ]\!]$

(Guards) $\mathscr{L}_{\mathtt{c}}[\![\ \mathtt{|}\ g_1 \cdots g_n\ ]\!]$

$\rightsquigarrow\ \mathtt{|}\ \mathscr{L}_{\mathtt{c}}[\![\ g_1\ ]\!] \cdots \mathscr{L}_{\mathtt{c}}[\![\ g_n\ ]\!]$

(Record) $\mathscr{L}_{\mathtt{c}}[\![\ e_{val}\mathtt{\{}\ l_1\mathtt{=}e_1\mathtt{,}\ \ldots\mathtt{,}\ l_n\mathtt{=}e_n\mathtt{\}}\ ]\!]$

$\rightsquigarrow\ \mathscr{L}_{\mathtt{c}}[\![\ e_{val}\ ]\!]\mathtt{\{}\ l_1\mathtt{=}\mathscr{L}_{\mathtt{c}}[\![\ e_1\ ]\!]\mathtt{,}\ \ldots\mathtt{,}\ l_n\mathtt{=}\mathscr{L}_{\mathtt{c}}[\![\ e_n\ ]\!]\mathtt{\}}$

Figure C.4.. Remaining rules for the transformation scheme $\mathscr{L}_c$ in Section 4.2.

$$\text{(Selection)} \quad \mathscr{L}_{\mathsf{c}}[\![\ \textit{expression}.\textit{label}\ ]\!]$$
$$\rightsquigarrow \quad \mathscr{L}_{\mathsf{c}}[\![\ \textit{expression}\ ]\!].\textit{label}$$

$$\text{(Any)} \quad \mathscr{L}_{\mathsf{c}}[\![\ \texttt{any(}\ e_1 \cdots e_n\texttt{)}\ ]\!]$$
$$\rightsquigarrow \quad \texttt{any(}\ \mathscr{L}_{\mathsf{c}}[\![\ e_1\ ]\!] \cdots \mathscr{L}_{\mathsf{c}}[\![\ e_n\ ]\!]\texttt{)}$$

$$\text{(Conditional)} \quad \mathscr{L}_{\mathsf{c}}[\![\ \texttt{if}\ \textit{predicate then else}\ ]\!]$$
$$\rightsquigarrow \quad \texttt{if}\ \mathscr{L}_{\mathsf{c}}[\![\ \textit{predicate}\ ]\!]\ \mathscr{L}_{\mathsf{c}}[\![\ \textit{then}\ ]\!]\ \mathscr{L}_{\mathsf{c}}[\![\ \textit{else}\ ]\!]$$

$$\text{(Application)} \quad \mathscr{L}_{\mathsf{c}}[\![\ \texttt{(}\ \textit{identifier}\ e_1 \cdots e_n\texttt{)}\ ]\!]$$
$$\rightsquigarrow \quad \texttt{(}\ \textit{identifier}\ \mathscr{L}_{\mathsf{c}}[\![\ e_1\ ]\!] \cdots \mathscr{L}_{\mathsf{c}}[\![\ e_n\ ]\!]\texttt{)}$$

Figure C.4.. Remaining rules for the transformation scheme $\mathscr{L}_c$ in Section 4.2 (contd.).