STATIC ANALYSIS OF MARTIN-LÖF'S INTUITIONISTIC TYPE THEORY

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Alastair J. Telford

September 1995

Abstract

Martin-Löf's intuitionistic type theory has been under investigation in recent years as a potential source for future functional programming languages. This is due to its properties which greatly aid the derivation of provably correct programs. These include the Curry-Howard correspondence (whereby logical formulas may be seen as specifications and proofs of logical formulas as programs) and strong normalisation (i.e. evaluation of every proof/program must terminate). Unfortunately, a corollary of these properties is that the programs may contain computationally irrelevant proof objects: proofs which are not to be printed as part of the result of a program.

We show how a series of static analyses may be used to improve the efficiency of type theory as a lazy functional programming language. In particular we show how variants of abstract interpretation may be used to eliminate unnecessary computations in the object code that results from a type theoretic program.

After an informal treatment of the application of abstract interpretation to type theory (where we discuss the features of type theory which make it particularly amenable to such an approach), we give formal proofs of correctness of our abstract interpretation techniques, with regard to the semantics of type theory.

We subsequently describe how we have implemented abstract interpretation techniques within the Ferdinand functional language compiler. Ferdinand was developed as a lazy functional programming system by Andrew Douglas at the University of Kent at Canterbury.

Finally, we show how other static analysis techniques may be applied to type theory. Some of these techniques use the abstract interpretation mechanisms previously discussed.

Dedication

To my mother, Joan.

Acknowledgements

Firstly, my heartfelt thanks go to my supervisor, Simon Thompson, for his outstanding expertise, encouragement and immense patience. I would also like to thank the other members of the Theoretical Computer Science group at UKC, especially Andy King and David Turner for their advice and support. Andrew Douglas deserves a special mention due to his help with Ferdinand. He, along with Gerry Nelson, also helped with sharing the angst of doing a PhD.

I would also like to thank Andy King in his capacity as my internal examiner, as well as Ray Turner of the University of Essex, my external, for their helpful comments and criticisms.

I should also like to thank the Engineering and Physical Sciences Research Council (formerly the Science and Engineering Research Council) for funding me for three years of this PhD work.

Finally, I am very grateful to the various friends who have supported me during some difficult times, particularly Danuta, Mercia, Polly, Terry, Dr. John Buss and Marina.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we intend to show how static analysis techniques may be applied to the intuitionistic type theory of Per Martin-Löf [95, 134]. Our aim is to demonstrate that the information generated by such analyses may be used to improve the efficiency of programs written in a functional system based upon type theory. Consequently, not only will the programs be derived as *witnesses* to a logical specification, but they will be *automatically* converted to a computationally efficient form.

This approach may be contrasted with suggestions by, amongst others, the Nuprl and Göteborg groups (see [30] and [114], respectively), that the type theory be altered (to form separate classifications of types and propositions, for instance) so that certain "computationally irrelevant" proof objects may be removed during program development. These methods have, primarily, to be employed at the discretion of the program developer to form more efficient programs. We argue instead that such modifications are unnecessary and that abstract interpretation, for instance, can be used to enhance the efficiency of type theory when it is implemented as a lazy functional programming language The principal goal, therefore, is to produce an *optimised form of lazy evaluation* for a language based upon type theory. This means that proof objects that definitely will not be required by the computation will be removed during compilation and that only those proof objects that have to be computed will be processed to normal form. The main technique which shall be investigated will be a form of abstract interpretation known as **backwards analysis**. With this technique it is possible, within a *single* analysis, to detect both computational redundancy *and* possibilities for optimisations such as transforming call-by-need evaluation to call-by-value.

1.1 Functional programming

High-level imperative languages, such as FORTRAN, were developed in order to speed up the development of programs that would behave correctly with respect to an informal specification. Such languages have an operational semantics that reflects the von Neumann computer architecture [18] on which they were developed: values are assigned to and retrieved from named memory locations. There is also an explicit flow of control, so that the overall meaning of a sequence of statements in a language depends upon their ordering.

The explicit flow of control, together with assignments to an *implicit global state* of such procedural languages means that the programs may be *referentially opaque*. In other words, two calls to the same procedure with the same arguments may produce *different* results, according to the position of the call within the *dynamic* execution of the program and what *side-effects* may be effected by each procedure. (Side-effects are alterations to the global state that are distinct from the input/output behaviour of a procedure.)

This referential opacity makes reasoning about imperative programs problematic: a semantics for a program must be given in terms of a state to state function. (Potentially this is worse if GOTO statements are included, as the semantics then has to be based additionally on continuations which denote the future path of the execution of the program.) Furthermore, the semantics of a procedure is not reflected in the types of its input and output variables. This also means that the potential for correct parallelisation of imperative programs is weakened.

To enable programs to be reasoned about more easily, the **functional programming** paradigm has been advocated [6] and developed. Functional programming languages are based on the λ-calculus developed by Church[1]. Pure functional programming languages, such as Miranda[2] [139] and Haskell [66], which contain no imperative features, are side-effect free and referentially transparent. These languages, however, not only remove the potential for actions upon the global state to be altered, but add the concept of *functions as first-class citizens*. This means that any function may be passed as an input parameter, or form the result of, another function. A function that takes a function as input or produces one as its result is termed *higher-order*.

Pure functional programming languages are also usually based upon a *lazy* reduction strategy. Lazy reduction is a refinement of the normal order (i.e. leftmost and outermost) reduction strategy for the lambda calculus whereby each argument to a function is evaluated at most once and structured data is not necessarily fully evaluated. In practice, however, some functional languages such as LML [3] are not fully lazy. (Some pure functional languages have been developed, such as HOPE [25], which combine a basically strict evaluation strategy with lazy evaluation of data structures such as lists and trees.)

It has been suggested by Hughes [73] that higher-order functions, together with lazy evaluation form the perfect "glue" for constructing large programs in a modular way. In particular, it allows the input to a function to be an infinite stream, with the function being applied demanding as little of the stream as is necessary for its computations. This allows input/output dialogues to be written

[1]Church's own account of his work is given in [28]. A comprehensive survey of lambda calculus is given by Barendregt in [7] and an introduction in [53].

[2]Miranda is a trademark of Research Software Ltd.

in pure functional languages. Functional languages which do not have this lazy evaluation strategy at all usually have impure features such as reads and writes upon the state of the machine in order to implement input and output[3] (see, for example, SML [57]).

There are some disadvantages with lazy functional languages, however. One major problem has been with regard to efficiency. Since the inputs to a function cannot be evaluated prior to calculating the function, *closures* have to be formed. Closures consist of an expression to be evaluated and an environment of variables (the formal parameters) and their associated expressions (the actual parameters). The formation of closures is costly in terms of space complexity and the associated memory retrievals have a deleterious effect on the time efficiency as well. One method of remedying this has been through **strictness analysis**, a form of *abstract interpretation*. This method has allowed the detection of which function parameters *must* be evaluated during the evaluation of the function. Consequently, such parameters can be evaluated prior to the execution of the function and closures may be simplified.

Secondly, whilst languages such as Miranda have polymorphic strong typing and allow rapid prototyping that may be seen as specifications for more efficient implementations [138], their type mechanism is not strong enough to reflect fully the specification of a program. For example, `[*] -> [*]` is the type of quicksort in Miranda. It is also the type of the identity function over polymorphic lists. Hence, it is not possible to have *integrated* programming and proofs of correctness within existing functional programming systems. It should be mentioned, however, that it is possible to develop functional applications readily from formal specification languages — an example of the development of a screen editor in Miranda from the specification language Z [42] is given in [16].

Finally, computations may not necessarily terminate. If a program terminates at all, it will do so under the lazy evaluation strategy, by the normalization theorem [7]. In this sense, therefore, the lazy semantics is the *greatest fixpoint* (in the sense of being the most informative). It is still impossible to decide in general, however, whether a program will terminate. Whilst this does not create the same problems as the implicit state and side-effects of imperative languages, it does make program analysis more complicated since each semantic domain is lifted by the undefined object. This is illustrated in the work of Thompson on Miranda and Haskell [133, 137, 136] where it is necessary to establish the truth of a predicate for both the defined objects of a type and the undefined value. In view of this Turner has proposed a particular paradigm of functional programming whereby termination will be guaranteed [141]. It would seem, however, that this system of *elementary strong functional programming* has the drawback of restricting the expressive power of a Miranda-like language whilst not providing the rich system of types and the "programs as proofs" correspondence present in the theory of Martin-Löf which we describe below.

[3] Input/output functions could be performed in a purely functional manner in a strict language by the use of continuation or monadic based input/output.

1.2 Type theory

Martin-Löf's intuitionistic type theory has received much attention over recent years as a basis for future functional programming languages. Originally, however, it was developed to provide a system to formalise the constructive mathematics[4] of systems such as Bishop[5] [13]. Bishop's work showed that constructive mathematics was not lacking the expressiveness of its classical counterpart, as had been thought even by Brouwer, the founder of the intuitionistic philosophy of mathematics [61].

In constructive mathematics for a logical formula to be proved an explicit *proof object* has to be exhibited for it. Consequently the **law of the excluded middle** of classical logic is not valid. As is stated by Bridges and Richman in [17]:

> The constructive mathematician interprets the logical connectives and quantifiers according to **intuitionistic logic**.

They go on to elaborate on the effect the rejection of the law of the excluded middle has on the connectives $\vee$ and $\Rightarrow$, and the quantifier $\exists$. (For instance, it is not possible to derive $\neg\neg P \Rightarrow P$, for an arbitrary predicate P, in intuitionistic logic.)

It is known by the **Curry-Howard** correspondence [35, 65] that the proof objects may be interpreted as programs — in Martin-Löf's theory they are in a form of Church's explicitly typed lambda calculus[6] — whilst the intuitionistic logical formulas which they *witness* are isomorphic to types (and specifications) in programming.

This has several important consequences in that programs may only be formed if they are proof objects of logical theorems in the system of type theory. Hence, programs may be developed with the aid of automatic theorem provers and should consequently be easier to transform correctly than those of previous programming languages. This is a consequence of the absence of undefined expressions which make mathematical laws invalid. For instance, in Miranda, the following idempotency law does not hold if it is possible for either `a` or `b` to be undefined.

```
a \/ (b \/ a) = b \/ a
```

Hence, we cannot, in general, make a program transformation by replacing the less efficient left hand side by the right hand side. Examples of program transformation in type theory are given in [134, Section 6.7]. Moreover, any language developed from type theory should already have most of its semantics explained by the term models (consisting of the reduction rules and the canonical proof objects) of the theory [98].

[4] [9] covers constructive mathematics thoroughly.

[5] It should be stressed that Martin-Löf was not only interested in formalising Bishop's work.

[6] As compared to languages such as Miranda which are based on Curry's implicitly typed calculus. Types may be assigned via the algorithms of Hindley [62] and Milner [101] and programmer-supplied types checked via Mycroft's algorithm [104].

Type theory provides two main improvements upon the functional languages that have been developed up to the present. Firstly, it provides a much richer type structure than contemporary functional languages. This type structure means that types may also be viewed as **specifications** unlike in traditional languages (both imperative and functional) where the types only help in making secure programs: the types are not precise enough to reflect the intension of the specification. For instance, Burstall's idea of *deliverables* [23, 26] may be represented within the types of the theory (although we would suggest that the explicit distinction made between propositions and types is unnecessary). Also, the availability of *dependent types*, where, for instance, the type of the second element of a pair may depend on the value of the first element, allows modules and abstract datatypes to be defined elegantly [92, 112]. Secondly, unlike languages such as Miranda, Haskell, etc., it provides **strong normalization**[7]: every program will terminate. This is a desirable property for a programming language to have as it means that as all programs are *total* we have a consequent improvement in the ability to verify programs. (Undefined objects do not have to be considered so the results of programs will truly lie within their declared type: if an expression is undefined it could be of any type.) Moreover, since the Church-Rosser theorem [7] also holds, every reduction sequence must result in the same normal form for a given expression. This means that we have, in a sense, more latitude in defining reduction sequences for programs than in languages such as Haskell.

Languages developed from type theory should continue the progress away from the traditional imperative languages and improve upon current practice in functional programming. It should be admitted that there are some drawbacks to the use of type theory as a functional programming language:

1. General recursion over arbitrary data structures is not available in type theory. Without this restriction strong normalization could not be guaranteed. However, this does mean that certain algorithms cannot be presented as elegantly or as efficiently as a language such as Miranda. Fortunately, however, Martin-Löf's system is deliberately an open one and new structures can be added to the system, provided that they do not produce inconsistency. For example, lists are added to the theory in [113] and Chisholm added parse trees in [27]. Attempts have been made (in [116], for instance) to introduce *well-founded recursion* which is more general than primitive recursion but which is guaranteed to terminate.

2. Some proof objects may be seen to be *computationally redundant* and break the Curry-Howard correspondence between proofs and programs. This idea is covered in Section 3.4.1 of [5] and is discussed in the section below.

[7] The strong normalization property is due to the fact that the terms of the theory are based on those of Church's explicitly typed lambda calculus. This was shown to be strongly normalising by Tait [128]. The proof is actually based on establishing a stronger property called variously *stability* [134] or *strong computability* [53].

1.2.1 Computationally redundant proof objects

This work is primarily concerned with the problem of computationally redundant proof objects in type theory. It may be seen that some of the proof objects are not relevant to the computation in which we are interested. For example, consider the index function upon lists where we wish to retrieve the nth element of a list l. This is defined in Miranda as the ! function:

```
(!)  :: [*] -> num -> *
(x:xs) ! 0 = x
(x:xs) ! (n+1) = xs ! n
```

However, in type theory, the function is defined as follows (using the informal presentation style of type theoretic functions that comes from [134] — typing is denoted by ':' and list construction by '::' i.e. the reverse of the Miranda convention), where A denotes an arbitrary type.

$$\mathit{index} \quad : \quad (\forall l : [A]).(\forall n : N).((n < \# l) \Rightarrow A)$$

$$\begin{aligned}
\mathit{index}\ []\ n\ p &\equiv_{df} \mathit{abort}_A\ p \\
\mathit{index}\ (a{::}x)\ 0\ p &\equiv_{df} a \\
\mathit{index}\ (a{::}x)\ (n+1)\ p &\equiv_{df} \mathit{index}\ x\ n\ p
\end{aligned}$$

Apart from the type (which is more explicit than its Miranda counterpart — the type asserts the domain of indices for which it makes sense to apply the index function for a given list), the parameter p and the first clause of the type theoretic definition form the difference between the two presentations.

The parameter p is a proof that the second parameter is less than the length of the list. This, and the first clause, guard against the possibility (which may happen in Miranda) that an out of range element may be asked for. The first clause takes care of the case where we have a pathological proof object (which will be of type *bottom*, $\perp$) that is effectively claiming that the empty list has a greater length than a natural number. This clause ensures that the function is *completely presented* so that we can recover easily from any mistakes in program development: an error of this kind shows us the type of the object that should have been correctly derived and the type scheme tells us the error in the derivation that occurred.

The *abort* expression simply returns an object of the required type A and halts. This object may effectively be anything, since we have detected an abnormality in the proof derivation: $\mathit{abort}_A\ p$ is a witness to *ex falso quodlibet*.

However, we may note that p is neither of the result type A nor is it printable: it exists solely as a check upon the correct derivation of a program. Nevertheless, it would seem that we have to evaluate it to normal form in order to evaluate to a printable value any program in which it occurs.

1.2.2 Removal of proof objects

In attempting to remove redundant proof objects, the main approach that has been tried has been to remove these proof objects from the theory itself. This was first done by introducing a **subset type** [111, 5] of the form:

$$\{ x : A \mid B \}$$

This represents the type of all elements, a, of A which satisfy $B[a/x]$. It is a weakening of the existential type with the second member of each pair (the witness that a satisfies the predicate B) being discarded.

The subset type has a number of undesirable properties, however. Firstly, the introduction and elimination rules do not adhere to the inversion principle [126] that elimination rules should follow in a uniform way from the introduction rules. Secondly, proof objects do not have a unique type ($a : A$ and also $a : \{ x : A \mid P \}$ for a range of predicates, P). Finally, as is shown in [124], $(\forall x : \{ z : A \mid B(z) \}).B(x)$ can only be derived if $(\forall x : A).B(x)$ is derivable anyway. (Even in the extensional version of Martin-Löf's type theory [97], Salvesen and Smith show that $(\forall x : \{ z : A \mid B(z) \}).B(x)$ cannot be derived for arbitrary formulas A and B.)

The *squash type* of the Nuprl group [30] sought to represent the judgement A *is true* via:

$$Triv : \{t : \top \mid A\}$$

This, of course, reduces the witnessing information further than the subset type. This too has disadvantages and Salvesen has shown [122] that it has a similar weakness with regard to universal quantification.

Thus a new theory has been produced by the University of Göteborg group, called the **subset theory** [114], which was first mooted in Salvesen and Smith's original 1987 piece on the subset type [123]. This theory distinguishes between propositions and types, and adds a notion of a predicate being "True". Whilst the new theory may be interpreted in the original, it clearly lacks the direct Curry–Howard correspondence: types are represented solely in set theoretic notation (e.g. Π, Σ) with the propositions expressed via the symbols of predicate logic (e.g. $\forall$, $\exists$). Moreover, it is argued by Thompson, in [135] and Section 7.4 of [134], that such an approach is not necessary as the main objectives of the development of the subset theory can be met without altering the basics of the system. The first objective of the subset theory was to make specifications (and hence the derived functions) more "natural". Thompson suggested that this aim may be met by a simple process of *skolemization* of the type specification (where in this case the type is viewed as a logical statement). In other words, we distinguish those objects in which we are interested and bring them to outermost level of scope in the type.

The other objective of the subset theory is to improve the *efficiency* of the system by removing the computationally redundant witnessing information. However, if a language based upon type theory is evaluated lazily, then only those objects that absolutely have to be computed will be processed to normal form.

Nevertheless, we still need to avoid the computationally wasteful creation of expressions (and their associated closures) which will never be used. To do this, we need to perform analyses upon type theory programs in order to find the most efficient means of computing such a program. We shall use *static analysis* techniques which to date have been developed to enhance traditional functional programming implementations. These techniques should help us to both eliminate the type theoretic witnessing information from our programs and to allow the remainder of the program to be evaluated in the most efficient way possible. Hence we strongly suggest that the distinction between propositions and types is unnecessary: the computationally redundant proofs of the propositions will be eliminated automatically by a compiler practising the techniques we present.

1.3 Static analysis and abstract interpretation

Static analysis is the term used to describe a variety of techniques which may be applied to computer programs when they are compiled. The idea is to deduce properties about a source program which may then be used to optimise the object code produced. The crucial point is, of course, that the properties are deduced without actually running the program. We shall focus upon the most commonly used static analysis technique for applicative languages known generally as **abstract interpretation**. More specifically, we shall use the form of abstract interpretation known as **backwards analysis**. Below we review the development of this increasingly important area of analysis of computer programs.

1.3.1 History of abstract interpretation

Patrick and Radhia Cousot were the first to attempt to develop abstract interpretation and relate it to the semantics of a programming language [32, 33]. The idea they developed was to use partial information about the inputs of a model imperative language to derive partial information about the outputs. They observed that the static semantics of a language did not have computational content in that it simply indicated the result (or a sequence of intermediate results) of a program. They established what was meant by a *safe* abstract interpretation of a language in that there would be *abstraction* and *concretisation* maps between the static semantics of a language and its abstract interpretation, and an abstraction followed by a concretisation would leave a result consistent with the static semantics. An introduction to the optimisation of imperative programs is given in [105].

Mycroft was the first to apply the ideas of abstract interpretation to functional languages in his PhD thesis in 1981 [103]. This analysis was rather restricted in that it only dealt with first-order programs and did not properly analyse lazy data structures. In 1985, Burn, Hankin and Abramsky described a method for the abstract interpretation of higher-order functions [21] and Abramsky extended the theory to polymorphic functions [1]. In 1987, Wadler made a major contribution to solving the problem of lazy data structures [144].

The above methods were all forms of *forwards* analysis, where abstract values are propagated from the inputs to discover (partial) information about the outputs of a program. A survey of this form of abstract interpretation of lazy functional programs is given in [20]. An alternative, however, is to do the reverse i.e. derive information about the inputs from the *context* of the output. The prime motivations for this approach were to try to capture the property of *head-strictness*, which would indicate whether lists were strict in their individual elements. Indeed, it was shown by Kamin [87] that this property could not be captured by forwards analysis. Hughes and Launchbury also showed that the analysis of products could be performed more accurately by a backwards analysis, with forwards analysis giving no advantage for sums [74]. It was thought, moreover, that the flow of information from a result to a function's parameters was more natural for properties such as strictness. In addition, backwards analysis has been seen to be more efficient than forwards analysis (although this is shown not to be a property of the direction of analysis in [39] — it is due to the fact that backwards analysis does not usually consider the relationships between parameters). A discussion of methods to improve the efficiency of forwards abstract interpretation is given in [54].

The backwards analysis method was first developed for the functional idiom by Johnsson in 1981 [81]. Backwards analysers were later developed by Hughes [67] and Wray [149]. Much of the theoretical work in this area has been done by Hughes [68, 69, 71, 70] with a main strand of the work being the presentation of backwards strictness analysis using projections[8] [147]. Relationships between forwards and backwards analysis are given in, for example, [39, 75]. Much of this work is reviewed in greater detail in Chapter 2.

1.3.2 Abstract interpretation used in this work

This work concentrates upon the backwards analysis techniques of Hughes, [69]. We have chosen this method for the following reasons:

1. As noted above, it is capable of detecting abstract properties of data structures more accurately than forwards analysis.

2. In the forms of forwards and backwards analysis commonly used, backwards analysis may be seen to be more efficient than forwards analysis.

3. For the properties which we wish to capture, such as computational relevance, there is a more natural flow of information backwards rather than forwards. The results of the function applications that we compute are clearly computationally relevant and so we see how this propagates backwards to the parameters of a function.

[8]Projections are continuous, idempotent functions over domains which approximate (in the sense of being less defined than) the identity function.

We examine a hierarchy of analyses, ranging from simple **neededness analysis**, which distinguishes between computationally relevant and redundant proof objects, to **sharing analysis**. Sharing analysis subsumes simple strictness analysis (where we distinguish between those objects which must be evaluated, i.e. are *strict*, and those which may not be evaluated, i.e. are *lazy*), detects parameters which will be evaluated more than once and so should be "shared" by different parts of the program, and differentiates between those objects that are needed and not needed by a computation.

1.4 The scope of this work

Chapter 2 gives more detail on the issue of computational irrelevance in type theory, techniques which have been used to try to solve the problem and develops the theory of backwards analysis as an alternative solution. The theory developed enables parameters that will be unused in a computation to be automatically detected. The work presented includes the sharing analysis presented by Hughes [69] and implemented by Fairbairn and Wray [48]. This sharing analysis detects whether parameters are used or unused, as well as determining whether functions are strict in certain parameters and whether inputs to functions may be shared during the reduction of a function application. Thus a single analysis allows both the irrelevant proof objects to be detected and for the optimisations that have been applied to other functional programming languages to be applied to type theoretic programs.

A formal account of the neededness aspect of the analysis with regard to the theory TT of [134] is given in Chapter 3. This includes a proof of correctness of the analysis with respect to the computational necessity of elements of the formal computation rules of type theory. This thus shows that the analysis is sound and will not indicate that a parameter is unused when, in fact, it may be required by the computation.

Chapter 4 describes how the ideas presented earlier were implemented within a compiler for a functional language, Ferdinand, based upon TT. This work thus enhances the research performed by Douglas [44] into producing a practical functional programming system based upon type theory. Of particular note in this Chapter is the presentation of a method for removing parameters that have been detected as redundant.

Chapter 5 reports on how other forms of static analysis may be applied to type theory, with particular focus on the time complexity of type theoretic programs. We show how the necessity analysis developed in Chapter 2 may be applied to the time analysis of the programs of TT, which was originally developed by Bjerner [14]. The use of the necessity analysis may be seen to provide an upper bound (worst case) for the time complexity of type theoretic programs. We also show how an analysis may be used to provide a lower bound (best case).

We present our conclusions and suggestions for further research in Chapter 6.

There are also two appendices: the first, Appendix A, gives examples of the application of backwards analysis to type theoretic programs, whilst the second Appendix B gives further documentation upon the implementation of the analysis within the Ferdinand compiler.

1.5 Related work

As mentioned above, this work is primarily designed to show that modifications to the type theory, such as the subset theory [114], are unnecessary in order to remove computationally irrelevant proof objects. The subset theory is embedded within the ALF theorem prover [4].

The AUTOMATH system of de Bruijn [40], like the subset theory, separated types and propositions in order to remove irrelevant (constructive) proofs from classical deductions. It has also been argued recently [143] that the proper place for the subset theory is to elide information in proofs (and thus reflect the actual practice of Bishop in his book [13]).

Paulin-Mohring [115] presents a method of extracting programs, with computationally irrelevant material removed, from proofs in the calculus of constructions [31] with a scheme for realizations added. This system makes a distinction between propositions that have a “computational informative” content and those that have only “logical” content. The process of realizing proofs is performed by marking parts of the propositions that are redundant computationally. Takayama [130] followed up Paulin-Mohring's work in designing a partially automated technique for pruning natural deduction proof trees as a prelude to the realization of executable functions in a non-type-theoretic version of constructive logic, QPC [129]. Berardi and Boerio [11, 15] built upon this by casting a lambda expression as a tree to be pruned. This was improved upon in [12] where a notion of subtyping was developed to produce a simplification relation which allowed optimized λ-terms to be deduced. We discuss this briefly in Section 2.2.

Luo has produced an extended calculus of constructions [89, 90] in which, again, types and propositions are kept separate. It also adds a single impredicative universe of propositions. It is intended to be a unification of the calculus of constructions and Martin-Löf's type theory. (One of Martin-Löf's earliest works on type theory [94] also had the impredicative notion of a type of all types.) The LEGO proof development system [91] was developed to accompany this work and McKinna has developed a category-theoretic approach to developing programs in that system [99].

Systems based upon Feferman's theory of types [49] may also be contrasted with this work. Such systems (e.g. *TK* [60] and *PX* [58]) separate entirely the theory of functions and operations and the theory of types, with a scheme of logical assertions being defined over the simple types. Moreover, programs, which, unlike those of Martin-Löf's type theory, are not strongly normalising, are extracted from proofs by a process of realizability. Such systems are described in general in [142]. Of particular note is the paper by Henson [59] which discusses how the realizability process removes computationally redundant proof objects.

Chapter 2

Backwards analysis of type theory

2.1 Introduction

In this chapter we explore first the computational relevance of proofs with regard to program development in type theory. We compare approaches to the specification of programs and examine how a separation between the purely proof theoretic and the computationally relevant is made. We argue that such a distinction is unnecessary given the form that we believe specifications should take and the abstract interpretation theory that we subsequently develop.

The development of the theory is preceded by a discussion of the relative merits of forwards and backwards analysis. We then proceed to develop the basic theory of a backwards analysis, based on the work of Hughes [67, 69, 72] of the system called TT in [134]. Of particular note is Section 2.9 which presents the treatment of elements of types which *exhibit computational redundancy* in the phraseology of [5].

We then show how the theory presented thus far may be used to analyse the *index* function presented in Section 1.2.1 of the introduction. This example demonstrates that the computationally irrelevant third argument to *index* may be automatically detected and removed.

Following the *index* example we develop the theory into more advanced areas, particularly higher-order functions. We subsequently give a formal presentation of the theory in order to make precise the preceding work.

Whilst our main concern is with regard to the terms of the theory that may be computed, we then discuss the analysis of the *types* of the theory. This is necessary since the types are "first-class citizens" (unlike the functional programming languages that were analysed by Hughes): terms may appear within types and types may be the inputs and results of functions.

Finally, we give the analysis of quicksort as a larger example which, as it calls the filter function, illustrates our approach to higher-order functions and present our conclusions.

2.2 The subset theory & program specification

The purpose of the introduction of the subset type [111] and, more recently, the subset theory [114] was to separate computationally redundant proof theoretic information from programs. This would have the benefit of increasing the efficiency of programs constructed in type theory since proof objects that would witness, for instance, the fact that a list was non-empty would not appear in the resulting program and would not have to be computed.

Elements of subset types are introduced by the following rule (which is Thompson's interpretation in Section 7.2 of [134] of the rule given in Section 18.5 of [114]):

$$\frac{a : A \qquad p : B[a/x]}{a : \{\, x : A \,|\, B \,\}} \quad (\mathit{Set\ Intro})$$

This should be compared with the introduction rule for the existentially quantified types:

$$\frac{a : A \qquad p : P[a/x]}{(a, p) : (\exists x : A).P} \quad (\exists\ \mathit{Intro})$$

Note that the premises to both rules are the same but that the introduction rule for *Set* discards what is the second component of the pair that witnesses membership of the existential type. It is this *information loss* that is precisely the point of the subset type in that the second component of the pair which witnesses the existential type *may* be a witness to a proposition such as the first element of the pair, a, being a sorted list.

Elements of a subset type are eliminated via the following rule:

$$\frac{a : \{\, x : A \,|\, B \,\} \qquad \begin{array}{c}[x : A;\ y : B]\\ \vdots \\ c(x) : C(x)\end{array}}{c(a) : C(a)} \quad (\mathit{Set\ Elim})$$

This rule has the side condition attached that y may not appear free in c or C. Again, this may be contrasted with its counterpart for existential types:

$$\frac{p : (\exists x : A).B \qquad \begin{array}{c}[x : A;\ y : B]\\ \vdots \\ c : C[(x, y)/z]\end{array}}{\mathit{Cases}_{x,y}\, p\, c\ : C[p/z]} \quad (\exists\ \mathit{Elim})$$

Here both c and C will, in general, depend on y as well as x and $\mathit{Cases}_{x,y}\, p\, c$ is computed by substituting the first and second components of the pair p for x and y, respectively, in c. The subset type, however, does not have an associated computation rule since no new selector is introduced: the computation rule is just that for $C(x)$.

The drawback of the subset type lies precisely in the fact that it does discard information. Its weakness in practice is shown in the integer division by two example given in Section 21.1 of [114]. There it is acknowledged that the subset type is too weak to interpret the following proposition as a set:

$$(\forall x : N).(\exists y : N).((I(N, x, (y * 2))) \vee (I(N, x, ((y * 2) + 1))))$$

(Above, $I(N, x, (y*2))$ and I(N,x,((y*2) +1)) are equality types.) Also, it is argued in Section 7.2 of [134] that functions to take the head and tail of a non-empty list cannot be derived via subset types. The problem lies in the fact that the rules for subsets encapsulate an *ad hoc* disposal of information; in the elimination rule, for instance, the assumption $y : B$ can be used in the proof of C, but it cannot occur in either c or C. As Martin-Löf says in [110]:

> So you have to use your skill in deciding whether to keep the information in a proof explicit or to forget it. And if you throw away something that you need later on, then you are in a bad position.

The theoretical weakness of the subset type is discussed in [124].

Propositions and types are separated in the subset theory which was proposed [123, 114] to overcome the problems of the subset type whilst separating the computationally relevant from the proof theoretic. Subsets are introduced by the rule

$$\frac{a : A \qquad P(a)\ true}{a : \{\, x : A \,|\, P(x) \,\}} \quad (Subset\ Intro)$$

Note that the premise $P(a)\ true$ has replaced $p : P[a/x]$ of the subset type. It is the need to have such propositional judgements that we regard as the weakness of the subset theory in that a parallel set of rules needs to be provided to form propositions and to derive judgements which show that a predicate is true. We suggest that the rules of Martin-Löf's type theory (which are presented in Chapter 3) are sufficiently complex to make any such augmentation undesirable from the viewpoint of people working with type theory in practice. More importantly, the separation of propositions and types does not increase the proof theoretic strength of the original theory, although it may be seen to increase the ease of expression.

A justification for the development of the subset type and theory given in [114] is that the specification of a function should be of the form:

$$(\forall x : A).(\exists y : B).P(x, y)$$

A proof of this type will be a function which returns a pair for each input, the first part of the pair being the value that we are interested in and the second part being a proof that x and y meet the specification relation, P. Even if we accept this, Thompson shows in [135] that it is still possible to convert the above form to the specification

$$(\exists f : A \Rightarrow B).(\forall x : A).P(x, f\,x)$$

This is done via the application of the *axiom of choice* and the elimination rule for implication. The axiom of choice[1] may be derived from the $\exists$ *Elim* rule given above. The above may be seen as a *Skolemization* process whereby we assert the existence of a function named 'f'. Thompson shows in Section 4 of [135] how the Dutch national flag partitioning problem [41] may be specified without recourse to the subset type used in Section 22.2 of [114]. Thompson also mentions that certain functions, such as hashing operations, do not naturally have the form of specification given in [114]. The Skolemization process thus results in specifications in which the proofs will be pairs of functions and witnesses that the behaviour of a function meets a predicate.

This leaves the other reason for the use of subsets, computational efficiency. The *lazy evaluation* strategy (whereby arguments to functions are calculated only if they are required to calculate the final result of a program) will, if used in a type theoretic program, ensure that the purely proof theoretic parts of the program are never actually calculated. However, the efficiency of a resulting object program may be hampered if computationally redundant proof objects are included within it, since closures (and hence machine space) will have to be allocated for the proof information which is never actually used.

We will show how the computationally irrelevant expressions will be detected by the use of the **ABSENT** expression in a backwards analysis of type theory. With the use of a suitable **context lattice** of values which abstract properties of programs, this will allow computational irrelevance to be detected and removed at compile time. Indeed, if p is the witness to

$$(\exists f : A \Rightarrow B).(\forall x : A).P(x, f\,x)$$

then if we use only the functional part via $Fst\,p$ then the second part of the pair p will be detected as *unused* and therefore may be removed during compilation. Here computationally redundant proof information occurs due to the form of the specification and can be readily detected and removed. However, it will not always be the case that the second part of a witness to an existential type can be discarded. (Such proof objects may be seen as variant records, for example, where the type of the second part of the pair depends upon the value of the first part.) Also, some computationally redundant proof information is *essential* so as to witness the fact that other inputs are valid. For instance, we would have to supply a proof that a divisor was non-zero or that a given index was meaningful with respect to a list (i.e. between zero and the length of the list). Thus it is these proof objects that maintain the strong normalization property of type theory. These proof objects have to be represented by the domain type, A, of the Skolem function, f, in the above formula. For instance, for the *tail* of list function over

[1] It should be noted that the axiom of choice that may be derived in Martin-Löf's type theory is intuitionistically valid in that it does *not* give rise to a proof of the law of the excluded middle.

natural number lists, a possible specification is:

$$\begin{array}{l}(\exists tl : ((\forall l : [N]).(nonempty\, l \Rightarrow [N])).\\ \quad (\forall l' : [N]).(\forall p : (nonempty\, l')).\\ \qquad I([N], l', ((hd\, l'\, p) : (tl\, l'\, p)))\end{array}$$

The above says that as input to the Skolem function, f, we require both a list and a proof that the list is non-empty. The second part of the specification relates the Skolem function tl to the hd (head of list) function whereby a list must be equivalent to constructing the list with its head and tail components. It should be noted here that hd also requires as input a proof that the list is non-empty.

Hence, whilst we can remove some purely proof theoretic information using the axiom of choice and the Fst selector, there will still remain some computationally redundant parameters in the functions in which we are interested for calculations, such as tl.

Berardi and Boerio's subtyping optimisation Berardi and Boerio, following on from the work of Paulin-Mohring [115] and Takayama [130], have developed an algorithm to detect and remove "useless computations" from a λ-calculus system based upon Gödel's system $\mathcal{T}$ [52]. They develop a notion of subtyping where Ω-types are used to develop a simplification relation. The base Ω-type consists of the natural numbers identified together i.e. one solitary element. The set of natural numbers, N, is considered as a subtype of Ω and each type of the simply typed lambda calculus is a subtype of some Ω-type. Optimisation consists of replacing computationally redundant terms of a type A with dummy constants of the corresponding Ω-type of which A is a subtype.

Whilst it would appear that their method could be extended to type theory to deal with computational redundancy, the system which we present has a significant advantage over theirs in that it is more modular (analysis and optimisation phases are separate) and can be used to obtain optimisations in addition to the elimination of computational redundancy, such as strictness detection. Moreover, we would suggest that our system is more easily extensible to new constructs in type theory than their algorithm.

2.3 Forms of abstract interpretation

The idea of abstract interpretation is to discover, without executing the program, *partial information* about the parameters of a program so that the program may be complied more efficiently. In other words, we simplify the range of values that objects may take so that we may compute a restricted amount of information about the variables in which we are interested. We therefore produce an *abstract semantics* for a particular interpretation of a language. The *actual semantics* for the language is, in a sense, too exact, in that it only stipulates the results of a computation in terms of a given expression: it does not give us data about parts of the computation or how the result may be categorised. Abstract interpretation allows us to categorise results. A simple example of this is the "rule of signs" for multiplication in arithmetic (two numbers of the same sign produce a positive number, two numbers of opposite sign produce a negative) in which we are only interested in the *sign* of a number rather than its actual value (e.g. -3).

The basic form of abstract interpretation is *forwards analysis* where we propagate *abstract values* as inputs to produce an abstract result. The rule of signs is an example of a forwards analysis. *Backwards analysis*, naturally, is the reverse: we take abstract information for the output of an expression to be analysed and propagate it to give information about the inputs of the program. We can thus tell, for example, whether an input parameter has to be evaluated in order to evaluate the main expression of the program.

2.3.1 Forwards analysis

Forwards analysis, pioneered for functional languages by Mycroft [102, 103], is probably the most natural form of abstract interpretation: instead of running the actual program on concrete arguments we "run" an *abstract* version of the program on *abstract* arguments. The abstract arguments have initial values assigned to them via an environment. These initial values represent *contexts* i.e. sets of expressions having a particular property. It is fairly straightforward to translate directly from a function in the concrete syntax of an implementation of type theory to produce an abstract version. In fact the main differences are simply the addition of some symbol such as $*$ to denote the fact that we are dealing with abstract variables and functions, and an environment, ρ say, which gives us the abstract values bound to the variables. Also, operators such as $\sqcup$ (to denote an "or"-like operation) and $\sqcap$ (like "and") are used to combine expressions in the abstract version of the program. Since we ensure that we are dealing with a *domain* of abstract values we may deal with recursion in a similar way to the method for backwards analysis we present later i.e. by finding the fixed point of an ascending chain of approximations. Here, however, we meet the prime drawback to forwards analysis since finding fixpoints of recursive equations involving abstract functions may, in the worst case, have an order of complexity exponential in the number of arguments. Even for Mycroft's simple two-point domain of values [103] (with $\mathbf{0}$ abstracting the strict property and $\mathbf{1}$ the non-strict) 2^n calculations of the abstract function will be required for just *one* iteration in the fixpoint calculation, where n denotes the number of parameters of the function. (There are two possibilities for each abstract input and thus 2^n permutations for all abstract parameters.) Similarly, 2^n comparisons have to be made after each iteration to determine whether the fixpoint has been reached. Moreover, in the worst case, the number of iterations to find the fixpoint may also be 2^n. The complexity of the calculations may be seen from the fact that fixpoint calculations with abstract values can be represented as a truth table in classical propositional logic (see Section 4.1 of [29]) and thus is equivalent in complexity to the satisfiability problem which is NP-complete [120].

This complexity difficulty is exacerbated when we abstract from data structures (by, for example, the four-point domain for lists devised by Wadler [144]) or higher-order functions (each abstract value of a functional argument will form a lattice of values — see [21]). There have been various attempts to improve this situation. Clack and Peyton Jones first proposed the idea of *frontiers* for forwards strictness analysis [29]. Frontiers allow the number of comparisons for each iteration to be reduced by partitioning the lattice of possible input abstract values into two portions. This idea was developed and formalised in [93, 76, 79]. In addition, an outline for the implementation of frontiers using lattices developed via the type class mechanism of Haskell was proposed in [82].

However, as the frontiers algorithm does not change the number of approximations that have to be calculated, this approach was found to be insufficient to improve computational efficiency in practice so Hankin and Hunt have developed their ideas of approximation further, which they show to be a form of widening and narrowing ([54]; widening and narrowing were first introduced by the Cousots to cope with large lattices in the abstract interpretation of imperative programs [33]). Widening allows the lattice of approximations to be traversed more quickly, to find a safe approximation to the fixpoint, whilst narrowing allows a more informative approximation to the fixpoint to be found. Unfortunately, whilst a narrowing operation can be defined as the meet over a lattice, there is no uniform method for defining a widening operator [34, Section 4]. The approximation method which we define over structured contexts, and is described in Sections 2.6.6 and 2.14.1, is an example of a widening operator.

Forwards analysis has a more serious defect with regard to the analysis of type theoretic programs in that it cannot gain information about components of data structures satisfactorily. For instance, it has been shown by Kamin [87] that head-strictness cannot be captured by the (usual) abstraction methods of forwards analysis.

2.3.2 Backwards analysis

As noted in [72], backwards analysis is, in fact, a particular type of forwards analysis where the abstract values are themselves functions upon contexts rather than being simply contexts themselves. (We shall, however, following the work of Hughes and others [68, 69, 72, 147], use the term *context* for the abstract values in our analysis.) The abstract values of backwards analysis may be seen as sets of continuations [68] whereby an abstract value such as "unused" will mean that the expression will definitely not be evaluated in the future computation of the program.

The backwards analysis of Hughes [68, 69, 72], however, has linear complexity in relation to the number of arguments. This is due to the fact that only a single input value (which represents a property of the *result* of the function) is required by each abstract function in the first-order case. Davis and Wadler have shown [39], however, that the complexity of the analyses is not due to the direction of the analyses but whether relationships between the variables of a function may be considered (what they term a *high-fidelity* analysis) or not. We may, for example, capture a property such as *joint redundancy* where if one parameter is needed for a particular application of a function then another will be unused in that application. For instance, we may form a function, *cond*, from the *if-then-else* construct:

$$cond\,a\,b\,c \;=\; if\,a\;then\,b\;else\,c$$

Here it may be seen that if parameter b is used then c will not be used and vice versa. Such information can be used to generate more efficient object code in that closures can be reduced more rapidly: as soon as any reduction of one parameter, p, say, takes place then information contained within the closure on other parameters which are jointly redundant with p can be discarded. An example of a forwards analysis which detects *joint strictness* (i.e. if one parameter is not necessarily evaluated then the other must be) is given for the *lesseq* function in Appendix A.2.1. A high-fidelity backwards analysis will have the same order of complexity as the forwards analyses that we have described above. The backwards analysis that we shall develop is a low-fidelity one in that we do not consider the possible properties of the parameters in conjunction. However, for higher-order functions, we are forced to make the analysis partly high-fidelity [69] in order to gain non-trivial information.

With regard to data structures, the work of Hughes and Launchbury [74] shows that backwards analysis is either better than (in the case of products) or incomparable with (in the case of sums) the corresponding forwards analysis in the first-order case.

Much of the work in the area of backwards analysis has been the modelling of contexts by Scott projections [147, 70]. Dybjer showed [45] that it is also possible to perform backwards analysis by using the inverse images of Scott-open sets (*cf* the model of abstract values in forwards analysis as Scott-closed sets). In this work, however, we shall take contexts to be sets of continuations.

There have been a number of works which relate forwards and backwards analysis. Apart from the works of Hughes and Launchbury [74] and Davis and Wadler [39] already mentioned, Burn has compared the power of forwards analysis with projection-based backwards analysis in the respect of strictness information about lists [19]. Hughes and Launchbury show in a later work that analyses may be reversed [75], although some loss of precision may occur if the reversals are not fully relational. Hunt developed the idea of analyses based on partial equivalence relations[2] [77, 78] which subsumes both projection-based analysis and forwards analysis based upon Scott-closed sets.

[2] A partial equivalence relation is a symmetric and transitive relation

2.4 Context lattices

2.4.1 Introduction to context lattices

The basic idea is that we are presented with some closed expression as (part of) a program in type theory which has to be evaluated to a normal form. In our analysis, we assume that this evaluation occurs in some sort of *context*: this context, in a sense, represents partial information about the evaluation, such as whether the result will be required a multiple number of times, whether it has to be stored on the heap or indeed whether it is necessary to evaluate the expression at all. To express this more formally, contexts represent sets of possible continuations of the program — for example, continuations where a certain sub-expression will definitely not be evaluated.

The information we obtain from a backwards analysis thus depends upon the *context lattice* which we use: the more distinct contexts we have, the more information we obtain about a program. Naturally, however, this will also mean that any analysis we do may be more complex. We only use basic lattice theory here, enough to give some flesh to the analysis which we are attempting. The reader is referred to "An Introduction to Abstract Interpretation" in [2] which covers relevant information about domain theory and the finding of fixed points, and to [37], which covers the necessary lattice theory. In particular, we shall use the Knaster-Tarski theorem which allows the least fixpoint of a recursive functional over a complete lattice to be found.

2.4.2 Lattice theory

Definition 1

A **poset** (partially ordered set) is a set, P, with an ordering relation, $\sqsubseteq$ ("less than or equal to") so that the following three properties hold:

1. *Reflexivity*: $a \in P$ implies that $a \sqsubseteq a$.
2. *Antisymmetry*: $a, b \in P$ implies that if $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$.
3. *Transitivity*: $a, b, c \in P$ implies that if $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$.

Definition 2

A **lattice** is a poset, L, where each pair of elements (as a set) has a least upper bound and a greatest lower bound. We call the least upper bound the *join* (denoted $\vee$ or $\sqcup$) and the greatest lower bound the *meet* (denoted $\wedge$ or $\sqcap$). More generally, the *join* and *meet* may range over any finite, non-empty subset of the underlying set.

Definition 3

A **complete lattice** is a lattice where *every* subset of the lattice has a least upper bound. (It can be shown that it follows that *every* subset will have a greatest lower bound as well.)

Result 1

Every complete lattice is bounded i.e. it has bottom and top elements.

Proof

See pp.27–29 of [37].

□

Result 2

Every lattice with a finite number of elements is complete.

Proof

See p.32 of [37].

□

Result 3

If A and B are complete lattices then their product, $A \times B$, and their linear sum, $A \oplus B$ are also complete lattices.

Proof

See p.33 of [37].

□

Definition 4

A map f between lattices L and M is **order-preserving** (or **monotone**) *iff* whenever $x \sqsubseteq y$,

$$f\ x \sqsubseteq f\ y$$

Definition 5 (Ascending Kleene chain)

The **ascending Kleene chain** for an order-preserving map $f : L \to L$ on a (complete) lattice L is the sequence

$$f^n\,(\bot)$$

where we use $\bot$ here to mean the bottom element of L. Note that

$$f^0\,(\bot) = \bot$$

and

$$f^{n+1}(\bot) = f(f^n(\bot))$$

Definition 6

x is termed a **fixpoint** of a map $f : L \to L$ on a lattice L if $f\ x = x$. The least such element of L, if it exists, is termed the **least fixpoint** and is denoted $\mu(f)$.

Result 4 (Knaster-Tarski theorem)

If L is a complete lattice and $f : L \to L$ is an order-preserving map then the least fixpoint of f, $\mu(f)$ exists. Furthermore, if L is finite then

$$\mu(f) = \bigsqcup_{n \geq 0} f^n(\bot)$$

That is, the least fixpoint is the least upper bound of the ascending Kleene chain formed for f.

Proof

The existence of the least fixpoint is shown on pp. 93–94 of [37]. Also, if the lattice is finite then the ascending Kleene chain must be finite and so, if

$$\nu = \bigsqcup_{n \geq 0} f^n(\bot)$$

then necessarily, as L is complete, ν is in L. Furthermore, there exists i such that $\nu = f^i(\bot)$ and for any $j \geq i$, $f^j(\bot) = \nu$. Hence, ν is a fixpoint. If ξ is any other fixpoint of f then $f^n(\xi) = \xi$. However, as $\bot \sqsubseteq \xi$ and f is order-preserving, $f^n(\bot) \sqsubseteq \xi$. Hence $\nu \sqsubseteq \xi$ and thus $\nu = \mu(f)$.

□

Definition 7 (Context lattice)

A **context lattice** is a finite lattice with a distinguished element **ABSENT**, and an operation, *contand* (&), which is an associative operation that is monotonic with respect to each of its arguments and for which **ABSENT** is the identity and the least element of the lattice, **CONTRA** is a zero. We also denote the context join as *contor* ($\sqcup$) for which **CONTRA** is, of course, the identity.

2.4.3 ABSENT and CONTRA

We shall first discuss the two contexts which must be present within any context lattice. They may not, however, be necessarily distinct from other contexts in the lattice. (Indeed, we shall see that in the neededness analysis lattice they are the same point.)

ABSENT is the context which results when a variable x is *computationally redundant* with respect to E. This may be due to the following possibilities:

1. x does not occur free in the expression E. For example, **ABSENT** is the relevant context with respect to x for the expression $y + 2$.

2. x only occurs in E as (part of) an applicand to a function which does not use that parameter. For example, suppose that the function *const* takes two parameters and simply returns the second as its result. Suppose then that E is the expression,
$$const\ x\ 3$$
ABSENT again pertains to x as it is computationally redundant here due to *const* not using its first parameter.

3. E has no computational content and is itself of a computationally redundant form. An example of this is an *abort* expression in TT which is used to ensure complete presentation in a strongly normalizing system: pathological proof objects of the empty type $\perp$ are eliminated using *abort* expressions. For example, for the *hd* of list function in TT we may have:
$$\begin{array}{rcl} hd & : & (\forall l : [A]).((nonempty\ l) \Rightarrow A) \\ hd\,[\,]\,p & \equiv_{df} & abort_A\,p \\ hd\,(a :: x)\,p & \equiv_{df} & a \end{array} \tag{1}$$
In (1), p must be a proof that the empty list is not empty and therefore is an impossible proof of type $\perp$. $abort_A\,p$ is a normal form of type A. It is *nonsensical* to evaluate p further: in this case p represents an error in proof derivation and its actual form is semantically irrelevant. It is sufficient to know that $p : \perp$ whilst it is axiomatic to type theory that $\perp$ is not inhabited.

As demonstrated above, the idea of the **ABSENT** context is vital to our development of a system which automatically detects computational redundancy in expressions of TT. In the neededness analysis lattice (Section 2.5.1) **ABSENT** and **CONTRA** both correspond to the context **U** representing the fact that a parameter is *unused* during a computation.

CONTRA represents the most precise context information we can assert about an object via a context lattice. It always corresponds to the bottom element of the context lattice. Its name comes from the fact that it represents CONTRAdictory information in the sharing analysis lattice (Section 2.5.4). In that lattice, if a variable has context **CONTRA** then it indicates that the parameter must both be used and unused by the computation.

We shall see that **ABSENT** is the identity for the context operator $\&$ and that **CONTRA** is the identity for the context operator $\sqcup$. We shall abbreviate **ABSENT** by **AB** and **CONTRA** by **CR**.

2.4.4 The contand and contor operations

There are two primitive operations upon each context lattice, **contand** ($\&$) and **contor** ($\sqcup$). **Contand** has to be defined according to the abstract semantics of each analysis. It should represent the idea of combining the properties of two contexts, in an analogous way to a logical-AND operation. For example, the result of applying **contand** to a context **U** denoting the fact that a parameter is definitely unused and a context **N** indicating that the parameter *may* be used should be **N**: this captures the idea that if a parameter is required by one or more sub-expressions then it is required for the computation as a whole. **Contand** is used to combine contexts resulting from different actual parameters to a function application. This is discussed in Section 2.8.3 on context propagation with respect to function applications. There are the following restrictions upon the definition of the & operator:

1. The **ABSENT** context (Section 2.4.3) must be an identity for &. This reflects the idea that if a parameter is not computationally needed by one sub-expression then the context for the parameter will only depend on other sub-expressions.

2. & should be associative and commutative so that

$$\mathbf{a} \,\&\, (\mathbf{b} \,\&\, \mathbf{c}) = (\mathbf{a} \,\&\, \mathbf{b}) \,\&\, \mathbf{c}$$

   and

$$\mathbf{a} \,\&\, \mathbf{b} = \mathbf{b} \,\&\, \mathbf{a}$$

   Associativity and commutativity guarantee the fact that the deduction of abstract properties can be computed in an order-independent way; if a parameter is needed in an application, for instance, then it is irrelevant whether that results from the first or last applicand.

3. & should be monotonic so that

$$\mathbf{a} \,\&\, \mathbf{b} \sqsubseteq \mathbf{a} \,\&\, \mathbf{c}$$

   whenever $\mathbf{b} \sqsubseteq \mathbf{c}$. This stipulation means that the combination of properties must preserve the *information ordering*. This is what we would demand intuitively as contexts higher up the lattice reflect less precise information than those lower down.

The companion operation to contand, **contor** ($\sqcup$), should, unlike contand, always be identical to the join operation ($\vee$) on the context lattice. Consequently, $\sqcup$ is associative, commutative and monotonic, and it has **CONTRA** has its identity. As the name implies this is somewhat similar to a disjunction of contexts. In sharing analysis, Section 2.5.4, it does indeed correspond to set union. We use $\sqcup$ to denote *uncertainty* in, for example, pattern-matching clauses, guarded expressions and *if-then-else* statements and, more generally, for computation rules which are defined by more than one clause. This reflects the fact that we do not know in advance which branch of a conditional expression will be evaluated. Section 2.8.2 shows how **contor** is used to define context propagation with respect to expressions which branch on different possibilities.

2.4.5 The strict operator

We also need a unary operator that can remove absence from a context so that the context which pertains to a parameter in the case that it *is* used in a computation will be produced. This will be convenient for the calculation of context functions, as shown in Section 2.7.2.

The operator to do this we call **strict**.

Definition 8

A context $\mathbf{c}'$ is equal to $\mathbf{strict\ c}$ *iff*

$$\mathbf{c} \sqcup \mathbf{ABSENT} = \mathbf{c}' \sqcup \mathbf{ABSENT}$$

and, if $\mathbf{c}''$ is any other context so that

$$\mathbf{c} \sqcup \mathbf{ABSENT} = \mathbf{c}'' \sqcup \mathbf{ABSENT}$$

then $\mathbf{c}' \sqsubseteq \mathbf{c}''$ where $\sqsubseteq$ partially orders the lattice of contexts.

For example, in the sharing analysis lattice, which is discussed in Section 2.5.4 and where $\sqcup$ corresponds to set union, this operation corresponds to subtracting the set $\{0\}$, the **ABSENT** context, from a context such as $\{0, 1\}$. In this case, $\{1\}$ will be the result of $\mathbf{strict}\ \{0, 1\}$. For the neededness analysis lattice (Section 2.5.1) **strict** is simply the identity function as $\mathbf{AB} = \mathbf{CR}$.

This operation is used to ensure the correct calculation of context functions (Section 2.7.2) over structured data (Section 2.6).

Finally, we show that **strict** is the identity over contexts not containing **ABSENT**, that it is a monotonic operator over contexts and that it preserves joins in contexts.

Result 5

If $\mathbf{ABSENT} \not\sqsubseteq \mathbf{c}$ then

$$\mathbf{strict\,c} = \mathbf{c}$$

Proof

Suppose that $\mathbf{ABSENT} \not\sqsubseteq \mathbf{c}$ but we have

$$\mathbf{strict\,c} = \mathbf{c}'$$

where $\mathbf{c} \neq \mathbf{c}'$. Then, by the definition, $\mathbf{c}$ is always an upper bound for the candidates for $\mathbf{strict\,c}$ and so

$$\mathbf{c}' \sqsubset \mathbf{c}$$

However, then it must follow that, as $\mathbf{ABSENT} \not\sqsubseteq \mathbf{c}$

$$\mathbf{c}' \sqcup \mathbf{ABSENT} \sqsubset \mathbf{c} \sqcup \mathbf{ABSENT}$$

which means that

$$\textbf{strict}\,\mathbf{c} \neq \mathbf{c}'$$

which contradicts our original assumption. Hence,

$$\textbf{strict}\,\mathbf{c} = \mathbf{c}$$

as required.

□

Result 6

The **strict** operator is order-preserving.

Proof

Suppose that we have two contexts, **a** and **b** such that

$$\mathbf{a} \sqsubseteq \mathbf{b}$$

Suppose we have,

$$\begin{aligned} \textbf{strict}\,\mathbf{a} &= \mathbf{a}' \\ \textbf{strict}\,\mathbf{b} &= \mathbf{b}' \end{aligned}$$

Then, as, from the definition of **strict**,

$$\textbf{strict}\,\mathbf{x} \sqsubseteq \mathbf{x}$$

it follows that

$$\mathbf{a}' \sqsubseteq \mathbf{a} \sqsubseteq \mathbf{b} \sqsubseteq \mathbf{b}'$$

This gives the result.

□

Similarly, we can prove that:

Result 7

The **strict** operator is uncertainty-preserving. That is,

$$\textbf{strict}\,(\mathbf{a} \sqcup \mathbf{b}) = (\textbf{strict}\,\mathbf{a}) \sqcup (\textbf{strict}\,\mathbf{b})$$

2.5 Lattices for the analysis of *TT*

In this section we develop context lattices which may be used for the abstract interpretation of programs in *TT*. We shall start by developing the neededness analysis lattice (Section 2.5.1), which allows the property of *computational redundancy* to be captured. We can, however, deduce other properties about programs in type theory such as strictness (Section 2.5.2). This allows classes of optimisations which have previously been applied to functional languages such as Haskell to be made to *TT* programs. We show how different properties may be combined into a single context lattice in Sections 2.5.3 and 2.5.4.

2.5.1 The neededness analysis lattice

Neededness analysis consists of a two-point context lattice as its basic abstract domain. This lattice allows distinctions to be made between those parameters which *might* be required by a computation and those which *definitely* will not be. It is the latter property which is essential to our study of computational redundancy in TT. We seek to determine which parameters are definitely computationally redundant (with respect to lazy evaluation) and those which may not be. For example, for the simple function *const*, which is defined as:

$$const\ x\ y \equiv_{df}\ x$$

we can readily see that the parameter y is *unused* by the computation, whilst x is always *needed* whenever the result of *const* is required by a computation. We assign the context **U** to denote the property of a parameter, such as y, being unused by the computation. Here, **ABSENT** corresponds to the abstract value **U**.

There is no decision procedure to show exactly which parameters are required by a computation and those which will definitely be unused; this undecidability property is proven in Result 15 on page 93. Consequently, the other point, **N**, in this lattice is less precise in terms of its informative content. It denotes the property that a parameter may or may not be used by a computation. In the above example, the context pertaining to the parameter x is **N**. Below is an example of a function where, in order to provide an abstraction that is sound (i.e. so that we will not determine a parameter as being unused when in fact it may be needed), we have to assign the **N** context as the abstract value of a parameter when in fact it may be unused:

$$condfn\ b\ x\ y \equiv_{df}\ if\ b\ then\ (x+1)\ else\ y$$

Here, b *must* be used in evaluating *condfn* and so its context is **N**. However, it is not necessarily the case that either x or y will definitely be used at all (although one of the two must be if *condfn* is called). For instance, *condfn* might only ever be called where b reduces to *False*. Consequently, y would in such a program be used but the parameter x would be unused. However, since both x and y *might* be used (we assume that addition always uses its arguments) they must each be assigned the abstract value **N**.

We order the two values by $\mathbf{U} \sqsubset \mathbf{N}$. Thus more precise information is ordered below the less precise. This reflects the idea of contexts as sets of possible continuations. **U** denotes all continuations (under a lazy evaluation strategy) where a parameter is not used. However, the context **N** denotes every continuation, both those where the parameter is unused and those where the parameter is evaluated. Consequently, **U** is a subset of **N** and hence the ordering that we have presented.

Since **CR** and **AB** are equal, the & and $\sqcup$ operations are consequently identical on this lattice. The **strict** operation is simply the identity function over contexts.

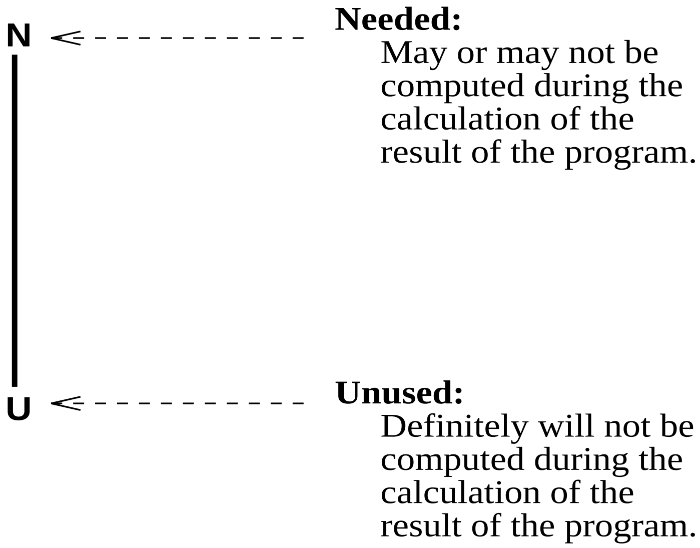

Figure 1: The neededness analysis lattice.

Once we detect a parameter has having the context **U**, we can then remove it from the object code produced at compile time. The neededness lattice is shown as a Hasse diagram in Figure 1.

2.5.2 The strictness analysis lattice

The most common analysis used to improve the efficiency of lazy functional programs is **strictness analysis**. Here we attempt to find which sub-expressions of a program may be evaluated and passed as values to functions rather than as names. This enables us to make the following gains in efficiency:

1. We avoid the creation and updating of an as-yet-unevaluated closure.
2. We may use a single piece of storage for an evaluated object rather than a pointer to a place in heap storage.
3. We eliminate extra evaluations of the same piece of code.

Here we wish to determine which parameters *must* be evaluated during a computation i.e. those in which a function is *strict*. If we determine that a parameter has the context **S** then the function may be evaluated by call-by-value on that parameter. If we return to our example,

$$condfn\ b\ x\ y \equiv_{df}\ if\ b\ then\ (x+1)\ else\ y$$

then the context pertaining to b should be **S** since b must be evaluated during the evaluation of *condfn*. However, both x and y should each have the context **L** to denote the fact that the function is "lazy" in those two arguments and that they

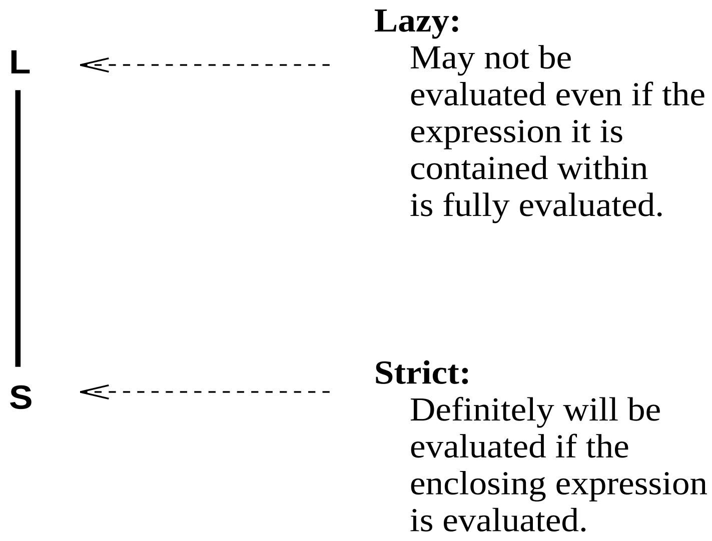

Figure 2: The strictness analysis lattice.

should be passed as names rather than values. In both cases x and y might not be used and so, in the interests of having a sound abstract interpretation, we assign the (less informative) context **L** to them. The lattice ordering is $\mathbf{S} \sqsubset \mathbf{L}$. This again reflects the fact that the more informative context **S** represents a subset of the continuations denoted by **L**. Also, as for neededness, strictness is an undecidable property over programs in TT.

In this analysis, & corresponds to the lattice meet whilst **ABSENT** is equivalent to **L**. (If a parameter is **ABSENT** then we obviously cannot say that it is definitely used.) The **strict** operator is equivalent to the constant function returning **S** i.e.

$$\mathbf{strict\ c} = \mathbf{S}$$

The Hasse diagram of the strictness lattice, first proposed in [81], is shown in Figure 2.

2.5.3 The strictness and absence analysis lattice

We may produce a lattice which encapsulates both the strictness and neededness properties. This extended analysis is called **strictness and absence analysis**.

To do this we construct the Cartesian closed product of the two sets of contexts, as shown in Table 1. Each possible pair reflects a unique possibility in that no redundant information is provided as a result of the product formation. If a parameter is both unused, **U**, and strict **S** then contradictory information has been found i.e. an error has occurred in the analysis. In the lattice that we are constructing, we give this the value **C** or **CONTRA**, representing the empty set of continuations. If the contexts **U** and **L** resulted then we can say that the parameter is definitely absent (i.e. computationally redundant), which we denote

Context pair	*Strictness and absence*
(**N**,**L**)	**L**
(**N**,**S**)	**S**
(**U**,**L**)	**A**
(**U**,**S**)	**C**

Table 1: Product of the neededness and strictness contexts.

by **A** or **ABSENT**. The other two possibilities are given the same labels as the strictness lattice values, namely **S** and **L**. The lattice ordering may be defined from the product construction i.e.

$$(\mathbf{a},\mathbf{b}) \sqsubseteq (\mathbf{c},\mathbf{d}) \textit{ iff } \mathbf{a} \sqsubseteq_{\mathbf{Nd}} \mathbf{c} \textit{ and } \mathbf{b} \sqsubseteq_{\mathbf{St}} \mathbf{d}$$

where $\sqsubseteq_{\mathbf{Nd}}$ and $\sqsubseteq_{\mathbf{St}}$ denote the orderings on the neededness and strictness lattices, respectively. Thus, for example, $\mathbf{A} \sqsubset \mathbf{L}$ but **A** and **S** are incomparable.

The context **C** ensures that we preserve a lattice structure, whilst maintaining the natural structure of the properties encoded. (A three-point chain would not be particularly natural since we would want the join of **A** and **S** to be **L**.) The four-point strictness-and-absence lattice is shown in Figure 3. Here & has the definition given in Table 2, which is reproduced from [69]. Note that **C** acts like a multiplicative zero, whilst **A** is an identity. The strictness and absence lattice is an example of what Hughes terms in [69] a *concrete context domain* where the **ABSENT** and **CONTRA** elements are distinct from the other elements in the lattice (**L** and **S**). The **strict** operation is equal to **S** for the contexts **S** and **L** and to **C** for **A** and **C**.

The contexts of this lattice may be illustrated in the following example.

$$condstra\; b\; x\; y \equiv_{df}\; \textit{if } b \textit{ then } (x+1) \textit{ else } 3$$

Here the context **S** pertains to b, **L** to x and **A** to y (since it is not used at all in the defining expression).

The strictness analysis lattice is what Hughes calls an *abstract context domain* (since we can perform an abstract interpretation projection from the concrete domain to the abstract one) as **ABSENT** is identified with **L** and **CONTRA** is identified with **S**. The neededness lattice is also a projection of the four-point domain, with **C** and **A** being mapped to **U**, and **S** and **L** being mapped to **N**.

2.5.4 The sharing analysis lattice

When implementing type theory we would like a mechanism which detects:

- Expressions that do not actually need to be evaluated during the computation (i.e. "*absent*" objects).

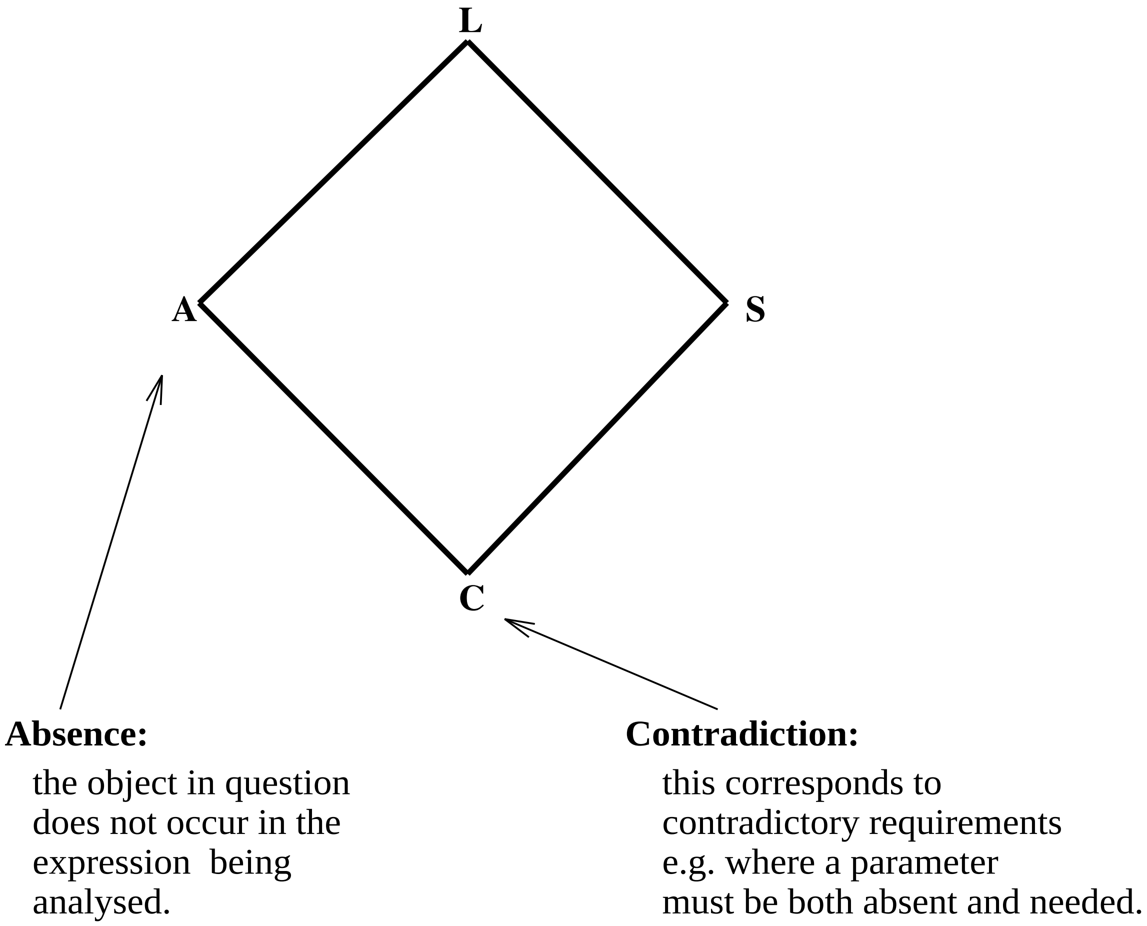

Figure 3: The strictness and absence analysis lattice.

&	C	A	S	L
C	C	C	C	C
A	C	A	S	L
S	C	S	S	S
L	C	L	S	L

Table 2: Definition of & for strictness and absence analysis.

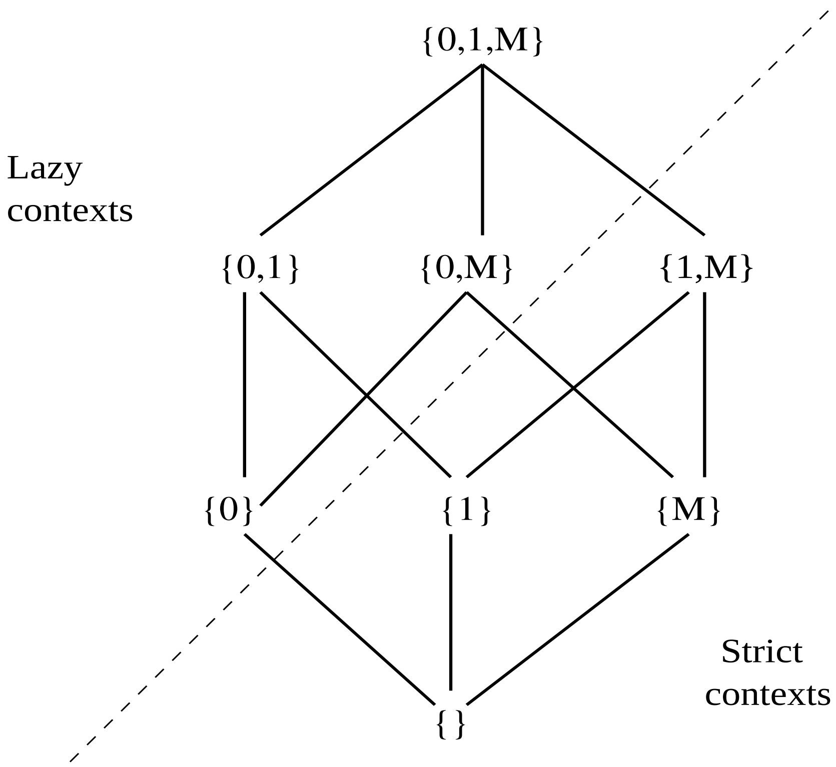

Figure 4: The sharing analysis lattice.

- Expressions that must be evaluated (so that we gain efficiency in places where the value of an expression may be stored rather than its code).
- Expressions to be shared (so that we may optimise call-by-need to call-by-name in the cases where an expression is used only once).

Sharing analysis is a form of abstract interpretation which finds such information: it subsumes strictness and absence analysis whilst also telling us which objects may be shared by different parts of the evaluation. (Details of an actual implementation of this method are given in [48].)

The sharing analysis lattice of context values consists of the power set of $\{0, 1, M\}$ with the join and meet operations on this lattice being set union and intersection, respectively. This lattice is shown in Figure 4. $0, 1, M$ are called **usage values**. They refer to how often a parameter is used in a computation:

- 0 means that the parameter is not used.
- 1 means that the parameter is used exactly once.
- M means that the parameter is used more than once.

Sets of these values denote *uncertainty*: $\{0, 1\}$, for instance, indicates that the object in question may not be used or may be used just once. Note that this

Context pair	*Full sharing equivalents*
(**L**,**M**)	{0,1,M}
(**L**,**1**)	{1}
(**S**,**M**)	{1,M}
(**S**,**1**)	{1}
(**A**,**M**)	{0}
(**A**,**1**)	{}
(**C**,**M**)	{}
(**C**,**1**)	{}

Table 3: Product of the strictness-and-absence and simple sharing contexts.

uncertainty dictates that we use a power set construction rather than, for example, a chain involving the abstract values **0**, **1** and **M**. If, say, we abstracted both the contexts $\{0, 1\}$ and $\{1\}$ by the value **1**, then our analysis would either be unsound or lacking in precision with respect to the properties which we are abstracting. If we deduced that the value **1** was propagated to a parameter of a function then it would be unsound to conclude that the function must be strict in that parameter; **1** might represent $\{0, 1\}$. Conversely, we thus must lose precision in the case when **1** actually represents $\{1\}$.

However, we can detect an **optimisation** of the sharing analysis lattice if we take the following approach to its construction. Firstly, we should note that in detecting possible call-by-need to call-by-name optimisations (i.e. closure simplification) we only have to distinguish between the cases where a parameter is used exactly once (which we shall denote by the context **1**) and those where a parameter is used some indeterminate number of times (**M**). Here, $\mathbf{1} \sqsubset \mathbf{M}$. Now, we can take the product of the strictness-and-absence lattice and our simple two-point sharing detection lattice. The resulting pairs and their equivalents in the full sharing lattice can be seen in Table 3. Note that there are two redundant pairs in the table in that three products are equivalent to the empty set (i.e. are contradictory). However, no equivalents to either $\{M\}$ or $\{0, M\}$ result. This is because the context **M** in the simple two point sharing analysis denotes *any* number of usages, including zero, one and multiple usage. Consequently, from this approach we deduce that neither $\{M\}$ nor $\{0, M\}$ are necessary in our sharing analysis and may be identified with $\{1, M\}$ and $\{0, 1, M\}$, respectively. This follows the intuition that in our analysis and optimisation we are not interested in deducing that a parameter must be used more than once — we only need to know when a parameter is used *exactly* once. Finally, we note that we could derive a four-point sharing-and-absence lattice from the neededness and two-point sharing lattices in a similar way to that of the strictness-and-absence lattice.

The following illustrates the contexts that we would expect to derive.

$$\mathit{condshar}\ b\,x\,y \equiv_{df}\ \mathit{if}\ b\ \mathit{then}\ (x+1)\ \mathit{else}\ (x+y)$$

Here, the context of b should be $\{1\}$ as it is used precisely once. The context of x should be $\{1, M\}$ since x is used more than once and the context of y should be $\{0, 1\}$ since it may be used once only or not at all.

Since the existence of different usage values within a set denotes uncertainty about possible continuations, we identify $\sqcup$ with set union, which is the join ($\vee$) on the lattice. $\{0\}$ is the **ABSENT** context and it follows from Definition 8 that on the sharing analysis lattice:

$$\textbf{strict c} = \mathbf{c} - \{0\}$$

& is more involved as it does not correspond to the meet or the join on this lattice.

Description of the & operator in sharing analysis

The contand operator (&) combines two contexts in a manner similar to that of logical "and": we wish to produce a resulting context which is equivalent to the meaning of both operands being true. For example, if one context tells us that a parameter must be used just once (i.e. the context $\{1\}$) whilst another tells us that the same object may not be used or may be used a multiple of times (i.e. {0,1,M}) then if both these contexts are true then the resulting context should tell us that the object in question is used once or a multiple of times. We are thus, in a sense, *adding* usage values to reflect this combination of contexts. (Here it is useful to remember that contexts are sets of possible program continuations.)

We thus arrive at the following definition of & in sharing analysis:

Definition 9

The *contand* (&) context operator is defined in sharing analysis as follows:

$$\mathbf{c}\,\&\,\mathbf{d} = \{a{+}b \mid a \in \mathbf{c}, b \in \mathbf{d}\}$$

(a and b are usage values.)
"+" is defined so that we form a simple, commutative monoid of usage values, with 0 as the obvious identity.

$$\begin{aligned} 0{+}1 &= 1 \\ 0{+}M &= M \\ 1{+}1 &= M \\ 1{+}M &= M \\ M{+}M &= M \end{aligned}$$

2.5.5 Properties of the context operations

Unfortunately, $\&$ and $\sqcup$ are not related in general, even though they correspond to greatest lower bound and least upper bound, respectively, in simple strictness analysis. In sharing analysis, for example, we may easily refute the notion that $\sqcup$ is distributive over $\&$:

$$\begin{aligned} \{1\} \sqcup (\{1\} \,\&\, \{0\}) &= \{1\} \sqcup \{1\} \\ &= \{1\} \end{aligned}$$

but

$$\begin{aligned} (\{1\} \sqcup \{1\}) \,\&\, (\{1\} \sqcup \{0\}) &= \{1\} \,\&\, \{1\} \\ &= \{M\} \end{aligned}$$

(As discussed in Section 2.5.4 we can equate the latter result with $\{1, M\}$.) However, we do have one useful result, which follows directly from the definition of $\&$ and $\sqsubset$ (which is the subset relation) for sharing analysis.

Distributivity law

Result 8

In sharing analysis,

$$\mathbf{c} \,\&\, (\mathbf{d} \sqcup \mathbf{e}) = (\mathbf{c} \,\&\, \mathbf{d}) \sqcup (\mathbf{c} \,\&\, \mathbf{e})$$

Proof

The result holds for the trivial cases where one or more of the contexts is **CONTRA**.

Suppose, where none of the three contexts is **CONTRA**, that

$$a \in (\mathbf{c} \,\&\, (\mathbf{d} \sqcup \mathbf{e}))$$

This is so iff $\exists b \in \mathbf{c}$ and $\exists c \in (\mathbf{d} \sqcup \mathbf{e})$ such that $a = b + c$.
From the definition of $\sqcup$ we have:

$$c \in (\mathbf{d} \sqcup \mathbf{e}) \iff (c \in \mathbf{d}) \; or \, (c \in \mathbf{e})$$

and:
$c \in \mathbf{d}$ iff $a \in (\mathbf{c} \,\&\, \mathbf{d})$.
$c \in \mathbf{e}$ iff $a \in (\mathbf{c} \,\&\, \mathbf{e})$.
This is true, if and only if,

$$a \in (\mathbf{c} \,\&\, \mathbf{d}) \sqcup (\mathbf{c} \,\&\, \mathbf{e})$$

This, together with the definition of $\sqsubset$, gives the result.
□

2.6 Structured types and contexts

2.6.1 Introduction to structured contexts

So far we have dealt only with *atomic* contexts — contexts which refer only to a variable as an object without structure and without type. Often, however, we wish to find out information about the components of a term of structured type. For example, we may want to find out information about the head of a list or the second part of a pair. This becomes particularly relevant when we consider functions defined by pattern matching on the structure of the type of a parameter.

To do this we introduce *structured contexts*. We thus broaden the domain of contexts so that contexts reflect the structure of the type in the actual syntax. We thus have types of contexts in one-to-one correspondence with the types of the language. Each of these structured contexts have constructors akin to those in the syntax of type theory. However, since the flow of information is *backwards*, it should be noted that *constructors* in the type theory (such as ::) give rise to *selectors* in the abstract interpretation and visa versa.

These structured contexts have two parts:

1. An **atomic** part which gives the context for the object as a whole e.g. for an entire list.

2. A **structured part** which gives the contexts of the components of the object e.g. the head and tail of a list parameter. These contexts are "glued" together by **context constructors**.

We are thus able to build up a set of context types which correspond to the types of a language based on intuitionistic type theory. In general, for a structured context:

$$\mathbf{AB} = \mathbf{AB}'_{[\mathbf{CR}]}$$

where $\mathbf{AB}'$ means the context **ABSENT** for the type of the atomic contexts and $[\mathbf{CR}]$ means that all the subscripted contexts have the value **CONTRA**.

2.6.2 Examples of structured contexts

The following are examples of structured contexts:

- $$\{1\}_{(\{1\},\{0\})}$$

  Tuple structured context. The two subscripted contexts refer to each element of the pair. In this case, the first part of the pair will be used exactly once, but the second part of the pair will be unused.

- $$\{0,1\}_{[\,],\,\{1\}\,::\,\{0,1\}}$$

  List structured context: the head of the list will definitely be used if the list itself is. (We include the empty list context for completeness to show that the expression may evaluate to the empty list and hence that the contexts for the head and the tail of the list will not be relevant in that case.)

- $$\{1,M\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})}$$

  Natural number structured context: the subscripted contexts refers to the predecessor of the natural number variable. (Again, **0** indicates that the natural number might evaluate to 0.)

- $$\{0,1,M\}_{\mathbf{Null},\,(\mathbf{Node}\,\{1,M\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})}\,\{0,1\}\,\{1,M\})}$$

  Binary tree (with natural number data carried at the nodes) structured context. Here **Null** refers to the possibility of an empty tree. Note that a natural number context is carried by the tree context, as each node in the tree carries a natural. There are two contexts in the subscripted part which represent the contexts pertaining to each of the two subtrees.

Note that nullary context constructors such as **0** and [] are included in the above examples. Since these do not contain any extra contextual information (they will be present in all structured contexts of the relevant types) we shall often omit them for the sake of notational convenience.

Note that we thus have *typed* contexts where the subscripted parts indicate the contexts of the components of the different head normal forms that may occur for each type. A complete list of the context types which may occur in the backwards analysis of TT is given in the formalisation of the backwards analysis, Section 2.14.1.

2.6.3 The at and str functions

We often wish to refer to the atomic part of a structured context. (Indeed, due to pattern matching upon a structured object, the atomic part of a context is quite often unchanged during the backwards analysis of a function.) In order to do this we introduce the functions **at** and **str**.

Definition 10

$$\begin{aligned}\mathbf{at}(\mathbf{c}) &= \textit{The atomic part of } \mathbf{c}\\ \mathbf{str}(\mathbf{c}) &= \textit{The structured part of } \mathbf{c}\end{aligned}$$

This informal definition may be made more precise by stipulating that contexts are pairs, consisting of an atomic part and a structured part. The structured part will contain context constructors and contexts (which may themselves be structured).

2.6.4 Semantics of the structured parts

It is possible for an atomic part of a context to be greater than or equal to **ABSENT**. Of course, if the atomic part was actually unused by the computation then the subscripted contexts would be meaningless. Hence, the structured part of a context is taken to be meaningful for the *strict* part of the atomic context. That is, the part of the context which describes possible continuations where the data structure will be used by the computation.

For example, in

$$\mathbf{L}_{[], \mathbf{S} :: \mathbf{S}}$$

the subscripted contexts mean that both the head and the tail must be used *if* the list itself is used.

When combining contexts, however, we have to factor in the possibility that the atomic parts do indeed correspond to **ABSENT**. This is why the definition of & in Section 2.6.5 below is not simply pointwise, as is the case with $\sqcup$.

This semantics of structured contexts is the reason also for the definition of the calculation of context functions given in Section 2.7.2, which allows the strict part of input contexts to be propagated to the structured components.

2.6.5 Definition of & upon structured contexts

Suppose we have $\mathbf{c}$ and $\mathbf{c}'$ of the following form:

$$\begin{aligned} \mathbf{c} &= \mathbf{a}_{\mathbf{C}\,\mathbf{c}_1 \ldots \mathbf{c}_m} \\ \mathbf{c}' &= \mathbf{a}'_{\mathbf{C}\,\mathbf{c}'_1 \ldots \mathbf{c}'_m} \end{aligned}$$

(In other words $\mathbf{c}$ and $\mathbf{c}'$ are structured contexts with the structured part consisting of the combination of m contexts (which themselves might be structured). These subscripted contexts are combined using the context constructor, $\mathbf{C}$.)

We then, using & upon atomic contexts, define & upon such structured contexts as follows:

$$\mathbf{c} \,\&\, \mathbf{c}' = (\mathbf{at}(\mathbf{c}) \,\&\, \mathbf{at}(\mathbf{c}'))_{\mathbf{C}\,[\Gamma]}$$

where, Γ is the combination of contexts given below.

$$\Gamma = \begin{cases} (ly\, \mathbf{c}'\, \mathbf{c}_1) \sqcup (ly\, \mathbf{c}\, \mathbf{c}'_1) \sqcup (\mathbf{c}_1 \,\&\, \mathbf{c}'_1), \\ \vdots \\ (ly\, \mathbf{c}'\, \mathbf{c}_m) \sqcup (ly\, \mathbf{c}\, \mathbf{c}'_m) \sqcup (\mathbf{c}_m \,\&\, \mathbf{c}'_m) \end{cases}$$

where

$$ly\,\mathbf{e}\,\mathbf{d} = \begin{cases} \mathbf{d} & \text{, if } \mathbf{e} \sqsupseteq \mathbf{ABSENT} \\ \mathbf{CONTRA} & \text{, otherwise} \end{cases}$$

where **CONTRA** is the bottom of the context domain that includes **d**. (($ly\,\mathbf{e}\,\mathbf{d}$) means "the context **d** with respect to the laziness of the atomic context **e**".) Note that & is still monotonic after this adjustment, since for any given context **e**, if $\mathbf{a} \sqsubseteq \mathbf{b}$ then

$$ly\,\mathbf{e}\,\mathbf{a} \sqsubseteq ly\,\mathbf{e}\,\mathbf{b}$$

The results of & are approximated as in 2.6.6.

2.6.6 Recursive data structures and context approximation

A problem with backwards analysis arises when we consider recursive data structures in type theory. The difficulty occurs because such structures may be of arbitrary size: we do not know in advance, for example, how long an arbitrary list may be. Whilst such structures are not infinite in the sense that a list such as `[1..]` is in Haskell (in type theory we need co-inductions to obtain such streams), their arbitrary size gives rise to infinite contexts. For example, a list context has, as its structured part, contexts for both the head of the list and the tail of the list. The tail of list context is, itself, a list context which is therefore structured and has a tail-of-list context which is again a list context and so on. We may retain a finite lattice of contexts by assuming that all head contexts and all tail contexts are the same for a particular list i.e. a list context is assumed to be of the form:

$$\mathbf{c}_{\mathbf{d}\,::\,\mathbf{e}_{\mathbf{d}\,::\,\mathbf{e}_{\mathbf{d}\,::\,\mathbf{e}_{\ddots}}}}$$

The above is represented as simply:

$$\mathbf{c}_{\mathbf{d}\,::\,\mathbf{e}}$$

Nevertheless, we must remember that **e** *is* a list context and that its (implicit) subscripted contexts must be used during a computation of contexts. During a computation we often find that a resulting context is not of the correct form. The following example, in simple strictness analysis and taken from [72], shows what may occur:

$$\mathbf{S}_{\mathbf{S}\,::\,\mathbf{L}} \,\&\, \mathbf{S}_{\mathbf{L}\,::\,\mathbf{S}} \;=\; \mathbf{S}_{\mathbf{S}\,::\,\mathbf{L}_{\mathbf{S}\,::\,\mathbf{L}_{\ddots}}} \,\&\, \mathbf{S}_{\mathbf{L}\,::\,\mathbf{S}_{\mathbf{L}\,::\,\mathbf{S}_{\ddots}}}$$

$$
\begin{aligned}
&= (\mathbf{S} \,\&\, \mathbf{S})_{(\mathbf{S} \,\&\, \mathbf{L}) :: ((\mathbf{L}_{\mathbf{S} :: \mathbf{L}_{\ddots}}) \,\&\, (\mathbf{S}_{\mathbf{L} :: \mathbf{S}_{\ddots}}))} \\
&= \mathbf{S}_{\mathbf{S} :: ((\mathbf{L}_{\mathbf{S} :: \mathbf{L}_{\ddots}}) \,\&\, (\mathbf{S}_{\mathbf{L} :: \mathbf{S}_{\ddots}}))} \\
&= \mathbf{S}_{\mathbf{S} :: ((\mathbf{L} \,\&\, \mathbf{S})_{((\mathbf{S} \,\&\, \mathbf{L}) \sqcup \mathbf{L}) :: ((\mathbf{L}_{\mathbf{S} :: \mathbf{L}} \,\&\, \mathbf{S}_{\mathbf{L} :: \mathbf{S}}) \sqcup \mathbf{S}_{\mathbf{L} :: \mathbf{S}})})}
\end{aligned}
$$

It may be observed that we have a repetition in the calculation (a repetition which is guaranteed by the restriction we have placed upon the form of structured contexts) and that we have as a result a context of the form:

$$
\mathbf{c}_{\mathbf{d} :: \mathbf{e}_{\mathbf{d}' :: \mathbf{e}'_{\mathbf{d}' :: \mathbf{e}'_{\ddots}}}}
$$

In such a situation we *approximate* such a context in order to maintain the convention of having finite structured contexts of the form:

$$
\mathbf{c}_{\mathbf{d} :: \mathbf{e}}
$$

We achieve this by taking the join of the head contexts to produce a final head context and similarly with the tail contexts i.e. using the notation above we have, as an approximation:

$$
\mathbf{c}_{(\mathbf{d} \sqcup \mathbf{d}') :: (\mathbf{e} \sqcup \mathbf{e}')}
$$

To denote this process of approximation we use the symbol $\overset{\sqsubset}{\sim}$. Going back to our example we have:

$$
\begin{aligned}
& \mathbf{S}_{\mathbf{S} :: ((\mathbf{L} \,\&\, \mathbf{S})_{((\mathbf{S} \,\&\, \mathbf{L}) \sqcup \mathbf{L}) :: ((\mathbf{L}_{\mathbf{S} :: \mathbf{L}} \,\&\, \mathbf{S}_{\mathbf{L} :: \mathbf{S}}) \sqcup \mathbf{S}_{\mathbf{L} :: \mathbf{S}})})} \\
\overset{\sqsubset}{\sim}\ & \mathbf{S}_{\mathbf{S} \sqcup ((\mathbf{S} \,\&\, \mathbf{L}) \sqcup \mathbf{L}) :: \mathbf{S} \sqcup ((\mathbf{L} \,\&\, \mathbf{S}) \sqcup \mathbf{S})} \\
=\ & \mathbf{S}_{\mathbf{S} \sqcup (\mathbf{S} \sqcup \mathbf{L}) :: \mathbf{S} \sqcup (\mathbf{S} \sqcup \mathbf{S})} \\
=\ & \mathbf{S}_{\mathbf{L} :: \mathbf{S}}
\end{aligned}
$$

These approximations are *safe* in that if $\mathbf{e}$ is the context that would actually result from the combination of two (finite) contexts and $\mathbf{e}'$ is an approximation to $\mathbf{e}$ then:

$$
\mathbf{e} \sqsubseteq \mathbf{e}'
$$

where $\sqsubseteq$ partially orders the lattice of structured contexts that contains both $\mathbf{e}$ and $\mathbf{e}'$. In other words, our approximate result will remain valid with regard to the actual semantics of a language based upon type theory but potentially there will be a reduction in the precision of the information we obtain. The approximation operator, $\precsim$, is an example of a widening operation [33, 34] in that it serves to find a fixpoint more readily, even if in general it will not be the least one.

In the TT system, there are two other recursively defined data structures, natural numbers and binary trees where natural number data is contained at the nodes. With both these cases, the procedure is the same: we perform the calculation with the structured contexts expanded so that the recursive nature of the contexts is made explicit. We then perform the calculation which will result in a structured context which is not of the required form, in that the recursive parts do not repeat from the first subscripted level downwards. This is shown in the case of the following tree context:

$$\mathbf{c}_{\mathbf{Node\,n\,l}_{(\mathbf{Node\,n_1\,l_1\,r_1})}\,\mathbf{r}_{(\mathbf{Node\,n_2\,l_2\,r_2})}}$$

Such a context is then approximated by joining corresponding parts of the structured context, so that left subtrees are joined with left subtrees etc.

We can also factor the idea of approximation into the definition of the $\&$ operator. For instance, we can produce an expression for the combination of two natural number contexts as follows:

$$\begin{array}{rcl}
\mathbf{c}_{\mathbf{Succ\,c_1}} \,\&\, \mathbf{d}_{\mathbf{Succ\,d_1}} & = & \mathbf{c}_{\mathbf{Succ\,c'}_{\mathbf{Succ\,c'}_{\ddots}}} \,\&\, \mathbf{d}_{\mathbf{Succ\,d'}_{\mathbf{Succ\,d'}_{\ddots}}} \\
& = & \mathbf{c} \,\&\, \mathbf{d}_{\mathbf{Succ}\,((\mathbf{c'} \& \mathbf{d'}) \sqcup \mathbf{a}_1 \sqcup \mathbf{b}_1)_{\mathbf{Succ}((\mathbf{c'} \& \mathbf{d'}) \sqcup \mathbf{a}_2 \sqcup \mathbf{b}_2)_{\mathbf{Succ}((\mathbf{c'} \& \mathbf{d'}) \sqcup \mathbf{a}_2 \sqcup \mathbf{b}_2)_{\ddots}}}} \\
& \precsim & \mathbf{c} \,\&\, \mathbf{d}_{\mathbf{Succ}\,((\mathbf{c'} \& \mathbf{d'}) \sqcup \mathbf{a}_1 \sqcup \mathbf{b}_1 \sqcup \mathbf{a}_2 \sqcup \mathbf{b}_2)}
\end{array}$$

In the above,

$$\begin{array}{rcl}
\mathbf{a}_1 & = & ly\,\mathbf{d}\,\mathbf{c'} \\
\mathbf{b}_1 & = & ly\,\mathbf{c}\,\mathbf{d'} \\
\mathbf{a}_2 & = & ly\,\mathbf{d'}\,\mathbf{c'} \\
\mathbf{b}_2 & = & ly\,\mathbf{c'}\,\mathbf{d'}
\end{array}$$

The formal definitions for $\&$ and the approximation operator for each of these recursively defined types are given in Section 2.14.1.

2.7 Context functions

2.7.1 Introduction to context functions

Context functions are used to calculate the context which pertains to each parameter of every function defined for a program in TT. That is, there will be a one-to-one correspondence between function parameters of TT programs and context functions. If a function in TT has the name f then we shall denote its context functions by $\mathbf{f_1} \ldots \mathbf{f_n}$, assuming that f has n parameters.

We shall see how context functions are derived from expressions in TT in the section on context propagation, 2.8. A context function is defined in terms of some context expression $\mathbf{E}$, where $\mathbf{E}$ contains one context variable, $\mathbf{v}$ which is bound by the context function. The basic abstract syntactical structure of a context expression, ce (of the syntactic domain $\mathbf{Cexp}$), is

$$ce ::= \mathbf{AB} \mid \mathbf{v} \mid ce_1 \,\&\, ce_2 \mid ce_1 \sqcup ce_2 \mid \mathbf{g_i}\ ce_1$$

ce_1 and ce_2 are also context expressions. This structure is a simplification of the full syntax of context expressions which is given in Section 2.14.2. That takes into account the additional complication of higher-order functions, the analysis of which is described in Section 2.11.

Context functions are evaluated by substituting an input context for occurrences of $\mathbf{v}$ and applying the basic context operations. For example, suppose that the context function $\mathbf{f}$ is defined as follows:

$$\mathbf{f}\ \mathbf{v} \equiv_{df} \mathbf{v} \,\&\, (\mathbf{AB} \sqcup \mathbf{v})$$

Then, in the strictness-and-absence lattice,

$$\begin{aligned} \mathbf{f}\,\mathbf{S} &= \mathbf{S} \,\&\, (\mathbf{A} \sqcup \mathbf{S}) \\ &= \mathbf{S} \,\&\, \mathbf{L} \\ &= \mathbf{S} \end{aligned}$$

The above example evaluates context functions over atomic contexts so that here, a context function $\mathbf{g}$ will have the type $\mathbf{C} \to \mathbf{C}$ where $\mathbf{C}$ is the context lattice being used. With structured contexts, however, the types of the input and output may differ.

2.7.2 Additional constraints upon context functions

For the neededness, strictness-and-absence and sharing lattices we impose the following additional constraints upon context functions:

1. *Absence* i.e.

$$\mathbf{f\ ABSENT} = \mathbf{ABSENT}$$

   This must be preserved as otherwise we may deduce that an expression must be evaluated when it need not be.

2. *Contradiction* i.e.

$$\mathbf{f\ CONTRA = CONTRA}$$

   This must be preserved as otherwise we would be adding possible program continuations which were not possible in the original expression that we are analysing.

3. *Uncertainty* i.e.

$$\mathbf{f\ (c \sqcup d) = (f\ c) \sqcup (f\ d)}$$

   This is not a constraint as such, but simply follows from the fact that context functions are defined by the & and $\sqcup$ operations.

Properties 1 and 3 also apply to the strictness lattice.

The above properties were stipulated by Hughes in [69] for *concrete context domains*. Both the strictness-and-absence and sharing lattices are examples of concrete context domains, where the **ABSENT** and **CONTRA** elements are made distinct from the other elements of the lattice. It is reasonable to impose properties 1 and 2 on context functions over the neededness lattice since, if the result of a function is unused then it follows that any parameter of that function must also be unused. Furthermore, property 2 follows immediately from property 1.

Properties 1 and 3 can be proved for the context functions which we derive, as is shown in Section 2.7.3.

We also have to make an alteration to the method by which context functions are calculated, in order to be consistent with the semantics of structured contexts (see Section 2.6.4).

$$\mathbf{f_i\ c} = \begin{cases} \mathbf{E[c'/v] \sqcup ABSENT}, & \text{if } \mathbf{ABSENT \sqsubseteq c} \\ \mathbf{E[c/v]}, & \text{otherwise} \end{cases}$$

In the above, $\mathbf{c'} = \mathbf{strict\ c}$ (**strict** is defined in Section 2.4.5). If we did not do this then only lazy contexts (i.e. those ordered above **ABSENT**) would appear in the structured parts of the result, if the input context was lazy. However, we desire that the contexts of the structured parts should be predicated on the assumption that the whole structure is used.

2.7.3 Order-preservation by context functions

In this section we prove that the context functions and expressions that we use in the analysis of TT are order-preserving (monotonic). Also, we show that the constraints of absence and uncertainty upon context function which we presented in Section 2.7.2 can be proven for the context functions of the form that we have given in 2.7.1.

Result 9 (Monotonicity of context functions)

The context functions used in our backwards analysis are order-preserving if and only if their defining context expressions are order-preserving.

Proof

We assume that we have a context function $\mathbf{f}$ defined in the following manner:

$$\mathbf{f}\ \mathbf{v} \equiv_{df} \mathbf{E}(\mathbf{v})$$

We aim to show that if $\mathbf{a} \sqsubseteq \mathbf{b}$ then

$$\mathbf{f}\ \mathbf{a} \sqsubseteq \mathbf{f}\ \mathbf{b}.$$

iff

$$\mathbf{E}[\mathbf{a}/\mathbf{v}] \sqsubseteq \mathbf{E}[\mathbf{b}/\mathbf{v}]$$

Suppose that $\mathbf{a} \sqsubseteq \mathbf{b}$ and, let $\mathbf{a}' = \mathbf{strict\ a}$ and $\mathbf{b}' = \mathbf{strict\ b}$ Then there are the following cases to consider.

1.

$$\mathbf{ABSENT} \not\sqsubseteq \mathbf{a}, \mathbf{b}$$

Then

$$\begin{aligned} \mathbf{f\ a} &= \mathbf{E}[\mathbf{a}/\mathbf{v}] \\ \mathbf{f\ b} &= \mathbf{E}[\mathbf{b}/\mathbf{v}] \end{aligned}$$

Hence, the result follows.

2.

$$\mathbf{ABSENT} \sqsubseteq \mathbf{a}, \mathbf{b}$$

Then, by the monotonicity of **strict** (Result 6), we have

$$\mathbf{a}' \sqsubseteq \mathbf{b}'$$

Also,

$$\begin{aligned} \mathbf{f\ a} &= \mathbf{E}[\mathbf{a}'/\mathbf{v}] \sqcup \mathbf{AB} \\ \mathbf{f\ b} &= \mathbf{E}[\mathbf{b}'/\mathbf{v}] \sqcup \mathbf{AB} \end{aligned}$$

It follows from the monotonicity of $\sqcup$ that

$$\mathbf{f\ a} \sqsubseteq \mathbf{f\ b}$$

iff

$$\mathbf{E}[\mathbf{a}'/\mathbf{v}] \sqsubseteq \mathbf{E}[\mathbf{b}'/\mathbf{v}]$$

The latter expression is equivalent to monotonicity of context expressions over strict contexts.

3.

$$\mathbf{ABSENT} \sqsubseteq \mathbf{b}$$

but

$$\mathbf{ABSENT} \not\sqsubseteq \mathbf{a}$$

Now, from Results 5 and 6,

$$\mathbf{a} = \mathbf{strict\ a} \sqsubseteq \mathbf{b}'$$

We thus have that:

$$\begin{aligned} \mathbf{f\ a} &= \mathbf{E}[\mathbf{a}/\mathbf{v}] \\ \mathbf{f\ b} &= \mathbf{E}[\mathbf{b}'/\mathbf{v}] \sqcup \mathbf{AB} \end{aligned}$$

Hence, from the monotonicity of $\sqcup$

$$\mathbf{f\ a} \sqsubseteq \mathbf{f\ b}$$

iff

$$\mathbf{E}[\mathbf{a}/\mathbf{v}] \sqsubseteq \mathbf{E}[\mathbf{b}'/\mathbf{v}]$$

Thus the result has been proven for all possible cases.
□

Result 10

The context expressions that we form in our backwards analysis are order-preserving.

Proof

We give an outline of the proof. The crucial point is that context expressions are built around the & and $\sqcup$ operators, which must be monotonic (see Section 2.4.4).

The proof is by structural induction over a context expression $\mathbf{E}$.

Base case Here the expression $\mathbf{E}$ must either be a constant, $\mathbf{AB}$, or a variable $\mathbf{v}$. If it is the former then we have proved monotonicity as,

$$\mathbf{f\ a} = \mathbf{f\ b} = \mathbf{AB}$$

when $\mathbf{a} \sqsubseteq \mathbf{b}$. In the latter case then, if $\mathbf{a} \sqsubseteq \mathbf{b}$,

$$\mathbf{f\ a} = \mathbf{a} \sqsubseteq \mathbf{b} = \mathbf{f\ b}$$

Inductive cases Here we have the following cases:

1. We need to prove that when $\mathbf{E}$ has the form, $ce_1 \sqcup ce_2$ it preserves monotonicity, assuming that both ce_1 and ce_2 do. This is follows immediately from the monotonicity of the contor operator, $\sqcup$. That is, if $\mathbf{a} \sqsubseteq \mathbf{b}$, then
$$\begin{aligned} ce_1(\mathbf{a}) &\sqsubseteq ce_1(\mathbf{b}) \\ ce_2(\mathbf{a}) &\sqsubseteq ce_2(\mathbf{b}) \end{aligned}$$
where $ce(\mathbf{x})$ means that $\mathbf{x}$ is substituted for free occurrences of the context variable $\mathbf{v}$ in $\mathbf{E}$. It follows that
$$(ce_1(\mathbf{a}) \sqcup ce_2(\mathbf{a})) \sqsubseteq (ce_1(\mathbf{b}) \sqcup ce_2(\mathbf{b}))$$

2. If we assume that monotonicity is preserved by sub-expressions ce_1 and ce_2 then, as $\&$ must be monotonic, the result follows similarly to the above.

3. If $\mathbf{E}$ has the form,
$$\mathbf{g}\ ce_1$$
then we may assume that monotonicity is preserved by the context function $\mathbf{g}$ i.e. that $\mathbf{a} \sqsubseteq \mathbf{b}$ implies
$$\mathbf{g}\ \mathbf{a} \sqsubseteq \mathbf{g}\ \mathbf{b}$$
This assumption is valid since $\mathbf{g}$ is monotonic *iff* its defining context expression is i.e. our induction hypothesis actually concerns the defining expression of $\mathbf{g}$.

   We also assume that if $\mathbf{a} \sqsubseteq \mathbf{b}$ then
$$ce_1(\mathbf{a}) \sqsubseteq ce_1(\mathbf{b})$$
   Since $ce_1(\mathbf{a}) \sqsubseteq ce_1(\mathbf{b})$ it follows that
$$\mathbf{g}\ (ce_1(\mathbf{a})) \sqsubseteq \mathbf{g}\ (ce_1(\mathbf{b}))$$

Hence the monotonicity result follows for context expressions.
□

Results 4, 9 and 10 guarantee the fact that the recursive equations that we form with respect to a context lattice may be solved by finding the limit of the ascending Kleene chain.

Result 11 (Uncertainty preservation by context functions)
The context functions used in backwards analysis preserve uncertainty i.e. for any contexts $\mathbf{a}$ and $\mathbf{b}$ we have:
$$\mathbf{f}\ (\mathbf{a} \sqcup \mathbf{b}) = \mathbf{f}\ \mathbf{a} \sqcup \mathbf{f}\ \mathbf{b}$$

Proof

We will show by structural induction that the result holds for the evaluation of context expressions. From this the result will then follow for context functions due to the preservation of uncertainty by the **strict** operator (Result 7).

Base case Both possibilities are trivial. For example, where the context expression $\mathbf{E}$ is $\mathbf{v}$ we have:

$$\begin{aligned} \mathbf{E}[(\mathbf{a} \sqcup \mathbf{b})/\mathbf{v}] &= \mathbf{a} \sqcup \mathbf{b} \\ &= \mathbf{E}[\mathbf{a}/\mathbf{v}] \sqcup \mathbf{E}[\mathbf{b}/\mathbf{v}] \end{aligned}$$

Inductive cases We give the case for the **contor** operation. The other two cases are similar.

$$E \equiv ce_1 \sqcup ce_2$$

Now, we have from the induction hypothesis that:

$$\begin{aligned} ce_1(\mathbf{a} \sqcup \mathbf{b}) &= ce_1(\mathbf{a}) \sqcup ce_1(\mathbf{b}) \\ ce_2(\mathbf{a} \sqcup \mathbf{b}) &= ce_2(\mathbf{a}) \sqcup ce_2(\mathbf{b}) \end{aligned}$$

From the associativity of $\sqcup$ we have:

$$\begin{aligned} ce_1(\mathbf{a} \sqcup \mathbf{b}) \sqcup ce_2((\mathbf{a} \sqcup \mathbf{b}) &= (ce_1(\mathbf{a}) \sqcup ce_1(\mathbf{b})) \sqcup (ce_2(\mathbf{a}) \sqcup ce_2(\mathbf{b})) \\ &= (ce_1(\mathbf{a}) \sqcup ce_2(\mathbf{a})) \sqcup (ce_1(\mathbf{b}) \sqcup ce_2(\mathbf{b})) \end{aligned}$$

This illustrates the proof method.
□

Result 12 (Absence preservation by context functions)
The context functions used in backwards analysis preserve absence i.e.

$$\mathbf{f\ ABSENT} = \mathbf{ABSENT}$$

Proof

This may be proved by a similar inductive argument to that given for Result 11.
□

Finally, we show that the requirement to preserve contradiction for all of our context lattices, apart from the strictness one, does not affect the monotonicity of context functions.

Result 13

The preservation of contradiction property (number 2 in Section 2.7.2) preserves the monotonicity of context functions.

Proof

Suppose that $\mathbf{a} = \mathbf{CONTRA}$ Then, for any context $\mathbf{b}$,

$$\mathbf{a} \sqsubseteq \mathbf{b}$$

Now, by the preservation of contradiction property,

$$\mathbf{f\ a} = \mathbf{CONTRA} \sqsubseteq \mathbf{f\ b}$$

Hence, monotonicity is preserved.

$\square$

Thus we have shown that the extra constraints imposed on context functions in Section 2.7.2 do not affect the monotonicity of context functions.

2.8 Context propagation

In this section we describe how contexts are *propagated* with respect to most of the expressions of type theory. This process of context propagation allows context expressions and functions to be formed with respect to a context variable. Such expressions will be evaluated with respect to certain input contexts for the analyses in which we are interested. In Section 2.9 we shall give definitions for expressions in TT which *exhibit computationally redundancy*, in the terminology of [5].

2.8.1 Basic definitions

Suppose then that we have some closed expression E, a variable, x, which is a formal parameter of the function in which E occurs, and some *initial context*, $\mathbf{c}$. When unambiguous, we shall often refer to actual parameter sub-expressions by the formal parameters (e.g. x) to which they correspond. (In order to gain context information about sub-expressions we have to deduce the contexts of the parameters to which they will be assigned.) $\mathbf{c}$ is the context of the entire expression E. The information that we are attempting to gain about x is itself a context: we call this mapping *context propagation* and it is described by the following notation:

$$\mathbf{c} \xrightarrow{E} x$$

$\mathbf{c}$ is said to be *propagated through* E to x.

This reversal of the "flow" of a function is what gives backwards analysis its name: contexts, the inputs to the *analysis*, propagate from the expression being evaluated to what correspond to the inputs of the *actual program*.

We start our analysis with the following axiom:

Definition 11

For any variable, x, where x is *not* of the type $\perp$, we have:

$$\mathbf{c} \xrightarrow{x} x = \mathbf{c}$$

where x is an arbitrary variable.

We use **ABSENT** in our definition for the converse situation where the variable whose context we are trying to find is not present in the TT expression. For example, the context which propagates to x from the expression $y + 2$ will be **ABSENT**.

Definition 12

$$\mathbf{c} \xrightarrow{y} x = \mathbf{ABSENT}$$

where x and y are distinct variables.

This is generalised by the following definition.

Definition 13

$$\mathbf{c} \xrightarrow{E} x = \mathbf{ABSENT}$$

where x does not occur free in E.

However, since expressions in TT are built up from applications, Definition 13 is unnecessary as it follows from Definition 12 and the method of propagating contexts with respect to function applications given in Section 2.8.3.

2.8.2 Conditional expressions

As noted in Section 2.4.4, the **contand** operation, $\&$, is used to combine together contexts in a manner similar to logical-AND. Also, the **contor** operation, $\sqcup$, is used to denote uncertainty and joins contexts together like logical-OR. These operations are useful in defining context propagation with respect to Booleans and selection over finite types, since we know that one sub-expression *must* be evaluated in order to determine which other sub-expression will be the result of the conditional.

Boolean conditionals and, more generally, selection over finite types are thus handled by the following definitions:

Definition 14

$$\mathbf{c} \xrightarrow{if\ b\ then\ c\ else\ d} x = (\mathbf{c} \xrightarrow{b} x) \,\&\, ((\mathbf{c} \xrightarrow{c} x) \sqcup (\mathbf{c} \xrightarrow{d} x))$$

Definition 15

$$\mathbf{c} \xrightarrow{cases_n\ v\ c_1 \ldots c_n} x = (\mathbf{c} \xrightarrow{v} x) \,\&\, (\bigsqcup_{i=1}^{i=n} (\mathbf{c} \xrightarrow{c_i} x))$$

The above definitions capture the intuition that we must evaluate the boolean expression in an *if-then-else* or the first argument of a $cases_n$ expression before evaluating one of the branches. However, we cannot know in advance, in general, which branch will be evaluated.

2.8.3 Function application

In a functional language based upon type theory we will often want to ascertain the context of x with respect to a function application of the form:

$$f\ E_1 \ldots E_n$$

where $E_1 \ldots E_n$ are expressions. $E_1 \ldots E_n$ are the actual parameters which will be substituted for f's formal parameters, $x_1 \ldots x_n$. We thus wish to find:

$$\mathbf{c} \xrightarrow{f\ E_1 \ldots E_n} x$$

Naturally x may occur in any of the subexpressions of the application of f. It is thus necessary to deduce the context of a formal parameter, x_i of f which will consequently give us the context to be propagated through the corresponding E_i to x. Starting with the context $\mathbf{c}$, as above, we denote the context of f's ith formal parameter as

$$\mathbf{f_i\ c}$$

We may form an expression, given in terms of $\mathbf{c}$, for $\mathbf{f_i\ c}$, as is described in Section 2.8.4. As mentioned above, $\mathbf{f_i\ c}$ is then used as the initial context to be propagated through E_i, the ith actual parameter of f, to x. That is, the resulting context is expressed as:

$$(\mathbf{f_i\ c}) \xrightarrow{E_i} x$$

Naturally, applying this procedure to each parameter we end up with n contexts. These contexts have to be combined using **contand**, $\&$, to produce the context which is derived from the original application — the resulting context represents the information common to all n contexts of the form, $(\mathbf{f_i\ c}) \xrightarrow{E_i} x$. Thus we have:

Definition 16

$$\mathbf{c} \xrightarrow{f\ E_1 \ldots E_n} x = ((\mathbf{f_1\ c}) \xrightarrow{E_1} x) \,\&\, ((\mathbf{f_2\ c}) \xrightarrow{E_2} x) \,\&\, \ldots \,\&\, ((\mathbf{f_n\ c}) \xrightarrow{E_n} x)$$

2.8.4 Function definitions

We may form an expression, given in terms of $\mathbf{c}$, for $\mathbf{f_i}\ \mathbf{c}$ — we thus call $\mathbf{f_i}\ \mathbf{c}$ the **context function of f's ith argument**. If f has n parameters it will have n context functions. Furthermore, as shown in Section 2.6, context types may be formed in correspondence to the types of the concrete semantics. Thus if we have a function f where:

$$f : T_1 \Rightarrow T_2 \Rightarrow \ \ldots \ \Rightarrow T_n$$

then $\mathbf{f_i}$ has the following context type:

$$\mathbf{f_i} : \mathbf{C}_{T_n} \rightarrow \mathbf{C}_{T_i}$$

Here, $\mathbf{C}_{T_n}$ is the type of contexts corresponding to the output type, T_n, in the original function. $\mathbf{C}_{T_i}$ is the type of contexts corresponding to the type of the ith input. We discuss how different context types are formed relative to the types of TT in Section 2.6 and, more formally, in Section 2.14.1. In general, the basic procedure to find an expression for a context function is defined as follows.

Definition 17

$$\mathbf{f_i}\ \mathbf{c} = \mathbf{c} \xrightarrow{E} x_i$$

where E is the expression over which f is abstracted. In other words, f is defined in the following way:

$$f\ x_1 \ldots x_n = E$$

We discuss the context functions that arise from the use of pattern matching to define type theoretic functions in Section 2.8.5.

2.8.5 Pattern matching

As with other functional programming systems we may define functions in type theory using *pattern matching*. Naturally, we must develop a theory of context propagation with regard to this kind of function definition.

In our theory we assume that functions are defined in the following way:

$$\begin{array}{rcl} f\ p_1 & = & E_1 \\ & \vdots & \\ f\ p_n & = & E_n \end{array}$$

We assume (and this is true in type theory for the basic computation rule selectors) that the p_is are exhaustive over the type whilst being mutually exclusive. In other words, any object of the type *must* match one and only one of the patterns given. To analyse such a function we may suppose that we have an application of the following form:

$$f\ e$$

where e is of the same type as each p_i.

In order to perform a pattern match we have to evaluate e to a certain extent until one of the patterns is matched i.e. the structure of the partially computed version of e will match one and only one of the p_is. If, however, there is only one pattern and none of the variables in the pattern is used on the right-hand side of the definition of f, then there is no need to evaluate e in order to perform the pattern match — we call such a pattern *trivial*. An example of a trivial pattern is that which occurs in the *case* selector for the $\top$ type. There we have

$$case\ Triv\ c \rightarrow c$$

There is no need to evaluate x in the expression

$$case\ x\ c$$

as, if x is of type $\top$, then it must evaluate to *Triv*. Furthermore, *Triv* does not contain any components which are used to evaluate c.

The patterns will contain sub-variables and constructors which together indicate the pattern being matched. The expression e will match a pattern p_i (i.e. $e =_{PM} p_i$) if p_i is of the form:

$$C_i\ p_{i,1} \ldots p_{i,m}$$

(where $p_{i,1} \ldots p_{i,m}$ are variables and C_i is a constructor)

and e may be reduced to the form:

$$C_i\ t_1 \ldots t_m$$

for some terms $t_1 \ldots t_m$. If $\mathbf{CTR}_i$ is the context constructor corresponding to C_i then we obtain, for $\mathbf{f_1\ c}$, where f is defined by a non-trivial pattern match:

$$\mathbf{at}(\mathbf{c})_{(1 \leq i \leq n)\ (\mathbf{CTR}_i((\mathbf{c} \overset{E_i}{\longrightarrow} p_{i,1}),\ldots(\mathbf{c} \overset{E_i}{\longrightarrow} p_{i,m})))}$$

Thus it is functional definitions by pattern matching which lead to the construction of structured contexts.

2.8.6 Other expressions in TT

We may analyse other expressions in TT using the methods described in the preceding sections. In particular, our approach consists of treating each of the selectors of type theory as a primitive function from which context functions may be derived. For example, the *Fst* selector over existential type elements has the following computation rule:

$$Fst\ (p, q) \rightarrow p$$

This gives rise to the following context function:

$$\mathbf{Fst_1\ c} = \mathbf{at}(\mathbf{c})_{(\mathbf{c},\mathbf{AB})}$$

The analysis of some selectors will depend upon the analysis of higher-order functions, which we describe in Section 2.11. We give the formal definitions of the context expressions derived from all the expressions of TT in Section 2.14.4. In Section 2.9 we discuss context propagation with respect to computationally redundant proofs in TT.

2.9 Computationally irrelevant proof objects

We now discuss how computationally redundant proof objects are dealt with by our system of analysis. We cover the $\perp$, $\top$ and equality types. These are the types which correspond to the class *prop* in the subset theory. ($\perp$ corresponds to propositions which are contradictory whilst $\top$ corresponds to a valid proposition. Equality types are the counterparts to propositions asserting equality between elements of a type.)

The flow of contextual information will ensure that any other terms constructed using expressions of these computationally redundant types will also have the computationally irrelevant information identified by the analysis (*cf* Section 3.4.1 of [5]). Thus the abstract property of computational redundancy will be traced through the program by our backwards analysis.

Computational redundancy occurs where we are simply interested in whether a type is inhabited or not. For example, if we have $t : \top$ then both the syntactic form of t and its computation to normal form are unimportant since, if the program is well-formed, then it must be the case that

$$t \twoheadrightarrow Triv$$

since $\top$ is inhabited by one element only. As is stated in [5],

> The important point to note about such types, and those exhibiting computational redundancy in general, is that their objects can always be transformed to equal objects containing no free variables.

2.9.1 The $\perp$ type and the *abort* term

The type theory selector *abort* provides us with a witness to *ex falso quodlibet*. It is included, for the sake of completeness, to guard against the possibility of an incorrect program derivation occurring. (The *abort* construct provides extra strength to programming in type theory: not only will any program which is correct in the system of type theory terminate — a syntactically correct Miranda program may not terminate — but programming errors may also be dealt with elegantly in the system logic rather than by some run-time system call.)

The *abort* object, capturing the spirit of *ex falso quodlibet*, is an arbitrary object of a type A. The term $abort_A\, p$ has no computation rule associated with it and is in normal form: it is nonsensical to try to reduce the pathological proof object p. Since p and $abort_A\, p$, where $p : \perp$, may not be reduced further, any parameter must be computationally irrelevant with respect to such expressions. This is similar to the idea that parameters are **ABSENT** with respect to constants such as 3.

Definition 18

For an initial context $\mathbf{c}$, an arbitrary type A and an object of type bottom, p:

$$\mathbf{c} \xrightarrow{p} x = \mathbf{ABSENT}$$

and

$$\mathbf{c} \xrightarrow{abort_A\; p} x = \mathbf{ABSENT}$$

Note that our idea of the *propagation of contexts* will mean that the **ABSENT** context will result for the bound variables of the functions corresponding to negation types (where $\neg A \equiv_{df} A \Rightarrow \bot$), for instance. For example, suppose we have the following:

$$\begin{array}{rcl} neg & : & (A \Rightarrow \bot) \\ neg\ a & \equiv_{df} & p(a) \end{array}$$

Here p is some arbitrary, pathological proof object, which may or may not refer to a. The context which is propagated to a here will be **ABSENT**, as an instance of the above definition, so that in neededness analysis a will be detected as being unused. If we have functions built from negated types then absence will be propagated to the relevant parameters. For instance, if we have:

$$\begin{array}{rcl} bnega & : & B \Rightarrow \neg A \\ bnega\ b\, a & \equiv_{df} & neg\ a \end{array}$$

Then **ABSENT** will be propagated as the context of a (as the sole parameter of neg is **ABSENT**) whilst b will also be detected as **ABSENT** since it does not appear in the defining expression. We can complicate matters by introducing conditionals or other expressions, but the analysis will still detect absence due to the $\bot$ type. As an example, suppose we have:

$$\begin{array}{rcl} NnegN & : & N \Rightarrow \neg N \\ NnegN\ m\, n & \equiv_{df} & if\ (m = 1)\ then\ neg\,(n+1)\ else\ neg\, n \end{array}$$

Here the second parameter, n, will be detected as **ABSENT**, although the first parameter, m, will have the context **N** in neededness analysis *if we start the analysis with* **N**. The starting context is, of course, essential in that if the input context to the analysis of the function was **ABSENT** then **ABSENT** would be the resulting context for all the parameters. Indeed, the input context to the context functions of $NnegN$ will *always* be **ABSENT** since the output type of the function is $\bot$. That is, the context propagated to a term corresponding to the output of *NnegN* will be **ABSENT** which will thus propagate to each of the parameters of *NnegN*.

An example of how computational redundancy may be detected with regard to proof objects of type $\perp$ and *abort* expressions may be seen in the analysis of the *index* function in Section 2.10.

2.9.2 The $\top$ witness

The single element type, $\top$, may be seen as corresponding to the judgement "*P is true*" in the subset theory. It has the following *elimination* and *computation* rules in the theory:

$$\frac{x : \top \qquad l : C(Triv)}{case\ x\ c : C(x)} \ (\top\ Elim)$$

$$case\ Triv\ c \rightarrow c$$

We assume that we are dealing entirely with closed terms. Hence, any occurrence of the expression *case p c* *must* compute to the value of c since p, being of type $\top$, must compute to *Triv*. Thus, as noted in Section 2.8.5, reducing the term p is unnecessary since we know that it may be of one form only. Indeed, when we abstract over p and c, expressions of the form *case p c* may be thought of as witnesses to types of the form,

$$\top \Rightarrow (A \Rightarrow A)$$

A simple function of this kind would be:

$$\begin{array}{rcl} casefn & : & \top \Rightarrow (A \Rightarrow A) \\ casefn\ p\ c & \equiv_{df} & case\ p\ c \end{array}$$

It can be seen that, by removing the computationally redundant p, we will have the identity function over A. Similarly, the dependent type $C(p)$ must be equivalent to $C(Triv)$.

Thus we make the following definition:

Definition 19

$$\mathbf{c} \xrightarrow{case\ p\ c} x = \mathbf{c} \xrightarrow{c} x$$

for any variable, x.

Note that this is saying that, if we regard the *case* selector as a function,

$$\mathbf{case_1\ c = ABSENT}$$

It should be observed that the difference between the *case* selector over the $\top$ type and the general *cases* selector over the finite types in general (i.e. N_n) lies in the fact that we have a *unique* pattern that *must* be matched for a term of the $\top$ type.

Also, the propagation of contexts means that if any function has $\top$ as its output type then the **ABSENT** context will pertain to the parameters of that function as well.

2.9.3 Equality types

The equality types, which are written in the form $I(A, a, b)$, denote the equality of two terms a and b of type A. The elimination and computation rules are:

$$\frac{c : I(A, a, b) \quad d : C(a, a, r(a))}{J(c, d) : C(a, b, c)} \quad (Equal\ Elim)$$

The above is equivalent to the Leibnitz law that equals may be substituted for equals — *some* occurrences of a are replaced by b in C.

$$J(r(a), d) \rightarrow d$$

Again, assuming that we are dealing with closed terms, A, a, b and c must all be bound with respect to an enclosing abstraction in the expression $J(c, d)$ and so a and b must be bound variables in d. Also, all closed terms of an equality type can be proven to be equal (see [134, Section 4.10.4]), so that for any a, b of type A,

$$r(a) \equiv r(b)$$

Here the a and b which occur in the terms exist only as *labels* for the purposes of complete presentation (so that we know how each equality term originated) and to ensure that each term in the TT system has a *unique* type — if we had a generic **eq** equality witness then that would belong to every equality type. Nevertheless, it should be stressed that the witnesses of each equality type are unique and *have no internal structure*. The purpose of a term such as $r(a)$ is simply to assert the validity of $I(A, a, b)$, say, where a and b are interconvertible by the reduction rules of TT. c must evaluate to a term which is equivalent to any other witness of $I(A, a, b)$. Also, $C(a, a, r(a))$ must be equivalent to $C(a, b, c)$. Thus we are in a similar situation as to that for the $\top$ type which has a unique witness. As in that instance, we can avoid computing the equality witness due to it being the sole inhabitant of its type.

We can bind the free variables to form the function:

$$\begin{array}{rcl} jayfn & : & (\forall a, b : A).(\forall c : I(A, a, b)).(C(a, a, r(a)) \Rightarrow C(a, b, c)) \\ jayfn\ a\ b\ c\ d & \equiv_{df} & J(c, d) \end{array}$$

Here, all parameters apart from d will be redundant and have the **ABSENT** context induced as their abstract values. As a result, the above function may be reduced to the identity function.

Consequently, for such closed terms, we state that it is not necessary to evaluate $c : I(A, a, b)$ (*cf* p.62 of [114] which says that c should be first be evaluated to compute the *open* term $J(c, d)$) and hence we have the following definition for our analysis:

Definition 20

$$\mathbf{c} \xrightarrow{J(c,d)} x = \mathbf{c} \xrightarrow{d} x$$

for any variable x.

It follows that computational redundancy from all equality types such as $I(A, a, b)$ will be propagated throughout a TT program. In neededness analysis, equality type parameters will be detected as unused.

2.10 Example: the *index* function

This section presents a backwards analysis performed on the arguments of index. The analysis shows that the first two arguments may or may not be used (i.e. they are *lazy*) but that the third argument, which witnesses the fact that the given index is less than the length of the list, is not actually relevant to the computation. We may thus produce an *optimized* form of the object code for this function which does not compute the value of the third argument.

2.10.1 Definition of the function in TT

The function is that first presented in Section 1.2.1:

$$index \quad : \quad (\forall l : [A]).(\forall n : N).((n < \#l) \Rightarrow A)$$

$$\begin{aligned} index\ []\ n\ p &\equiv_{df} abort_A\ p & (2)\\ index\ (a{::}x)\ 0\ p &\equiv_{df} a & (3)\\ index\ (a{::}x)\ (n+1)\ p &\equiv_{df} index\ x\ n\ p & (4) \end{aligned}$$

2.10.2 Analysis of the first argument

We first formulate the context function of the first argument of *index* for an arbitrary initial context $\mathbf{c}$. The *index* function may be divided into two parts: the first which deals with the case that the first argument evaluates to [] (clause (2) of the *index* function) and the second which deals with a non-empty first argument. We, naturally, do not know which of these parts will apply in the actual execution of the function: this uncertainty is shown in the sharing analysis by the $\sqcup$ operator. In other words, we are joining together the contexts which result from each of the possible two parts. Here the structured part has two context variables which have to be evaluated, namely the arguments of :: — these context variables gives us information about the head and tail parts of the argument. We may thus form the following expression for the context function of the first argument of *index*:

$$\mathbf{index_1\ c} = \mathbf{at}(\mathbf{c})_{[],\, (\mathbf{c} \xrightarrow{(3),(4)} a) :: (\mathbf{c} \xrightarrow{(3),(4)} x)}$$

The head and tail contexts may, as they refer to clauses (3) and (4) of *index*, be split into two parts. Here the two cases arise from the form of the second argument which may be zero or not. For this part of the analysis we thus have:

$$\begin{aligned}\mathbf{c} \xrightarrow{(3),(4)} a &= (\mathbf{c} \xrightarrow{a} a) \sqcup (\mathbf{c} \xrightarrow{index\; x\; n\; p} a)\\ &= \mathbf{c} \sqcup \mathbf{AB}\end{aligned}$$

The above follows from a not being present in the expression $index\, x\, n\, p$. Also,

$$\begin{aligned}\mathbf{c} \xrightarrow{(3),(4)} x &= (\mathbf{c} \xrightarrow{a} x) \sqcup (\mathbf{c} \xrightarrow{index\; x\; n\; p} x)\\ &= \mathbf{AB} \sqcup \mathbf{index_1 c}\end{aligned}$$

For the sake of notational convenience, we shall leave out the [] context constructor as this information is invariant in what follows. We have thus to solve the following recursive equation:

$$\mathbf{index_1 c} = \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{index_1 c})}$$

The above may be solved by performing the following fixpoint iteration, using the ascending Kleene chain of pre-fixpoints. The first (zeroth) approximation to the fixpoint is defined to be $\mathbf{CR}$ whilst the $(n+1)$th approximation is formed by substituting the nth approximation to the fixpoint for every occurrence of $(\mathbf{index_1\, c})$ in the above equation. As shown below, the third in a series of fixpoint iterations gives the fixpoint.

$$\begin{aligned}
(\mathbf{index_1\, c})^0 &= \mathbf{CR}\\
(\mathbf{index_1\, c})^1 &= \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: \mathbf{AB}}\\
(\mathbf{index_1\, c})^2 &= \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: \mathbf{AB}})}\\
&\sqsubseteq\!\!\!\sim \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{at(c)})}\\
(\mathbf{index_1\, c})^3 &= \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{at(c)})})}\\
&\sqsubseteq\!\!\!\sim \mathbf{at(c)}_{(\mathbf{c} \sqcup \mathbf{AB}) :: (\mathbf{AB} \sqcup \mathbf{at(c)})}
\end{aligned}$$

Note that we have to make approximations to the resulting context after the second iteration since the context has more than one level of subscripting: we assume that the list contexts have the non-empty list subscript $\mathbf{d} :: \mathbf{e}$. The latter case means that we assume that $\mathbf{e}$ represents a list context of the form $\mathbf{e}_{\mathbf{d} :: \mathbf{e}}$.

The above illustrates how a fixpoint solution may be found by using purely *algebraic* manipulation upon an arbitrary argument $\mathbf{c}$. Mechanically, however, this may not be straightforward in more complicated cases.

As an example of a concrete rather than an algebraic result, the following is the result produced when the context function is applied to a strict, single-use argument. The argument is a member of the type of polymorphic contexts: these are basically atomic contexts that are used when the corresponding type in the concrete semantics is polymorphic. (Here the result of *index* is an element of an arbitrary type, A.)

$$\begin{aligned}\mathbf{index_1}\ \{1\}_{\mathbf{Poly}} &= \{1\}_{(\{0,1\}_{\mathbf{Poly}} :: (\{0\} \sqcup \{1\}))} \\ &= \{1\}_{(\{0,1\}_{\mathbf{Poly}} :: \{0,1\})}\end{aligned}$$

2.10.3 Analysis of the second argument

$$\begin{aligned}\mathbf{index_2\ c} &= \mathbf{c} \xrightarrow{abort_A\ p} n \sqcup (\mathbf{at(c)}_{\mathbf{0}, (\mathbf{Succ(index_2\ c)})}) \\ &= \mathbf{AB}\ \sqcup\ \mathbf{at(c)}_{(\mathbf{Succ(index_2\ c)})}\end{aligned}$$

(We are using a similar notational shorthand to that which we employed for the first argument.)

The solution to this is again found by a fixpoint iteration:

$$\begin{aligned}(\mathbf{index_2\ c})^1 &= \mathbf{AB}\ \sqcup\ \mathbf{at(c)}_{\mathbf{Succ(CR)}} \\ (\mathbf{index_2\ c})^2 &= \mathbf{AB} \sqcup \mathbf{at(c)}_{\mathbf{Succ(AB} \sqcup \mathbf{at(c)}_{\mathbf{Succ(CR)}})} \\ &\sqsupseteq\!\!\!\sim \mathbf{AB} \sqcup \mathbf{at(c)}_{\mathbf{Succ(AB} \sqcup \mathbf{at(c))}}\end{aligned}$$

It follows immediately that this is the least fixpoint. Hence, the second argument is lazy and even if it is used it may not be fully evaluated. This is illustrated by the following result:

$$\mathbf{index_2}\ \{1\}_{\mathbf{Poly}} = \{0,1\}_{[], \mathbf{Succ}(\{0,1\})}$$

However, it should be noted that the second parameter will be fully evaluated in the two non-pathological cases.

2.10.4 Analysis of the third argument

The analysis of the third argument, which witnesses the fact that the index is not greater than the length of the list, is fairly simple:

$$\begin{aligned}\mathbf{index_3\ c} &= \mathbf{AB} \sqcup (\mathbf{AB} \sqcup \mathbf{index_3\ c}) \\ &= \mathbf{AB} \sqcup \mathbf{index_3\ c}\end{aligned}$$

The least fixpoint solution of the above is, trivially, **ABSENT**. We have that:

$$\mathbf{index_3}\ \{1\} = \{0\}_{\mathbf{Path,Triv}}$$

where the subscript, "**Path,Triv**" denotes the fact that the proof object p is of a dependent type which may be either $\perp$ or $\top$. We conclude that the third argument is not "necessary" when computing an application of *index* to normal form.

2.11 Higher-order functions

Hughes and Launchbury wrote in [75] that:

> ...backwards analyses in general have difficulty with higher-order functions.

This difficulty is due to applications of formal parameters in the defining expression of a higher-order function. If a parameter, f, is applied to some subexpression, e, then we will not, in general, be able to determine the precise context of a parameter x with respect to $f\,e$. We will only be able to deduce safely that the parameter x has the context $\top$ (the top element) in the appropriate finite lattice.

To obtain more precise information we need to extend the backwards analysis so that each context function which results from a higher-order function has extra parameters which are the context functions that correspond to each functional argument.

2.11.1 Hughes's approach to higher-order functions

As a first attempt, Hughes argues [69] that higher-order functions will have higher-order context functions as their counterparts in backwards analysis since the context functions of parameters are unknowns (they may be arbitrary context functions). This means that the corresponding context functions will take extra parameters, one for each functional parameter in the original function. The contexts that are propagated to the parameters are found by supplying a set of possible functions for each context function parameter.

Hughes says, however, that such an approach (which was implemented in the Ponder compiler by Wray [149, 150]) is only possible for the simplest type of higher-order function where the parameters of functions are not extracted from data structures and where functions are not defined by partial application. An example of such a function in a Miranda-like language would be:

```
compose :: (** -> ***) -> (* -> **)-> * -> ***
compose f g x = f(g x)
```

Hughes gives the following as an example of a function which would *not* be covered by this simple analysis:

```
composeall :: [(* -> *)] -> * -> *
composeall [] x = x
composeall (f:r) x = f(composeall r x)
```

Here the functions are extracted from a list. He thus develops a more sophisticated theory, with a simple forwards analysis included in the form of an environment of abstract values to guide the subsequent backwards analysis. The idea is that using such a forwards analysis we should be able to gain context information for all possible contextual arguments of a higher-order function. This forwards analysis is a form of arity check upon the expression being analysed: objects of ground type have their abstract values to be defined as **1** which means that the original version of the theory still holds. Otherwise, the abstract values are those of context function spaces (which may themselves include abstract values).

This naturally makes the entire analysis more complicated with, in particular, an environment of abstract values being included. Context propagation is consequently much more complex for analyses involving parameters that are not of ground type. For this reason, although he formalises completely how higher-order constructs should be analysed, Hughes chooses not to give an example of this form of backwards analysis.

2.11.2 The approach to higher-order functions used

Our approach is similar to that of Wray mentioned above. There are three points to be followed when analysing higher-order functions:

1. The formation of context expressions from applications of formal parameters may proceed in the same way as for named functions, except that instead of the name of the context function we have a variable name. This will allow context expressions to be substituted for the variable name. So we have variable context function applications of the form:

$$(p)_{\mathbf{1}}$$

2. Each context function application will require, additionally, a set of context expressions as parameters. (Recursively, these expressions may also require context expression parameters.) So instead of having, say,

$$\mathbf{filter_2\;c}$$

we may have

$$\mathbf{filter_2\;[lesseq_1\;[plus_2\;[AB, AB]], AB]\;c}$$

which would be part of the context expression resulting from

$$\mathit{filter}\,(\mathit{lesseq}\,(1+2))\,l$$

(where l is some list constant). This will also apply with regard to formal parameters so we may have:

$$(p\,[(a)_{\mathbf{1}}\,[]])_{\mathbf{1}}$$

3. When the actual parameter context expressions are substituted for variables the context function indices and the lists of actual parameter expressions have to be suitably combined. For instance, if we substituted

$$\mathbf{lesseq_1\;[plus_2\;[AB, AB]]}$$

for p in

$$(p\,[\mathbf{AB}])_{\mathbf{1}}$$

we would get:

$$\mathbf{lesseq_2}\,[\mathbf{plus_2}\,[\mathbf{AB}, \mathbf{AB}], \mathbf{AB}]$$

The above means that each context propagation is performed relative to a set of context expressions. Context propagation thus will have the following general form:

$$[\mathbf{e}_1 \ldots \mathbf{e}_n]\ \mathbf{c} \xrightarrow{e} x$$

For higher-order functions, function applications are of the following general form:

$$ap\ f\ a$$

which has the informal semantics that f is evaluated to a lambda abstraction which is then applied to a. We then make the following definition:

$$[\mathbf{a}_1 \ldots \mathbf{a}_n]\ \mathbf{c} \xrightarrow{ap\ f\ a} x = ([\mathbf{a}_1 \ldots \mathbf{a}_n]\ (\mathbf{F}\ \mathbf{c}) \xrightarrow{a} x)\ \&\ ([\mathbf{a}_1 \ldots \mathbf{a}_n]\ \mathbf{c} \xrightarrow{f} x)$$

Where

$$\begin{array}{l}
\mathbf{F} \equiv_{df} \mathbf{efm}\ f\ 1\ [\mathbf{efm}\ a\ 0\ []\ [\mathbf{a}_1 \ldots \mathbf{a}_n]] \\
\mathbf{efm}\ f\ i\ [\mathbf{b}_1 \ldots \mathbf{b}_j][\mathbf{c}_1 \ldots \mathbf{c}_k] \equiv_{df} \\
\quad \begin{cases} \mathbf{f_i}\ [\mathbf{c}_1 \ldots \mathbf{c}_k] & \text{if } f \text{ is a named function} \\ \mathbf{efm}\ g\ (i+1)\ [\mathbf{d}, \mathbf{b}_1 \ldots \mathbf{b}_j]\ [\mathbf{c}_1 \ldots \mathbf{c}_k], & \text{if } f \text{ is of the form } ap\ g\ d \\ \mathbf{G}\ l\ i\ [\mathbf{b}_1 \ldots \mathbf{b}_j]\ [\mathbf{c}_1 \ldots \mathbf{c}_k], & \text{if } f \text{ is the parameter } x_i \end{cases} \\
\mathbf{d} \equiv_{df} (\mathbf{efm}\ d\ 0\ [\mathbf{c}_1 \ldots \mathbf{c}_k]) \\
\mathbf{G}\ \ l\ i\ [\mathbf{b}_1 \ldots \mathbf{b}_j]\ [\mathbf{c}_1 \ldots \mathbf{c}_k] \equiv_{df} \\
\quad \begin{cases} \mathbf{fn_{i+r}}\ [\mathbf{d}_1 \ldots \mathbf{d}_m, \mathbf{b}_1 \ldots \mathbf{b}_j], & \text{if } l \leq k \text{ and} \\ & \mathbf{c}_l \text{ is of the form } \mathbf{fn_r}\ [\mathbf{d}_1 \ldots \mathbf{d}_m] \\ \\ \mathbf{top_q}, & \text{otherwise} \end{cases}
\end{array}$$

The last clause in the definition of $\mathbf{G}$ expresses the idea that if not all the necessary expressions are supplied then we must safely approximate by using the **top** context function which produces the top element of the context lattice for an input context.

Functions defined by partial application

We may overcome the problem of functions defined by partial application by adding extra variables to a function definition in order to make sure that it becomes fully applied. This is easily done in the monomorphic case since we know the type (and the arity) of the function and therefore will know precisely how many variables to add.

In the polymorphic case we may add an arbitrary *vector* of variables to each definition in order to generalise the function to accept an arbitrary number of arguments. For instance, if we consider the primitive recursion operator in type theory as a function i.e.

$$\begin{array}{rcl} prim\, 0\, c\, f & \equiv_{df} & c \\ prim\, (n+1)\, c\, f & \equiv_{df} & f\, n\, (prim\, n\, c\, f) \end{array}$$

then we may imagine this to be extended to take an arbitrary vector of additional parameters, $\overrightarrow{x}$, thus:

$$\begin{array}{rcl} prim'\, 0\, c\, f\, \overrightarrow{x} & \equiv_{df} & c\, \overrightarrow{x} \\ prim'\, (n+1)\, c\, f\, \overrightarrow{x} & \equiv_{df} & f\, n\, (prim'\, n\, c\, f)\, \overrightarrow{x} \end{array}$$

Hence, for $i > 3$, we get:

$$\mathbf{prim_i}\, [\mathbf{n}, \mathbf{c}, \mathbf{f}, \overrightarrow{\mathbf{x}}] \equiv_{df} (\mathbf{c_i}\ \overrightarrow{\mathbf{x}}) \sqcup (\mathbf{f_i}\, [\mathbf{AB}, \mathbf{prim_3}\, [\mathbf{AB}, \mathbf{c}, \mathbf{f}], \overrightarrow{\mathbf{x}}])$$

where $\overrightarrow{\mathbf{x}}$ is the set of context expressions corresponding to each of the parameters in $\overrightarrow{x}$.

Functions extracted from data structures

Unfortunately, the above method does not cope with functions extracted from data structures. If we had an explicit list of functions then the method could cope since in order to analyse

$$composeall\, [id_N, addone]\, e$$

we could just form a context function which gave the least upper bound of the results of the context functions corresponding to id_N and $addone$. However, this does not work with, for example,

$$composeall\, (adders\, [1, 2, 3])\, e$$

where the list of functions is itself formed by another function. (In this case, $adders$ forms the $(x+)$ function for each member, x, of the input list.)

Hence, if we have any occurrence of an application of a pattern matching variable then we must safely approximate by using the **top** context function.

2.12 Polymorphism

By building up a set of context expressions, we can allow contexts of various types to propagate. We analyse polymorphic variables by assuming that their contexts are basically atomic and tagging their structured parts to indicate that they are polymorphic. For example, we might have:

$$\{1\}_{\mathbf{Poly}_A}$$

The A above is used to indicate polymorphism with respect to a type variable A.

However, whenever a context function, **f**, that corresponds to a function with a polymorphic result type, is applied within another context function, the polymorphic structured part may be replaced by the structured part of the input context to **f**. Hence, **f** will have various instantiations, depending on the contexts with which it is called.

2.13 Example: The *quicksort* function

As a larger example and, especially, to illustrate our ideas with regard to higher-order functions, we present the analysis of the *quicksort* function.

2.13.1 The analysis of the *filter* function

As part of our analysis of the quicksort function, we show how the higher-order function, *filter* may be analysed.

Definition of filter

$$\begin{array}{rcll} \mathit{filter} & : & (A \Rightarrow \mathit{bool}) \Rightarrow [A] \Rightarrow [A] & \\ & & & \\ \mathit{filter}\, p\, [] & \equiv_{df} & [] & \\ \mathit{filter}\, p\, (a :: x) & \equiv_{df} & a :: (\mathit{filter}\, p\, x) & , \textbf{if } (p\ a) \\ \mathit{filter}\, p\, (a :: x) & \equiv_{df} & \mathit{filter}\, p\, x & , \textbf{otherwise} \end{array}$$

The first argument

It may be shown that the first argument of filter (the predicate which operates on the second argument, the list) will always have the atomic context:

$$\{0, 1, M\}$$

if the initial context is non-trivial (i.e. not **AB** or **CR**.) This means that the functional argument is "fully lazy" i.e. we cannot determine in advance whether it has to be evaluated or whether it may be evaluated more than once.

The reason for the above is as follows.

$$\begin{array}{rcl} \mathbf{filter_1\ c} & = & \mathbf{c} \xrightarrow{[]} p \sqcup ((\mathbf{c} \xrightarrow{p\,a} p) \,\&\, ((\mathbf{c} \xrightarrow{a::(\mathit{filter}\ p\, x)} p) \sqcup (\mathbf{c} \xrightarrow{\mathit{filter}\ p\, x} p))) \\ & = & \mathbf{AB} \sqcup (\mathbf{at(c)}_{\mathbf{Fun}} \,\&\, ((\mathbf{filter_1}\,(::_2\,\mathbf{c}) \sqcup (\mathbf{filter_1\ c}))) \end{array}$$

Now, for the input context, $\{1\}_{[], \{1\} :: \{1\}}$,

$$::_2\, \{1\}_{[], \{1\} :: \{1\}} = \{1\}_{[], \{1\} :: \{1\}}$$

Hence,

$$\mathbf{filter_1}\,\{1\}_{[],\{1\}::\{1\}} = \{0\}_{\mathbf{Fun}} \sqcup (\{1\}_{\mathbf{Fun}} \,\&\, (\mathbf{filter_1}\,\{1\}_{[],\{1\}::\{1\}}))$$

A fixpoint iteration gives the solution:

$$\{0,1,M\}_{\mathbf{Fun}}$$

The analysis of the second argument to filter

The analysis of the second argument of *filter* proceeds as follows.

$$\mathbf{filter_2}\,[p,l]\;\mathbf{c} = \mathbf{at}(\mathbf{c})_{[],(\mathbf{d}_1::\mathbf{d}_2)}$$

$$\begin{aligned}\mathbf{d}_1 \quad = \quad & ((\mathbf{at}(\mathbf{c})_{\mathbf{TF}}) \xrightarrow{p\;a} a) \\ & \& \\ & ((\mathbf{c} \xrightarrow{a::(\mathit{filter}\;p\;x)} a) \sqcup (\mathbf{c} \xrightarrow{\mathit{filter}\;p\;x} a))\end{aligned}$$

(Above we are writing **TF** to stand for **True, False**.)

$$(\mathbf{at}(\mathbf{c})_{\mathbf{TF}}) \xrightarrow{p\;a} a \quad = \quad (p)_{\mathbf{1}}\,[\mathbf{top_0}\,[]]\,(\mathbf{at}(\mathbf{c})_{\mathbf{TF}})$$

In the interests of conciseness, we shall omit the context expression argument $[\mathbf{top_0}\,[]]$ — this argument simply means that if the predicate parameter p applies its argument (which is the head of the input list to *filter*) then the **TOP** context will result.

$$\mathbf{c} \xrightarrow{a::(\mathit{filter}\;p\;x)} a = ::_{\mathbf{1}}\,\mathbf{c}$$

$$\begin{aligned}\mathbf{d}_2 \quad &= \quad (\mathbf{c} \xrightarrow{a::\mathit{filter}\;p\;x} x) \sqcup (\mathbf{c} \xrightarrow{\mathit{filter}\;p\;x} x) \\ &= \quad ((::_{\mathbf{2}}\,\mathbf{c}) \xrightarrow{\mathit{filter}\;p\;x} x) \sqcup (\mathbf{c} \xrightarrow{\mathit{filter}\;p\;x} x) \\ &= \quad (\mathbf{filter_2}\,[(p)_{\mathbf{0}}\,[],\mathbf{top_0}\,[]]\,(::_{\mathbf{2}}\,\mathbf{c})) \sqcup (\mathbf{filter_2}\,[(p)_{\mathbf{0}}\,[],\mathbf{top_0}\,[]]\,\mathbf{c})\end{aligned}$$

(The second context parameter is **AB** since a list cannot be applied to an argument.) Hence we obtain,

$$\mathbf{filter_2}\,[p,l]\;\mathbf{c} \quad = \quad \mathbf{at}(\mathbf{c})_{\mathbf{n}_1::\mathbf{n}_2}$$

Where

$$\begin{aligned}\mathbf{n}_1 \quad &= \quad ((f)_1(\mathbf{at}(\mathbf{c})_{\mathbf{TF}}))\,\&\,((::_{\mathbf{1}}\mathbf{c}) \sqcup \mathbf{AB}) \\ \mathbf{n}_2 \quad &= \quad (\mathbf{filter_2}\,[(p)_{\mathbf{0}}\,[],\mathbf{top_0}\,[]]\,(::_{\mathbf{2}}\mathbf{c})) \sqcup (\mathbf{filter_2}\,[(p)_{\mathbf{0}}\,[],\mathbf{top_0}\,[]]\,\mathbf{c})\end{aligned}$$

Writing simply p instead of $(p)_{\mathbf{0}}\,[]$ and x instead of $\mathbf{top_0}\,[]$ we have,

$$\mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c}) \;=\; \mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})_{\mathbf{r}_1\,::\,\mathbf{r}_2}$$

where

$$\begin{aligned}\mathbf{r}_1 &= ((p)_{\mathbf{1}}\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})_{\mathbf{TF}}))\,\&\,((::_{\mathbf{1}}\,\mathbf{c})\sqcup\mathbf{AB})\\ \mathbf{r}_2 &= \mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c})\end{aligned}$$

(The above follows since $::_{\mathbf{2}}\,(::_{\mathbf{2}}\,\mathbf{c}) = ::_{\mathbf{2}}\,\mathbf{c}$ due to the stipulation placed upon list contexts to ensure finiteness.) The above recursive equation may be solved by a fixed-point iteration, as shown below:

$$\begin{aligned}(\mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c}))^0 &= \mathbf{CR}\\ (\mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c}))^1 &= \mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})_{\mathbf{r}_1\,::\,\mathbf{CR}}\\ (\mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c}))^2 &= \mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})_{\mathbf{r}_1\,::\,\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})}\\ (\mathbf{filter_2}\,[p,x]\,(::_{\mathbf{2}}\,\mathbf{c}))^3 &= \mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})_{\mathbf{r}_1\,::\,\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})}\end{aligned}$$

This establishes the fixpoint.

Consequently,

$$\begin{aligned}\mathbf{filter_2}\,[p,l]\,\mathbf{c} &= \mathbf{filter_2}\,[p,x]\,\mathbf{c}\\ &\sqsubseteq\!\!\!\sim \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup(\mathbf{filter_2}\,[p,x]\,\mathbf{c}))}\end{aligned}$$

We may again employ a fixpoint iteration in order to find the least fixpoint of this recursive expression.

$$\begin{aligned}(\mathbf{filter_2}\,[p,x]\,\mathbf{c})^1 &= \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})}\\ (\mathbf{filter_2}\,[p,x]\,\mathbf{c})^2 &= \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup\mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c}))})}\\ &\sqsubseteq\!\!\!\sim \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup\mathbf{at}(\mathbf{c}))}\\ (\mathbf{filter_2}\,[p,x]\,\mathbf{c})^3 &= \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup\mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup\mathbf{at}(\mathbf{c}))})}\\ &= \mathbf{at}(\mathbf{c})_{(\mathbf{n}_1\sqcup\mathbf{r}_1)\,::\,(\mathbf{at}(::_{\mathbf{2}}\,\mathbf{c})\sqcup\mathbf{at}(\mathbf{c}))}\end{aligned}$$

Thus we have found the fixpoint and thus have a non-recursive expression for the context function of the second argument of filter.

2.13.2 Definition of quicksort in type theory

We shall use the definition of quicksort which appears on p. 213 of [Thompson, 1991] i.e.

$$qsort : (\forall n : N).(\forall\, l : [N]).((\#l \leq n) \Rightarrow [N])$$

$$
\begin{array}{rcll}
qsort\, n\, []\, p & \equiv_{df} & [] & (5)\\
qsort\, 0\, (a :: x)\, p & \equiv_{df} & abort_{[N]}\, p' & (6)\\
qsort\, (n+1)\, (a :: x)\, p & \equiv_{df} & & \\
 & & qsort\, n\, (filter\, (lesseq\, a)\, x)\, p_1 & (7)\\
 & & +\!\!+\, [a] +\!\!+ & (8)\\
 & & qsort\, n\, (filter\, (greater\, a)\, x)\, p_2 & (9)
\end{array}
$$

Note that this is the actual sorting algorithm, although a function based on this, which takes just a list as its single argument, may be given by the expression:

$$qsort\ (\#l)\, l\ Triv$$

Each argument p_i is the result of a function g_i of type:

$$
\begin{array}{l}
(\forall a : N).(\forall n : N).\\
\quad (\forall x : [N]).(\forall p : (\#(a{::}x) \leq (n+1))).\\
\quad\quad (\#(filter\ (ord_i\ a)x) \leq n)
\end{array}
$$

where

$$
\begin{array}{rcl}
ord_1 & \equiv_{df} & lesseq\\
ord_2 & \equiv_{df} & greater
\end{array}
$$

We shall write h for g_1 and j for g_2.

2.13.3 Analysis of the first argument

We assume that the natural number argument is absent with respect to the terms p_1 and p_2. This is valid since an analysis shows that the third argument is unused (see section 2.13.5).

$$\mathbf{qsort_1\ c} = \mathbf{AB} \sqcup \mathbf{at(c)}_{\mathbf{0},\,\mathbf{Succ}\,(\mathbf{c} \xrightarrow{(7)+\!+[a]+\!+(9)} n)}$$

$$
\begin{array}{rcl}
\mathbf{c} \xrightarrow{((7)+\!+[a]+\!+(9))} n & = & (+\!\!+_{\mathbf{1}}\ \mathbf{c}) \xrightarrow{(7)\ +\!+\ [a]} n\\
 & & \&\, (+\!\!+_{\mathbf{2}}\ \mathbf{c}) \xrightarrow{(9)} n
\end{array}
$$

$$
\begin{array}{rcl}
(+\!\!+_{\mathbf{1}}\ \mathbf{c}) \xrightarrow{(7)\ +\!+\ [a]} n & = & +\!\!+_{\mathbf{1}}(+\!\!+_{\mathbf{1}}\ \mathbf{c}) \xrightarrow{(7)} n\\
 & = & \mathbf{c} \xrightarrow{(7)} n
\end{array}
$$

(Since $+\!\!+_1\ \mathbf{c}\ =\ \mathbf{c}$.)

$$= \mathbf{qsort}_1\ \mathbf{c}$$

$$
\begin{aligned}
(+\!\!+_2\ \mathbf{c}) \overset{(9)}{\longrightarrow} n &= \mathbf{qsort_1}(+\!\!+_2\ \mathbf{c}) \\
&= \mathbf{qsort_1}(\mathbf{c} \sqcup ::_{\mathbf{2}}(\mathbf{c}))
\end{aligned}
$$

Then we have:

$$\mathbf{qsort_1\ c} = \mathbf{AB} \sqcup \mathbf{at}(\mathbf{c})_{\mathbf{0},\,\mathbf{Succ}(\mathbf{qsort_1\ c}\ \&\ \mathbf{qsort_1}(\mathbf{c}\ \sqcup\ ::_{\mathbf{2}}(\mathbf{c})))}$$

Now,

$$::_{\mathbf{2}}\ \{1\}_{[],\,(\{1\}_{\mathbf{0},\,\mathbf{Succ}(\{1\})})\,::\,\{1\}} = \{1\}_{[],\,(\{1\}_{\mathbf{0},\,\mathbf{Succ}(\{1\})})\,::\,\{1\}}$$

and so,

$$\mathbf{qsort_1\ d} = \{0,1\}_{\mathbf{0},\,\mathbf{Succ}((\mathbf{qsort_1\ d})\,\&\,(\mathbf{qsort_1\ d}))}$$

where

$$\mathbf{d} = \{1\}_{[],\,(\{1\}_{\mathbf{0},\,\mathbf{Succ}(\{1\})})\,::\,\{1\}}$$

This may be solved by the following fixpoint iteration

$$
\begin{aligned}
(\mathbf{qsort}_1\ \mathbf{d})^0 &= \{\} \\
(\mathbf{qsort}_1\ \mathbf{d})^1 &= \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{\})} \\
(\mathbf{qsort}_1\ \mathbf{d})^2 &= \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{\})}\,\&\,\{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{\})})} \\
&= \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\}_{\mathbf{0},\,\mathbf{Succ}(\{\})})} \\
&\precsim \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})} \\
(\mathbf{qsort}_1\ \mathbf{d})^3 &= \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})}\,\&\,\{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})})} \\
&\precsim \{0,1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1,M\})}
\end{aligned}
$$

This establishes the fixpoint.

This result may be seen to be disappointing since, by this analysis, we cannot eliminate the natural number argument from an object code version of the *qsort* function. The analysis detects that the first argument may be used since pattern matching is performed upon it. We do not think that it is possible for a safe analysis, even one capable of detecting the dependencies of properties between parameters, to avoid detecting the first argument as being used because of the pattern matching; the clauses constructed by pattern matching are a syntactic sugaring of a primitive recursion. Without the natural number argument it would not be possible to formulate quicksort as a primitive recursive function. We speculate that the only way to eliminate such an argument is to introduce a form of terminating general recursion, as proposed by [116].

2.13.4 Analysis of the second argument

$$\mathbf{qsort_2\ c} \quad = \quad \mathbf{at(c)}_{[\,]\sqcup \mathbf{c}_1 :: \mathbf{c}_2}$$

Where

$$\begin{aligned}
\mathbf{c}_1 &= \mathbf{AB} \sqcup \mathbf{c} \xrightarrow{(7)+\!\!+[a]+\!\!+(9)} a \\
\mathbf{c}_2 &= \mathbf{AB} \sqcup \mathbf{c} \xrightarrow{(7)+\!\!+[a]+\!\!+(9)} x
\end{aligned}$$

$$\begin{aligned}
\mathbf{c} \xrightarrow{(7)+\!\!+[a]+\!\!+(9)} a &= [(+\!\!+_1\ \mathbf{c} \xrightarrow{(7)} a) \,\&\, (+\!\!+_2\ \mathbf{c} \xrightarrow{[a]} a)] \\
&\quad \&\, (+\!\!+_2\ \mathbf{c} \xrightarrow{(9)} a) \\
+\!\!+_2\ \mathbf{c} \xrightarrow{[a]} a &= ::_1(+\!\!+_2\ \mathbf{c}) \\
&= ::_1\ \mathbf{c} \\
+\!\!+_1\ \mathbf{c} \xrightarrow{(7)} a &= \mathbf{qsort_2}(+\!\!+_1\ \mathbf{c}) \xrightarrow{\mathit{filter}\ (\mathit{lesseq}\ a)\ x} a \\
&= \mathbf{filter_1}(\mathbf{qsort_2}(+\!\!+_1\ \mathbf{c})) \xrightarrow{\mathit{lesseq}\ a} a \\
&= \mathbf{lesseq_1}\ (\mathbf{filter_1}(\mathbf{qsort_2}(+\!\!+_1\ \mathbf{c}))) \\
+\!\!+_2\ \mathbf{c} \xrightarrow{(9)} a &= \mathbf{greater_1}(\mathbf{filter_1}(\mathbf{qsort_2}(+\!\!+_2\ \mathbf{c}))) \\
&= \mathbf{lesseq_1}(\mathbf{filter_1}(\mathbf{qsort_2}(+\!\!+_2\ \mathbf{c})))
\end{aligned}$$

$$\begin{aligned}
\mathbf{c} \xrightarrow{(7)+\!\!+[a]+\!\!+(9)} x &= [(+\!\!+_1\ \mathbf{c} \xrightarrow{(7)} x) \,\&\, (+\!\!+_2\ \mathbf{c} \xrightarrow{[a]} x)] \\
&\quad \&\, (+\!\!+_2\ \mathbf{c} \xrightarrow{(9)} x) \\
+\!\!+_2\ \mathbf{c} \xrightarrow{[a]} x &= \mathbf{AB} \\
+\!\!+_1\ \mathbf{c} \xrightarrow{(7)} x &= \mathbf{filter_2}\ [\mathbf{lesseq_1}\ [\mathbf{top_0}\ [\,]]](\mathbf{qsort_2}(+\!\!+_1\ \mathbf{c})) \\
+\!\!+_2\ \mathbf{c} \xrightarrow{(9)} x &= \mathbf{filter_2}\ [\mathbf{greater_1}\ [\mathbf{top_0}\ [\,]]](\mathbf{qsort_2}(+\!\!+_2\ \mathbf{c}))
\end{aligned}$$

Note that:

$$\begin{aligned}
+\!\!+_1\ \mathbf{c} &= \mathbf{c} \\
+\!\!+_2\ \mathbf{c} &= \mathbf{c} \sqcup ::_2(\mathbf{c})
\end{aligned}$$

In particular,

$$
\begin{aligned}
+\!\!+_{\mathbf{1}}\,\{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}} &= \{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}} \\
+\!\!+_{\mathbf{2}}\,\{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}} &= \{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}}
\end{aligned}
$$

Hence we have a recursive expression in

$$\mathbf{qsort_2\,d}$$

where

$$\mathbf{d} = \{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}}$$

This may be solved by a fixpoint iteration. (We omit nullary constructors to simplify the notation.)

$$(\mathbf{qsort_2\,d})^0 = \{\}$$

$$(\mathbf{qsort_2\,d})^1 = \{1\}_{(\{0,1\}_{\mathbf{Succ}(\{1\})})::\{0\}}$$

This is so since,

$$::_{\mathbf{1}}\,\{1\}_{[],(\{1\}_{\mathbf{0},\mathbf{Succ}(\{1\})})::\{1\}} = \{1\}_{\mathbf{Succ}(\{1\})}$$

and all the other contexts must be $\{\}$, due to the preservation of contradiction property of context functions in sharing analysis.

$$(\mathbf{qsort_2\,d})^2 = \{1\}_{\mathbf{e}::\mathbf{m}}$$

where

$$
\begin{aligned}
\mathbf{e} &= \mathbf{AB} \sqcup (\{1\}_{\mathbf{Succ}(\{1\})} \,\&\, \{0,1,M\}_{\mathbf{Succ}(\{0,1,M\})}) \\
&= \{0,1,M\}_{\mathbf{Succ}(\{1,M\})}
\end{aligned}
$$

(This is due to the result for $\mathbf{filter_1}$.)

$$
\begin{aligned}
\mathbf{m} &= \mathbf{AB} \sqcup ((\mathbf{filter_2}\,[\mathbf{lesseq_1}]\,(\{1\}_{\{1\}_{\mathbf{Succ}(\{1\})}::\{\}})) \\
&\& \quad (\mathbf{filter_2}\,[\mathbf{lesseq_1}]\,(\{1\}_{\{1\}_{\mathbf{Succ}(\{1\})}::\{\}})))
\end{aligned}
$$

Now, if we let

$$\mathbf{n} = \{1\}_{\{1\}_{\mathbf{Succ}(\{1\})}::\{0\}}$$

then

$$\begin{aligned}
\mathbf{filter_2}\,[\mathbf{lesseq_1}]\,\mathbf{n} &= \{1\}_{\mathbf{r}::\mathbf{s}} \\
\mathbf{r} &= (\mathbf{lesseq_2}\,\{1\} \,\&\, (\{0,1\}_{\mathbf{Succ}(\{1\})})) \\
&\quad \sqcup (\mathbf{lesseq_2}\,\{0\} \,\&\, \{0,1\}_{\mathbf{Succ}(\{1\})}) \\
&= \{1, M\}_{\mathbf{Succ}(\{1,M\})} \\
\mathbf{s} &= \{0\} \sqcup \{1\} \\
&= \{0,1\}
\end{aligned}$$

This is so since,

$$\mathbf{lesseq_2}\,\{1\} = \{0,1\}_{\mathbf{Succ}(\{0,1\})}$$

At the next iteration we thus get the fixpoint which is

$$\{1\}_{[],\{0,1,M\}_{\mathbf{0},\mathbf{Succ}(\{0,1,M\})}::\{0,1,M\}}$$

2.13.5 Analysis of the third argument

$$\begin{aligned}
\mathbf{qsort_3\ c} &= (\mathbf{c} \xrightarrow{[]} p) \sqcup (\mathbf{c} \xrightarrow{abort_{[N]}\ p'} p) \sqcup (\mathbf{c} \xrightarrow{(3)} p) \\
&= \mathbf{AB} \sqcup \mathbf{AB} \sqcup\ ((\mathbf{qsort_3\ c} \xrightarrow{h\ a\ n\ x\ p} p) \,\&\, (\mathbf{qsort_3\ c} \xrightarrow{j\ a\ n\ x\ p} p)) \\
&= \mathbf{AB} \sqcup ((\mathbf{h_4}(\mathbf{qsort_3\ c})) \,\&\, (\mathbf{j_4}(\mathbf{qsort_3\ c})))
\end{aligned}$$

Now, as both $\mathbf{h_4}$ and $\mathbf{j_4}$ are context functions they must preserve absence and contradiction over the sharing analysis lattice. Hence we get:

$$\begin{aligned}
(\mathbf{qsort_3\ c})^1 &= \mathbf{AB} \\
(\mathbf{qsort_3\ c})^2 &= \mathbf{AB}
\end{aligned}$$

This gives us that,

$$\mathbf{qsort_3}\,\{1\} = \{0\}$$

Thus the third argument of quicksort in type theory is never used.

2.14 Formalisation of the backwards analysis

In order to make the preceeding ideas more precise, we give a formalisation of the backwards analysis in the form of a denotational abstract semantics for each of the constructs of TT. We firstly stipulate the form that contexts may take and the operations that may be applied to structured contexts. Then we present the syntax of context expressions and a semantic function from the domain of context expressions to contexts. Finally, we show how the expressions of TT are mapped to context expressions.

2.14.1 Structured contexts

In order to determine the contexts of components of data structures, structured contexts are formed via the product, $\mathbf{C} \times \mathbf{SC}$, where $\mathbf{C}$ is the context lattice set (see Section 2.4) and $\mathbf{SC}$ is the set of the structured parts, which is enumerated below. This enables types to be given to contexts and we define the **ABSENT** (abbreviated **AB**) and **CONTRA** (abbreviated **CR**) constants, and the $\sqcup$ and & operators for each type. ($\sqcup$ is defined pointwise over the structured contexts.) In each the lattice **TOP** is the dual of the **CONTRA** i.e. the structured **TOP** may be formed from the definition of the structured **CONTRA** by replacing every occurrence of **CONTRA** (from the basic context lattice) in the structure with **TOP**. These definitions are recursive in that they appeal to the definitions of **ABSENT** etc. upon the components of each structured context. (The function *ly* is used below. It is the same function as that defined in Section 2.6.5.) The approximation operator, **apx** is defined for all the recursive types — for non-recursive types it is equivalent to the identity.

The function **at**, which finds the atomic part of a context, is equivalent to the *fst* projection on $\mathbf{C} \times \mathbf{SC}$. Similarly, **str**, is equivalent to *snd*. The lattice ordering, $\sqsubseteq$, is extended to structured contexts via the lexicographical ordering on the product:

$$s_1 \sqsubseteq s_2$$

iff either

$$(\mathbf{at}\ s_1) \sqsubset (\mathbf{at}\ s_2)$$

or, if $(\mathbf{at}\ s_1) = (\mathbf{at}\ s_2)$, then for each corresponding pair of components, p_1, p_2, of the structured parts,

$$p_1 \sqsubseteq p_2$$

- $$\mathbf{C}_{Bot} = \mathbf{C} \times \{\mathbf{Path}\}$$

  $\mathbf{C}_{Bot}$ is the set of contexts for the $\perp$ type and such contexts are written:

  $$<\mathbf{c}, \mathbf{Path}>$$

  For $\mathbf{C}_{Bot}$,

  $$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, \mathbf{Path}> \\ \mathbf{CR} & = & <\mathbf{CR}, \mathbf{Path}> \\ <\mathbf{c}, \mathbf{Path}> \sqcup <\mathbf{d}, \mathbf{Path}> & = & <\mathbf{c} \sqcup \mathbf{d}, \mathbf{Path}> \\ <\mathbf{c}, \mathbf{Path}> \,\&\, <\mathbf{d}, \mathbf{Path}> & = & <(\mathbf{c} \,\&\, \mathbf{d}), \mathbf{Path}> \\ <\mathbf{c}, \mathbf{Path}> \sqsubseteq <\mathbf{d}, \mathbf{Path}> & \mathit{iff} & \mathbf{c} \sqsubseteq \mathbf{d} \end{array}$$

- $$\mathbf{C}_{Top} = \mathbf{C} \times \{\mathbf{Triv}\}$$

$\mathbf{C}_{Top}$ is the set of contexts for the $\top$ type. These contexts have the form:

$$<\mathbf{c}, \mathbf{Triv}>$$

For $\mathbf{C}_{Top}$,

$$\begin{aligned}
\mathbf{AB} &= <\mathbf{AB}, \mathbf{Triv}> \\
\mathbf{CR} &= <\mathbf{CR}, \mathbf{Triv}> \\
<\mathbf{c}, \mathbf{Triv}> \sqcup <\mathbf{d}, \mathbf{Triv}> &= <(\mathbf{c} \sqcup \mathbf{d}), \mathbf{Triv}> \\
<\mathbf{c}, \mathbf{Triv}> \,\&\, <\mathbf{d}, \mathbf{Triv}> &= <(\mathbf{c} \,\&\, \mathbf{d}), \mathbf{Triv}> \\
<\mathbf{c}, \mathbf{Triv}> \sqsubseteq <\mathbf{d}, \mathbf{Triv}> &\quad \textit{iff} \quad \mathbf{c} \sqsubseteq \mathbf{d}
\end{aligned}$$

•

$$\mathbf{C}_{Bool} = \mathbf{C} \times (\{\mathbf{True}\} \times \{\mathbf{False}\})$$

$\mathbf{C}_{Bool}$ is the set of contexts for the *Bool* type. Each context of this type is written as:

$$<\mathbf{c}, (\mathbf{True}, \mathbf{False})>$$

For $\mathbf{C}_{Bool}$,

$$\begin{aligned}
\mathbf{AB} &= <\mathbf{AB}, (\mathbf{True}, \mathbf{False})> \\
\mathbf{CR} &= <\mathbf{CR}, (\mathbf{True}, \mathbf{False})>
\end{aligned}$$

$$\begin{aligned}
<\mathbf{c}, (\mathbf{True}, \mathbf{False})> \sqcup <\mathbf{d}, (\mathbf{True}, \mathbf{False})> &= <(\mathbf{c} \sqcup \mathbf{d}), (\mathbf{True}, \mathbf{False})> \\
<\mathbf{c}, (\mathbf{True}, \mathbf{False})> \,\&\, <\mathbf{d}, (\mathbf{True}, \mathbf{False})> &= <(\mathbf{c} \,\&\, \mathbf{d}), (\mathbf{True}, \mathbf{False})> \\
<\mathbf{c}, (\mathbf{True}, \mathbf{False})> \sqsubseteq <\mathbf{d}, (\mathbf{True}, \mathbf{False})> &\quad \textit{iff} \quad \mathbf{c} \sqsubseteq \mathbf{d}
\end{aligned}$$

•

$$\mathbf{C}_{N_n} = \mathbf{C} \times (\{\mathbf{1_n}\} \times \cdots \times \{\mathbf{n_n}\})$$

$\mathbf{C}_{N_n}$ is the set of contexts for each finite type, N_n. These contexts are written:

$$<\mathbf{c}, (\mathbf{1_n}, \ldots \mathbf{n_n})>$$

For $\mathbf{C}_{N_n}$,

$$\begin{aligned}
\mathbf{AB} &= <\mathbf{AB}, (\mathbf{1_n}, \ldots \mathbf{n_n})> \\
\mathbf{CR} &= <\mathbf{CR}, (\mathbf{1_n}, \ldots \mathbf{n_n})>
\end{aligned}$$

$$\begin{aligned}
<\mathbf{c}, (\mathbf{1_n}, \ldots \mathbf{n_n})> \sqcup <\mathbf{d}, (\mathbf{1_n}, \ldots \mathbf{n_n})> &= <(\mathbf{c} \sqcup \mathbf{d}), (\mathbf{1_n}, \ldots \mathbf{n_n})> \\
<\mathbf{c}, (\mathbf{1_n}, \ldots \mathbf{n_n})> \,\&\, <\mathbf{d}, (\mathbf{1_n}, \ldots \mathbf{n_n})> &= <(\mathbf{c} \,\&\, \mathbf{d}), (\mathbf{1_n}, \ldots \mathbf{n_n})> \\
<\mathbf{c}, (\mathbf{1_n}, \ldots \mathbf{n_n})> \sqsubseteq <\mathbf{d}, (\mathbf{1_n}, \ldots \mathbf{n_n})> &\quad \textit{iff} \quad \mathbf{c} \sqsubseteq \mathbf{d}
\end{aligned}$$

•

$$\mathbf{C}_{I(A,a,b)} = \mathbf{C} \times \{\mathbf{Eq}_{I(A,a,b)}\}$$

$\mathbf{C}_{I(A,a,b)}$ is the set of contexts for each equality type, $I(A,a,b)$. The contexts are written:

$$<\mathbf{c}, \mathbf{Eq}_{I(A,a,b)}>$$

For $\mathbf{C}_{I(A,a,b)}$,

$$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, \mathbf{Eq}_{I(A,a,b)}> \\ \mathbf{CR} & = & <\mathbf{CR}, \mathbf{Eq}_{I(A,a,b)}> \end{array}$$

$$\begin{array}{rcl} <\mathbf{c}, \mathbf{Eq}_{I(A,a,b)}> \sqcup <\mathbf{d}, \mathbf{Eq}_{I(A,a,b)}> & = & <(\mathbf{c} \sqcup \mathbf{d}), \mathbf{Eq}_{I(A,a,b)}> \\ <\mathbf{c}, \mathbf{Eq}_{I(A,a,b)}> \,\&\, <\mathbf{d}, \mathbf{Eq}_{I(A,a,b)}> & = & <(\mathbf{c} \,\&\, \mathbf{d}), \mathbf{Eq}_{I(A,a,b)}> \\ <\mathbf{c}, \mathbf{Eq}_{I(A,a,b)}> \sqsubseteq <\mathbf{d}, \mathbf{Eq}_{I(A,a,b)}> & \textit{iff} & \mathbf{c} \sqsubseteq \mathbf{d} \end{array}$$

•

$$\mathbf{C}_{(\forall x:A).B} = \mathbf{C} \times \{\mathbf{Fun}\}$$

$\mathbf{C}_{(\forall x:A).B}$ and each context is written:

$$<\mathbf{c}, \mathbf{Fun}>$$

For $\mathbf{C}_{(\forall x:A).B}$,

$$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, \mathbf{Fun}> \\ \mathbf{CR} & = & <\mathbf{CR}, \mathbf{Fun}> \end{array}$$

$$\begin{array}{rcl} <\mathbf{c}, \mathbf{Fun}> \sqcup <\mathbf{d}, \mathbf{Fun}> & = & <(\mathbf{c} \sqcup \mathbf{d}), \mathbf{Fun}> \\ <\mathbf{c}, \mathbf{Fun}> \,\&\, <\mathbf{d}, \mathbf{Fun}> & = & <(\mathbf{c} \,\&\, \mathbf{d}), \mathbf{Fun}> \\ <\mathbf{c}, \mathbf{Fun}> \sqsubseteq <\mathbf{d}, \mathbf{Fun}> & \textit{iff} & \mathbf{c} \sqsubseteq \mathbf{d} \end{array}$$

•

$$\mathbf{C}_{(\exists x:A).B} = \mathbf{C} \times (\mathbf{C}_A \times \mathbf{C}_B)$$

$\mathbf{C}_{(\exists x:A).B}$ is the set of contexts for each product type. Each context of this type is written in the form:

$$<\mathbf{c}, (\mathbf{Pair\,a\,b})>$$

This may be represented graphically as:

$$\mathbf{c}_{(\mathbf{a},\mathbf{b})}$$

For $\mathbf{C}_{(\exists x:A).B}$,

$$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, (\mathbf{Pair\,CR\,CR})> \\ \mathbf{CR} & = & <\mathbf{CR}, (\mathbf{Pair\,CR\,CR})> \end{array}$$

$$\begin{array}{rcl} <\mathbf{c}, (\mathbf{Pair\,c}_1\,\mathbf{c}_2)> \sqcup <\mathbf{d}, (\mathbf{Pair\,d}_1\,\mathbf{d}_2)> & = & <(\mathbf{c} \sqcup \mathbf{d}), (\mathbf{Pair\,w\,x})> \\ <\mathbf{c}, (\mathbf{Pair\,c}_1\,\mathbf{c}_2)> \,\&\, <\mathbf{d}, (\mathbf{Pair\,d}_1\,\mathbf{d}_2)> & = & <(\mathbf{c} \,\&\, \mathbf{d}), (\mathbf{Pair\,y\,z})> \\ <\mathbf{c}, (\mathbf{Pair\,c}_1\,\mathbf{c}_2)> \sqsubseteq <\mathbf{d}, (\mathbf{Pair\,d}_1\,\mathbf{d}_2)> & \mathit{iff} & (\mathbf{c} \sqsubset \mathbf{d}) \ \mathit{or} \\ & & ((\mathbf{c} = \mathbf{d}) \ \mathit{and} \\ & & \quad (\mathbf{c}_1 \sqsubseteq \mathbf{d}_1) \ \mathit{and} \ (\mathbf{c}_2 \sqsubseteq \mathbf{d}_2)) \end{array}$$

where

$$\begin{array}{rcl} \mathbf{w} & = & \mathbf{c}_1 \sqcup \mathbf{d}_1 \\ \mathbf{x} & = & \mathbf{c}_2 \sqcup \mathbf{d}_2 \\ \mathbf{y} & = & (\mathbf{c}_1 \,\&\, \mathbf{d}_1) \sqcup (ly\ \mathbf{c}\,\mathbf{d}_1) \sqcup (ly\ \mathbf{d}\,\mathbf{c}_1) \\ \mathbf{z} & = & (\mathbf{c}_2 \,\&\, \mathbf{d}_2) \sqcup (ly\ \mathbf{c}\,\mathbf{d}_2) \sqcup (ly\ \mathbf{d}\,\mathbf{c}_2) \end{array}$$

- $$\mathbf{C}_{A \vee B} = \mathbf{C} \times (\mathbf{C}_A \times \mathbf{C}_B)$$

$\mathbf{C}_{A \vee B}$ is the set of contexts for each disjunction type. Each context of this type is written in the form:

$$<\mathbf{c}, ((\mathbf{inl}\,a), (\mathbf{inr}\,b))>$$

This has a graphical counterpart in:

$$\mathbf{c}_{(\mathbf{inl}\,a),(\mathbf{inr}\,b)}$$

For $\mathbf{C}_{A \vee B}$,

$$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, (\mathbf{inl\,CR}, \mathbf{inrCR})> \\ \mathbf{CR} & = & <\mathbf{CR}, (\mathbf{inl\,CR}, \mathbf{inrCR})> \end{array}$$

$$\begin{array}{rl} <\mathbf{c}, ((\mathbf{inl\,c}_1), (\mathbf{inr\,c}_2))> \sqcup <\mathbf{d}, ((\mathbf{inl\,d}_1), (\mathbf{inr\,d}_2))> & = \\ <(\mathbf{c} \sqcup \mathbf{d}), ((\mathbf{inl\,w}), (\mathbf{inr\,x}))> & \\ <\mathbf{c}, ((\mathbf{inl\,c}_1), (\mathbf{inr\,c}_2))> \,\&\, <\mathbf{d}, ((\mathbf{inl\,d}_1), (\mathbf{inr\,d}_2))> & = \\ <(\mathbf{c} \sqcup \mathbf{d}), ((\mathbf{inl\,y}), (\mathbf{inr\,z}))> & \\ <\mathbf{c}, ((\mathbf{inl\,c}_1), (\mathbf{inr\,c}_2))> \sqsubseteq <\mathbf{d}, ((\mathbf{inl\,d}_1), (\mathbf{inr\,d}_2))> & \mathit{iff} \\ (\mathbf{c} \sqsubset \mathbf{d}) \ \mathit{or} & \\ ((\mathbf{c} = \mathbf{d}) \ \mathit{and} \ (\mathbf{c}_1 \sqsubseteq \mathbf{d}_1) \ \mathit{and} \ (\mathbf{c}_2 \sqsubseteq \mathbf{d}_2)) & \end{array}$$

where

$$
\begin{aligned}
\mathbf{w} &= \mathbf{c}_1 \sqcup \mathbf{d}_1 \\
\mathbf{x} &= \mathbf{c}_2 \sqcup \mathbf{d}_2 \\
\mathbf{y} &= (\mathbf{c}_1 \mathbin{\&} \mathbf{d}_1) \sqcup (ly\ \mathbf{c}\, \mathbf{d}_1) \sqcup (ly\ \mathbf{d}\, \mathbf{c}_1) \\
\mathbf{z} &= (\mathbf{c}_2 \mathbin{\&} \mathbf{d}_2) \sqcup (ly\ \mathbf{c}\, \mathbf{d}_2) \sqcup (ly\ \mathbf{d}\, \mathbf{c}_2)
\end{aligned}
$$

- $$\mathbf{C}_{Nat} = \mathbf{C} \times (\{\mathbf{0}\} \times \mathbf{C})$$

  $\mathbf{C}_{Nat}$ is the set of contexts for expressions of natural number type. Such contexts are written:

  $$<\mathbf{c}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{d})>$$

  These contexts may be written in the graphical form as follows:

  $$\mathbf{c}_{\mathbf{0},\ (\mathbf{Succ}\, \mathbf{d})}$$

  For $\mathbf{C}_{Nat}$,

  $$
  \begin{aligned}
  \mathbf{AB} &= <\mathbf{AB}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{CR})> \\
  \mathbf{CR} &= <\mathbf{CR}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{CR})>
  \end{aligned}
  $$

  $$
  \begin{aligned}
  <\mathbf{c}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{c}_1)> \sqcup <\mathbf{d}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{d}_1)> &\quad = \\
  <(\mathbf{c} \sqcup \mathbf{d}), (\mathbf{0}, \mathbf{Succ}\ \mathbf{x})> & \\
  <\mathbf{c}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{c}_1)> \mathbin{\&} <\mathbf{d}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{d}_1)> &\quad = \\
  <(\mathbf{c} \mathbin{\&} \mathbf{d}), (\mathbf{0}, \mathbf{Succ}\ \mathbf{y})> & \\
  <\mathbf{c}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{c}_1)> \sqsubseteq <\mathbf{d}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{d}_1)> &\quad \textit{iff} \\
  (\mathbf{c} \sqsubset \mathbf{d})\ \textit{or} & \\
  ((\mathbf{c} = \mathbf{d})\ \textit{and}\ (\mathbf{c}_1 \sqsubseteq \mathbf{d}_1)) &
  \end{aligned}
  $$

  where

  $$
  \begin{aligned}
  \mathbf{x} &= \mathbf{c}_1 \sqcup \mathbf{d}_1 \\
  \mathbf{y} &= (\mathbf{c}_1 \mathbin{\&} \mathbf{d}_1) \sqcup (ly\ \mathbf{c}\, \mathbf{d}_1) \sqcup (ly\ \mathbf{d}\, \mathbf{c}_1) \sqcup (ly\ \mathbf{c}_1\, \mathbf{d}_1) \sqcup (ly\ \mathbf{d}_1\, \mathbf{c}_1)
  \end{aligned}
  $$

  apx maps from $\{\mathbf{0}\} \times \mathbf{C}_{Nat}$ to $\{\mathbf{0}\} \times \mathbf{C}$ as is shown:

  $$\mathbf{apx}\ (\mathbf{0}, (\mathbf{Succ} <\mathbf{c}, (\mathbf{0}, \mathbf{Succ}\ \mathbf{d})>)) = (\mathbf{0}, (\mathbf{Succ}\ (\mathbf{c} \sqcup \mathbf{d}))$$

  Note that in our definition of the & operator, **apx** has been factored in, along with the idea of using

  $$<\mathbf{c}, (\mathbf{0}, \mathbf{Succ} <\mathbf{c}_1,\ (\mathbf{0}, \mathbf{Succ}\ \mathbf{c}_1)>)>$$

  and

  $$<\mathbf{d}, (\mathbf{0}, \mathbf{Succ} <\mathbf{d}_1,\ (\mathbf{0}, \mathbf{Succ}\ \mathbf{d}_1)>)>$$

  This concept is explained in Section 2.6.6.

•

$$\mathbf{C}_{[A]} = \mathbf{C} \times (\{[]\} \times (\mathbf{C}_A \times \mathbf{C}))$$

$\mathbf{C}_{[A]}$ is the set of contexts for lists. These contexts are written:

$$<\mathbf{c}, ([], (\mathbf{cons\,h\,t}))>$$

These contexts may be written in the following graphical form:

$$\mathbf{c}_{[], (\mathbf{h} :: \mathbf{t})}$$

For $\mathbf{C}_{[A]}$,

$$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, ([], (\mathbf{cons\,CR\,CR}))> \\ \mathbf{CR} & = & <\mathbf{CR}, ([], (\mathbf{cons\,CR\,CR}))> \end{array}$$

$$\begin{array}{rl} <\mathbf{c}, ([], (\mathbf{cons\,c}_1\,\mathbf{c}_2))> \sqcup <\mathbf{d}, ([], (\mathbf{cons\,d}_1\,\mathbf{d}_2))> & = \\ <(\mathbf{c} \sqcup \mathbf{d}), ([], (\mathbf{cons\,w\,x}))> & \\ <\mathbf{c}, ([], (\mathbf{cons\,c}_1\,\mathbf{c}_2))> \,\&\, <\mathbf{d}, ([], (\mathbf{cons\,d}_1\,\mathbf{d}_2))> & = \\ <(\mathbf{c} \,\&\, \mathbf{d}), ([], (\mathbf{cons\,y\,z}))> & \\ <\mathbf{c}, ([], (\mathbf{cons\,c}_1\,\mathbf{c}_2))> \sqsubseteq <\mathbf{d}, ([], (\mathbf{cons\,d}_1\,\mathbf{d}_2))> & \textit{iff} \\ (\mathbf{c} \sqsubset \mathbf{d}) \; or & \\ ((\mathbf{c} = \mathbf{d}) \; and \; (\mathbf{c}_1 \sqsubseteq \mathbf{d}_1) \; and \; (\mathbf{c}_2 \sqsubseteq \mathbf{d}_2)) & \end{array}$$

where

$$\begin{array}{rcl} \mathbf{w} & = & \mathbf{c}_1 \sqcup \mathbf{d}_1 \\ \mathbf{x} & = & \mathbf{c}_2 \sqcup \mathbf{d}_2 \\ \mathbf{y} & = & (\mathbf{c}_1 \,\&\, \mathbf{d}_1) \sqcup (ly\; \mathbf{cd}_1) \sqcup (ly\; \mathbf{dc}_1) \sqcup (ly\; \mathbf{c}_2\mathbf{d}_1) \sqcup (ly\; \mathbf{d}_2\mathbf{c}_1) \\ \mathbf{z} & = & (\mathbf{c}_2 \,\&\, \mathbf{d}_2) \sqcup (ly\; \mathbf{cd}_2) \sqcup (ly\; \mathbf{dc}_2) \sqcup (ly\; \mathbf{c}_2\mathbf{d}_2) \sqcup (ly\; \mathbf{d}_2\mathbf{c}_2) \end{array}$$

apx maps from $\{[]\} \times (\mathbf{C}_A \times \mathbf{C}_{[A]})$ to $\{[]\} \times (\mathbf{C}_A \times \mathbf{C})$ as is shown:

$$\mathbf{apx}\, ([], (\mathbf{cons\,h} <\mathbf{t}, ([], \mathbf{cons\,d\,e})>)) = ([], (\mathbf{cons}\, (\mathbf{h} \sqcup \mathbf{d})\, (\mathbf{t} \sqcup \mathbf{e})))$$

Again, **apx** has been factored into the definition of the & operator.

•

$$\mathbf{C}_{Tree} = \mathbf{C} \times (\{\mathbf{Null}\} \times (\mathbf{C}_{Nat} \times \mathbf{C} \times \mathbf{C}))$$

$\mathbf{C}_{Tree}$ is the set of contexts for binary trees which carry natural numbers at the nodes. These contexts are written:

$$<\mathbf{c}, (\mathbf{Null}, (\mathbf{Node\,n\,l\,r}))>$$

Consequently, these contexts may be written using the following form of graphical notation:

$$\mathbf{c}_{\mathbf{Null},\,(\mathbf{Node\,n\,l\,r})}$$

For $\mathbf{C}_{Tree}$,

$$\begin{array}{rcl}
\mathbf{AB} & = & <\mathbf{AB},(\mathbf{Null},(\mathbf{Node\,CR\,CR\,CR}))> \\
\mathbf{CR} & = & <\mathbf{CR},(\mathbf{Null},(\mathbf{Node\,CR\,CR\,CR}))>
\end{array}$$

$$\begin{array}{rl}
<\mathbf{c},(\mathbf{Null},(\mathbf{Node\,c_1\,c_2\,c_3}))> \sqcup <\mathbf{d},(\mathbf{Null},(\mathbf{Node\,d_1\,d_2\,d_3}))> & = \\
<(\mathbf{c}\sqcup\mathbf{d}),(\mathbf{Null},(\mathbf{Node\,n_1\,l_1\,r_1}))> & \\
<\mathbf{c},(\mathbf{Null},(\mathbf{Node\,c_1\,c_2\,c_3}))> \,\&\, <\mathbf{d},(\mathbf{Null},(\mathbf{Node\,d_1\,d_2\,d_3}))> & = \\
<(\mathbf{c}\,\&\,\mathbf{d}),(\mathbf{Null},(\mathbf{Node\,n_2\,l_2\,r_2}))> & \\
<\mathbf{c},(\mathbf{Null},(\mathbf{Node\,c_1\,c_2\,c_3}))> \sqsubseteq <\mathbf{d},(\mathbf{Null},(\mathbf{Node\,d_1\,d_2\,d_3}))> & \mathit{iff} \\
(\mathbf{c}\sqsubset\mathbf{d})\ \mathit{or} & \\
((\mathbf{c}=\mathbf{d})\ \mathit{and}\ (\mathbf{c_1}\sqsubseteq\mathbf{d_1})\ \mathit{and}\ (\mathbf{c_2}\sqsubseteq\mathbf{d_2})\ \mathit{and}\ (\mathbf{c_3}\sqsubseteq\mathbf{d_3})) &
\end{array}$$

where

$$\begin{array}{rcl}
\mathbf{n_1} & = & \mathbf{c_1}\sqcup\mathbf{d_1} \\
\mathbf{l_1} & = & \mathbf{c_2}\sqcup\mathbf{d_2} \\
\mathbf{r_1} & = & \mathbf{c_3}\sqcup\mathbf{d_3} \\
\mathbf{n_2} & = & (\mathbf{c_1}\,\&\,\mathbf{d_1})\sqcup \\
 & & (ly\ \mathbf{c\,d_1})\sqcup(ly\ \mathbf{d\,c_1})\sqcup \\
 & & (ly\ \mathbf{c_2\,d_1})\sqcup(ly\ \mathbf{d_2\,c_1})\sqcup \\
 & & (ly\ \mathbf{c_3\,d_1})\sqcup(ly\ \mathbf{d_3\,c_1}) \\
\mathbf{l_2} & = & (\mathbf{c_2}\,\&\,\mathbf{d_2})\sqcup \\
 & & (ly\ \mathbf{c\,d_2})\sqcup(ly\ \mathbf{d\,c_2})\sqcup \\
 & & (ly\ \mathbf{c_2\,d_2})\sqcup(ly\ \mathbf{d_2\,c_2})\sqcup \\
 & & (ly\ \mathbf{c_3\,d_2})\sqcup(ly\ \mathbf{d_3\,c_2}) \\
\mathbf{r_2} & = & (\mathbf{c_3}\,\&\,\mathbf{d_3})\sqcup \\
 & & (ly\ \mathbf{c\,d_3})\sqcup(ly\ \mathbf{d\,c_3})\sqcup \\
 & & ly\ \mathbf{c_2\,d_3})\sqcup(ly\ \mathbf{d_2\,c_3})\sqcup \\
 & & (ly\ \mathbf{c_3\,d_3})\sqcup(ly\ \mathbf{d_3\,c_3})
\end{array}$$

apx maps from $\{\mathbf{Null}\}\times(\mathbf{C}_{Nat}\times\mathbf{C}_{Tree})\times\mathbf{C}_{Tree}$ to $\{\mathbf{Null}\}\times\mathbf{C}$ as is shown:

$$\begin{array}{rl}
\mathbf{apx}\ (\mathbf{Null},(\mathbf{Node\,n}<\mathbf{l},(\mathbf{Null},\mathbf{Node\,n_1\,l_1\,r_1})><\mathbf{r},(\mathbf{Null},\mathbf{Node\,n_2\,l_2\,r_2})>)) & = \\
(\mathbf{Null},(\mathbf{Node}\,(\mathbf{n}\sqcup\mathbf{n_1}\sqcup\mathbf{n_2})\,(\mathbf{l}\sqcup\mathbf{l_1}\sqcup\mathbf{l_2})\,(\mathbf{r}\sqcup\mathbf{r_1}\sqcup\mathbf{r_2}))) &
\end{array}$$

(Note that this could be applied when one of the subtree contexts is simply in **C**.) As previously, **apx** has been built into the definition of the & operator.

- $$\mathbf{C}_{Poly_A} = \mathbf{C} \times \{\mathbf{Poly}_A\}$$

  $\mathbf{C}_{Poly_A}$ is the set of contexts for polymorphic terms (with respect to a type variable, A). These contexts may be written as follows:

  $$<\mathbf{c}, \mathbf{Poly}_A>$$

  For $\mathbf{C}_{Poly_A}$,

  $$\begin{array}{rcl} \mathbf{AB} & = & <\mathbf{AB}, \mathbf{Poly}_A> \\ \mathbf{CR} & = & <\mathbf{CR}, \mathbf{Poly}_A> \end{array}$$

  $$\begin{array}{rcl} <\mathbf{c}, \mathbf{Poly}_A> \sqcup <\mathbf{d}, \mathbf{Poly}_A> & = & <(\mathbf{c} \sqcup \mathbf{d}), \mathbf{Poly}_A> \\ <\mathbf{c}, \mathbf{Poly}_A> \,\&\, <\mathbf{d}, \mathbf{Poly}_A> & = & <(\mathbf{c} \,\&\, \mathbf{d}), \mathbf{Poly}_A> \\ <\mathbf{c}, \mathbf{Poly}_A> \sqsubseteq <\mathbf{d}, \mathbf{Poly}_A> & \textit{iff} & \mathbf{c} \sqsubseteq \mathbf{d} \end{array}$$

2.14.2 Syntax of context expressions

The abstract syntax of context expressions (i.e. the domain **Cexp**) is as follows:

ce	$::=$	$\mathbf{ab}$	Absent
	$\mid$	$\mathbf{cr}$	Bottom
	$\mid$	$\mathbf{tp}$	Top
	$\mid$	$\mathbf{c}$	Context variable
	$\mid$	$\mathbf{at}(ce_1)$	Atomic part operation
	$\mid$	$ce_1 \sqcup ce_2$	Contor operation
	$\mid$	$ce_1 \,\&\, ce_2$	Contand operation
	$\mid$	$\lambda<f_1 \ldots f_j>.\lambda c.\, ce_1$	Context function definition
	$\mid$	$ce_1\ ce_2$	Context function application
	$\mid$	$\mathbf{fname_i}<a_1 \ldots a_j>$	Named context function
	$\mid$	$(f_i)_\mathbf{j}<a_1 \ldots a_k>$	Formal parameter context function
	$\mid$	$<ce_1, sce_1>$	Structured context expressions

Above, ce_i refers to an instance of ce. The abstract syntax for structured parts of context expressions is described below.

sce	$::=$	$\mathbf{Path}$	Correspondent to a pathological proof
	$\mid$	$\mathbf{Triv}$	Truth witness constructor
	$\mid$	$\mathbf{Eq}_{(A,a,b)}$	Equality witness
	$\mid$	$(\mathbf{True}, \mathbf{False})$	Booleans
	$\mid$	$(\mathbf{1}_n^1, \ldots \mathbf{n}_n^n)$	Finite types
	$\mid$	$\mathbf{pair}\; ce_1\; ce_2$	Products
	$\mid$	$(\mathbf{Inl}\; ce_1, \mathbf{Inr}\; ce_2)$	Sums
	$\mid$	$(\mathbf{0}, \mathbf{Succ}\; ce_1)$	Natural numbers
	$\mid$	$([], \mathbf{Cons}\; ce_1\; ce_2)$	Lists
	$\mid$	$(\mathbf{Null}, \mathbf{Node}\; ce_1\; ce_2\; ce_3)$	Trees
	$\mid$	$\mathbf{Poly}_A$	Polymorphic type

2.14.3 Semantics of context expressions

We give a semantics for context expressions[3]. The semantic function is:

$$\mathcal{V} : \mathbf{Cexp} \rightarrow \mathbf{ExpEnv} \rightarrow \mathbf{Cexp}^n \rightarrow \mathbf{C}_B \rightarrow \mathbf{C}_A$$

An **ExpEnv** is an environment of context function definitions i.e.

$$\mathbf{ExpEnv} : (\mathbf{FunName} \times \mathbf{Nat}) \rightarrow \mathbf{Cexp}$$

The clauses are as follows (with constants and operations being over contexts of the appropriate type):

- $$\mathcal{V}\,[\![\,\mathbf{ab}\,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} = \mathbf{AB}$$
- $$\mathcal{V}\,[\![\,\mathbf{cr}\,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} = \mathbf{CR}$$
- $$\mathcal{V}\,[\![\,\mathbf{tp}\,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} = \mathbf{TOP}$$
- $$\mathcal{V}\,[\![\,\mathbf{v}\,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} = \mathbf{c}$$

[3]Note that some of the operations (e.g. $\sqcup$) in the syntax of context expressions and their semantic counterparts have the same syntactic form. We trust that this does not cause confusion.

- $$\mathcal{V}\,[\![\, \mathbf{at}(ce_1) \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} = \mathbf{at}(\mathcal{V}\,[\![\, ce_1 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c})$$

- $$\begin{array}{lcl} \mathcal{V}\,[\![\, ce_1 \sqcup ce_2 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} & = & \\ & & \mathcal{V}\,[\![\, ce_1 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} \\ & & \sqcup \\ & & \mathcal{V}\,[\![\, ce_2 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} \end{array}$$

- $$\begin{array}{lcl} \mathcal{V}\,[\![\, ce_1 \,\&\, ce_2 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} & = & \\ & & \mathcal{V}\,[\![\, ce_1 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} \\ & & \& \\ & & \mathcal{V}\,[\![\, ce_2 \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} \end{array}$$

- $$\mathcal{V}\,[\![\, \lambda < f_1 \ldots f_m > . \lambda \mathbf{v}. ce \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c} \;=$$
$$\begin{cases} \mathbf{CR}, & \text{if the analysis is not strictness and } \mathbf{c} = \mathbf{CR} \\ \mathbf{AB}, & \text{if } \mathbf{c} = \mathbf{AB} \\ \mathcal{V}\,[\![\, ce' \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c}', & \text{if } \mathbf{AB} \sqsubseteq \mathbf{c} \\ \mathcal{V}\,[\![\, ce' \,]\!]\, \rho < e_1 \ldots e_n > \mathbf{c}, & \text{otherwise} \end{cases}$$

Where

$$\begin{array}{rcl} \mathbf{c}' & = & \mathbf{strict\ c} \\ ce' & = & ce\,[\mathbf{subst} < e_1 \ldots e_n > < f_1 \ldots f_m >] \end{array}$$

$ce\,[\mathbf{subst} < e_1 \ldots e_n > < f_1 \ldots f_m >]$ means that, assuming each e_i is of the form

$$\mathbf{fname_u} < a_1 \ldots a_j >$$

then any occurrence of

$$(f_i)_{\mathbf{v}} < b_1 \ldots b_k >$$

in ce is replaced by

$$\mathbf{fname_w} < a_1 \ldots a_j, b_1 \ldots b_k >$$

where $\mathbf{w}$ is the index formed from the sum of the numerals corresponding to $\mathbf{u}$ and $\mathbf{v}$.

- $$\mathcal{V}\,[\![\, ce_1\; ce_2 \,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} = \mathcal{V}\,[\![\, ce_1 \,]\!]\;\rho <e_1 \ldots e_n> \mathbf{c}'$$

  Where $\mathbf{c}' = \mathcal{V}\,[\![\, ce_2 \,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c}$

- $$\mathcal{V}\,[\![\, (f_i)_{\mathbf{j}} <a_1 \ldots a_k> \,]\!]\,\rho <e_1 \ldots e_k> \mathbf{c} = \mathbf{top_i} <>$$

  (Since f_i must represent a variable that could not be replaced during substitutions of context expressions. It thus represents an arbitrary context function and so must be replaced by the **top** context function.)

- $$\mathcal{V}\,[\![\, \mathbf{fname_i} <a_1 \ldots a_j> \,]\!]\,\rho <e_1 \ldots e_k> \mathbf{c} = \mathbf{fix}\,(\mathcal{V}\,[\![\, F \,]\!]\,\rho <a_1 \ldots a_j> \mathbf{c})$$

  where **fix** is the least fixpoint operator.

- $$\begin{aligned}\mathcal{V}\,[\![\, <ce_1, sce_1> \,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c} \quad &= \\ &<(\mathcal{V}\,[\![\, ce_1 \,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c}), \mathbf{apx}\,(\mathcal{V}'\,[\![\, sce_1 \,]\!]\,\rho <e_1 \ldots e_n> \mathbf{c})>\end{aligned}$$

  Where $\mathcal{V}'$ (which is not given explicitly here) is $\mathcal{V}$ mapped over each context expression in the structured part of the expression.

2.14.4 Formation of context expressions

Context expressions are formed from the expressions of TT via the function:

$$\mathcal{E} : \mathbf{TTexp} \rightarrow \mathbf{var} \rightarrow \mathbf{Cexp}$$

For each function definition of the form

$$f\; x_1 \ldots x_n \equiv_{df} E$$

we add the n-context functions of f to an environment, ρ. Each context function, $\mathbf{f_i}$ is formed by:

$$\mathcal{E}\,[\![\, E \,]\!]\, x_i$$

A modified environment, ρ', is thus formed by:

$$\rho' = [<\mathbf{f}, \mathbf{i}> \mapsto \mathcal{E}\,[\![\, E \,]\!]\, x_i]\,\rho$$

$\mathcal{E}$ is defined by the following clauses:

- $$\begin{array}{rcl} \mathcal{E} [\![p]\!] x & = & \mathbf{ab} \\ \mathcal{E} [\![abort_A \, p]\!] x & = & \mathbf{ab} \end{array}$$

  whenever $p : \bot$

- $$\mathcal{E} [\![y]\!] x = \begin{cases} \mathbf{v}, & \text{if } y \equiv x \\ \mathbf{ab}, & \text{otherwise} \end{cases}$$

- $$\mathcal{E} [\![J(c,d)]\!] x = \mathcal{E} [\![d]\!] x$$

- $$\mathcal{E} [\![case \; x \; c]\!] x = \mathcal{E} [\![c]\!] x$$

- $$\mathcal{E} [\![if \; b \; then \; c \; else \; d]\!] x = \mathcal{E} [\![b]\!] x \;\&\; (\mathcal{E} [\![c]\!] x \sqcup \mathcal{E} [\![d]\!] x)$$

- $$\mathcal{E} [\![cases_n \; v \, c_1 \ldots c_n]\!] x = \mathcal{E} [\![v]\!] x \;\&\; (\bigsqcup_{i=1}^{i=n} \mathcal{E} [\![c_i]\!] x)$$

- $$\mathcal{E} [\![Fst \; p]\!] x = <\mathbf{at}(\mathbf{v}), ((\mathcal{E} [\![p]\!] x), \mathbf{ab})>$$

- $$\mathcal{E} [\![Snd \; p]\!] x = <\mathbf{at}(\mathbf{v}), (\mathbf{ab}, (\mathcal{E} [\![p]\!] x))>$$

- $$\mathcal{E} [\![cases \; i \; f \; g]\!] x = <\mathbf{at}(\mathbf{v}), (\mathbf{inl\,a}, \mathbf{inr\,b})> \& \, \mathcal{E} [\![f]\!] x \, \& \, \mathcal{E} [\![g]\!] x$$

  In the above,

  $$\begin{array}{rcl} \mathbf{a} & = & \mathbf{efm} [\![f]\!] 1 <\mathbf{befm} [\![i]\!]> \mathbf{v} \\ \mathbf{b} & = & \mathbf{efm} [\![g]\!] 1 <\mathbf{befm} [\![i]\!]> \mathbf{v} \end{array}$$

- $$\mathcal{E} [\![ap \; f \; a]\!] x = ((\mathcal{E} [\![a]\!] x) \, (\mathbf{efm} [\![f]\!] 1 <\mathbf{befm} [\![a]\!]> \mathbf{v})) \, \& \, (\mathcal{E} [\![f]\!] x)$$

- $$\begin{aligned}\mathcal{E}\,[\![\, prim\; n\, c\, f \,]\!]\, x &= (\mathbf{prim_1}\; \mathbf{pargs}(\mathcal{E}\,[\![\, n \,]\!]\, x))\,\&\\ &\quad (\mathbf{prim_2}\; \mathbf{pargs}(\mathcal{E}\,[\![\, c \,]\!]\, x))\,\&\\ &\quad (\mathbf{prim_3}\; \mathbf{pargs}(\mathcal{E}\,[\![\, f \,]\!]\, x))\end{aligned}$$

  where $\mathbf{pargs} = <\mathbf{abs_i}<>, \mathbf{befm}\,[\![\, c \,]\!], \mathbf{befm}\,[\![\, f \,]\!]>$

- $$\begin{aligned}\mathcal{E}\,[\![\, lrec\; l\, c\, f \,]\!]\, x &= (\mathbf{lrec_1}\; \mathbf{largs}(\mathcal{E}\,[\![\, l \,]\!]\, x))\,\&\\ &\quad (\mathbf{lrec_2}\; \mathbf{largs}(\mathcal{E}\,[\![\, c \,]\!]\, x))\,\&\\ &\quad (\mathbf{lrec_3}\; \mathbf{largs}(\mathcal{E}\,[\![\, f \,]\!]\, x))\end{aligned}$$

  where $\mathbf{largs} = <\mathbf{abs_i}<>, \mathbf{befm}\,[\![\, c \,]\!], \mathbf{befm}\,[\![\, f \,]\!]>$

- $$\begin{aligned}\mathcal{E}\,[\![\, trec\; t\, c\, f \,]\!]\, x &= (\mathbf{trec_1}\; \mathbf{targs}(\mathcal{E}\,[\![\, l \,]\!]\, x))\,\&\\ &\quad (\mathbf{trec_2}\; \mathbf{targs}(\mathcal{E}\,[\![\, c \,]\!]\, x))\,\&\\ &\quad (\mathbf{trec_3}\; \mathbf{targs}(\mathcal{E}\,[\![\, f \,]\!]\, x))\end{aligned}$$

  where $\mathbf{targs} = <\mathbf{abs_i}<>, \mathbf{befm}\,[\![\, c \,]\!], \mathbf{befm}\,[\![\, f \,]\!]>$

Above, **efm** is an context expression former for functions. Its function space is

$$\mathbf{efm} : \mathbf{TTexp} \to \mathbf{Nat} \to \mathbf{Cexp}^n \to \mathbf{Cexp}$$

It is defined thus:

$$\mathbf{efm}[\![\, f \,]\!]\, i <a_1 \ldots a_n> \equiv_{df} \begin{cases} \mathbf{f_i} <a_1 \ldots a_n>, & \text{if } f \in \mathbf{Fnames}\\ \mathbf{efm}\,[\![\, g \,]\!]\,(i+1)<(\mathbf{befm}\,[\![\, b \,]\!]), a_1 \ldots a_n>, & \text{if } f \equiv ap\; g\; b\\ \mathbf{top_i}<>, & \text{if } f \in \mathbf{PMvar}\\ (f)_\mathbf{i} <a_1 \ldots a_n>, & \text{if } f \in \mathbf{FPvar}\\ \mathbf{abs_i}<>, & \text{if } f \in \mathbf{Consts}\end{cases}$$

where **Fnames**, **PMvar**, **Consts** and **FPvar** are the syntactic domains of named functions, pattern matching variables, constants and formal parameter names, respectively. **befm** is also used above. It is defined as

$$\mathbf{befm}\,[\![\, f \,]\!] \equiv_{df} \mathbf{efm}\,[\![\, f \,]\!]\, 0 <>$$

The context functions for the primitive recursive operators are of the following form:

$$\begin{aligned}\mathbf{prim_1} <n, c, f> \mathbf{c} &\equiv_{df} <\mathbf{at}(\mathbf{c}), (\mathbf{0}, \mathbf{succ\, s})>\\ \mathbf{prim_2} <n, c, f> \mathbf{c} &\equiv_{df} \mathbf{c} \sqcup (\mathbf{prim_2}\; \mathbf{prargs}\; ((f)_\mathbf{2}\; \mathbf{fargs}\; \mathbf{c}))\\ \mathbf{prim_3} <n, c, f> \mathbf{c} &\equiv_{df} \mathbf{ab} \sqcup \mathbf{c}\end{aligned}$$

Above,

$$\begin{aligned}
\mathbf{s} &= \mathbf{ab} \sqcup (((f)_1\, \mathbf{fargs\, c})\,\&\,(\mathbf{prim_1\, prargs}\,((f)_1\, \mathbf{fargs\, c})))\\
\mathbf{fargs} &= <\mathbf{abs_i} <>, (\mathbf{prim_3\, prargs})>\\
\mathbf{prargs} &= <\mathbf{abs_i} <>, (\mathbf{befm} [\![c]\!]), (\mathbf{befm} [\![f]\!])>
\end{aligned}$$

The context functions for *lrec* and *trec* are similar to the **prim** functions listed above.

The context functions $\mathbf{abs_i}$ and $\mathbf{top_i}$ are simply constant functions that always return **ABSENT** or **TOP**, respectively.

2.15 Analysis of types

In type theory, unlike in languages such as Miranda, types are "first-class citizens" i.e. types may be the inputs and results of functions. Also, terms may occur in types. Such mixing of types and terms is facilitated by two constructs of higher-order logic which occur in the theory, equality types and universes. The equality types, of the form $I(A, a, b)$, allow terms to occur within types, whilst the system of universes allows *every* type to be given a type itself. For example, $bool : U_0$ where U_0 is the base universe in a hierarchical system of non-cumulative universes.

For example, we may form the following function (taken from p.204 of [134]):

$$\begin{aligned}
nonempty &: [A] \Rightarrow U_0\\
nonempty\, [] &\equiv_{df} \bot\\
nonempty\, (a :: x) &\equiv_{df} \top
\end{aligned}$$

This definition may be taken a step further. The function ranges over a type variable A. We may quantify over this variable as follows:

$$\begin{aligned}
nonempty' &: (\forall A : U_0).[A] \Rightarrow U_0\\
nonempty'\, A\, [] &\equiv_{df} \bot\\
nonempty'\, A\, (a :: x) &\equiv_{df} \top
\end{aligned}$$

Note that the type variable may be seen to be unused with respect to the definition of function, but is required by the type definition. It should also be noted that we may go further if we admit transfinite universes, such as U_ω, since we may then range over the indices of universes.

Whilst there would appear to be less of a scope for optimisations with regard to the analysis of type information, it may be useful to determine how much a term has to be evaluated in order to typecheck another term of a dependent type. Also, expressions may be detected as being shared by a type and its associated term.

The analysis of types in terms is as before, with atomic contexts representing whether a type variable is needed or unused, strict or lazy etc. For example,

$$\mathbf{nonempty'_1 c} = (\mathbf{c} \xrightarrow{\bot} A) \sqcup (\mathbf{c} \xrightarrow{\top} A)$$

We explain context expressions such as

$$\mathbf{c} \xrightarrow{T} x$$

where the term T represents type information, in Section 2.15.1 below.

2.15.1 Analysis of terms in types

The analysis of the base types (including booleans, natural numbers and natural numbers) of the system is straightforward. These types are simply *constants* of some universe. For example, we have as the formation rule for the type of natural numbers:

$$\frac{}{N : U_0} \quad (N\ Form)$$

The pertinent point for such types is that they are formed without any premises and thus cannot depend on any terms.

We thus have,

$$\mathbf{c} \xrightarrow{B} x = \mathbf{AB}$$

where B is one of U_k (where k is a natural number), $\bot$, $\top$, *bool*, N_n, N and *tree*.

For type variables, such as V, the context propagated is just the same as for the original case i.e.

$$\mathbf{c} \xrightarrow{V} x = \begin{cases} \mathbf{c}, & \text{if } V \equiv x \\ \mathbf{AB}, & \text{otherwise} \end{cases}$$

If we have a list type, denoted $[A]$, then the context propagated is just that for the type carried by the list i.e.

$$\mathbf{c} \xrightarrow{[A]} x = \mathbf{c} \xrightarrow{A} x$$

Disjunction types and non-dependent products and function spaces are all dealt with similarly: the resulting context is the combination of the contexts propagated through the types that are used to form the type e.g.

$$\mathbf{c} \xrightarrow{A \vee B} x = (\mathbf{c} \xrightarrow{A} x) \,\&\, (\mathbf{c} \xrightarrow{B} x)$$

The reason for the above is that a type constructor such as $\Rightarrow$ is just like a pairing operation for terms.

Likewise, equality types are the combination of the contexts propagated through each of their three components i.e.

$$\mathbf{c} \xrightarrow{I(A,a,b)} x = (\mathbf{c} \xrightarrow{A} x) \,\&\, (\mathbf{c} \xrightarrow{a} x) \,\&\, (\mathbf{c} \xrightarrow{b} x)$$

Suppose we have a dependent type which is defined via functions with universes such as *nonempty* above. These dependent types induce context functions as before e.g.

$$\mathbf{nonempty_1\ c} = \mathbf{at(c)}_{[], (\mathbf{AB} :: \mathbf{AB})}$$

In general, we use the notation $\mathbf{P_i}$ to denote the ith context function of a dependent type, P, and equivalently we use the notation $\mathbf{P}_x$ to denote a context function of a formal parameter x.

There are also context functions **induced by the quantifiers**. For example, if we have

$$(\forall x : A).B$$

then there is an induced context function, $\mathbf{B}_x$, which is defined as

$$\mathbf{B}_x\ \mathbf{c} \equiv_{df} \mathbf{c} \xrightarrow{B} x$$

Such context functions are invoked whenever a substitution is made into a type family. For example, one of the elimination rules given in [134] for the existential type is as follows:

$$\frac{p : (\exists x : A).P}{Snd\,p :\ P[Fst\,p/x]} \quad (\exists\ Elim_S)$$

Now,

$$\mathbf{c} \xrightarrow{P[Fst\,p/x]} y = (\mathbf{Fst_1}\ (\mathbf{P}_x\ \mathbf{c})) \xrightarrow{p} y$$

In general, for such a substitution in a dependent type we have,

$$\mathbf{c} \xrightarrow{A[s/y]} x = (\mathbf{A}_y\ \mathbf{c}) \xrightarrow{s} x$$

The general form for the type of a function is:

$$(\forall x_1 : A_1).\ldots(\forall x_n : A_n).B$$

For each function parameter, x_i, bound by the quantification, the context of x_i with respect to the type of the function is just:

$$\mathbf{P}_{x_i}\ \mathbf{c}$$

where P is just B if $i = n$ or, otherwise,

$$(\forall x_{i+1} : A_{i+1}).\ldots(\forall x_n : A_n).B$$

In the case of a variable which is not bound in the quantified type, the context propagated is just the combination of the contexts resulting from each of the constituent types e.g.

$$\mathbf{c} \xrightarrow{(\exists a : A).P} x = (\mathbf{c} \xrightarrow{A} x)\,\&\,(\mathbf{c} \xrightarrow{P} x)$$

To denote the context function with respect to the type information, a superscript, $\mathbf{T}$, is used. For example, we have,

$$\mathbf{nonempty'}_1^{\mathbf{T}}\ \mathbf{c} \quad = \quad \mathbf{c} \xrightarrow{(\forall A : U_0).[A] \Rightarrow U_0} A$$

$$
\begin{aligned}
&= (\mathbf{c} \xrightarrow{U_0} A) \,\&\, (\mathbf{c} \xrightarrow{[A] \Rightarrow U_0} A) \\
&= \mathbf{AB} \,\&\, (\mathbf{c} \xrightarrow{[A] \Rightarrow U_0} A) \\
&= (\mathbf{c} \xrightarrow{[A]} A) \,\&\, (\mathbf{c} \xrightarrow{U_0} A) \\
&= (\mathbf{c} \xrightarrow{A} A) \,\&\, \mathbf{AB} \\
&= \mathbf{c}
\end{aligned}
$$

Consequently, the first argument to *nonempty′* is needed with respect to the typing information.

2.16 Conclusion

We have developed a theory of backwards analysis, based upon the work of Hughes, which is capable of automatically detecting computational irrelevance within a type theoretic program. We chose the backwards analysis form of abstract interpretation due to its ability to capture abstract properties of data structures more precisely than forwards analysis and for its superior efficiency in the first-order case.

It has been shown that a hierarchy of analyses, including neededness, may be employed to gain information about different properties of a program. This hierarchy culminates in *sharing analysis*, which subsumes both neededness and strictness analysis, and also indicates whether an expression may be shared during computation. Thus, as well as removing computational redundancy, we can perform optimisations on a type theoretic program due to information that results from just a single backwards analysis.

Subjects for possible further investigation in the areas which we have described are the analysis of higher-order functions which produce or apply functions contained within data structures, and the analysis of polymorphic functions.

We have described the application of backwards analysis to the whole of the system TT described in [134], including the *list* and (binary) *tree*, well-founded types. Future work may focus upon applying the techniques which we have described to the general case of well-founded types, the W-types, which are described in Section 5.10.2 of [134]. Related to this would be a study of the backwards analysis of possible schemes for well-founded recursion in type theory [116, 109, 121]. There is also scope for work on inductively defined [100, 46] and co-inductive types [134, Section 7.11] which are the least and greatest fixpoints of recursive type equations, respectively.

Chapter 3

Correctness of the neededness analysis

3.1 Introduction

In this chapter the formal rules of Martin-Löf's intuitionistic type theory and a characterisation of neededness of expressions in the theory are given. In particular, it is shown how the backwards analysis that has been developed may be demonstrated to be *safe* with respect to neededness i.e. consistent with the semantics of type theory.

It is necessary to be able to show that the analysis that has been developed is correct (i.e. the abstract information that is deduced is consistent with the original semantics of the program being analysed) since if it is not then there will be potentially catastrophic and unpredictable consequences for the program optimised as a result of the analysis. It may well be the case that the resulting program may not be strongly normalising if we incorrectly remove arguments that are, in fact, needed by the computation.

In order to define safety rigorously, a definition of an unused function argument must first be given. Since this property of neededness is undecidable, we cannot show that the analysis will always detect an unused argument (i.e. that the analysis is *complete*). Instead it remains to prove that the analysis is *sound* i.e. that any function which does require an argument, x, in order to be evaluated, will be shown by the analysis to have x as a needed parameter. This proof is done for each of the constructs of type theory.

Our characterisation of an unused argument is slightly different from that usually presented for functional programming languages. For instance, a function is termed *strict* in its argument *iff*:

$$f \perp = \perp$$

where $\perp$ is the undefined element that inhabits every semantic domain. Similarly, a function parameter is termed *unused* or absent *iff*:

$$f\ a = f \perp$$

for every possible a. However, we do not have the element $\perp$ inhabiting every type in TT. Hence the idea of a computationally absent parameter has to be modified. Furthermore, we need to ensure safety at two levels. The first level is the *atomic* one, where the need to evaluate parts of the sub-structure of a parameter is not considered. It then remains to examine the *structured* level, where the safety of the analysis with respect to the *components* (e.g. the head of a list) of a parameter is considered.

We do not prove the safety of the analysis with respect to strictness since TT is strongly normalising and has the Church-Rosser property. This means that every reduction sequence for a term must terminate with the same normal form. Consequently, in TT, unlike in programming languages such as Miranda, denoting a function as being strict in its argument cannot affect the semantics of a type theoretic program. In this sense, therefore, strictness analysis must be safe with respect to type theory.

3.2 Definitions and theorems

In this section we present the main definitions which will characterise our safety argument and the main theorems of this chapter.

3.2.1 Abstractions of context lattices

In the discussion that follows we shall simply be concerned with the contexts $\mathbf{N}$ and $\mathbf{U}$ i.e. whether a parameter (or component of a parameter) is needed or unused. Information on whether a parameter is needed is embedded within the sharing and strictness-and-absence lattices: we need to perform an abstraction on the contexts of these lattices to give contexts in $\{\mathbf{N}, \mathbf{U}\}$. These **abstractions of the context lattices** are as follows for atomic values:

$$\mathbf{abscxt}\ c = \begin{cases} \mathbf{U}, & \text{if } c \sqsubseteq \mathbf{AB} \\ \mathbf{N}, & \text{otherwise} \end{cases}$$

So, for instance, $\mathbf{abscxt}\{0\} = \mathbf{U}$ and $\mathbf{abscxt}\{0,1\} = \mathbf{N}$. It may be noted that the context lattices that we have used are all abstractions of the eight-point sharing analysis lattice (based on the power set of $\{0, 1, M\}$). Note, for example, that we would map both points of the simple strictness lattice, $\mathbf{S}$ (which corresponds to non-empty subsets of $\{1, M\}$ in the sharing lattice) and $\mathbf{L}$ ($\{0,1\}, \{0, M\}$ and $\{0, 1, M\}$) to $\mathbf{N}$.

The idea of abstractions of context domains/lattices comes from [69].

3.2.2 Basic definitions

Definition 21

We say that a single parameter function f, of the generalised function space type $(\forall x : A).B$, is **independent** of its argument *iff*

1.
$$B(a) = B(b)$$

2.
$$f\ a = f\ b$$

for any a,b of type A. This can be expressed within TT in that f is independent of its argument if we can derive:

$$(\forall a, b : A).((I(U_k, B(a), B(b))) \wedge (I(B(a), (f\ a), (f\ b))))$$

where U_k is a universe containing both $B(a)$ and $B(b)$.

Equivalently, we say that f's first parameter is **unused**.

Note that we are primarily concerned with *term reduction* rather than type checking, which we shall assume has been done as a separate phase. Consequently, the use of the input element within the *type* of the output shall not be considered in general. Also, the definition of independence ensures that if a parameter is unused then we will be dealing with the non-dependent function space. Below, therefore, we shall use $A \Rightarrow B$ to mean the generalised function space, $(\forall x : A).B$, where the *type* of the result depends upon the input element. Also, in the definitions which follow, we shall implicitly assume that the type equality of condition (1) holds so that the assertions of equality between applications of a function to different arguments is meaningful.

The above may be extended naturally to functions of more than one argument.

Definition 22

If there exists a j such that

$$f\ a_1 \ldots a_j \ldots a_n = f\ b_1 \ldots b_j \ldots b_n$$

whenever $a_i = b_i$ for all i such that $1 \leq i \leq n$ and $i \neq j$, then we say that f is **independent of its jth argument** (parameter).

Also, we say that the jth argument (parameter) is unused.

Evidently, there is the converse definition:

Definition 23

f is **dependent** upon its jth parameter if it is not independent of that argument.

Similarly, we say that the jth argument of f is **needed** if it is not unused.

If f is independent of its argument then we may remove its formal parameter in its definition to form a new function f'. Similarly, we may replace a call $f\ c$, say, with f'. Since it is intended to produce a modified form of f with the unused parameters removed, it is essential that the analysis only detects arguments which are *definitely* unused. Otherwise, we are certain to produce an "optimised" program which gives incorrect results.

The analysis uses the (atomic) abstract values:
N ("May be needed if the result of the function is needed")
and
U ("Always unused")

It is necessary to show that if a function's jth argument is needed then the backwards analysis will show that the context for that parameter will be **N**. We shall use the symbol

$$\mathcal{N}$$

to denote a context whose atomic part is **N**.

An abstraction map, **abstr**, from the definition of (in)dependent parameters in the concrete semantics to the neededness context lattice is defined as follows:

Definition 24

$$\mathbf{abstr}\ fj = \begin{cases} \mathbf{N}, & \text{if } f \text{ is dependent on its } j\text{th parameter} \\ \mathbf{U}, & \text{if } f \text{ is independent of its } j\text{th parameter} \end{cases}$$

Definition 25 (Safety)
The analysis is safe, with respect to neededness, if whenever the jth argument of f is needed then

$$\mathbf{at}(\mathbf{f_j}\ \mathcal{N}) = \mathbf{N}$$

where $\mathbf{f_j}$ is the context function of f's jth parameter.

From definitions 24 and 25 we obtain:

Result 14
The analysis is safe if and only if:

$$\mathbf{abstr}\ fj \sqsubseteq \mathbf{at}(\mathbf{f_j}\ \mathcal{N})$$

Proof
Suppose the contrary i.e.

$$\mathbf{at}(\mathbf{f_j}\ \mathcal{N}) \sqsubset \mathbf{abstr}\ fj$$

Now,

$$\mathbf{at}(\mathbf{f_j}\ \mathcal{N}) \sqsubset \mathbf{abstr}\ fj$$

iff $\mathbf{at}(\mathbf{f_j}\ \mathcal{N}) = \mathbf{U}$ and $\mathbf{abstr}\ fj = \mathbf{N}$
iff $\mathbf{at}(\mathbf{f_j}\ \mathcal{N}) = \mathbf{U}$ and f is dependent on its jth parameter.

iff the analysis is unsafe.

□

So, by the above result, safety may be proved by confirming that the inequality holds.

3.2.3 Main theorems

We present the two main theorems which are relevant to establishing the correctness of the analysis. Firstly, we show that the property of absence is undecidable so that there does not exist an algorithm with which to calculate the **abstr** mapping. As a consequence, it is impossible for our neededness analysis to be complete for TT in the sense that there will be some programs for which absence cannot be determined precisely. Secondly, we state the safety result that the analysis does not detect parameters (or components of parameters) as being unused when in fact they may be used. This is a soundness result in the sense that for every function that we can prove, by the analysis, that it does not use its argument it will be valid to do so.

Result 15 (Undecidability of the absence property)
The absence property is undecidable.

Proof
The undecidability of the absence property is a consequence of the undecidability of extensional equality, which we shall demonstrate. Suppose that we have a function to simulate the operation of a Turing machine in TT over a finite number of steps, which will return *True* if it halts within that number of steps. That is, we have the following function:

$$\begin{array}{rcl} \mathit{turingsim} & : & N \Rightarrow \mathit{TuringMach} \Rightarrow \mathit{Bool} \\ \mathit{turingsim}\; n\; t & \equiv_{df} & \begin{cases} \mathit{True}, & t \text{ halts in } n \text{ steps} \\ \mathit{False}, & \text{otherwise} \end{cases} \end{array}$$

In the above, *TuringMach* is the type of Turing machine representations. Suppose that we can compute **abstr** for any function and any parameter. Then we will be able to calculate:

$$\mathbf{abstr}\; \mathit{turingsim}\; 1$$

That is, we will be able to determine whether

$$\mathit{turingsim}\; m\; t = \mathit{turingsim}\; n\; t$$

for all m and n, for an arbitrary t. If this held then we would have a solution to the halting problem. This is impossible and so our original assumption about the decidability of **abstr** must be false.

□

Result 16 (Correctness of the neededness analysis)
Our neededness analysis is safe with respect to the absence property. That is, if the analysis detects a parameter as being unused then that parameter will not be required during computation with a lazy evaluation strategy. Formally,

$$\mathbf{abstr}\ f\ j \sqsubseteq \mathbf{at}(\mathbf{f_j}\ \mathcal{N})$$

Also, where the data is structured, the analysis is sound for each component of the data. That is,

$$\mathbf{abstr}_{prj}\ f\ j \sqsubseteq (\mathbf{at} \circ \mathbf{prj} \circ \mathbf{str} \circ \mathbf{f_j})\ \mathcal{N}$$

In the above, $\mathbf{abstr}_{prj}$ is the abstraction function with respect to the component of the parameter extracted via the projection, *prj*. **prj** is the projection over contexts which is the counterpart to *prj*.

The rest of this chapter will be devoted to proving Result 16. This proof will be performed using the method presented in Section 3.2.4.

3.2.4 Method of proof

The approach is to consider the base types of the system and then to examine types constructed from other types. In proving the safety of our necessity analysis we shall thus appeal to the *principle of induction over types*. This consists of showing that our analysis gives a safe analysis for:

1. Functions over the base types of the system.

2. *Assuming* that safety holds for types $A_1 \ldots A_n$ showing that it holds for a type B constructed from $A_1 \ldots A_n$ and for functions whose inputs are from $A_1 \ldots A_n$.

We shall use $\mathcal{N}_C$ to indicate an input context corresponding to a type C where the atomic part is $\mathbf{N}$ and where any components of the structured part of $\mathcal{N}_C$ are also $\mathbf{N}$ at the atomic level. In other words, we are simply assuming that the result of a function is needed. (The analysis will be trivially safe in the case that the result is not needed.)

If the type is structured then we will additionally prove safety for each of the components of the data structure. (Note that the natural numbers, being based on the Peano arithmetic system, are considered to be structured here.)

3.3 Base types

The first step in proving correctness is to examine the base types of the system i.e. those which are not constructed from other types.

3.3.1 The empty type, $\perp$

$\perp$, which may be interpreted as the absurd proposition, is not inhabited by any elements and consequently does not have an introduction rule. Its formation rule is:

$$\frac{}{\perp : U_0} \quad (\perp\ \mathit{Form})$$

Its elimination rule is:

$$\frac{p : \perp}{\mathit{abort}_A\ p} \quad (\perp\ \mathit{Elim})$$

In $\mathit{abort}_A\, p$, the p occurs in order to adhere to the *principle of complete presentation*. Essentially, due to the notion of "ex falso quodlibet", $\mathit{abort}_A\ p$ may represent *any* element of type A. $\mathit{abort}_A\ p$, of course, has no computational content: it is nonsensical to try to reduce a term which has resulted from a pathological proof object that should have never been derived. For any given type, A, we may say that all "abort_A" terms are equivalent so that if

$$\begin{aligned} f\ a &= \mathit{abort}_A\, p \\ f\ b &= \mathit{abort}_A\, q \end{aligned}$$

then $f\ a = f\ b$. Furthermore, since $\perp$ is uninhabited, the condition that, in the case that $f\ :\ \perp \Rightarrow C$,

$$f\ a = f\ b$$

for *any* $a, b : \perp$, is vacuously satisfied as $\perp$ is uninhabited. ($\perp$ is isomorphic to the empty set.) Hence, if $f\ :\ \perp \Rightarrow C$, then f must be independent of its first argument. Thus the safety condition must also hold as

$$\mathbf{abstr}\ f\ j = \mathbf{U}$$

where the jth parameter is of type $\perp$.

3.3.2 The single-element type, $\top$

The type $\top$, which may be viewed as the "true" proposition, contains just one element, $Triv$. Its formation, introduction, elimination and computation rules are as follows:

$$\frac{}{\top : U_0} \quad (\top\ \mathit{Form}) \qquad \frac{}{\mathit{Triv} : \top} \quad (\top\ \mathit{Intro})$$

$$\frac{x : \top \qquad c : C(\mathit{Triv})}{\mathit{case}\ x\ c : C(x)} \quad (\top\ \mathit{Elim})$$

$$\mathit{case}\ \mathit{Triv}\ c\ \rightarrow\ c$$

Clearly, as there is only one inhabitant of $\top$,

$$f\ a = f\ b$$

for any elements $a, b : \top$, as necessarily $a = b = Triv$. This is exhibited in the computation rule for *case* which is independent of its first argument and consequently:

$$\mathbf{abstr}\ case\ 1 = \mathbf{U}$$

Hence the safety condition is satisfied with regard to the first argument to *case*.

For the second argument of *case*, the abstract interpretation gives the following:

$$\mathbf{at}(\mathbf{case_2}\ \mathcal{N}) = \mathbf{N}$$

Hence, it must follow that the safety condition is met i.e.

$$\mathbf{abstr}\ case\ 2 \sqsubseteq \mathbf{at}(\mathbf{case_2}\ \mathbf{N})$$

3.3.3 Booleans

The boolean type is the finite type inhabited by two elements denoted *True* and *False*. Its formation and introduction rules are:

$$\frac{}{bool : U_0}\ (bool\ Form)$$

$$\frac{}{True : bool}\ (bool\ Intro_t) \qquad \frac{}{False : bool}\ (bool\ Intro_f)$$

Its elimination rule is

$$\frac{tr : bool \qquad l : C[True/x] \qquad d : C[False/x]}{if\ tr\ then\ l\ else\ d\ : C[tr/x]}\ (bool\ Elim)$$

and the computation rule is defined via pattern matching on the boolean argument.

$$\begin{array}{lcl} if\ True\ then\ l\ else\ d & \rightarrow & l \\ if\ False\ then\ l\ else\ d & \rightarrow & d \end{array}$$

Here, the abstract interpretation results in the (atomic) context $\mathbf{N}$ for each of the three arguments to the *if-then-else* construct. ($\mathbf{N}$ results for the first argument due to pattern matching.) Consequently, the abstract interpretation is safe over the booleans.

3.3.4 Finite types in general

We denote N_n to be the finite type inhabited by exactly n elements. The finite types are formed and elements of N_n are introduced by the following rules:

$$\frac{}{N_n : U_0}\ (N_n\ Form) \qquad \frac{}{1_n : N_n}\ (N_n\ Intro_1) \qquad \cdots \qquad \frac{}{n_n : N_n}\ (N_n\ Intro_n)$$

The elimination rule is:

$$\frac{e : N_n \qquad l_1 : C[c_1/x] \ldots l_n : C[c_n/x]}{cases_n\ e\ c_1 \ldots c_n : C[e/x]}\ (N_n\ Elim)$$

There are n associated parts to the computation rule:

$$\begin{array}{rcl} cases_n\ 1_n\ c_1 \ldots c_n & \rightarrow & c_1 \\ \vdots & \vdots & \vdots \\ cases_n\ n_n\ c_1 \ldots c_n & \rightarrow & c_n \end{array}$$

As in the boolean case, the abstract interpretation will show that, for all $\mathbf{i}$, when $n \geq 2$,

$$\mathbf{at}((\mathbf{cases}_n)_\mathbf{i}\ \mathcal{N}) = \mathbf{N}$$

This is so since c_i appears as the result to one of the parts of computation rule and pattern matching is performed on the element of type N_n. Again, this indicates that the analysis must be safe on N_n.

3.4 Inductive cases

We now examine the inductive cases. That is, we prove the safety property for the cases of types that are constructed from other types (products, lists, trees, ...), or where the safety argument with respect to the computation rule of the type depends upon the hypothesis that the safety result holds over the type of the argument which is being abstracted. In most cases, apart from the equality and function types, the data will be structured so that it will be necessary to consider correctness with regard to the *components* of parameters as well as the parameters themselves.

3.4.1 Equality types

The equality types form the primitive predicates upon each type A. They are written as

$$a =_A b$$

or

$$I(A, a, b)$$

The above means that equality between two elements a and b of the type A holds.

The formation rule is the first that we have exhibited where terms may be embedded in the type:

$$\frac{A : U_n \qquad a : A \qquad b : A}{I(A, a, b) : U_n} \quad (\textit{Equality Form})$$

The introduction rule depends upon the relation of convertibility between terms. An explanation of this equivalence relation[1] ($\twoheadleftarrow\!\twoheadrightarrow$) is given on pp. 118-119 of [134].

$$\frac{a \twoheadleftarrow\!\twoheadrightarrow b \qquad a : A \qquad b : A}{r(a) : I(A, a, b)} \quad (\textit{Equality Intro})$$

[1]Briefly, $\twoheadleftarrow\!\twoheadrightarrow$, is the smallest equivalence relation extending the reduction relation, $\twoheadrightarrow$. The latter is the reflexive, transitive closure of the computation rules that we are presenting.

$r(a)$ *witnesses* the equality between two terms.

The elimination rule for the equality types which embodies Leibnitz's law (that "equals may be substituted for equals") is:

$$\frac{c : I(A, a, b) \qquad d : C(a, a, r(a))}{J(c, d) : C(a, b, c)} \quad (\textit{Equality Elim})$$

The computation rule is:

$$J(c, d) \rightarrow d$$

Note the similarity in this computation rule and that for $\top$. In fact, it can be shown (p. 117 of [134]) that all elements of the equality type are equal. Moreover, if we regard the selector J as a two-parameter function (by currying) then we see that J is independent of its first argument. The analysis must consequently be safe with respect to the equality witness, c. Conversely, $\mathbf{at}(\mathbf{J}_2\, \mathcal{N}) = \mathbf{N}$ in our analysis and so safety is guaranteed in this case as well.

3.4.2 Product types

In the case of products it is sufficient to consider the generalised type of products, $(\exists x : A) . B$ i.e. where the type of the second component of the pair may depend upon the first component. The formation and introduction rules are:

$$\frac{A : U_m \qquad \begin{array}{c}[x : A] \\ \vdots \\ P : U_n\end{array}}{(\exists x : A).P : U_{max(m,n)}} \quad (\exists\ \textit{Form}) \qquad \frac{a : A \qquad p : P[a/x]}{(a, p) : (\exists x : A).P} \quad (\exists\ \textit{Intro})$$

There are two elimination rules for this type (which can be shown to be equivalent to the single elimination rule, ($\exists$ E), given on p.139 of [134]):

$$\frac{p : (\exists x : A).P}{\textit{Fst}\ p : A} \quad (\exists\ \textit{Elim}_f) \qquad \frac{p : (\exists x : A).P}{\textit{Snd}\ p : P[\textit{Fst}\ p/x]} \quad (\exists\ \textit{Elim}_s)$$

The computation rules for products are:

$$\begin{array}{rcl} \textit{Fst}(a, b) & \rightarrow & a \\ \textit{Snd}(a, b) & \rightarrow & b \end{array}$$

For each of the computation rules, the analysis detects that the parameter of type $\exists x : A . B$ must be used — due to the pattern matching over the pair. That is (for the atomic parts of the analysis),

$$\mathbf{Fst}_1\, \mathcal{N} = \mathbf{N}$$

and

$$\mathbf{Snd}_1\, \mathcal{N} = \mathbf{N}$$

However, the analysis also produces contexts for the components of the pair. Consequently, it is necessary to expand the definition of a function being independent of its arguments.

We may say that a function is independent of *components* of its argument as follows:

Definition 26 (Dependent product — second component)
Suppose that:

1. $f : (\exists x : A \,.\, B) \Rightarrow C$
2. $a : A$, where a is arbitrary.

Then, if
$$f(a, b) = f(a, c)$$
for *any* $b, c : B(a)$, we say that
f is **independent of the second component of its argument**.

The case of the first component of the generalised product is more complicated due to the possible variation in the type of the second.

Definition 27 ((*Non*-)dependent product — first component)
Suppose that:
$$f : (\exists x : A \,.\, B) \Rightarrow C$$
If then, for *any* $a, b : A$,
$$B(a) \equiv B(b)$$
(i.e. we have a non-dependent product which is isomorphic to $A \wedge B$) and for an arbitrary $c : B$,
$$f(a, c) = f(b, c)$$
then we say that
f is **independent of the first component of its argument** (of non-dependent product type).

Definition 28 (Dependent product — first component)
If, $f : (\exists x : A \,.\, B) \Rightarrow C$ and
$$f(a, b) = f(c, d)$$
for *any* $a, c : A, b : B(a), d : B(c)$, we say that
f is **independent of both components of its argument**.

Note that, if *we consider type checking and term reduction as one indivisible process* it is impossible for a function to be independent of the first component of a product if the second component is needed and its type depends upon the first component. However, we are primarily concerned with the behaviour of the reduction of terms of the system and assume that type checking has already been completed as an independent phase. (This is the case in the Ferdinand system, for instance.) Making this assumption, we may apply the definition 27 to *all* elements of product types, even if the type of the second component depends upon the first.

We have, naturally, the dual notion of *dependence* upon components of a pair and the abstraction map, **abstr**, is extended to the components of products as follows:

Definition 29

$$\mathbf{abstr}_{fst} f j = \begin{cases} \mathbf{N}, & \text{if } f \text{ is dependent on the first component of its } j\text{th parameter} \\ \mathbf{U}, & \text{if } f \text{ is independent of the first component of its } j\text{th parameter} \end{cases}$$

Likewise, we may define $\mathbf{abstr}_{snd}\, f j$.

Definition 30 (Safety for components of products)
We say that, for a function $f : (\exists x : A\,.\,B) \Rightarrow C$, the analysis is safe with regard to the components of the input parameter *iff*:

$$(\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{f}_1)\, \mathcal{N}_C = \mathbf{U}$$

implies that f is independent of its first component of its parameter.
Above,
str extracts the structured part of a context.
fst projects the first component of the structured part of a context i.e.

$$\mathbf{fst} \;:\; \mathbf{C}_A \times \mathbf{C}_B \rightarrow \mathbf{C}_A$$

$\mathcal{N}_C$ is a needed element of the structured contexts corresponding to the type C i.e. with atomic part equal to $\mathbf{N}$.

We have a similar definition of safety for the second component of an element of product type. (The context projection **snd** simply needs to be substituted for **fst** in the above.)

As in the basic, atomic case, the above definition implies that safety is assured if and only if:

$$\mathbf{abstr}_{fst}\, f j \sqsubseteq (\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C$$

and

$$\mathbf{abstr}_{snd}\, f j \sqsubseteq (\mathbf{at} \circ \mathbf{snd} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C$$

If we assume that type checking has been done as a separate phase, then we see that *Fst* and *Snd* are independent of the second and first components, respectively, of their parameters. Conversely, the analysis gives:

$$\mathbf{Fst}_1\, (\mathcal{N}_A) = \mathbf{N}_{(\mathcal{N}_A, \mathbf{U})}$$

and so

$$(\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{Fst}_1)\, \mathcal{N}_A = \mathbf{N}$$

Similarly,

$$(\mathbf{at} \circ \mathbf{snd} \circ \mathbf{str} \circ \mathbf{Snd}_1)\, \mathcal{N}_B = \mathbf{N}$$

Hence, the analysis is safe with respect to the components of the pair in each case as well.

3.4.3 Disjunction types

The cases where the input type of a function is that of a disjunction type (i.e. $A \vee B$) are relatively straightforward. These types are formed as follows:

$$\frac{A : U_m \qquad B : U_n}{(A \vee B) : U_{max(m,n)}} \quad (\vee\, Form)$$

Elements of disjunction types are introduced by the following rules:

$$\frac{q : A}{inl\; q : (A \vee B)} \quad (\vee\, Intro_l) \qquad\qquad \frac{q : B}{inr\; q : (A \vee B)} \quad (\vee\, Intro_r)$$

The elimination rule is as follows:

$$\frac{p : (A \vee B) \qquad q : (\forall x : A).C[inl\, x/z] \qquad r : (\forall y : B).C[inr\, y/z]}{cases\; p\, q\, r : C[p/z]} \quad (\vee\, Elim)$$

The computation rule is defined via pattern matching on the proof object of the disjunction type:

$$\begin{array}{rcl} cases\; (inl\; a)\, q\, r & \rightarrow & q\, a \\ cases\; (inr\; b)\, q\, r & \rightarrow & r\, b \end{array}$$

This pattern matching means that the analysis shows that the first parameter of *cases* is needed whenever the result of *cases* is needed. The analysis also indicates that the two other arguments will also be required. Hence, at the atomic level, the analysis must be safe with regard to disjunction types.

It is also necessary to check safety within the structure of the disjunctive proof object.

Definition 31 (Disjunction type)
If $f : (A \vee B) \Rightarrow C$ and for *any* $a, b : A$:

$$f(inl\; a) = f(inl\; b)$$

Then we say that f is **independent of a left injection component of its parameter**

A similar definition may be formulated for a right injection component.

The **abstr** mapping is extended naturally, in a similar manner to that done for product types, and the analysis will be safe for a function f with jth parameter of type $A \vee B$ if and only if:

$$\mathbf{abstr}_{inl}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C \tag{10}$$

and

$$\mathbf{abstr}_{inr}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{snd} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C \tag{11}$$

The jth parameter of the function f must reduce to the forms *inl a* and *inr b* in (10) and (11), respectively. (Note that, as with the product, **fst** and **snd** are used as the projections from the context domain $\mathbf{C}_{A \vee B}$ since the structured part of that domain consists of pairs of contexts from $\mathbf{C}_A$ and $\mathbf{C}_B$.)

It follows from the definitions of *cases* and **abstr** that

$$\mathbf{abstr}_{inl}\, cases\, 1 = \mathbf{abstr}\, q\, 1$$

Also, the analysis gives the result:

$$\mathbf{at}(\mathbf{q_1}\, \mathcal{N}_C) \sqsubseteq (\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{cases_1})\, \mathcal{N}_C$$

(Normally, the analysis will give equality in the above. However, if the parameter q is a partial application then the analysis may give a less precise result.) As an induction hypothesis, we assume that safety holds over the type of q, $(\forall x : A).C[inl/x]$. That is,

$$\mathbf{abstr}\, q\, 1 \sqsubseteq \mathbf{at}(\mathbf{q_1}\, \mathcal{N}_C)$$

This implies that,

$$\mathbf{abstr}_{inl}\, cases\, 1 \sqsubseteq (\mathbf{at} \circ \mathbf{fst} \circ \mathbf{str} \circ \mathbf{cases_1})\, \mathcal{N}_C$$

Hence, by the principle of induction over types, safety has been shown for a left injection component of the first parameter of *cases*. Similarly, safety for a right injection component may also be proven.

3.4.4 Function types

Function types, which are represented in the general case by universal quantification, have the following formation, introduction and elimination rules. (Below, U_i represents the ith universe.)

$$\frac{A : U_m \quad \begin{array}{c}[x : A] \\ \vdots \\ P : U_n\end{array}}{(\forall x : A).P \; : U_{max\,(m,n)}} \quad (\forall\ Form)$$

$$\frac{\begin{array}{c}[x:A]\\ \vdots \\ p:P\end{array}}{(\lambda x:A).p\,:\,(\forall x:A).P}\;(\forall\ Intro)$$

$$\frac{a:A \qquad f:(\forall x:A).P}{(ap\ f\ a):P[a/x]}\;(\forall\ Elim)$$

The computation rule is:

$$ap\ ((\lambda x:A).p)\ a \rightarrow p[a/x]$$

Note that we know that the analysis must be safe in the function being applied in the computation rule for the universally quantified type since:

$$\mathbf{at}(\mathbf{ap_1}\,\mathcal{N}) = \mathbf{N}$$

Suppose, as the induction hypothesis, that the analysis is safe with respect to the function f which ranges over A. Then

$$\mathbf{abstr}\ f\ 1 \sqsubseteq \mathbf{at}(\mathbf{f_1}\,\mathcal{N})$$

Now, ap requires its second argument *iff* f requires its first argument i.e.

$$\mathbf{abstr}\ ap\ 2 = \mathbf{abstr}\ f\ 1$$

Also,

$$\mathbf{ap_2\ c} = \mathbf{f_1\ c}$$

Consequently, from the induction hypothesis:

$$\mathbf{abstr}\ ap\ 2 \sqsubseteq \mathbf{at}(\mathbf{f_1}\,\mathcal{N}) = \mathbf{at}(\mathbf{ap_2}\,\mathcal{N})$$

3.4.5 Natural numbers

The natural number type is formed simply by:

$$\frac{}{N:U_0}\;(Nat\ Form)$$

Natural numbers in the theory have the same structure as in Heyting arithmetic (and its classical counterpart, Peano arithmetic), as illustrated by the following introduction rules:

$$\frac{}{0:N}\;(Nat\ Intro_0) \qquad\qquad \frac{n:N}{(succ\,n):N}\;(Nat\ Intro_s)$$

Natural numbers are eliminated by the following rule.

$$\frac{n:N \qquad l:C[0/x] \qquad f:(\forall n:N).(C[n/x] \Rightarrow C[(succ\,n)/x])}{prim\,n\,c\,f : C[n/x]}\;(Nat\ Elim)$$

Note that the proof object deduced is a primitive recursion over natural numbers and the elimination rule encapsulates proof by induction. The computation rule simply states the general form of primitive recursion over the natural numbers.

$$\begin{aligned} prim\, 0\, c\, f &\rightarrow c & (12)\\ prim\, (succ\, n)\, c\, f &\rightarrow f\, n\, (prim\, n\, c\, f) & (13)\end{aligned}$$

The safety at the top-level is assured as the analysis indicates that the first parameter of *prim* may be used (as the argument of natural number type must be reduced to weak-head normal form for pattern matching to be performed). Also, the second argument, c, may be used (due to the base case, (12)). Finally, f will be shown by the analysis to be "needed" due to its application in clause (13).

However, in keeping with the way the naturals are defined, the analysis treats natural numbers as "structured": here the structured part reflects whether a non-zero number needs to be evaluated beyond the form $(succ\, n)$. This will indicate whether the predecessor to the element of natural number type has to be evaluated. We thus make the following definition.

Definition 32

If $f : N \Rightarrow C$ and for *any* a,b:N

$$f\,(succ\ a) = f\,(succ\ b)$$

then we say that f is **independent of the predecessor of its parameter**.

Again, the abstraction mapping, **abstr**, is extended to predecessor contexts so that the following safety condition applies to a function g of type $N \Rightarrow C$:

$$\mathbf{abstr}_{pred}\ g\ j \sqsubseteq (\mathbf{at} \circ \mathbf{pred} \circ \mathbf{str} \circ \mathbf{g_j})\, \mathcal{N}_C$$

With regard to the computation rule for *prim*, if f is independent of both its arguments then:

$$\mathbf{abstr}_{pred}\ prim\, 1 = \mathbf{U}$$

Hence,

$$\mathbf{abstr}_{pred}\ prim\, 1 \sqsubseteq (\mathbf{abstr}\, f\, 1) \sqcup (\mathbf{abstr}\, f\, 2)$$

Now, we assume the following, as an induction hypothesis in an induction over types[2]:

$$\mathbf{abstr}\, f\, 1 \sqsubseteq \mathbf{at}(\mathbf{f_1}\, \mathbf{c'})$$

and

$$\mathbf{abstr}\, f\, 2 \sqsubseteq \mathbf{at}(\mathbf{f_2}\, \mathbf{c''})$$

[2]Note that we are eliding the fact that the context functions may depend, in the higher order case, upon the arguments n and $prim\, n\, c\, f$. This omission, for the sake of notational convenience, does not affect the validity of the argument presented.

where $\mathbf{at}(\mathbf{c}') = \mathbf{N}$ and $\mathbf{at}(\mathbf{c}'') = \mathbf{N}$. Now, as noted above, the first parameter of *prim* must be used at the atomic level, and so,

$$\mathbf{at}(\mathbf{prim_1}\,\mathcal{N}_C) = \mathbf{N}$$

It then follows from the induction hypothesis that:

$$\begin{aligned}
\mathbf{abstr}_{pred}\, prim\, 1 &\sqsubseteq \mathbf{at}(\mathbf{f_1}\,\mathcal{N}_C) \sqcup \mathbf{at}(\mathbf{f_2}(\mathbf{prim_1}\,\mathcal{N}_C)) && (14)\\
&= \mathbf{at}((\mathbf{f_1}\,\mathcal{N}_C) \sqcup (\mathbf{f_2}\,(\mathbf{prim_1}\,\mathcal{N}_C))) && (15)\\
&= \mathbf{at}(\mathbf{AB} \sqcup ((\mathbf{f_1}\,\mathcal{N}_C)\,\&\,(\mathbf{f_2}\,(\mathbf{prim_1}\,\mathcal{N}_C)))) && (16)\\
&\sqsubseteq (\mathbf{at} \circ \mathbf{pred} \circ \mathbf{str} \circ \mathbf{prim_1})\,\mathcal{N}_C && (17)
\end{aligned}$$

Note that:

(15) Follows from the definition of $\sqcup$ over structured contexts.

(16) Follows from the equivalence of $\&$ and $\sqcup$ over the lattice of values of necessity.

Hence, safety has been proven for primitive recursion over the natural numbers via the principle of induction over types.

3.4.6 Lists

The following is the formation rule for list types:

$$\frac{A : U_n}{[A] : U_n} \quad (\mathit{List\ Form})$$

List structures, composed with elements of a type A, are introduced via the following rules:

$$\frac{}{[] : [A]} \quad (\mathit{List\ Intro}_e) \qquad\qquad \frac{a : A \qquad l : [A]}{(a :: l) : [A]} \quad (\mathit{List\ Intro}_n)$$

The elimination rule is equivalent to a proof by induction over lists. The proof itself is the generalised form of primitive recursion over lists.

$$\frac{\begin{array}{l} l : [A] \\ s : C[[]/x] \\ f : (\forall a : A).(\forall l : [A]).(C[l/x] \Rightarrow C[(a :: l)/x]) \end{array}}{\mathit{lrec}\; l\; s\; f : C[l/x]} \quad (\mathit{List\ Elim})$$

The elimination rule for *lrec* shows how primitive recursion over lists is reduced.

$$\begin{aligned}
\mathit{lrec}\;[]\,s\,f &\;\rightarrow\; s\\
\mathit{lrec}\;(a :: l)\;s\;f &\;\rightarrow\; f\;a\;l\;(\mathit{lrec}\;l\;s\;f)
\end{aligned}$$

Due to the definition via pattern matching on the first argument, the analysis will indicate that the list argument will definitely be used. Similarly, the second and third arguments will be detected as used (due to the result of the first clause and the application in the second clause, respectively). Consequently, the analysis must be safe at the atomic level with regard to recursion over lists.

Since the analysis also attempts to determine whether the head and tail components of a list parameter are used, we require the following definitions (which naturally may be extended to functions of any number of parameters):

Definition 33

If $f : [A] \Rightarrow C$ and for *any* $a, b : A$ and $l : [A]$

$$f\,(a :: l) = f\,(b :: l)$$

then we say that f **is independent of the head component of its argument.**

Definition 34

If $f : [A] \Rightarrow C$ and for *any* $a : A$ and $l, m : [A]$

$$f\,(a :: l) = f\,(a :: m)$$

then we say that f **is independent of the tail component of its argument.**

Corresponding to the above definitions there is an extension to the **abstr** mapping so that safety may be characterised by the following equations:

$$\mathbf{abstr}_{hd}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{hd} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C \tag{18}$$

$$\mathbf{abstr}_{tl}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{tl} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N}_C \tag{19}$$

(Naturally, the above only apply when the argument of list type reduces to a non-empty value. Also, in (18), **hd** projects from the structured part of $\mathbf{C}_{[A]}$ to $\mathbf{C}_A$ and, in (19) **tl** is a projection from the structured part of $\mathbf{C}_{[A]}$ to $\mathbf{C}_{[A]}$.)

With respect to head components,

$$\mathbf{abstr}_{hd}\, lrec\, 1 = \mathbf{abstr}\, f\, 1$$

Now, as an induction hypothesis, we assume that, for the type of f $((\forall a : A).(\forall l : [A].(C[l/x] \Rightarrow C[(a :: l)/x])))$,

$$\mathbf{abstr}\, f 1 \sqsubseteq \mathbf{at}(\mathbf{f_1}\, \mathcal{N})$$

Hence,

$$\begin{aligned}\mathbf{abstr}_{hd}\, lrec\, 1 \;&\sqsubseteq\; \mathbf{at}(\mathbf{f_1}\, \mathcal{N}) \\ &\sqsubseteq\; (\mathbf{at} \circ \mathbf{tl} \circ \mathbf{str} \circ \mathbf{lrec_1})\, \mathcal{N}_C\end{aligned}$$

and safety has thus been shown (via the principle of induction over types) with respect to list recursion and head components of lists.

For tail components, if the function f (the third parameter to *lrec*), does not use its second or third parameters then certainly,

$$\mathbf{abstr}_{tl}\, lrec\, 1 = \mathbf{U}$$

Consequently,

$$\mathbf{abstr}_{tl}\, lrec\, 1 \sqsubseteq (\mathbf{abstr}\, f\, 2) \sqcup (\mathbf{abstr}\, f\, 3)$$

Again, it may be assumed as an induction hypothesis that:

$$(\mathbf{abstr}\, f\, 2) \sqsubseteq \mathbf{at}(\mathbf{f_2}\, \mathbf{c}')$$

and

$$(\mathbf{abstr}\, f\, 3) \sqsubseteq \mathbf{at}(\mathbf{f_3}\, \mathbf{c}'')$$

where $\mathbf{at}(\mathbf{c}') \sqsubseteq \mathbf{N}$ and $\mathbf{at}(\mathbf{c}'') \sqsubseteq \mathbf{N}$. Also, as have already noted,

$$\mathbf{at}(\mathbf{lrec_1}\mathcal{N}_C) = \mathbf{N}$$

and so:

$$\begin{aligned}
\mathbf{abstr}_{tl}\, lrec\, 1 &\sqsubseteq (\mathbf{abstr}\, f\, 2) \sqcup (\mathbf{abstr}\, f\, 3) && (20)\\
&\sqsubseteq \mathbf{at}(\mathbf{f_2}\, \mathcal{N}_C \sqcup \mathbf{at}(\mathbf{f_3}\, (\mathbf{lrec_1}\mathcal{N}_C)) && (21)\\
&= \mathbf{at}(\mathbf{AB} \sqcup (\mathbf{f_2}\, \mathcal{N}_C \sqcup \mathbf{at}(\mathbf{f_3}\, (\mathbf{lrec_1}\mathcal{N}_C)))) && (22)\\
&\sqsubseteq (\mathbf{at} \circ \mathbf{tl} \circ \mathbf{str} \circ \mathbf{lrec_1})\, \mathcal{N}_C && (23)
\end{aligned}$$

Hence it has been proven (by the principle of induction over types) that the analysis is safe with respect to list recursion and tail components of lists.

3.4.7 Binary trees

Binary tree types and objects, containing elements of natural number[3] type may be formed and introduced by the following rules:

$$\frac{}{tree : U_0} \quad (tree\ Form)$$

$$\frac{}{Null : tree} \quad (tree\ Intro_N) \qquad \frac{n : N \qquad u : tree \qquad v : tree}{(Bnode\; n\, u\, v) : tree} \quad (tree\ Intro_T)$$

[3]In fact, an arbitrary type may be substituted instead of "natural number": we have assumed that the node information consists of naturals simply to be consistent with the presentation given in [134].

The elimination rule corresponds to proof by induction over the tree:

$$\frac{\begin{array}{l} t : tree \\ l : C[Null/x] \\ f : (\forall n : N).(\forall u : tree).(\forall v : tree). \\ \qquad (C[u/x] \Rightarrow C[v/x] \Rightarrow C[(Bnode\, n\, u\, v)/x]) \end{array}}{trec\; t\; c\; f : C[t/x]} \quad (tree\ Elim)$$

The proof object that is derived by the elimination rule corresponds to primitive recursion over the binary tree. It thus has the computation rule:

$$\begin{array}{rcl} trec\; Null\; c\; f & \to & c \\ trec\; (Bnode\; n\; u\; v)\; c\; f & \to & f\; n\; u\; v\; (trec\; u\; c\; f)\; (trec\; v\; c\; f) \end{array}$$

Note that the analysis shows that, if its result is required, then all three arguments to *trec* must be needed, at least at the atomic level. It must follow, therefore, that the analysis is safe with respect to *trec* at the atomic level.

It is also necessary to ensure safety for the components of a binary tree type object. In fact, by extending the definition of **abstr** in the obvious way, we may obtain the following safety conditions for each component of non-empty binary tree object:

$$\begin{array}{rcll} \mathbf{abstr}_{node}\; g\; j & \sqsubseteq & (\mathbf{at} \circ \mathbf{node} \circ \mathbf{str} \circ \mathbf{g_j})\, \mathcal{N}_C & (24) \\ \mathbf{abstr}_{lsub}\; g\; j & \sqsubseteq & (\mathbf{at} \circ \mathbf{lsub} \circ \mathbf{str} \circ \mathbf{g_j})\, \mathcal{N}_C & (25) \\ \mathbf{abstr}_{rsub}\; g\; j & \sqsubseteq & (\mathbf{at} \circ \mathbf{rsub} \circ \mathbf{str} \circ \mathbf{g_j})\, \mathcal{N}_C & (26) \end{array}$$

(Above, g is a function whose jth parameter is of *tree* type. **node**, **lsub** and **rsub** are projections from the structured part of a **tree** context. The first is a projection to natural number contexts and the latter two are projections to **tree** contexts.)

(24) for *trec* follows simply from the induction hypothesis that

$$\mathbf{abstr}\; f\; 1 \sqsubseteq \mathbf{at}(\mathbf{f_1}(\mathcal{N}_C))$$

(25) may be shown with respect to *trec* as follows. If the third parameter to *trec* is independent of its second and fourth arguments then

$$\mathbf{abstr}_{lsub}\; trec\; 1 = \mathbf{U}$$

Thus,

$$\mathbf{abstr}_{lsub}\; trec\; 1 \sqsubseteq (\mathbf{abstr}\; f\; 2) \sqcup (\mathbf{abstr}\; f\; 4)$$

We may assume as an induction hypothesis that,

$$\begin{array}{rcll} \mathbf{abstr}\; f\; 2 & \sqsubseteq & \mathbf{at}(\mathbf{f_2}\mathcal{C}') & (27) \\ \mathbf{abstr}\; f\; 4 & \sqsubseteq & \mathbf{at}(\mathbf{f_4}\mathcal{C}'') & (28) \end{array}$$

where $\mathbf{at}(\mathcal{C}') = \mathbf{at}(\mathcal{C}'') = \mathbf{N}$. Now since the analysis indicates that the first parameter of *trec* must be needed at the atomic level (i.e. $\mathbf{at}(\mathbf{trec_1}\,\mathcal{N}_C = \mathbf{N})$),

$$\begin{aligned}
\mathbf{abstr}_{lsub}\ \mathit{trec}\ 1 &\sqsubseteq (\mathbf{abstr}\ f\ 2) \sqcup (\mathbf{abstr}\ f\ 4) & (29)\\
&\sqsubseteq \mathbf{at}(\mathbf{f_2}\mathcal{N}_C) \sqcup \mathbf{at}(\mathbf{f_4}(\mathbf{trec_1}\,\mathcal{N}_C)) & (30)\\
&\sqsubseteq \mathbf{at}(\mathbf{AB} \sqcup ((\mathbf{f_2}\mathcal{N}_C)\,\&\,(\mathbf{f_4}(\mathbf{trec_1}\,\mathcal{N}_C))) & (31)\\
&\sqsubseteq (\mathbf{at} \circ \mathbf{lsub} \circ \mathbf{str} \circ \mathbf{trec_1})\,\mathcal{N}_C & (32)
\end{aligned}$$

(31) follows from (30) due to the definitions of **at** and $\sqcup$ and the fact that $\sqcup$ is equivalent to & over the neededness lattice. It has thus been shown by the principle of induction over types that the analysis is safe with respect to *trec* and the left subtree component of a binary tree proof object. It may be shown similarly that (26) holds with regard to *trec*. (The above argument may basically be altered by considering the third and fifth parameters of f instead of the second and fourth.)

3.5 Conclusion

We have developed the idea of the result of a function being independent of its parameter. Additionally, the idea of safety with respect to neededness for the analysis has been defined. Safety has been shown at the atomic level and also for the components of elements of structured proof objects. This proof of safety has used the principle of induction over types.

Consequently, whilst the necessity of computing a sub-expression within a type theory program is undecidable in general without reducing the entire program, we have shown that the analysis that has been presented will always indicate correctly that "needed" parameters will indeed have to be evaluated.

Chapter 4

Implementation within Ferdinand

In order to investigate the practicability and effectiveness of the techniques described earlier, an implementation was developed within a prototype compiler for a language based upon intuitionistic type theory. We sought to modify the Ferdinand language compiler [44] so that an optimised form of the object code would be produced. Ferdinand compiles its scripts into Functional Language Intermediate Code (FLIC; [86]). The aim was to produce a fully higher-order implementation of backwards analysis within the compiler. The compiler itself, and the implementation of backwards analysis which we describe, were both written in the Miranda[1] functional programming language [139]. We give an overview of Ferdinand and a detailed examination of each of the phases that constituted the implementation of the abstract interpretation mechanism in Ferdinand. Additionally, an account is given of the theoretical and practical basis for the removal of computationally redundant parameters. We discuss the results produced by the system and conclude with the areas of future work that may be be undertaken with regard to this project.

4.1 An overview of Ferdinand

Ferdinand, which is described in full detail in [44], is a functional programming language based on the theory TT given in [134]. However, it differs in that it has a system of *cumulative* universes which is claimed to be similar to that proposed by Luo [90]. Also, a form of extensional equality is used where each equality type represents extensional equality i.e. extensional equality is axiomatic to the system. Also, a single proof object, eq, is the sole inhabitant of *every* non-empty equality type. This leads to a system which is not strongly normalising and so extensional equality was not permitted over functions with universes as their result types. Even with this restriction it was not proven that all Ferdinand programs

[1] Miranda is a trademark of Research Software Ltd.

do indeed terminate[2], although we have assumed this when applying some optimisation techniques to Ferdinand. If the system is *not* strongly normalising then any program which includes the use of the equality eliminator, J, may be analysed incorrectly.

Ferdinand generalises sums and products to form arbitrarily sized existential and universal quantifier types i.e. we may form tuples of any length and similarly have any number of constructors for sum types. However, in order to translate the terms of Ferdinand into the primitive recursive selectors of the theory (e.g. *prec* and *lrec* for natural numbers and lists, respectively), the following restriction is placed upon functions defined by pattern matching:

> ...the only valid recursive calls are those which are performed on arguments which are *structurally* smaller, proper sub-expressions of the pattern declared, which have the correct type.
>
> (Taken from [44])

Furthermore, patterns are matched on a "best-fit" basis, rather than the sequential scheme of Miranda, for example. This is explained further in Section 3.3.2 of [44].

Ferdinand is compiled into FLIC code. The FLIC code is then translated into machine code via the fc compiler, which was produced by Thomas as part of the PG-TIM project (see [132] and, in particular, the manual for the system, [131]). FLIC is compiled using a lazy evaluation model by fc and, consequently, Ferdinand programs may be seen to run lazily. Unfortunately, fc recognises only *strictness annotations* and does not allow comments in FLIC code. (The strictness annotations are used by fc to ensure that unnecessary closures are not formed during the graph reduction process.) This means that annotations which indicate that a term is only used once (i.e. the term is unshared) cannot be processed by the fc compiler. Consequently, although the system that we have implemented is capable of detecting which parameters may be shared or used only once, this information cannot be used to produce any optimisations upon the FLIC code produced.

The Ferdinand system has been implemented in Miranda. Since the Miranda system [139] used is an interpretive one, compiling scripts to an intermediate code, Ferdinand is actually invoked by executing a Bourne shell script within the UNIX operating system. This script runs Miranda which interprets the top-level `ferdinand` function acting upon an input program. The actions of the Ferdinand compiler may be modified by certain input parameters to its top level function: the input parameters are determined by options given to the Bourne shell script. The shell script of the original Ferdinand system (`ferd`), written by Douglas, was significantly altered to allow options which determined the kind of optimisation to be performed, thus producing the script `ferd2`. The new script made several enhancements to the original, including automatically running the fc compiler

[2]Douglas is currently working on a strong normalisation proof for Ferdinand — this may result in a restriction to the universes that can be used in the system.

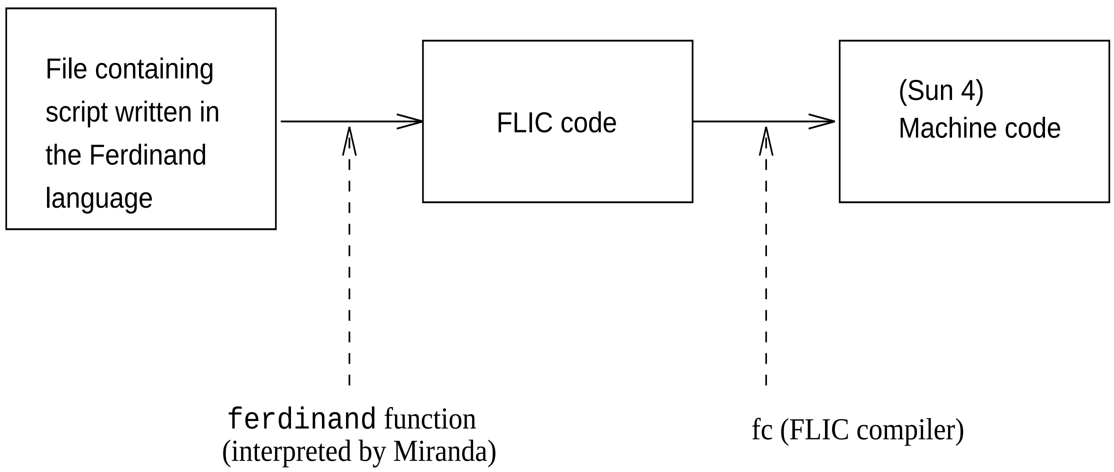

Figure 5: The process performed by the `ferd2` shell script.

on the FLIC code produced. An additional script, `mferd`, was also developed which allowed Ferdinand programs to be compiled automatically with respect to every possible analysis/optimisation performed by the modified compiler. Figure 5 shows the process performed by the `ferd2` script. A UNIX manual page for `ferd2` and `mferd` was also developed as part of this work and is in Appendix B.4.

The input to the Ferdinand system should consist of a file containing a set of function definitions. One of these should be distinguished as the main expression to be evaluated. Ferdinand would syntactically analyse the script (and any scripts which may have been recursively included), perform type checking and translate a resulting set of combinator definitions to FLIC. We have added an additional phase of program analysis and modified the translation phase so that, using the information gleaned from the analysis, optimised code was produced. One problematic aspect of the implementation was that the types of the combinators were lost during the lambda lifting phase. The phases of the modified form of the Ferdinand compiler are shown in Figure 6. Further information on compiling lazy functional languages in general (and hence all the phases covered by the Ferdinand and fc compilers) is given in [119].

4.2 Development of the implementation

The following is an account of the main phases involved in the implementation of backwards analysis within the Ferdinand compiler. In this implementation we aimed to produce a system that would build up, during each Ferdinand compilation, a database of the context functions that would result from the script being compiled. The system was designed to cope with higher-order functions. The information included in this structure then allowed us to produce optimised FLIC code.

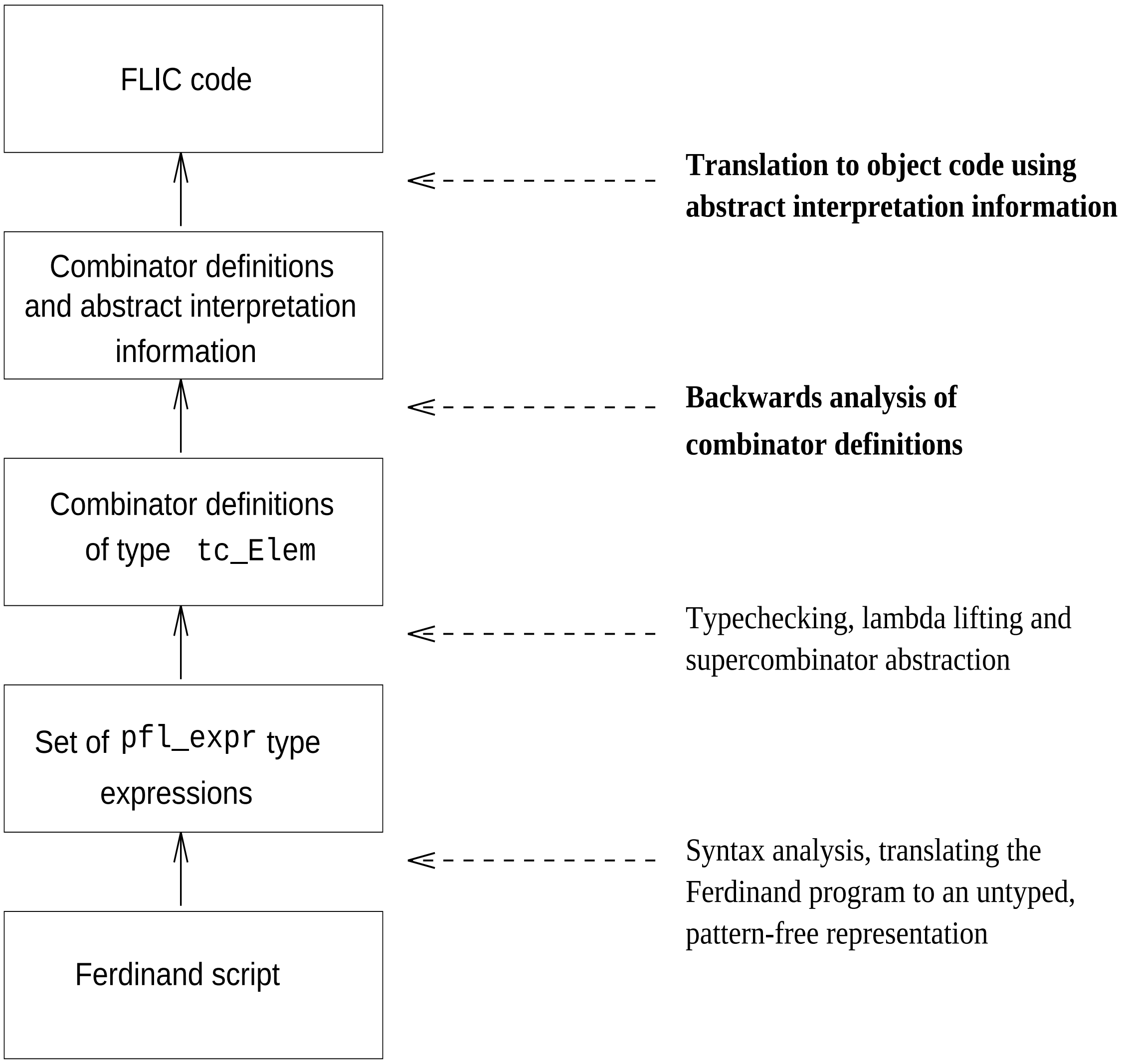

Figure 6: The phases of the Ferdinand compilation process, with the stages added shown in bold.

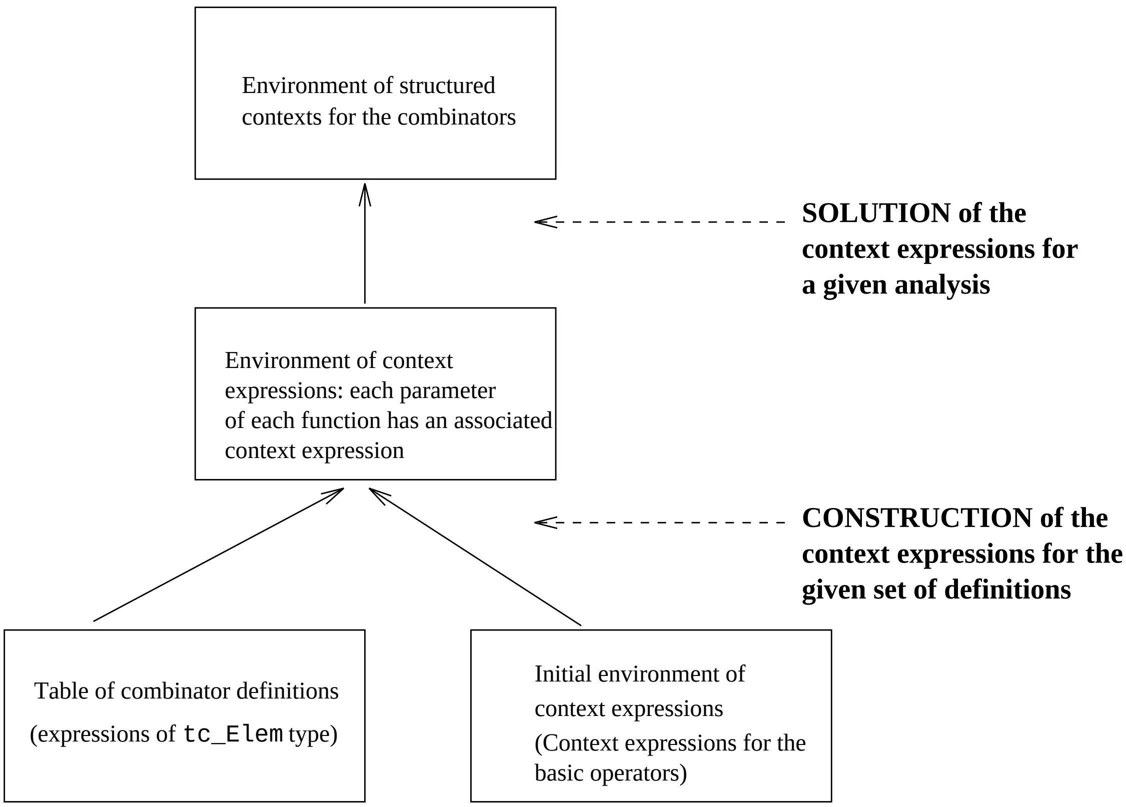

Figure 7: The analysis and translation phases added to Ferdinand.

The key phases to this project were the translation from the `tc_Elem` type of the Ferdinand compiler to context expressions and the solution of such expressions. Figure 7 shows these phases. The construction and solution of context expressions depended upon, as a foundation, a database of the context expressions for the basic operations of Ferdinand and the implementation of context lattice structures. In our implementation we attempted to produce a system which was largely independent of the particular form of analysis involved. Our aim was to produce an analysis mechanism which would cope with analyses other than neededness analysis and, in particular, analyses such as sharing analysis which would provide neededness information. This would eventually enable us to produce a combination of static analyses which should consequently produce the best optimisations of the code produced by Ferdinand.

4.2.1 Top-level functions

Since the changes to the Ferdinand system were entirely at the end of the compilation process (at the point where FLIC code was produced) it was only necessary to alter the top-level module, `newmain.m`, of the existing Ferdinand system. The alterations included the addition of a parameter to the compilation function, `ferdinand` that would indicate the kind of analysis/optimisation being undertaken. Also, an environment (accessed via combinator names) of combinator definitions was built up and the main function of the new analysis/translation system, `analysed_translation`, applied to a pair of the top-level expression and the combinator definition environment. (It is necessary to include the translation of the top-level expression, since optimisations due to the removal of unused parameters may affect the form of this expression.) `analysed_translation` has the type:

```
analysed_translation ::
    analysis_type ->
        (tc_Elem, fnDefn_Env) ->
            (flic_simple_part, flic_program)
```

The first parameter is the kind of analysis being performed and the second the pair of the expression and the definitions environment. The function produces a pair of results in abstract FLIC syntax: `flic_simple_part` is the type of the translated form of the top-level expression, whilst `flic_program` is the type of the translated form of the combinators. (In the abstract, a `flic_program` is a set consisting of `flic_binding` elements which are pairs of names and FLIC expressions, of type `flic_simple_part`.)

The translation function is partitioned into two cases. The first case deals with simple translations where no analysis has to be performed. This is so where the translation should be the same as for the original Ferdinand (i.e. following a lazy evaluation strategy) and in the case where an eager version (with all parameters denoted as being strict) should be produced. The second case is where analysis and optimisation has to be performed. The three main stages of this process were implemented via the functions `all_anno_translate`, `all_Context_Expr_Val` and `all_Context_Prop_Expr`. These functions have the types shown in Figure 8. Each of these functions may be seen to produce a new environment which is related to the original environment of combinator definitions. (A `flic_program` can be seen as an environment of FLIC function definitions.) The latter two functions are defined via a `foldr` over the list of combinator names to build up a new environment from the previous one e.g. a `context_Exp_Env` (an environment of context expressions) is constructed from a `fnDefn_Env` (the environment of combinator definitions) by `all_Context_Prop_Expr`. Each of the three functions named above calls a subsidiary function which looks up the function name in the relevant environment and applies a function to create a new version of a result environment. For instance, `all_Context_Prop_Expr` calls:

```
cxt_expr_form ::
    fnDefn_Env ->
        fn_name ->
            context_Exp_Env ->
                context_Exp_Env
```

`cxt_expr_form` takes an environment of function definitions, a function name, an existing context expression environment and produces a new expression environment. (The new environment is formed by adding the set of context expressions pertaining to the given function to the original environment.)

```
all_anno_translate ::
    analysis_type ->
        context_Env ->
            fnDefn_Env ->
                [fn_name] ->
                    flic_program

all_Context_Expr_Val ::
    analysis_type ->
        context_Exp_Env ->
            [fn_name] ->
                context_Env

all_Context_Prop_Expr ::
    fnDefn_Env ->
       [fn_name] ->
           context_Exp_Env
```

Figure 8: The Miranda types of the top-level functions.

4.2.2 Main data structures

The analysis and translation phase added to Ferdinand may be seen as transformations between data structures: we start with a set of function definitions and from that form a database of context expressions. That collection of expressions may then be solved to form a database of contexts which is then used with the original set of function definitions to produce an optimised set of FLIC definitions. The environment of function definitions (type `fnDefn_Env`) was modelled as an instance of a simple abstract type, `environment`, with values bound to names.

Each of the databases of context information, however, required a more sophisticated data structure, where each (function) name was bound to a *table* containing context information. Each table would contain information on each parameter of the function to which the table was bound. Moreover, it was important that information in the table could be accessed by both indices (e.g. to look up the ith context function of a named function) and variables (e.g. to determine whether the variable could be omitted from the resulting code). Hence, two abstract data structures were created: `idx_table` ("indexed tables") and `fnParam_Env` ("environments of parameter expressions"). The latter were implemented in terms of the former viz.:

```
fnParam_Env * == environment (idx_table *)
```

This concept is illustrated in Figure 9. The `context_Expr_Env` and `context_Env` are instantiations of `fnParam_Env` with the types of the values contained within the index tables being `context_Expression` and `structured_context`, respectively. That is,

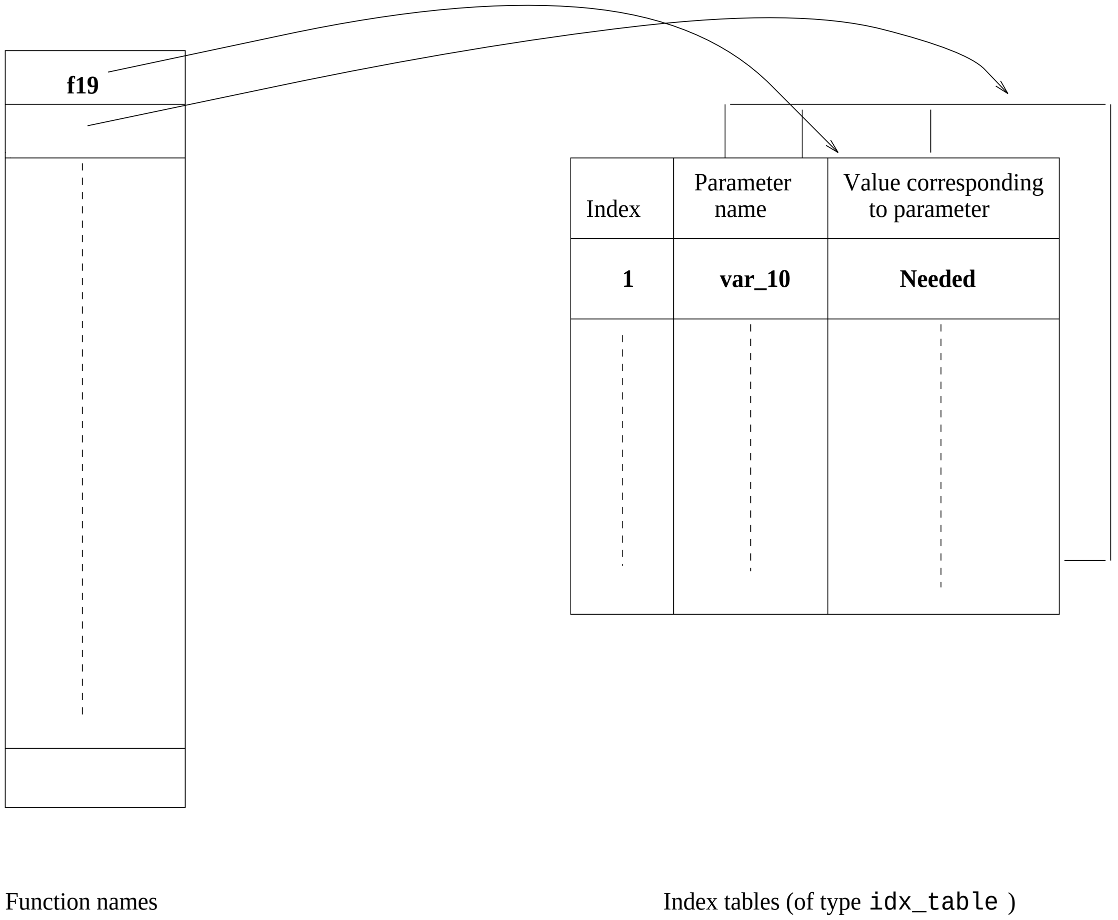

Figure 9: Conceptual diagram of `fnParam_Env` and `idx_table` types.

```
context_Exp_Env == fnParam_Env context_Expression
context_Env == fnParam_Env structured_context
```

In order to ensure correctness, the result of each lookup in an environment was given by a term of the abstract type `value`. This is similar to the exception monad given in [146].

4.2.3 Formation of context expressions

The formation of a set of (analysis independent) context expressions for each parameter of each function represented the first stage in the optimisation process. The main function (which is the counterpart of the $\mathcal{E}$ operator of Section 2.14.4) to do this was:

```
context_Prop_Expr ::
    prop_Var ->
        [param] ->
            tc_Elem ->
                context_Expression
```

The type `prop_Var` enables a distinction to be made between the case where context information is being propagated to a variable and the case where a proxy context expression is being formed as part of a list of supplementary parameters to a context function. (These supplementary expressions will only be needed in the case of higher-order functions.) The second parameter is a list of the formal parameters of the function being analysed and the final parameter is the defining expression of the function.

Naturally, `context_Prop_Expr` is defined via pattern matching over the `tc_Elem` type, which represents types and terms of the type theory. For example,

```
context_Prop_Expr v params (Jay tc_El1 tc_El2)
=
    context_Prop_Expr v params tc_El2
```

The above captures the semantics of the J selector over the equality types i.e. that only the second part of the pair that J is acting upon is of computational significance (see Section 2.9.3).

Below we describe the implementation of the type of context expressions, elaborate on the process of expression construction and give a description of the initial environment of context expressions, formed for the basic operators of Ferdinand.

Implementation type of context expressions

Context expressions are implemented via the following algebraic type (*cf* Section 2.14.2):

```
context_Expr ::=
    ABSENT                                                  |
    CONTRA                                                  |
    E_TOP                                                   |
    Initial                                                 |
    Cxt structured_context                                  |
    Contor context_Expression context_Expression            |
    Contand context_Expression context_Expression           |
    Func fn_name num [context_Expression]                   |
    Lambda_Cxt num num [context_Expression]                 |
    App_Cxt context_Expression context_Expression           |
    Struct context_Expression struct_context_Expr           |
    STR context_Expression                                  |
    Fixpoint
```

The above have the following meanings:

- `ABSENT`, `CONTRA` and `E_TOP` correspond to the (analysis independent) **ABSENT**, **CONTRA** and **TOP**, respectively (see Section 2.4.3).

- `Initial` indicates that the initial, or input, context to the expression should be substituted at this point.

- `Cxt structured_context` represents a constant context expression i.e. a structured context. (This has little use during expression construction since we are forming analysis independent expressions. However, it has been employed during testing.)

- `Contor` and `Contand` represent the $\sqcup$ and $\&$ context operators, respectively (see 2.4.4).

- `Func fn_name num [context_Expression]` represents a context function, $\mathbf{f_i}$, say — the second component indicates the index of the context function, whilst the first component indicates the function name. The third component is a list of context expressions corresponding to a list of actual parameters. These supplementary expressions will be used if the context function is higher-order (see 2.11.2).

- `Lambda_Cxt num num [context_Expression]` indicates that a supplementary context expression should be substituted at this point. The first component indicates the number of the parameter that should be substituted e.g. if it is 3 then the third expression in the list of supplementary expressions should be used. The second component is the index to be used when applying the resulting context expression. Finally, the third component is a list of *additional* supplementary expressions. This construct is necessary to allow higher-order functions to be analysed more precisely — see Section 2.11.2.

- `App_Cxt` represents the application of one context expression to another.

- `Struct context_Expression struct_context_Expr` represents a structured context expression (see below). The first component is an atomic expression, whilst the second component is a structured set of context expressions (the homologue of structured parts of contexts).
- `STR context_Expression` represents the **strict** function (see 2.4.5) upon the result of a context expression.
- `Fixpoint` indicates that the current approximation to the fixpoint of an iteration should be substituted at this point.

The structured context expressions allow, for example, a context expression to be given for the head of a list. Each structured context expression is a counterpart to a structured type in Ferdinand and is implemented via the following:

```
struct_context_Expr ::=
    Cex_END                                                              |
    Cex_TOP                                                              |
    Cex_TRUNC                                                            |
    Cex_INIT                                                             |
    Cex_Succ context_Expression                                          |
    Cex_Cons context_Expression context_Expression                       |
    Cex_Node context_Expression context_Expression context_Expression    |
    Cex_Tuple [context_Expression]                                       |
    Cex_Inj context_Expression
```

The last five cases are reasonably self-explanatory since they are just the context constructions (see Section 2.6.1) corresponding to those of the types of Ferdinand. `Cex_END` denotes the end of a structured context expression (and is hence equivalent to the bottom element of the context lattice, **CONTRA**). `Cex_TOP` denotes an arbitrary structured expression where all basic elements will be equivalent to the top element of the underlying lattice (e.g. a structured list context with both head and tail needed). `Cex_INIT` is similar to `Cex_TOP` except that instead of the top element of the basic lattice we shall have the initial context of the relevant lattice. (The initial context will, of course, depend upon the analysis employed. For example, in the case of neededness analysis, the initial context will be **N** (needed).) `Cex_TRUNC` indicates that a recursive structure has been terminated at that point (in order to preserve a finite set of lattices). `Cex_TRUNC` may thus be seen as denoting a cycle in the graph of the structured context.

The structured context expressions are not actually used during the formation of context expressions. This is because the type checking phase of Ferdinand reduces all recursive expressions (and pattern matching) to instances of the primitive recursive selectors, *prec*, *lrec* and *trec*. Consequently, the structured expressions only occur within the initial context expression environment (see page 123), where they are used for some of the basic operators.

Main functions of context expression formation

The context_Prop_Expr function depends upon the following functions to construct context expressions from application expressions (of type tc_Elem).

```
cxt_appl_mk ::
    prop_Var ->
        [param] ->
           tc_Elem ->
               context_Expression
```

cxt_appl_mk forms a context expression relative to an input variable (the first parameter to cxt_appl_mk) and the set of formal parameters for the function being analysed, from an application (the third parameter). This is done by partitioning the application into the name of the function being applied and the actual parameters. The name of the function being applied determines whether cxt_fn_app (for named functions) or cxt_lda_app (where the function being applied is a parameter of the function under analysis) should be used.

```
cxt_fn_app ::
    prop_Var ->
        [param] ->
            fn_name ->
                [tc_Elem] ->
                     context_Expression
```

cxt_fn_app produces a context expression corresponding to a named function (the third parameter) being applied to a list of applicands (the fourth parameter). This calls a function, cxt_app_form which actually forms the context expression for each applicand. The type of this function is:

```
cxt_app_form ::
    prop_Var ->
        [param] ->
            fn_name ->
                num ->
                    tc_Elem ->
                        [context_Expression] ->
                            context_Expression
```

The fourth parameter, of type num denotes the index of the context function to be applied. The fifth parameter, of tc_Elem type, is the applicand expression. The final parameter is a list of context expressions. The context expressions thus formed are combined by recursively using the Contand constructor (to indicate that the results of the context expressions should be combined).

cxt_lda_app is similar to cxt_fn_app except that the function being applied is a parameter of the current function being analysed. Hence, this will result in a context expression involving the application(s) of a Lambda_Cxt form. Like cxt_fn_app, cxt_lda_app calls a function, cxt_lda_form to construct the actual expression for each applicand.

Finally, a function is required to form supplementary context expressions i.e. expressions which may be regarded as extra actual parameters to context functions and which are used if and only if the context function is higher-order.

```
cxt_free_prop_exp ::
    [param] ->
        tc_Elem ->
            context_Expression
```

In fact, `cxt_free_prop_exp` just invokes `context_Prop_Expr` again, but with the variable being set to `Input`. This means that the context expressions built up by `cxt_fn_app` and `cxt_lda_app` will differ slightly (with only one context function application being formed and no combination of expressions via `Contand`). The clauses of `context_Prop_Expr` for variable names also vary according to the form of the variable in question e.g.

```
context_Prop_Expr Input params
    (Assume vr tc_El)
=
    Lambda_Cxt vind 0 []
    || Input matches any variable: the context lambda abstraction
    || indicates that we have to input a context expression at this point
    || instead of the variable.
    , if ok_par_index
=
    ABSENT
    , otherwise
    || If vr does not correspond to any formal parameter. Really this
    || could be considered an error, as vr must be a variable in scope.
    where
        || vind is the parameter index of the variable.
        vind
        =
            param_index vr params
        ||
        || ok_par_index indicates whether the variable is amongst
        || the formal parameters.
        ok_par_index
        =
            vind > 0
||
context_Prop_Expr (Ord v) params
    (Assume vr tc_El)
=
    Initial
    , if v = vr         || i.e. the variables match.
=
    ABSENT
    , otherwise         || i.e. the variables are different.
```

Initial context expression environment

There are some operators of Ferdinand which are built-in to the language: most of these operators are not primitive to TT but, on the other hand, they form a more restricted set than that of a functional language like Miranda. For example, the subtraction operator is not primitive to Ferdinand. Corresponding to these operators, an initial environment of context expressions was required: the context expressions of the combinators of the program being compiled would then be added to these primitives to form a new context expression environment.

Analysing these operators and building the expression database for them was fairly trivial, apart from the cases of the primitive recursive operators upon natural numbers, lists and binary trees. For example, the context expression for the length of list operator is:

```
Struct Initial (Cex_Cons ABSENT Initial)
```

The context expression for the first parameter of *prec* is:

```
Struct Initial (Cex_Succ (Contor ABSENT cl2exp))
```

where

- `cl2exp` is the context expression arising from the second clause of the definition of prec.

  ```
  cl2exp
  =
      Contand
          (App_Cxt (Lambda_Cxt 3 1 cl2acts) Initial)
          (App_Cxt (Func "Prec" 1 prec_acts) fapp2)
  ```

- `fapp2` is the context expression formed from the application of the parameter f to its second argument $(prec\, n\, c\, f)$.

  ```
  fapp2
  =
      App_Cxt (Lambda_Cxt 3 2 cl2acts) Initial
  ```

- `cl2acts` is the actual parameter list resulting from the second clause of the definition of *prec*.

  ```
  cl2acts
  =
      [nat_exp,prec3]
  ```

- `nat_exp` is a context expression for the successor of the natural number argument.

```
nat_exp
=
     E_TOP
```

- prec3 is a context expression corresponding to *prec n c f*.

```
prec3
=
    App_Cxt (Func "Prec" 3 prec_acts) Initial
```

- prec_acts consists of the context expressions corresponding to the arguments of *prec*.

```
prec_acts
=
    [nat_exp,(Lambda_Cxt 3 1 []),(Lambda_Cxt 3 0 [])]
```

The Ferdinand standard environment is a script (stdenv.fe) which, unlike in Miranda, is only loaded if it is explicitly inserted within a script. Whilst these functions are not built into the compiler, it was originally envisaged that a database of context expressions for these functions, such as map, would be added to the expressions for the basic operations. However, whilst the functions themselves were tractable to analysis (the only differences from their Miranda counterparts being explicit polymorphism via universes, rather than implicit polymorphism, and some additional proof objects), it was realised that to do this would require a substantial alteration to the existing type checking phase of the Ferdinand compiler so that the standard environment functions would be recognised rather than compiled to a set of combinators like the other functions. In addition, each of the standard environment functions would have to be encoded in FLIC. Whilst it would seem that it is essential for the future development of Ferdinand as an efficient programming language that these standard environment functions (and their context expressions) be fully integrated within the compiler, it was believed that this was not germane to the current work.

4.2.4 Evaluation of context expressions

The main function (which is the counterpart of the $\mathcal{V}$ function of Section 2.14.3) used to find the structured context resulting from a context expression and an input context was:

```
context_Expr_Evaluation ::
    analysis_type ->
        structured_context ->
            context_Exp_Env ->
                fn_name ->
                    num ->
                        [context_Expression] ->
                            context_Expression ->
                                structured_context
```

The first parameter denotes the kind of analysis being performed (and which determines the primitive context functions that are employed). The second parameter is the input context and the third is the environment of context function expressions. The fourth and fifth parameters are the name of the function being analysed and the index of the context function being evaluated, respectively. (These are necessary so that recursive context expressions may be detected.) The sixth parameter is a list of input context expressions: these are the supplementary expressions which will be used if the context function is higher-order. The final parameter is the context expression to be evaluated.

In general, evaluation of context functions is done in the following stages:

1. Using the function `context_Expr_Reduction` to produce a triple of type

   ```
   (structured_context, bool, context_Expression)
   ```

   The first component of the triple is a structured context which will be the result of the function if no recursion has occurred. If recursion has occurred then it will be the first non-bottom approximation to the least fixpoint as the bottom context is substituted for any recursive form found. The second component indicates whether the expression being evaluated contains any recursion (and, consequently, whether a fixpoint iteration should be performed). The final component is a modified form of the original context expression being evaluated: it is this form that will be used in any subsequent iterations. It is optimised, in the sense that any occurrences of recursion in the original expression will be replaced by the constructor `Fixpoint` which will indicate in subsequent iterations that the current approximation to the fixpoint should be substituted at this point. This means that `Fixpoint` will occur at the points in the expression where the bottom context was substituted to form the first approximation to the fixpoint.

2. The calculation of the least fixpoint (see Section 2.4.2) for a recursive context expression. This uses the `context_Expr_Valuation` function to produce successive approximations to the fixpoint:

   ```
   least_fixpoint antype iCxt fpCxt cExpEnv fNm vInd act_Exprs cE
   =
       cxt_limit antype (iterate cxt_expr_fn fpCxt)
       where
           || cxt_expr_fn is a function based upon the context
           || expression whose value is being calculated.
           cxt_expr_fn cxt
           =
               context_Expr_Valuation
                   antype iCxt cxt cExpEnv fNm vInd act_Exprs cE
   ```

   `context_Expr_Valuation` is very similar to the function `context_Expr_Reduction` except that only the structured context value of the expression is returned.

(It is already known, of course, that the expression is recursive and the context expression has already been simplified.) The cxt_limit is similar to the limit function of the Miranda standard environment, in that it checks whether two successful members of a list of approximations to the fixpoint are equal contexts for a given analysis.

Both context_Expr_Reduction and context_Expr_Valuation determine the values of expressions using the rule:

$$e\,\mathbf{c} = \mathbf{AB} \sqcup (e\,(\mathbf{strict}\ \mathbf{c}))$$

if $\mathbf{AB} \sqsubseteq \mathbf{c}$. (See Sections 2.4.5 and 2.6.4). Each function calls subsidiary functions which are defined via pattern matching over the structure of context expressions. In addition, there are functions to reduce structured context expressions.

The most significant part of context expression evaluation is concerned with the evaluation of a context function application. This is shown in the clause of the function context_Expr_Reduce (which is called by context_Expr_Reduction) given below.

```
context_Expr_Reduce antype iCxt fpCxt cExpEnv fNm vInd act_Exprs
    (App_Cxt (Func fname ind a_cExps) cE)
=
    (fpCxt, True, Fixpoint)
    , if recursive_call
=
    (context_Fn_val, input_is_recursive, new_appl_expr)
    , if ~ recursive_call & index_known
=
    (top_cxt, input_is_recursive, E_TOP)
    , if ~ recursive_call & ~ index_known
    where
        recursive_call
        =
            context_Expr_Match antype
                (App_Cxt (Func fname ind subst_cExps) (Cxt str_cE_val))
                (App_Cxt (Func fNm vInd act_Exprs) (Cxt iCxt))
        ||
        new_appl_expr
        =
            App_Cxt (Func fname ind a_cExps) cE'
        ||
        index_known
        =
            ind <= fname_size
        ||
        top_cxt
        =
            lattice_SCTOP antype
```

```
||
fname_size
=
    value_elim size_err_message fn_size_lkup
||
fn_size_lkup
=
    fn_size cExpEnv fname
||
context_Fn_val
=
    context_Expr_Evaluation
        antype cE_val cExpEnv fname ind subst_cExps cExpFn_exp
||
(vname, cExpFn_exp)
=
    value_elim error_message tag_cExpFn_exp
||
tag_cExpFn_exp
=
    idx_lkup_param cExpEnv fname ind
||
subst_cExps
=
    sub_actual_exps antype act_Exprs a_cExps
||
(cE_val, input_is_recursive, cE')
=
    context_Expr_Reduce
        antype iCxt fpCxt cExpEnv fNm vInd act_Exprs cE
||
str_cE_val
=
    lattice_SCSTR antype cE_val
||
error_message
=
    "context_Expr_Reduce: Error in lookup of " ++
    fname ++ " " ++ (shownum ind)
||
size_err_message
=
    "context_Expr_Reduce: Error in lookup of size of " ++ fname
```

In the above code fragment:

- In the first case, above, we have a recursive expression and the current approximation to the fixpoint is substituted at this point. In the second case the context function expression does not result in a recursion. However, the context function for the given function and index is known. In the third case, the context function expression does not result in a recursion, but the context function for the given function and index is unknown.

- `recursive_call` indicates whether we have a recursive call of the function. This is discovered by attempting to match the components of the current function being evaluated and that specified by the `Func` construction.

- `new_appl_exp` is the new application expression to be used in any subsequent fixpoint iterations.

- `index_known` indicates whether the context function for the given function and index is known.

- `top_cxt` is the top element of the context lattice.

- `fname_size` is the size (i.e. number of parameters) of `fname`.

- `fn_size_lkup` is the result of looking up the size of `fname`.

- `context_Fn_val` is the value of the context function looked-up, relative to the initial context `cE_val`.

- `cExpFn_exp` is the untagged context expression found from the current environment of context expressions, `cExpEnv`, and `vname` is the name of its corresponding variable. The call of `value_elim` will trap any errors that have occurred during the lookup.

- `tag_cExpFn_exp` is the tagged context expression found from the current environment of context expressions, `cExpEnv`.

- `subst_cExps` are the actual parameter context expressions of the function, with appropriate substitutions made for lambda context expressions.

- `cE_val` is the result of evaluating the expression `cE`, which will be the initial context given to the context expression looked up. `input_is_recursive` indicates whether the expression for the initial context is recursive. `cE'` is the expression to be used in any fixpoint iteration.

- `str_cE_val` is the strict version of `cE_val`.

- `error_message` is the header error message to be used when a lookup has resulted in some error occurring.

- `size_err_message` is the header error message to be used when a lookup for the function's size has given an erroneous result.

It should be noted from the above that it is possible for the index of the context function not to be recognised. This will occur for a function defined by partial application where the number of actual parameters supplied to the function is less than the number of formal parameters. In this case, a safe approximation is taken by defining the results as the top structured context and top context expressions. It is possible to be more precise than this, although matters are complicated by the fact that we do not know the precise types of the combinators produced by the Ferdinand type checking phase, since they are lost during lambda lifting. Methods to remedy this (and thus produce a system which is fully higher-order) are discussed in the conclusion, Section 4.5.

Two important aspects of context expression reduction are evident in the given code fragment, firstly that of the recognition of recursive patterns and secondly the substitution of actual context expressions for "dummy expressions" which refer to formal parameters (the `Lambda_cxt` form). In the case of the former it is necessary to check that the function name, the index of the context function, the input context *and* the actual parameter expressions (for higher-orders) must all match. To ensure that the supplementary expressions match a function `context_Expr_Match` (and a subsidiary function to deal with structured expressions) is used. The substitution of expressions is dealt with by a function `sub_actual_cxt_exp` (and, again, a function to deal with structured expressions).

Of course, evaluation of expressions depends upon operations over the underlying set of structured contexts. These are described below.

Structured Contexts

Structured contexts (see Sections 2.6 and 2.14.1) were implemented within the module `context_calc.m` as the following type:

```
structured_context == (context_Lattice_Value, struct_context)
```

That is, we simply follow the theory in having structured contexts represented by a pair of an *atomic part* ranging over the basic lattice values (represented by the `context_Lattice_Value` type) and a *structured part* consisting of *context constructors* and basic lattice values (represented by the `struct_context` type). The `struct_context` is an algebraic type similar to that for structured context expressions. (We considered the "atomic types" of Ferdinand to be the following: `Char`, `Boolt` and `Triv`. As our phrase suggests, we assumed that variables of these types had no "structure" and their contexts would thus be atomic ones.) Operations, such as the combination of structured contexts, were defined over the structures using the relevant functions over the lattices of simple values. For example, the following is the implementation of the $\sqcup$ operator over structured contexts:

```
lattice_SCOR an_ty (at1, str1) (at2, str2)
=
    (at', str')
    where
        at' = lattice_COR an_ty at1 at2
```

```
        str' = struct_COR an_ty str1 str2
```

struct_COR is, naturally, the $\sqcup$ operator over the structured part of contexts and lattice_COR the $\sqcup$ operation over the basic, atomic values.

Functions were developed which gave the top element of a given structure (i.e. a structured context with all values contained being the top element of the basic lattice) and the initial element of a given structure. This allowed contexts to be built which would match other structures — for instance, we could produce a list context with the atomic context and the head and tail contexts all being the top elements of the context lattice. Also, functions to determine equality over structured contexts were developed, which facilitated a cxt_limit function. This was similar to the limit function of the Miranda standard environment but with equality over structured contexts (the lattice_SEQ function) replacing '='.

Most importantly, as required by the functions to calculate $\&$, this module included functions to approximate contexts for the recursive types, list and tree. The function used to do this over structured parts was:

```
approx_Cxt ::
    analysis_type ->
        struct_context ->
            struct_context
```

Thus the structured part of context was approximated with respect to a particular analysis. This was implemented in general by gathering together the contexts corresponding to similar parts of the structure (e.g. the head contexts for lists) in one list and mapping the lattice_SCOR function over those contexts. The results of these subsidiary computations are then reassembled as a new structured part.

Basic context lattices

The aim in the implementation of the basic lattice/domain theoretic aspects (see Section 2.4) of the static analysis was to produce a series of functions and structures which would be *generic* in the sense that they would be capable of acting upon more than one form of backwards analysis. To that end, we attempted to emulate a simplified version of the lattice theory implemented in Haskell by Jones [82]. Whilst, since we used the Miranda language as our implementation tool, we did not enjoy the full power of Haskell's type classes and could not therefore produce a generic lattice structure, we did implement a generalised context_lattice type with associated functions (in the module general_lattice.m). This allowed us to abstract the implementation with functions parameterised upon the type of analysis being performed. (If it were possible to use Jones's implementation of lattices, however, the system of type classes would mean that the indicator of the kind of analysis being used would not have to be stated explicitly throughout the context evaluation phase.)

The lattices that were implemented were mainly two-point ones e.g. Needed and Unused for neededness analysis (see 2.5). (Note that we choose to be "explicit" with regard to the basic contexts used: a more efficient implementation for each two-point domain would have been to make each set of contexts equivalent to the Miranda `bool` type.) The four-point lattice of strictness and absence analysis (shown in 2.5) was also constructed, as was the eight-point one (arising from a power set construction) of full sharing analysis (see 2.5.4). However, the latter could not be usefully employed due to problems in the translation phase which we outline below.

4.2.5 Translation into FLIC

The final stage of the modified implementation was the translation into FLIC. In the original version of Ferdinand the `tc_Elem` expressions were translated directly into strings of characters. However, we decided instead to produce an abstract syntax of FLIC and provide pretty-printing functions upon that abstract syntax. The benefits of this were to allow errors to be detected more easily and to enable future modifications to the FLIC syntax to be made more smoothly. (Indeed, the abstract syntax developed is really that of the 1990 revision of FLIC, as described in [86] and we translate into the earlier version of FLIC of [117]. This was basically a smaller version of the 1990 language with minor variations in the concrete syntax. However, it was the only variant of FLIC accepted by the fc compiler.)

A separate module was produced for each kind of translation. As will be explained in the section below, the optimisations requiring neededness information would not only need reference to the context information of the parameters of the function being translated but also the entire context environment.

In order to provide functions which could be used by all the translation processes, two modules were created. One was developed to capture the idea of predicates upon variables with regard to contexts (e.g. to determine whether a variable is strict or not by a lookup in the indexed table of contexts for the function being translated) and the other to give generic translation functions. The latter set of functions are mainly concerned with implementing the algorithm for the removal of unused parameters which is described below.

Strictness annotations, which indicate to the FLIC compiler that the input may be evaluated before evaluating the function and so that a closure need not be formed for that parameter, are given via the FLIC concrete syntax, `[!]`. This indicates that the function which follows is strict in its argument. (Multiple argument functions are defined by nesting of lambda abstractions.) For example, the translation function for simple strictness analysis is:

```
st_flictrans ::
    idx_table structured_context ->
        tc_Elem ->
            flic_simple_part

st_flictrans par_cxts tc_El
=
```

```
    st_flictrans_lambda par_cxts tc_El
    , if isLambda tc_El
=
    new_flictrans tc_El
    , otherwise
    || Strictness information is only used with lambda abstractions.

st_flictrans_lambda par_cxts (Lambda v t c)
=
    Anno STR v_abstraction
    , if v_is_Strict
    || i.e. we add the FLIC annotation to denote strictness in that
    || variable.
=
    v_abstraction
    , otherwise
||
    where
        || v_abstraction is the lambda abstraction, binding v.
        v_abstraction
        =
            Abs (Single_Abs v (Simple fl_c))
            || Note that although Single_Abs is used, fl_c may contain
            || subsequent abstractions.
        ||
        || fl_c is the FLIC translation of the defining expression.
        fl_c
        =
            st_flictrans par_cxts c
        ||
        || v_is_Strict indicates whether v is a "strict" variable or not.
        v_is_Strict
        =
            lkup_pred is_Str par_cxts v
```

(Above, `new_flictrans` is the function for performing the translation into FLIC without the use of any abstract interpretation information. The first parameter to `st_flictrans` is the table of context information for the function being translated, whilst the second parameter is the expression to be translated.)

Unfortunately, it was unclear as to how sharing analysis information could be used to improve the efficiency of the code produced: no annotations other than that for strictness are recognised by the fc compiler. Thus, whilst it would seem that the fc compiler implements full laziness, there would appear to be no way of informing the compiler that an expression is used at most once.

4.3 Optimisation due to neededness

In this section we describe how neededness analysis information may be used to optimise the object code produced by a lazy functional language with particular reference to the practicalities of this operation within the Ferdinand compiler.

4.3.1 First-order case

Example

In the first-order case, we have only complete applications e.g.

$$\mathbf{add13}\ 1\ 2\ 3$$

where

$$\mathbf{add13} \equiv_{df} \lambda x.\lambda y.\lambda z.\ (x + z)$$

Such functions are easily transformed. In the example given, the second parameter of **add13** is unused and so we may remove it completely, to form a new definition, **add13'**, where

$$\mathbf{add13'} \equiv_{df} \lambda x.\lambda z.\ (x + z)$$

Similarly, the call of **add13** 1 2 3 is transformed to become:

$$\mathbf{add13'}\ 1\ 3$$

Generalisation

Generally, in the first-order setting, we have a function, f, say, where

$$f \equiv_{df} \lambda x_1 \ldots x_m.\ E$$

We may transform the definition of f to be f', where

$$f' \equiv_{df} \lambda y_1 \ldots y_l.\ E'$$

where $l \leq m$. The sequence of variables, $\langle y_i \rangle$, is the natural sequence derived from the variable indices of the elements of $\mathrm{Nd}\,\{x_1 \ldots x_m\}$ i.e. the subset of the variables of f which are needed. Also, E' is formed from E by making any necessary transformations of calls of functions within E. In addition, any occurrence of an unused variable in E is replaced by $\perp$ in E'.

Transformations of calls of functions are handled recursively as follows. We have the application:

$$f\ e_1 \ldots e_m$$

This becomes:

$$f'\ e'_1 \ldots e'_l$$

Each e'_i corresponds to a needed actual parameter of f and unused actual parameters will simply be deleted from the call. Also, each actual parameter of f' is, of course, the result of call transformation as well.

4.3.2 Higher-order case

Example

In the higher-order case, the scenario is complicated by partial applications e.g.

$$\mathbf{map}\,(\mathbf{const}\,3)\,l$$

or

$$\mathbf{map}\,(\mathbf{const}_2\,3)\,l$$

where

$$\begin{array}{rcl}\mathbf{const} & \equiv_{df} & \lambda x.\lambda y.\ x \\ \mathbf{const}_2 & \equiv_{df} & \lambda x.\lambda y.\ y\end{array}$$

We may transform the defining expressions as before. The new functions are:

$$\begin{array}{rcl}\mathbf{const'} & \equiv_{df} & \lambda x.\ x \\ \mathbf{const}_2' & \equiv_{df} & \lambda y.\ y\end{array}$$

(So, in effect, **const** and $\mathbf{const}_2$ are both mapped to the identity function.)

The calls above are transformed, respectively, to:

$$\mathbf{map}\,(\lambda x.\mathbf{const'}\,3)\,l$$

and

$$\mathbf{map}\,(\mathbf{const}_2')\,l$$

Note that in the first case we have to add a "dummy abstraction" so that the function argument is applied with the correct arity.

Generalisation

The definitions of higher-order functions are transformed as before, with unused bound variables being removed. In addition, applications of a formal parameter are translated as normal since we do not have any information about the neededness of the arguments of the formal parameter. An alteration to the method occurs when we analyse calls of the form:

$$(\ldots((\mathbf{f}\,e_1)\,e_2)\ldots e_n)$$

where **f** is not a parameter of the function being analysed. We also assume that the arity (i.e. the number of formal parameters) of **f** is m.

For $1 \leq i \leq n$ and $i \leq m$ we either remove e_i completely from the application if the ith parameter of **f** is unused or, alternatively, transform e_i to become e_j' (with $j \leq i$). As we mentioned in the first-order case, this is a recursive process: each e_j' results from a transformation of its sub-expressions.

Consequently, we now have as our first step:

$$(\ldots((\mathbf{f'}\, e_1')\, e_2') \ldots e_l')$$

where **f'** is the transformation of **f** and $l \leq n \leq m$. If $m \leq n$ then **f** and, consequently, **f'** are fully applied (to the first n arguments) and the situation is equivalent to the first-order case. If $m > n$ then we must add dummy abstractions to ensure that the applications which we form are of the correct arity (i.e. we do not have any leftover variables). These abstractions effectively throw away any unused arguments. We thus obtain:

$$\lambda x_1 \ldots x_k.(\ldots((\ldots((\mathbf{f'}\, e_1')\, e_2') \ldots e_l')y_1) \ldots y_j)$$

The y_j variables represent the j (with $j \leq (m-n)$) needed variables of **f** amongst those having indices ranging from $n+1$ to m. Each y_v will be syntactically the same as some x_w. We can omit binding any new variables if all those with indices from $n+1$ to m are needed since

$$\lambda x_1 \ldots x_k.\ (\ldots(e\, x_1) \ldots x_k)$$

is **n**-equivalent to e. This is why, in our example above, **map** ($\mathbf{const}_2$ 3) l is transformed to become **map** ($\mathbf{const}_2'$) l rather than **map** ($\lambda x.\mathbf{const}_2'\, x$) l.

4.3.3 Practical considerations

When translating functions we shall assume that two environments have been constructed to store neededness analysis information. The first environment, L, consists of the neededness information of the formal parameters of the particular function being translated. L itself may be considered a subset of the main environment M. This should contain neededness information about all the formal parameters of each function defined in the program. In the case of the Ferdinand compiler, M is of type `context_Env`. L is of type `idx_table structured_context` i.e. an indexed table of structured contexts. M consists of bindings of function names to indexed tables of contexts.

The first stage in translating the definition of a function consists of forming lambda abstractions using the needed variables. The variables are looked up in L and either ignored, if they are unused, or translated to a lambda abstraction if needed. The abstraction formed will thus contain a subset of the original variables.

The second stage consists of translating the defining expression of a function. If the innermost function of an application is a formal parameter of the function being translated, then we translate the entire application as normal. Otherwise, the following steps should be followed:

1. Find the leftmost and innermost function being applied and determine the size of the application (i.e. the number applicands), a, say. This may be done by putting all the components of the application in a list.

2. Determine the arity, in terms of the number of formal parameters, of the function being applied (F, say). This arity, b, say, may be found from a lookup in M.

3. If $b > a$ then, again via M, determine the number c of the parameters of F which have indices ranging from $a + 1$ to b and are also unused.

4. If $c > 0$ then preface the application with $(b - a)$ distinct bound variables. These are the dummy abstractions. Similarly, extra applications are added for those new bound variables which are needed.

5. For each of the b parameters of F and using M, translate each actual parameter that corresponds to a needed variable of F and ignore each actual parameter that would be unused by F.

6. If $a > b$ then the sub-expressions indexed from $a + 1$ to b are simply translated.

The last two stages may be done by filtering the list of applicands so that only needed sub-expressions remain and then forming a FLIC application by folding translation over the list. If any sub-expression contains an unused formal parameter then that variable is translated to ABORT in FLIC syntax.

The main functions used to perform the above translation, `ndd_flic_appl_expr` and `ndd_opt_appl_trans` are given in Appendix B.1.2. These are functions are generic, in the sense that they are used in the translation based upon neededness and that based upon strictness and absence.

4.4 Results

Qualitative assessment An illustration of the effects of applying the optimised translation phase is given by the following example. The Ferdinand version of the index function (see 2.10) is shown in Figure 10[3]. The FLIC code produced when no analysis/optimisation is applied is shown in Figure 11 and the FLIC code produced when strictness-and-absence analysis is applied is shown in Figure 12. The FLIC programs have the form:

```
EXPORT (Pmain)

&
<List of function names>
<The main expression, _topdog>
<Combinator definitions, in the same order as in the list>
```

[3]Note that the index function presented is in a slightly different form to that given in [134]. This is due to a problem in the type checking phase of the Ferdinand compiler.

```
lt :: nat -> nat -> (Un 0)

lt m 0 = bot
lt 0 (n+1) = top
lt (m+1) (n+1) = lt m n

index :: (a :: Un 0) =>
            (l :: (List a)) -> (n :: nat) -> ((lt n  (# l)) -> a)
index [] 0 p = abort p a
index [] (n+1) p = abort p a
index (a:x) 0 p = a
index (a:x) (n+1) p = index x n p

main :: nat
main = index [1,2,3,4,5,6] 4  unique
```

Figure 10: The Ferdinand version of the index function.

The optimised piece of code shows how strictness annotations may be added, with unused parameters discarded and some dummy variables inserted. A sample of the `tc_Elem` expressions and context expressions for this program are given in Appendices B.2.1 and B.2.2, respectively. In addition, the structured contexts that result are given in Appendix B.2.3.

Quantitative results In an attempt to quantify the efficacy of the optimisations implemented, some Ferdinand programs (which are given in Appendix B.3) were compiled with respect to each possible optimisation method. (The fc compiler was used with optimisation disabled of the C code produced as an intermediate phase.) There were two controls for each test program. The first was the executable produced with no alteration to the translation (i.e. as in the original Ferdinand). The second was a fully eager executable (with every parameter marked as strict). Execution times and the number of garbage collections required were recorded for each program run. Due to the fc compiler, these statistics were given as part of the output produced by the program executed. Due to variances within the system being used (SunOS 4.1.2 running on a Sun4 architecture) each program was run eleven times to produce a mean execution time. These times are shown in Table 4. The numbers of garbage collections required during each execution are shown in Table 5. In the tables, the following are used to indicate the analysis/optimisation types:

None No analysis optimisation performed i.e. the code that would have been produced by the original Ferdinand results.

Eager A fully eager version of the program is produced i.e. with every parameter of each function annotated as being strict.

```
EXPORT (Pmain)

&
(_topdog f13 f12 f11 f10 f9 f8 f7 f6 f5 f4 f3 f2 f0 f14 __printit Pmain)
((((("f14" (PACK-2-1 1 (PACK-2-1 2 (PACK-2-1 3 (PACK-2-1 4
    (PACK-2-1 5 (PACK-2-1 6 PACK-0-0))))))) 4) PACK-0-0))
(\ "_Var25" ("_lrec" "_Var25" "f4" "f12"))
(\ "_Var17" ("f11" "_Var17"))
(\ "_Var17" (\ "_Var18" ("f10" "_Var17")))
(\ "_Var17" (\ "%@hyp_Var18" (("f9" "_Var17") "%@hyp_Var18")))
(\ "_Var17" (\ "%@hyp_Var18" (\ "_Var19" ("_prec" "_Var19"
    ("f5" "_Var17") ("f8" "%@hyp_Var18")))))
(\ "%@hyp_Var18" (\ "_Var14" (("f7" "%@hyp_Var18") "_Var14")))
(\ "%@hyp_Var18" (\ "_Var14" (\ "%@hyp_Var14" (("f6" "%@hyp_Var18") "_Var14"))))
(\ "%@hyp_Var18" (\ "_Var14" (\ "_Var15" (("%@hyp_Var18" "_Var14") "_Var15"))))
(\ "_Var17" (\ "_Var20" "_Var17"))
(\ "_Var26" ("_prec" "_Var26" "f0" "f3"))
(\ "_Var23" "f2")
(\ "%@hyp_Var23" "f0")
(\ "_Var27" PACK-0-0)
"f13"
 (\"pr_input" ((_itos "pr_input")))
 (\"__input" (_@ ("__printit" "_topdog") (PACK-2-1 '\n PACK-0-0))))
```

Figure 11: The FLIC code produced for the index function with no optimisation.

Need The program produced when neededness analysis has been employed.

Strict Strictness analysis/optimisation.

Strab Strictness and absence analysis/optimisation.

Commentary on the results The results are extremely encouraging in some of the cases examined, such as permsort, whilst in others, such as acker, they appear to be rather disappointing.

Results for the two simple programs acker and polymap, were disappointing, with no statistically significant[4] improvements for the various analyses. Indeed, the strictness and strictness and absence analyses actually gave slightly worse execution time results than the original Ferdinand compiler for the acker script. However, for the translations which added strictness annotations, including the simple eager translation, reduced the number of garbage collections from 370 to 351. However, since these programs result in fewer FLIC combinators, as shown in Table 7, there would appear to be less scope for optimisations to have a significant effect. In addition, the speed of the process is constrained by the efficiency of the

[4] At the 95% confidence level with Student's t-test for the difference of two means. (The assumption was made that the underlying variances of the program run times were the same.)

```
EXPORT (Pmain)

&
(_topdog f13 f12 f11 f10 f9 f8 f7 f6 f5 f4 f3 f2 f0 f14 __printit Pmain)
((("f14" (PACK-2-1 1 (PACK-2-1 2 (PACK-2-1 3 (PACK-2-1 4
    (PACK-2-1 5 (PACK-2-1 6 PACK-0-0)))))) 4 PACK-0-0))
[!] (\ "_Var25" ("_lrec" "_Var25" "f4" "f12"))
(\ "_Var17" (\ "dummy_1" ("f11" "_Var17")))
(\ "_Var17" ("f10" "_Var17"))
(\ "_Var17" (\ "%@hyp_Var18" ("f9" "_Var17" "%@hyp_Var18")))
(\ "_Var17" (\ "%@hyp_Var18" [!] (\ "_Var19" ("_prec" "_Var19"
    (\ "dummy_1" ("f5" "_Var17")) ("f8" "%@hyp_Var18")))))
[!] (\ "%@hyp_Var18" (\ "_Var14" (\ "dummy_1" ("f7" "%@hyp_Var18" "_Var14"))))
[!] (\ "%@hyp_Var18" (\ "_Var14" ("f6" "%@hyp_Var18" "_Var14")))
[!] (\ "%@hyp_Var18" (\ "_Var14" (\ "_Var15" ("%@hyp_Var18" "_Var14" "_Var15"))))
[!] (\ "_Var17" "_Var17")
[!] (\ "_Var26" ("_prec" "_Var26" (\ "dummy_1" ("f0")) (\ "dummy_1" ("f3"))))
(\ "dummy_1" ("f2"))
"f0"
PACK-0-0
"f13"
 (\"pr_input" ((_itos "pr_input")))
 (\"__input" (_@ ("__printit" "_topdog") (PACK-2-1 '\n PACK-0-0))))
```

Figure 12: The FLIC code produced for the index function with strictness-and-absence optimisation.

Test program	*Translation method*				
	None	Eager	Need	Strict	Strab
acker	121.82	129.24	121.53	127.78	134.47
polymap	7.26	10.04	7.27	7.42	7.16
bubblesort	86.58	94.46	95.31	96.47	87.07
mergesort	15.31	30.25	14.19	13.01	14.58
permsort	29.51	16.74	26.98	14.32	12.70
treesort	19.45	49.95	20.34	18.04	20.49

Table 4: Mean execution time in seconds for each executable produced.

Test program	*Translation method*				
	None	Eager	Need	Strict	Strab
acker	370	351	370	351	351
polymap	19	28	19	19	19
bubblesort	318	338	319	318	318
mergesort	56	101	56	54	54
permsort	94	61	89	48	42
treesort	74	169	74	73	73

Table 5: Number of garbage collections for each executable.

Test program	Translation method				
	None	Eager	Need	Strict	Strab
acker	15.7	15.5	16.4	16.2	16.4
polymap	19.4	19.3	21.4	21.0	21.3
bubblesort	220.3	219.6	296.1	293.4	297.9
mergesort	47.8	47.1	70.1	67.9	70.5
permsort	204.8	206.9	234.8	233.1	224.3
treesort	184.4	184.8	225.0	223.5	222.9

Table 6: Compilation times in seconds (single compilation in each case).

Test program	Combinators
acker	7
polymap	13
bubblesort	47
mergesort	53
permsort	70
treesort	59

Table 7: Number of combinators produced in the FLIC code for each test program.

object code produced by the fc compiler and, in particular, the way it deals with strictness annotations, natural numbers and output generally.

More promising results were observed with the mergesort and permsort scripts. Each of the three analyses produced statistically significant improvements for mergesort with need, strict and strab producing 7.9%, 17.7% and 5.0% speed gains. It is unclear why strictness and absence analysis should not produce improvements at least as good as the others. The simple eager translation produced a marked degradation of performance with mergesort, almost doubling both the execution time and the number of garbage collections.

As already mentioned, the most spectacular results were produced in the case of permsort. Need, strict and strab produced improvements in speed of, respectively, 9.4%, 106.1% and 132.4% and garbage collections were reduced by 5, 46 and 52, respectively. The last figure represents a 55% reduction in the number of garbage collections. The simple eager translation increased the execution speed by 76.3% and reduced the number of garbage collections by a third. It should be noted, however, that this was still significantly worse than strict or strab: the latter two translation methods were 16.9% and 31.8% faster than the eager translation, with similarly large reductions in the number of garbage collections.

The treesort and bubblesort tests produced results which may be seen to be anomalous: in the case of the former, strictness analysis produced a speed increase of 7.8% but the neededness and strictness and absence analyses produced slightly worse results compared to the executable produced by the original Ferdinand. In the case of bubblesort, neededness and strictness analyses produced degradations in performance, whilst the strictness and absence analysis produced results similar to the original. This is unexpected, and disappointing since the bubblesort program, which may be seen in Appendix B.3.4 includes a number of computationally redundant proof objects. As a number of parameters have been removed, and some dummy variables added, in the resulting FLIC code, it is unclear why the performance should not have been enhanced.

It should be noted that the fully eager translation did produce a severe degradation of performance, in terms of both execution time and garbage collections, for some of the cases examined and in the other cases, such as with permsort, it did not produce results as good as that for strictness analysis. This was the case even with Ackermann's function where no (lazy) list structures were involved. This would appear to be a result of the way the fc compiler handles strictness annotations and garbage collections.

It should be noted, from Tables 4 and 5 that optimisations which reduced garbage collections the most consequently reduced execution times most.

Enhancements to the optimisation process, in particular to remove more unused expressions, are considered in the conclusion below.

Compilation times[5] The degradation in compilation times due to backwards analysis and optimisation varies, for strictness and absence analysis, from 4.5% in the case of acker to 47.5% for mergesort, as can be seen in Table 6. There appears to be a weak correlation between the compilation times and the number of FLIC combinators produced for each test program. (Table 7 shows the number of combinators for each program.) Programs with a greater number of combinators produce worse compilation times for each of the abstract analyses, as would be expected, since calculations have to be done for each parameter of every function. However, the addition of the analyses does not produce as severe a degradation in performance as might be feared. We speculate that this is due to the following reasons:

1. The original Ferdinand compiler is itself a prototype which has not as yet been optimised. Compared to the complex phases that the main part of the compiler performs, such as type checking and lambda lifting, the abstract interpretation module is not particularly significant.

2. The abstract analyses that we have implemented and tested have small basic domains: the most complicated is strictness and absence analysis which consists of just four points. Whilst structured contexts add to the complexity, this is alleviated by point 3, below.

3. The Ferdinand system is a Miranda program which is thus evaluated *lazily*. Often it is the case that only the atomic parts of contexts have to be calculated in order to determine the form of the output to be produced. For example, if, with neededness analysis, the atomic part of a list context evaluates to `Needed` then there may be no need to calculate the abstract values of the head and tail parts, since the optimisations are based upon whether the entire parameter is needed or not. Only when a standard function which uses pattern matching, such as lrec, is used will the subscripted parts be calculated.

An unexpected feature of these results is that the strictness and absence analysis compilation times are little worse than, or in some cases better, than those for neededness analysis.

4.5 Conclusion

The theory of backwards analysis has been implemented within a compiler for a functional language system, Ferdinand, that is based upon type theory. The implementation was intended to cope with various different analyses within one generic framework. Most importantly, the analysis was able to deal with higher-order

[5]It should be noted that the compilation times are for a *single* compilation only and therefore, because of variation in the processing speed by the operating system, are statistically less precise than the execution times detailed above.

functions, although with some simplifications in the case of partial applications. (This simplification was due simply to the fact that the types of the Ferdinand combinators were lost during the lambda lifting phase of the compiler.)

The implementation was designed with modularity rather than efficiency in mind. This has the beneficial aspect that modifications to the compiler should be more straightforward to create in future. However, it has meant that, for instance, lookups upon the same name in different structures are performed. It was originally envisaged, in the interests of efficiency, that the function definitions, of `tc_Elem` type, would be paired with their context information in a single structure. This, however, was found not to be easily tractable. A further loss of efficiency may be perceived due to the lattice operations being generic and not specialised with regard to any particular lattice. Furthermore, it is not necessary in many cases to use structured contexts and expressions: it would be better if these could be reduced to the basic lattice values. Also, as mentioned, it would be more elegant to use Jones's implementation strategy for lattices using the type classes of Haskell [82].

There have been some encouraging results produced as a result of this exercise. In particular, there have been some small, yet significant improvements produced by neededness analysis alone. Moreover, there have been some dramatic improvements produced by the strictness and strictness and absence analyses. It should also be noted that applying a simple eager reduction strategy to the programs does not generally give good results. In other cases, results have not been so impressive and, in some cases where the results appear to be anomalous, may deserve further study. However, we conjecture that some of these more disappointing results are due to the following factors:

1. The fc compiler does optimisations of its own upon the FLIC code provided. (In testing we attempted to keep these to a minimum, in particular disabling the optimisation of the C language code produced by fc as an intermediate step.)

2. It would be better if, as in the case of strictness, we could use annotations to denote the fact that a parameter was unused, with the optimisations being performed within the fc compiler.

3. As noted above, some approximations arise with regard to partially applied functions. This problem is analysed further below.

4. More redundant code could be removed and this is discussed below.

It has been noted that the optimisations do produce satisfyingly tangible results in terms of alterations to the code produced.

Further optimisations

Functions defined by partial application It would be more satisfactory if we could take account of definitions by partial application without approximating by the topmost context in the case where the context function was unknown i.e. in the case where a function is defined with x parameters but it is applied to y, where $y > x$ — the context functions for indices ranging from $x + 1$ to y will be undefined. If this could be overcome then we would have a *fully higher-order analysis*. We could do this easily if we had accurate type information for each combinator. However, such type information is discarded during the lambda lifting phase of the compiler.

Alternatively, we could adopt an "on the fly method" where for each function, we generate an additional context expression (representing application to other unnamed variables). When we then encounter the case where we require the context expression of a parameter that does not exist for a function, we use the additional context expression. This is similar to the concept of generating context expressions (which may be used in a higher-order context function) for each actual parameter of an application in the Ferdinand code. This idea is similar to the concept of producing context functions for an arbitrary vector of additional parameters given in Section 2.11.2. However, that addition would only be necessary for functions of polymorphic result type. Here, though, since we do not have accurate type information, it would be necessary to create an additional expression for *every* function.

Removal of extra redundant code The FLIC code may be optimised further to remove certain inefficiencies that may occur in the code produced. The inefficient code that we describe was present in the original version of Ferdinand. It may be argued, however, that, the following might reap the full benefits of the analyses performed, especially as the forms described might occur more often due to the removal of unused parameters.

Trivial definition removal — remove functions which have definitions of the form "f_i" where "f_i" is another function name. This will include constants.

Identity function detection and removal — replace functions which reduce to the form $\lambda x.x$ with the (untyped) identity function and applications that are equivalent to $(\lambda x.x)e$ with e.

Eta-equivalent expression removal — Function definitions of the form,

$$\lambda x_1 \ldots \lambda x_n.(\ldots(e\,x_n)\ldots x_1)$$

where each x_i is not free in e, may be replaced simply by e. Eta-equivalent expression removal is the general form of the trivial and identity function cases described above. Apart from analysing the FLIC code produced, it is possible that variables which are "not needed" may be detected by abstract interpretation. The needed context would have to be split into two, one context indicating that the parameter was needed generally and the other meaning that the parameter was only required by an expression which may be reduced to an eta-equivalent form. In addition, if the latter context applied, then the contexts of other parameters would decide whether the parameter could be removed. A discussion of how selective eta-expansion, as a form of *partial evaluation* [83], can be used to specialise terms in λ-calculus is given in [36].

Polyvariant specialization We only have one set of backwards analysis results for each function. This means that we simply take the function and determine the contexts for its parameters relative to one, set input context in each case. Optimisation of the function is done only with regard to this one set of data. It may be, however, that during the analysis of another function, f, say, a function, g, is analysed to produce different results from that produced by the stand-alone analysis. In order to optimise f fully, a specialized version of g, g', should be produced which is based upon the contexts propagated during the analysis of f. f should then call g' instead of g in the optimised code produced. This is known as *polyvariant specialisation* (as opposed to the monovariant specialisation we have done) and is explained with regard to partial evaluation in [83].

It should be noted, however, that the above optimisations would mean that alterations would have to be made to the list of combinator names produced at the top of the FLIC code produced.

Standard environment functions A major improvement to the efficiency of Ferdinand would be to have the standard environment functions fully integrated within the compiler. This would involve an alteration to the built-in tables of functions, so that each function of the standard environment had a FLIC representation. Also, it would be necessary to modify the initial context expression environment so that it included context expressions for each parameter of every standard environment function.

Scope for further modifications

Other analyses There are the following stages to be performed when adding any new analyses to the compiler:

1. A module containing the basic lattice definitions should be created. This should include a 10-tuple comprising all the basic definitions: the lattice top, bottom, absent and initial contexts; the strict operation; the lattice ordering and equality predicates; the $\sqcup$ and & operations; and the lattice meet. This module should then be integrated within the `general_lattice` module by adding to the algebraic types contained therein. In particular, the `analysis_type` enumeration of all possible analyses should be extended.

2. A module to translate the `tc_Elem` expressions into FLIC should be created. This will probably require the use of predicates upon structured contexts and so it may be necessary to extend the `cxt_pred` module as a result. To assist

in the creation of a new translation module, a generic translation template, `temp_translate` has been formulated.

Extensions to the Ferdinand language Extensions to the Ferdinand language are more problematic with regard to alterations that have to be made to the analysis and translation phases. Firstly, new clauses within the context expression formation function will have to be created for each new language construct. More importantly, for each new Ferdinand data type the syntax for structured expressions and contexts will have to be extended to maintain correspondences between the formation functions of Ferdinand and the context selectors and vice versa. It follows that the initial context expression environment will have to be extended to have context expressions for each new selector over expressions. Another consequence will be that all functions for expression matching, substitution etc. will have to be extended for the new construct. If the new data structure is recursive then new cases will have to be added to the context approximation function (so that finite lattices are maintained).

Chapter 5

Other static analyses of type theory

5.1 Introduction

In this chapter we examine how other static analysis techniques may be applied to type theory. We shall focus particularly on developing a new approach to the time complexity analysis of type theoretic programs, which uses the abstract interpretation techniques discussed earlier.

5.2 Time complexity of type theory

An active area of computer science has been the investigation the measurement of the *complexity*, in terms of space or time, of programs. The development of such metrics is useful to compare the efficiency of separate algorithms which meet the same specification. In TT, there will usually be more than one witness to each function specification. For instance, a function to sort a list of natural numbers may have the following type:

$$\exists f : ([N] \Rightarrow [N]).\forall l : [N] \, . \, (Sorted\, f\, l) \wedge (Perm\, (f l)\, l)$$

(The intention of the *Sorted* and *Perm* predicates, whose details we omit, is to ensure that the resulting list, (fl), is both sorted and a permutation of the original list, respectively.) There are several functions which are proofs of the above, such as quicksort [134, pages 211–218] or insertion sort [111]. Additionally, sorting algorithms may not be defined by primitive recursion, as in the two examples cited, but, in an augmented type theory, employ well-founded recursion. For example, Paulson [116] defines quicksort using such a relation.

Furthermore, it is desirable to automate this process so that we may have a program which assists in the measurement of complexity. (Algorithmic methods are limited, however, in that the measurement of time or space complexity is an undecidable process.) For example, pioneering work in this area was performed by

Wegbreit [148] who produced a system, "Metric", which analysed the complexity of simple Lisp programs relative to the properties of the input.

The idea of time complexity analysis is that we can give some closed-form expression that expresses the number of computation steps required to analyse a program by some metric, usually with respect to the complexity of possible inputs, such as the length of a list or the modulus of a natural number.

In this section we give an account of the work of Bjerner on the time complexity analysis of type theoretic programs [14]. We shall give an example of the kind of analysis performed and this will provide a contrast with the work that follows in the next section. There we shall use our proceeding work on the abstract interpretation of TT and some of the ideas of Wadler [145] to produce a time-complexity analysis for the whole of TT. We believe that the analysis that we shall present is more modular than Bjerner's in that a time complexity measure can be derived from an absence analysis that has been previously calculated. Hence, the results of the absence analysis may be used in a dual role, both as a starting point for optimisations and to calculate the time complexity under the assumption that a fully lazy evaluation strategy is employed. Also, our work, being built upon the analysis of TT, is of greater scope than Bjerner's: his analysis covers a primitive recursive subset of lazy functional languages. For instance, we may derive complexity measures for terms in types or functions which return types. Additionally, our method of time complexity analysis should be more easily extensible than that of Bjerner, in that adding new types to the theory should involve little more work than extending the absence analysis to those types.

5.2.1 Analysis of strict languages

As is pointed out by Wadler [145], the analysis of strict functional languages is relatively straightforward. This follows since arguments to a function *must* be calculated before computing the value of the function, the time complexity of a strict functional program may be found using the following composition rule:

$$[\![\, f(g\,x) \,]\!]^T = [\![\, f\,y \,]\!]^T + [\![\, g\,x \,]\!]^T$$

In the above,

$$\begin{array}{cl} [\![\, e \,]\!]^T & \text{is the time complexity of the expression } e \\ y & \text{is the normal form of } g\,x \end{array}$$

The only exception to this are conditional constructs such as *if-then-else* which will have the following complexity:

$$[\![\, \mathit{if}\ b\ \mathit{then}\ \mathit{texp}\ \mathit{else}\ \mathit{fexp} \,]\!]^T = [\![\, b \,]\!]^T + \begin{cases} [\![\, \mathit{texp} \,]\!]^T, & \text{if } b \twoheadrightarrow \mathit{True} \\ [\![\, \mathit{fexp} \,]\!]^T, & \text{otherwise} \end{cases}$$

Worst or best case complexities can be derived for the different normal forms of b. For example, a simple expression for the worst case in the above would be:

$$[\![\, b \,]\!]^T + \mathit{max}\,([\![\, \mathit{texp} \,]\!]^T, [\![\, \mathit{fexp} \,]\!]^T)$$

Here the maximum is taken of the complexity of the expressions of each branch of the conditional. More precise estimates may be found by estimating the probability that the conditional expression, b, will evaluate to *True*.

In lazy functional functional languages, however, the arguments to each function are not necessarily evaluated. Hence the composition rule mentioned above is not applicable. The above outline for the time complexity of a strict functional language thus provides a method for obtaining an *upper bound* for the time complexity of a type theoretic based functional programming language. (This does mean that the time complexity, in terms of the number of sub-computations involved, of using a strict evaluation strategy for a TT based language is typically worse than that of a lazy one. However, in practice, on an actual machine, the lazy evaluation strategy will generally give worse performance in terms of the clock time required to evaluate a given program. This is due to the fact that the *space* complexity of the lazy strategy is generally worse, due to the closures that have to be formed during evaluation. Thus the lazy evaluation strategy will typically require more memory accesses in the machine, with a consequent degradation in the speed of computation.)

Bjerner's method for approaching the problem of finding the time complexity of type theoretic programs evaluated using a lazy evaluation strategy is outlined below.

5.2.2 Description of Bjerner's method

Bjerner uses notation similar to that used in [114]: Bjerner separates the time complexity of an expression in type theory[1] by specifying two component parts:

Constant time cost – the time complexity due to the evaluation of a function body.

Constant evaluation degree – a measure of how much an argument has to be evaluated in order for the entire application to be evaluated.

The evaluation degrees are necessary in order to account for the lazy evaluation strategy.

Since we will typically not be dealing with fully applied expressions, the time complexity will be given in terms of a function relative to either or both of the following:

- The **size** of an input.
- The particular normal form of the input.

[1] We shall use the notation and system of TT given in [134].

The size of an expression in TT

Definition 35

The **size** of an expression, $e : A$, (denoted $|e|$) in TT is the number of constructors of the type A that appear in the normal form of e. This is equivalent to the following:

$$|e| = 1 + \sum_{p=1}^{k} |a_p|$$

where $C(a_1 \dots a_p)$ is the value of e i.e. $e \twoheadrightarrow C(a_1 \dots a_p)$ and C is a constructor of the type of e.

For example, the size of the number 3, is 4, since the formal representation of 3 in type theory is:

$$Succ(Succ(Succ\ 0))$$

In the above, there are three instances of the N selector, *Succ*, and one of the selector 0. In general,

$$|n| = \mathcal{F}\ [\![\, n \,]\!] + 1$$

where $\mathcal{F}$ is the semantic function mapping from the TT type N to the natural numbers. Similarly, for a list, l, its size is:

$$|l| = length\ l + 1$$

How time complexity is measured

Bjerner takes one cycle of the following **computation procedure** to count as a unit of the time complexity metric for a program. The computation procedure is iteratively applied until a *computational normal form* (similar to head normal form, but taking into account the possible sharing of arguments) is achieved:

1. Compute the value of the **major argument**. The major argument of an expression, $e : A$, is the leftmost and outermost proper sub-expression of type A in e. For example, in

   $$prim\ n\ c\ f$$

   the major argument is n, since $prim\ n\ c\ f$ and n are both of type N.

2. The construction of the **selected program**. This is the expression associated with the pattern of the major argument for the relevant computation rule.

3. The computation of the selected program.

In other words, this corresponds to a single iteration of the application of the computation rules.

The above computation procedure leads to a definition of the **time cost function** in terms of the sequence of needed programs (SNP).

Definition 36

The **sequence of needed programs**, $SNP(e)$ for an expression e, consists of:

- The expression itself.
- Additionally, if e is a non-canonical program, the concatenation of the two sequences, $SNP(m)$ and $SNP(s)$, where m is the major argument of e and s is the selected program.

Definition 37

The time cost function, $\phi(e)$ for an expression, e, is defined as follows:

$$\phi(e) = \begin{cases} 0, & \text{if } e \text{ is canonical} \\ 1 + \phi(m) + \phi(s), & \text{if } e \text{ is non-canonical} \\ & \text{with } m \text{ the major argument and } s \text{ the selected program} \end{cases}$$

It is straightforward to see why the time cost function is so defined in light of the computation procedure above: the 1 comes from the application of the procedure for the whole expression, whilst the other two expressions are the time costs associated with computing the major argument (the first step) and the selected program (the last step).

The time cost is linked to the length of the sequence of needed programs by the following theorem (4.2 in [14]).

$$Length(SNP(e)) = 1 + 2 \times \phi(e)$$

A proof by induction (on the length of $SNP(e)$) of this is given by Bjerner.

Time cost of abstraction applications

The sequence of needed programs can similarly be used to give the following result (theorem 5.3 in [14]) for abstraction applications:

$$\phi(e(a)) = \phi(e(a^V)) + k \times \phi(a)$$

where k is the number of occurrences of the subsequence $SNP(a)$ in $SNP(e(a))$ and a^V is the *value* of a i.e.

$$a^V = C(a_1 \ldots a_n)$$

where C is a constructor of the type of A and $a \twoheadrightarrow a^V$.

It follows by induction from the above that have the **abstraction application time cost principle**:

> The time cost of $(e(a))$ is equal to the time cost of each program derivative of a (i.e. components of the value of a) multiplied by the number of occurrences.

V	Gives the evaluation degree of an expression so that $e^{V}(e)$ (e evaluated to the degree $V(e)$) is the value of e i.e. e evaluated to canonical form (but not necessarily normal form).
W	Gives the evaluation degree of an expression so that $e^{W}(e)$ is the normal form of e.
Spine	Gives the evaluation degree of e required to "unfold" the recursive parts of e (e.g. the tail of a list, the subtrees of a tree).

Table 8: Many-valued evaluation degrees.

Time cost of arguments to applications

We must also determine the time cost of an argument to an application. As has been already discussed, this is problematic since we do not know whether arguments will be evaluated under the lazy evaluation strategy. Bjerner's solution to this problem is to introduce the idea of *evaluation degrees*. These measure the extent to which a sub-expression must be evaluated in evaluating the enclosing expression.

Evaluation degrees

Definition 38

Evaluation degrees are defined as follows:

- $\perp$ is an evaluation degree.
- $[\sigma_1 \ldots \sigma_n]$ is an evaluation degree.

A more restrictive definition, that of **proper evaluation degrees** is also made by Bjerner. The additional restriction is that the only proper evaluation degree for an abstraction is $\perp$.

(We shall see in Section 5.4 that the evaluation degree $\perp$ corresponds to the context **U** in the neededness analysis lattice. Indeed, proper evaluation degrees are combined with the $\sqcup$ operator; this gives the least upper bound of two evaluation degrees, whilst $\perp$ is implicitly the bottom element of the lattice of evaluation degrees.)

Bjerner also defines "many-valued evaluation degrees" which may vary according to the form of the expression to which they refer. The three evaluation degree functions are given in Table 8.

Relative evaluation degrees

The evaluation degrees of sub-expressions are relative to the enclosing expression being evaluated. These are described in Table 9. Hence, $\frac{a}{e}\,\tau$ is the generalisation of $\psi(a, e)$ and $\frac{a:\sigma}{e}\,\tau$ is the generalisation of $\Psi(\sigma, a, e)$.

$\psi(a,e)$	The evaluation degree of a for e.
$\frac{a}{e}\,\tau$	The *smallest* evaluation degree to which a has to be evaluated when e is evaluated to degree τ.
$\Psi(\sigma,a,e)$	The evaluation degree of a for e when a is already evaluated to degree σ.
$\frac{a:\sigma}{e}\,\tau$	The evaluation degree of a for e when e has to be evaluated to degree τ and a has already been evaluated to degree σ.

Table 9: Relative evaluation degrees.

Time costs relative to proper evaluation degrees

The time cost of an expression that has to be evaluated to the degree σ is denoted by:

$$a \downarrow \sigma$$

where σ is a proper evaluation degree for a. If it is assumed that arguments to functions are evaluated at most once, the following result is obtained (corollary 5.1 in [14]):

$$e(a) \downarrow \tau = (e(a^W) \downarrow \tau) + (a \downarrow \frac{a^W}{e(a^W)}\,\tau) \tag{33}$$

where a^W means $a^{W(a)}$.

Note that the above is a generalisation of the abstraction application time cost principle (see Section 5.2.2 above).

Open expressions

Normally we attempt to find the time complexity of a particular function definition, relative to an *arbitrary* argument. However, the equation (33) is expressed with respect to *fully evaluated* arguments. Bjerner describes how the **constant time cost** ($e(x) \downarrow \tau$, where x is an arbitrary input) and the **constant evaluation degree** ($\frac{x}{e(x)}\,\tau$) may be expressed as a function of the fully evaluated form of the input. The function obtained may thus be relative to one or both of the following:

- The size of the input.
- The possible canonical forms of the input (or worst/average/best case scenarios based upon the various canonical forms).

Bjerner presents some additional rules which generalise the above for an arbitrary vector of unknown input arguments.

Specialist rules for each type

Having formulated a set of general rules for open expressions, Bjerner specialises these for the main datatypes of type theory. For **natural numbers** and **lists**, this leads to the following set of additional terminology:

Course of values sequence This is the sequence of fully evaluated values of the relevant recursion operator (*prim* or *lrec*) for each possible input that is structurally less than or equal to the given major argument.

Recursive Pattern This is the sequence of evaluation degrees for each one of the course of values sequence.

Recursion Depth The number of iterations that must actually be performed to calculate the value of the recursion operator expression.

Component Evaluation Degrees These denote how far structural sub-components of the major argument have to be evaluated when evaluating the functional argument to the recursion operator.

In each case the evaluation degree of the major argument is defined in terms of a specialised subsidiary function applied to the component evaluation degrees and the recursion depth.

Boolean type expressions are analysed by taking the various cases possible arising from whether the boolean argument reduced to *True* or *False*.

With **higher-order functions** the results obtained for open expressions are used with the *ap* selector which applies a function to its argument. With higher-order functions *recurrence relations* are obtained as expressions for the constant time cost and the constant evaluation degree. This may be compared with the recursive equations, which are solved by fixpoint iterations, we obtain during abstract interpretation.

5.3 Example of Bjerner's Analysis

In this section we present an example of the application of Bjerner's analysis to a *TT* program. The *max* function may be defined informally thus:

$$\begin{aligned} max\,0\,n &\equiv_{df} n \\ max\,(succ\;m)\,n &\equiv_{df} succ\,(max\;m\,(pred\,n)) \end{aligned}$$

where *pred* is the predecessor function on the natural numbers, including 0. The above is a naive definition of a function to compute the maximum of two natural numbers. The formal definitions of the functions are as follows:

$$max \equiv_{df} \lambda m.\lambda n.(prec\;m\,n\,(\lambda a.\lambda b.(succ(b(pred\,n))))) \qquad (34)$$

$$pred \equiv_{df} \lambda n.(prec\;n\,0\,(\lambda p.\lambda v.\;p)) \qquad (35)$$

Now the **constant time cost** of *pred* when fully evaluated is:

$$\mathit{pred}\, m \downarrow W = \rho + 0 \downarrow \tau_1 + \sum_{j=1}^{|n|-1} ((\lambda p.\lambda v.\ p)\, n_j\, z_{(j)}) \downarrow \tau_{j+1} \tag{36}$$

where n is a fully evaluated value equivalent to the arbitrary input m. ρ is the **recursion depth** of the function, *pred*. It measures the number of recursive iterations that have to be performed. (If the primitive recursion is translated to an imperative form, via a tail-recursive representation, then it will equal the number of times that a while-loop is performed.) The recursion depth naturally depends upon whether the recursive calls of *prim* have to be evaluated at all. The **recursive pattern**, which is the sequence $\langle \tau_i \rangle$, denotes the extent to which the recursive calls have to be evaluated. The values of the recursive calls are the **course of values sequence**, denoted $z_{(j)}$ where:

$$z_{(j)} = (\mathit{prim}\ y_j\ d\ f)^W = \begin{cases} d^W, & \text{if } j = 1 \\ f\ y_{j-1}\ z_{j-1}^W, & \text{if } |y| \geq j > 1 \end{cases}$$

In the above, y_j represents a canonical form of a natural number e.g. y_1 corresponds to 0, whilst y_4 corresponds to 3. $|n|$ represents the **size** of the natural number n.

Returning to the example, the recursive pattern is calculated as follows:

$$\tau = \frac{(\lambda p.\lambda v.\ p)\, n\, z}{n}\, W = \frac{n}{n}\, W = W$$

for any values of n, z. The evaluation degree W shows that the expression, n, must be fully evaluated if *pred* is fully evaluated.

In the above, therefore, each one of the recursive pattern is equal to W. It also follows that the recursion depth is equal to $|n|$. The **component evaluation degrees** (which represent the degree to which the components of the argument have to be evaluated) are thus calculated as follows:

$$\sigma = \frac{n}{(\lambda p.\lambda v.\ p)\, n\, z}\, W = W$$

Now,

$$((\lambda p.\lambda v.\ p)\, n_j\, z_{(j)}) \downarrow \tau_{j+1} = n_j \downarrow \tau_{j+1} = 0$$

for all j since n_j is a (fully evaluated) component of the fully evaluated n.

It follows that:

$$\mathit{pred}\, m \downarrow W = 1 \tag{37}$$

where '1' indicates one cycle of the computation procedure.

The constant evaluation degree of *pred* may be found as follows.

$$\begin{aligned} \frac{y}{\mathit{pred}\, y}\, W &= \mathit{RMA}\, (\sigma_1 \ldots \sigma_{|y|-1})(\rho) && (38) \\ &= W && (39) \end{aligned}$$

In the above, RMA means the "rec major argument". (Bjerner refers to $prim$ as **rec**.) (38) follows from theorem 7.2 in [14]. (39) follows from the definition of RMA in [14] and the fact that the recursion depth is equal to $|y|$.

We treat the two-argument function, max, in a higher-order manner.

$$\begin{aligned}(max\,x\,y) \downarrow W &= (ap\,(max\,x_1)\,y) \downarrow W && (40)\\ &= 1 + w^{(0)}\,y \downarrow W + (max\,x_1) \downarrow W \qquad \{w = max\,x_1\} && (41)\\ &= 2 + w^{(0)}\,y \downarrow W && (42)\\ &= 2 + y \downarrow W && (43)\end{aligned}$$

where $w^{(0)}$ is the only component of the fully evaluated form of $max\,x_1$.

$$\begin{aligned}(ap\,(max\,x_{x+1})\,y) \downarrow W &= 1 + w^{(0)}\,y \downarrow W + (max\,x_{i+1}) \downarrow W\\ &\qquad \{w = max\,x_{i+1}\} && (44)\\ &= 2 + w^{(0)}\,y \downarrow W && (45)\\ &= 2 + ((\lambda n.Succ\,(ap\,(max\,x_i)\,(pred\,n)))\,y) \downarrow W && (46)\end{aligned}$$

Let

$$f' \equiv_{df} \lambda n.Succ\,(ap\,(max\,x_i)\,(pred\,n))$$

$$\begin{aligned}(f'\,y) \downarrow W &= (f'\,y') \downarrow W + y \downarrow (\frac{y'}{(f'\,y')}W) \qquad \{y' = y\} && (47)\\ &= \phi(f'\,y') + (f'[y'/n] \downarrow W) + y \downarrow (\frac{y'}{(f'[y'/n])}W) && (48)\\ &= 0 + (Succ\,(ap\,(max\,x_i)\,(pred\,y'))) \downarrow W\\ &\quad +\\ &\quad y \downarrow (\frac{y'}{ap\,(max\,x_i)\,(pred\,y')}W) && (49)\\ &= 1 + (ap\,(max\,x_i)\,(y_{|y'|-1})) \downarrow W + y \downarrow W && (50)\end{aligned}$$

(50) follows since

$$\begin{aligned}\frac{p}{(ap\,(max\,x_i)p)}W &= \frac{y'}{pred\,y'}W\\ &= W\end{aligned}$$

where p is a fully evaluated version of $pred\,y'$. (See the constant evaluation degree of $pred$ above (39).)

The equations (50) and (43) indicate that we have a simple recurrence relation. This may be solved to produce the following equality for the **constant time cost** of max:

$$max\,x\,y \downarrow W = 3 \times |x| - 1 + (y \downarrow W) \qquad (51)$$

A similar recurrence relation for the **constant evaluation degree** of x can also be constructed, which when solved gives W. This means that both parameters of max must be fully evaluated.

5.4 Neededness analysis aids time analysis

Wadler observed [145] that the work of Bjerner covered in Section 5.2 could be applied to lazy functional programming languages in general. Moreover, Wadler noted that abstract interpretation techniques could be used to assist in the process of analysing the time complexity of functional programs.

In this section we comment on Wadler's approach (which is based upon the concrete strictness analysis domain) and show how the simple neededness analysis will perform the same function. Moreover, we indicate how a neededness analysis may be used to give an upper bound (or *worst case*) for the time complexity of a program, whilst a simple strictness analysis may give a lower bound (or *best case*).

5.4.1 Description of Wadler's method

The technique proposed by Wadler [145] consists of replacing the evaluation degrees of Bjerner by the contexts of abstract interpretation. (Correspondingly, the evaluation degrees are undecidable in general, like contexts.) In particular, Wadler uses the concrete context domain for strictness analysis. This is a four-point domain which is equivalent to the strictness and absence lattice presented earlier in Section 2.5. It should also be noted that the contexts used by Wadler are domain projections, as discussed in [147]. Wadler's method is to use the contexts and context functions (termed "projection transformers" by Wadler) to determine whether the parameters to functions are needed by the computation and hence whether they make a contribution to the time complexity of the whole expression being analysed. The scheme devised by Wadler was applied to a simple language consisting of constants, variables, *if-then-else* conditionals, and function definitions, as described below.

Definition of Wadler's lazy time analysis

$$f^T\ x_1 \ldots x_n\ \mathbf{c}$$

is the number of call steps (called iterations of the computation procedure in [14]) required to evaluate

$$f\ x_1 \ldots x_n$$

relative to the context $\mathbf{c}$.

Now, it is assumed that f must have a definition of the form:

$$f\ x_1 \ldots x_n \equiv_{df} e$$

This is defined as having the following time complexity relative to a given context:

$$f^T\ x_1 \ldots x_n\ \mathbf{c} = \begin{cases} 0, & \text{if } \mathbf{c} = \mathbf{AB} \\ 1 + e^T\ \mathbf{c}, & \text{otherwise} \end{cases} \tag{52}$$

In the above, $e^T\,\mathbf{c}$ is the time complexity of the defining expression, e.

The time complexity of expressions is defined as follows. $e^T\,\mathbf{AB} = 0$ but if $\mathbf{c} \neq \mathbf{AB}$ then $e^T\,\mathbf{c}$ has the following definition:

$$
\begin{aligned}
k^T\,\mathbf{c} &= 0 && (53)\\
x^T\,\mathbf{c} &= 0 && (54)\\
(f\,e_1 \ldots e_n)^T\,\mathbf{c} &= (f^T\,e_1 \ldots e_n\,\mathbf{c}) + \\
&\quad (e_1^T\,(\mathbf{f_1}\,\mathbf{c})) + \ldots + (e_n^T\,(\mathbf{f_n}\,\mathbf{c})) && (55)\\
(\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)^T\,\mathbf{c} &= (e_0^T\,\mathbf{S} + (\mathit{if}\ e_0\ \mathit{then}\ e_1^T\,\mathbf{c}\ \mathit{else}\ e_2^T\,\mathbf{c}) && (56)
\end{aligned}
$$

In the above, $\mathbf{f_i}$ is the ith context function of the function f, that is, $\mathbf{f_i}\,\mathbf{c}$ gives the context corresponding to the ith parameter of f. Clauses (53), (54), (55) and (56), correspond to the time complexities of constants, variables, applications and conditionals, respectively.

As in the case of Bjener's method, conditionals are typically analysed by taking the worst, best or average cases, according to the possible outcomes of the boolean conditional expression (e_0 above).

5.4.2 Applying the Wadler method to type theory

We describe how Wadler's method may be extended so that it can be applied to type theory, thus providing an alternative method to that of Bjerner, which was described in Section 5.2. We suggest that this method is more easily mechanised than Bjerner's, since we have already provided a practical method for neededness analysis in Chapter 4. It is unclear, however, how Bjerner's method, which involves more *ad hoc* algebraic manipulation, may be automated.

It should be noted that the Wadler lazy time analysis method may be simplified immediately, since the analysis only depends on whether the given context is **ABSENT** or not. This means that the **neededness analysis** lattice (see Section 2.5) would be suitable for time analysis, as it makes a distinction between those which definitely will be unused by the computation and those which may be evaluated. Thus time analysis only requires the two-point neededness lattice rather than the four-point strictness and absence analysis one.

The time cost semantic function, $\mathcal{T}$ mapping from TT expressions and contexts to time complexity expressions is defined as follows. We assume that we have some environment, σ, of named function definitions. Also, $\mathcal{V}$ is the context expression evaluation function (see Section 2.14.3), ρ is an environment of context function definitions and $<e_1 \ldots e_n>$ is a set of input context expressions. The context expression formation functions, **efm** and **befm**, used below are described in 2.14.4. For ease of presentation, we have omitted σ, ρ and $<e_1 \ldots e_n>$ wherever they are not essential.

$\mathcal{T}$ is defined by the following clauses:

- $$\mathcal{T}\,[\![\, e \,]\!]\,\mathbf{U} = 0$$

for any expression e.

- $$\mathcal{T}\,[\![\, f \,]\!]\,\mathbf{c} = 1 + \mathcal{T}\,[\![\, e \,]\!]\,\mathbf{c}$$

  where f is a function name so that $\sigma\, f = e$.

- $$\mathcal{T}\,[\![\, \lambda a.\ e \,]\!]\,\mathbf{c} = 1 + \mathcal{T}\,[\![\, e \,]\!]\,\mathbf{c}$$

- $$\mathcal{T}\,[\![\, C(a_1 \ldots a_n) \,]\!]\,\mathbf{c} \quad = \quad \mathcal{T}\,[\![\, a_1 \,]\!]\,\mathbf{c}_1 + \ldots + \mathcal{T}\,[\![\, a_n \,]\!]\,\mathbf{c}_n$$

  where
  $$\mathbf{c}_i = (\mathcal{V}[\![\, \mathbf{C_i} \,]\!]\ \rho <e_1 \ldots e_n> \mathbf{c})$$
  where $\mathbf{C_i}$ is the ith context function corresponding to the TT constructor, C.

- $$\begin{aligned} \mathcal{T}\,[\![\, p \,]\!]\,\mathbf{c} &= 0 \\ \mathcal{T}\,[\![\, abort_A\ p \,]\!]\,\mathbf{c} &= 0 \end{aligned}$$

  whenever $p : \bot$

- $$\mathcal{T}\,[\![\, y \,]\!]\,\mathbf{c} = 0$$

  where y is a variable.

- $$\mathcal{T}\,[\![\, J(c,d) \,]\!]\,\mathbf{c} = \mathcal{T}\,[\![\, d \,]\!]\,\mathbf{c}$$

- $$\mathcal{T}\,[\![\, case\ x\ c \,]\!]\,\mathbf{c} = \mathcal{T}\,[\![\, c \,]\!]\,\mathbf{c}$$

- $$\mathcal{T}\,[\![\, if\ b\ then\ c\ else\ d \,]\!]\,\mathbf{c} = \mathcal{T}\,[\![\, b \,]\!]\,\mathbf{c} + \begin{cases} \mathcal{T}\,[\![\, c \,]\!]\,x, & \text{if } b \twoheadrightarrow True \\ \mathcal{T}\,[\![\, d \,]\!]\,x, & \text{otherwise} \end{cases}$$

- $$\mathcal{T}\,[\![\, cases_n\ v\ c_1 \ldots c_n \,]\!]\,\mathbf{c} = \mathcal{T}\,[\![\, v \,]\!]\,\mathbf{c} + \begin{cases} \mathcal{T}\,[\![\, c_1 \,]\!]\,\mathbf{c}, & \text{if } v \twoheadrightarrow 1_n \\ \vdots & \vdots \\ \mathcal{T}\,[\![\, c_n \,]\!]\,\mathbf{c}, & \text{if } v \twoheadrightarrow n_n \end{cases}$$

- $$\mathcal{T}\ [\![\ Fst(p,q)\ ]\!]\ \mathbf{c} = \mathcal{T}\ [\![\ p\ ]\!]\ \mathbf{c}$$

- $$\mathcal{T}\ [\![\ Snd(p,q)\ ]\!]\ \mathbf{c} = \mathcal{T}\ [\![\ q\ ]\!]\ \mathbf{c}$$

- $$\mathcal{T}\ [\![\ cases\ i\ f\ g\ ]\!]\ \mathbf{c} = \mathcal{T}\ [\![\ i\ ]\!]\ \mathbf{c}\ + \begin{cases} \mathcal{T}\ [\![\ ap\ f\ q\ ]\!]\ \mathbf{c}, & \text{if } i \twoheadrightarrow inl\ q \\ \mathcal{T}\ [\![\ ap\ g\ r\ ]\!]\ \mathbf{c}, & \text{if } i \twoheadrightarrow inr\ r \end{cases}$$

- $$\mathcal{T}\ [\![\ ap\ f\ a\ ]\!]\ \mathbf{c} = (\mathcal{T}\ [\![\ f\ ]\!]\ \mathbf{c}) + (\mathcal{T}\ [\![\ a\ ]\!]\ \mathbf{c}')$$

  where

  $$\mathbf{c}' = \mathcal{V}[\![\ cexp\ ]\!]\rho < e_1 \dots e_n > \mathbf{c}\ cexp = \mathbf{efm}\ [\![\ f\ ]\!]\ 1 < \mathbf{befm}\ [\![\ a\ ]\!] > \mathbf{v}$$

- $$\begin{aligned} \mathcal{T}\ [\![\ prim\ n\ c\ f\ ]\!]\ \mathbf{c} \quad = \quad & (|n| - 1) \times (\mathcal{T}\ [\![\ f'\ ]\!]\ \mathbf{c}) && (57) \\ & + \\ & (\mathcal{T}\ [\![\ n\ ]\!]\ \mathbf{d}) && (58) \\ & + \\ & (\mathcal{T}\ [\![\ c\ ]\!]\ \mathbf{e}) && (59) \\ & + \\ & (\mathcal{T}\ [\![\ f\ ]\!]\ \mathbf{m}) && (60) \end{aligned}$$

  where $|n|$ represents the size of n and f' is the normal form of f. Also,

  $$\mathbf{d} \quad = \quad \mathcal{V}[\![\ (\mathbf{prim_1\ pargs})\ ]\!]\rho < e_1 \dots e_n > \mathbf{c}$$

  (Similarly for $\mathbf{e}$ and $\mathbf{m}$ with regard to $\mathbf{prim_2}$ and $\mathbf{prim_3}$, respectively.) In the above, $\mathbf{pargs} = < \mathbf{abs_i} <>, \mathbf{befm}\ [\![\ c\ ]\!], \mathbf{befm}\ [\![\ f\ ]\!] >$. The context functions of *prim*, $\mathbf{prim_1}$, $\mathbf{prim_2}$ and $\mathbf{prim_3}$ are defined in 2.14.4.

  (57) arises from the solution to a recurrence relation for the time complexity of *prim*.

- The time complexity expressions for *lrec* and *trec* are similar to that for the **prim** function listed above.

5.4.3 Strictness analysis and lower bounds

It has been noted above that time complexity analysis is undecidable in general, just as properties such as necessity are undecidable. The use of neededness analysis thus gives an upper bound for the complexity of a program, since some parameters will be detected as unused when, in fact, they will be unused by the computation.

Suppose, however, we took the time complexity of each actual parameter to be 0 if there was the possibility that it would not be used. This may be done by employing the strictness analysis lattice, where **L** (which corresponds to "might be evaluated") is the abstract value which will induce a time complexity of 0.

5.5 Additional forms of static analysis

Type theoretic programs may be *specialized* to produce computationally more efficient forms, using the ideas of **partial evaluation** [83]. This consists of producing new forms of functions where possible normal forms have been substituted for one or more of the *static* parameters, where the possible values of a parameter can be obtained during compilation. The process is similar to the idea of currying and definition by partial application in functional programming. It would be possible in type theory to thus specialise each of the primitive recursive operators by *unfolding* their calls according to possible values of the major argument (i.e. the one which controls the number of iterations performed). We speculate that the strongly normalizing nature of type theory will mean that such specialization may be performed more easily in that a large number of arguments may be detected as static. However, to detect that an argument is static in general requires *binding time analysis*. Indeed, this can be combined with strictness analysis, but the flow of information is more naturally forwards [88].

It should be noted, however, that the strong normalization property of type theory does place a restriction on the idea of building program generators, such as compilers, by self-application. (Such techniques are known as the Futamura projections, which are described in [50, 83]. Jorgensen has shown how such ideas can be used to produce *efficient* compilers for lazy functional languages [84, 85].) This self-application is impossible in type theory (unlike the untyped lambda calculus in which a self-interpreter can be built [8]) and so we would have to restrict the type theory to, say, the ωth universe in order to build an interpreter in a type theory including the $\omega + 1$th universe.

It would also be interesting to investigate a development of Burstall and Darlington's fold/unfold program transformation technique [24] in the setting of TT where strong normalization is guaranteed. In particular, there would be a need to establish fold rules which were consistent with the proof rules of TT.

Additional static analyses that may be applied to type theory include space complexity analysis and the automatic detection of possible parallel processing (which has been investigated for lazy functional programs in [64]).

5.6 Conclusion

We have described Bjerner's method for calculating the time complexity of type theoretic programs, under a lazy evaluation strategy. Subsequently, we have expanded upon Wadler's idea of using abstract interpretation to detect the computational demand required of each sub-expression, in order to produce a method of time analysis based upon the neededness analysis of Chapter 2. This time complexity analysis could thus be used mechanically in the implementation presented in Chapter 4.

In addition, other possible forms of static analysis and optimisation, such as partial evaluation have been discussed. We speculate that a combination of these analyses may be used to provide an efficient implementation of type theory.

Chapter 6

Conclusions

In this chapter we present a review of the foregoing work, outline areas for possible further research and give our conclusions.

6.1 Review

We have discussed how modifications to type theory have been made in order to remove computationally redundant proof objects, so as to improve the efficiency of type theory as a functional programming language. However, we have argued that such modifications complicate the original theory of Martin-Löf and, moreover, are unnecessary if abstract interpretation techniques such as the one that we have developed are employed.

In Chapter 2 we presented a scheme for backwards analysis of type theory, building on the ideas of Hughes [69, 72]. This backwards analysis method allows computationally redundant proofs to be detected automatically at compile-time. Furthermore, the property of whether a parameter is required by the computation may be detected by a single analysis which will also detect other properties of the program. These other properties, which indicate whether expressions are required for evaluation or if they may be shared, can also be used to make compile-time optimisations on type theoretic programs. The basic theory of backwards analysis was extended to structured data and higher-order functions and two examples were given which demonstrated how the backwards analysis technique may detect computationally redundant parameters and, dually, those parameters which must be computed. To give the analysis a firm theoretical backing, the analysis was formalised by giving a denotational abstract semantics to each of the constructs of the TT system presented in [134]. Finally, since types and terms may be intertwined in type theory, we discussed how types might be given an abstract interpretation.

The backwards analysis techniques developed would be valueless if they produced results which were incorrect with respect to the computational semantics of type theory. Thus in Chapter 3 we gave a proof of the correctness of the abstract interpretation in conjunction with a formal presentation of the rules of type theory. This showed that the analysis was indeed safe with regard to the property of neededness and so any parameter that was detected by the analysis as being computationally redundant would indeed not be required in a normal order computation of the program.

In Chapter 4 we reported on how the abstract interpretation was implemented within the Ferdinand functional programming language, which is based upon type theory. The implementation showed that the backwards analysis scheme could have practical benefits and there were signs of promise for efficient, practical programming systems to be developed from type theory.

Chapter 5 showed how other static analysis techniques may be applied to type theory and how the backwards analysis techniques that were presented earlier to detect neededness may be used to refine time complexity analysis of type theoretic programs. This gave an insight into how type theory may be analysed and optimised further.

6.2 Possible developments of the work presented

In Chapter 2 the analysis we presented did not indicate possible relationships with regard to an abstract property between the parameters for a function. So our analysis did not detect joint neededness (where each parameter is needed if and only if the others are), for example. If this method was developed then it might allow further optimisations to be made to the programs of type theory. However, this conversion to a *high-fidelity* analysis will make the analysis less efficient. High-fidelity analyses for a basic lazy functional programming language have been given in [38, 39].

It would also be useful if a practical scheme could be developed to analyse precisely functions which apply functions extracted from lists. The present system simply approximates such occurrences by the top element of the lattice, since it is possible for an arbitrary function to be included within a data structure: the top element will always be a safe abstraction of such applications.

An investigation of possible schemes for the resolution of typing information within type theoretic systems would be instructive. This would give a background to the discussion of the analysis of types in Chapter 2. (In the Ferdinand system of Chapter 4, type checking is carried out as a separate phase before abstract interpretation and code generation.) It may be more efficient to attempt to integrate the type checking and code generation phases if terms are detected as being required in both type checking and in program computation.

There are several possible areas for future improvement of the **Ferdinand** system and the backwards analysis that we have integrated within it. One of the main difficulties encountered in implementing backwards analysis within the compiler was the fact that the types of functions had been lost during the lambda lifting phase. If the types could be preserved throughout type checking then this would allow the backwards analysis to proceed more efficiently and also to give more precise results with regard to functions defined by partial application.

We suggest that other possible improvements to the Ferdinand implementation would be:

- A mechanism for implementing an optimisation based upon sharing analysis. We were able to determine which expressions may or may not be shared within an implementation but it was unclear how optimisations to the resulting FLIC code could be made. We speculated that these optimisations could only be done by some mechanism within the back-end to the compilation process (the fc compiler).

- Further optimisations could be made to the FLIC code produced by Ferdinand. This is due to the fact that many trivial function definitions (which could be removed from the code) remain at the end of the compilation process. For instance, some function definitions consist solely of the name of another function. In general, we would like to be able to remove such definitions which are eta-equivalent to others.

- A full integration of the standard environment functions within Ferdinand would, we speculate, improve efficiency greatly. At present the standard environment is not treated differently to any other script and is not automatically included by the compiler. If the standard environment was included automatically then the initial environment of context expressions could similarly be augmented to provide the context functions of functions such as *map*. The full-integration of the standard environment within the compiler would thus have the following benefits:

  1. Efficient, object code forms of the standard environment could be called by the compiler for each occurrence of a standard function.
  2. The backwards analysis process would be faster (since the standard functions would not have to be re-analysed during every compilation) and probably more precise since the pre-supplied context information, which would be derived by hand, would not have as many safe approximations included as those generated by an automated analysis.

- Partial evaluation may be employed within the Ferdinand compiler, in conjunction with the backwards analysis optimisations. This would allow function definitions to be folded or unfolded to reach a more efficient form. The Ferdinand system would, in its current form, appear to be particularly amenable to such an approach since there are no input or output streams and all programs are translated to the primitive recursive forms of type theory. Hence, there would appear to be a sizeable amount of static information available during a Ferdinand compilation which may be used to create specialized programs.

- It would be helpful to have an estimate of the efficiency of the backwards analysis phase in the Ferdinand compiler and, in general, to have a set of statistics describing the computations required to analyse a Ferdinand program. This may be done by adding an output monad which will capture the trace of the execution [146].

In Chapter 3 we proved that the analysis was correct with respect to the property of neededness. However, it remains to be shown that the optimisations thus employed in the Ferdinand compiler in Chapter 4 are correct. Burn and Le Métayer have done this with respect to strictness analysis of lazy functional programs [22]. Their method consists of giving a continuation-passing model to a program and its optimised form. It is unclear, however, whether this method would be appropriate for the optimisation that we have given.

6.3 Areas for further research

6.3.1 Additions to type theory

In the work presented, we have applied our backwards analysis techniques to the whole of the system TT [134], including lists. In order to have a fully-featured programming system, however, some additions need to be made to the system.

Inductively defined types

In the system studied, the inductively defined data structures, *list* and *tree* have been included. These are examples of how rules for new types were added to the open system of type theory [113]. Indeed, new structures modelling mutually recursive types have been added to type theory in Section 6 of [5], following on from the development of parse trees by Chisholm [27].

It may be desirable, however, to have a general method of producing inductively defined types, such as the W types. The W types were proposed in [96] and have the following introduction, elimination and computation rules (as given in Section 5.10 of [134]):

$$\frac{a : A \qquad f : (B(a) \Rightarrow (W\, x : A).B(x))}{node\; a\; f : (W\, x : A).B(x)} \; (W\; Intro)$$

Above, the function f gives the set of predecessors to the node.

$$\frac{w : (W\, x : A).B(x) \qquad R : Ind(A, B, C)}{(Rec\; w\; R) : C(w)} \; (W\; Elim)$$

Here $Ind(A, B, C)$ represents the induction step over the W-type. It is a generalisation of a formula such as

$$(\forall n : N).(C[n/x] \Rightarrow C[(succ\; n)/x])$$

which is the induction step for natural numbers. The computation rule is:

$$Rec\; (node\; a\; f)\; R \rightarrow R\; a\; f\; (\lambda x.Rec\; (f\; x) R)$$

This generalises a computation rule such as:

$$\begin{array}{rcl} lrec\,[]\,s\,f & \rightarrow & s \\ lrec\,(a :: l)\,s\,f & \rightarrow & f\,a\,l\,(lrec\,l\,s\,f) \end{array}$$

The problem with the W type, however, is that it contains many elements which are *extensionally equal* but different intensionally. (See pp. 188-190 of [134] with regard to modelling the *tree* type via a W type.) The potential difficulties of implementing the W types in practice are given in [43].

It may also be seen that the use of W types will produce a *loss of precision* in any backwards analysis upon them. For instance, for the *tree* type we were able to deduce an atomic context for the entire structure with contexts for the natural term and the left and right subtrees at each node. However, if *tree* is formulated as a W-type then we will only be able, for the structured part of the context, to deduce a context for the natural number object and a *single* context representing the subtrees. Indeed, it would appear that this is the best that may be hoped for, since our backwards analysis is *purely intensional* in nature: we rely upon the syntax of pattern matching to induce structured contexts. Hence, unless the structural recursion over W types is made explicit by pattern matching over various constructs, our structured contexts will not capture very accurate information about a structured type.

Conversely, however, an excess of structured information leads to a rapid expansion in the size of the context lattices and thus the performance of a backwards analyser may suffer. It is therefore an open problem as to how much contextual information about an arbitrary data structure is required in order to make significant gains in efficiency in the resulting object code.

An alternative method to form inductively defined structures is to take the least fixed point of a monotonic operator, Θ, over types [100, 46]. The introduction rules for inductively defined types then follow from the fact that we have the following convertibility relation:

$$Fix\ \Theta \twoheadleftarrow\!\!\twoheadrightarrow \Theta\,(Fix\ \Theta)$$

The computation rule for these types is similar to the above and has the following form:

$$fix\ g \rightarrow g\,(fix\ g)$$

This arises from the elimination rule:

$$\frac{\begin{array}{c}[\,T \subseteq Fix\ \Theta\,] \\ \vdots \\ g : (\forall x : T).C \Rightarrow ((\forall y : \Theta T).C[y/x])\end{array}}{fix\ g : (\forall z : Fix\ \Theta).C[z/x]}\quad (Ind\ Elim)$$

(Above, $T \subseteq Fix\ \Theta$ is the judgement introduced by Mendler [100] to indicate the ordering of a type hierarchy with regard to recursive equations for types.)

The computation rule (which is a fixpoint calculation), as seen above, may be seen to give little promise for deducing contexts of parts of structures. However, it is argued in [46] that introduction and elimination rules for each inductively defined type may be read from the definition of the corresponding monotonic operator, Θ. It would seem, therefore, that using this approach there would be more scope for a backwards analyser to deduce the contexts of substructures of inductively defined types. However, as noted on p.310 of [134] the disadvantage of this system is that equality between types becomes undecidable.

Another formulation for a general scheme of inductively defined types is given by Luo in Section 9.2.2 of [90] as a development of the extended calculus of constructions. The method used there is similar to the scheme of W types which we have mentioned above. It would be interesting to determine how much contextual information may be deduced from the structures of Luo's system.

Well-founded recursion

The recursion schemes that we have seen for lists, trees and the more general forms of inductively defined types have all been forms of primitive recursion over the structure of types. There have been suggestions for adding restricted forms of general recursion which preserve the strong normalisation property of type theory.

The idea, which has been proposed by Paulson [116] and, in collaboration, by Saaman and Malcolm [121] and Nordström [109], has been to use *well-founded* orderings (which are partial orders with no infinitely descending chains). Paulson makes a definition of a well-founded ordering and it then remains to prove that a given partial order is thus well-founded. The approach of Nordström, Saaman and Malcolm, however, is to use sequences of elements of a partial order which do not form an infinite descending chain — these are called the *accessible elements* of a type. However, it has the disadvantage that additions have to be made to the system, particularly the addition of a proposition indicating membership of the accessible elements at each type [121]. We hypothesise that the latter system is one that is implicitly formulated in terms of the subset theory, whilst the system of Paulson follows the pattern of the original type theory. This is also borne out by the computation rules for the two systems. For that of Nordström, Saaman and Malcolm, we have:

$$rec\; e\; p \rightarrow e\; p\, (rec\; e)$$

where p is a member of the accessible elements of the type and e simply calculates the values at each node. In Paulson's system, however, we have:

$$\mathbf{wfrec}\; PFx \rightarrow F(\lambda y.\lambda r.(\mathbf{wfrec}\; PFy))$$

(Originally the place of $\rightarrow$ was taken by equality but the reduction symbol may be validly inserted instead.) Here the first parameter, P is a proof that an element y is less than x with respect to the well founded relation. It is readily seen that such a proof is computationally irrelevant but we suggest that a backwards analysis would automatically detect this redundancy. (Indeed, Paulson does this *ad hoc* for his derivation of a quicksort algorithm in [116, Section 5].)

Co-inductions

As a dual notion to the idea of presenting inductively defined types as the least fixed point of a monotonic recursive type equation, we may form the greatest fixed point of such equations. These constructs are thus *infinite* and allow us to characterise lazy streams in type theory. The details of how such infinite objects are defined in type theory by **co-inductions** are given in Section 7.11 of [134].

The interesting aspect of these infinite data structures is that the contexts of their recursive parts must be lazy (i.e. a proper tail context of an infinite list must be $\sqsupseteq$ **ABSENT**), since it is obviously impossible to evaluate completely an infinite list. Thus the distinction made in type theory made between finite and infinite data structures will allow us to make a division between the contexts used for the two kinds of structure. The knowledge that recursive parts of an infinite data structure must be lazy should allow the analysis to be more efficient and precise than in the case of lazy languages such as Miranda, where the finite and infinite structures are not differentiated by their types.

6.3.2 Other static analyses

In Chapter 5 we mentioned the scope for future work using static analysis techniques such as partial evaluation [83] and how we may determine other properties of type theoretic programs such as the detection of the possibility of parallelisation [64]. An interesting alternative method to the abstract interpretation techniques that we have employed is **abstract reduction** [107] which has been applied to the Concurrent Clean lazy functional programming language [108] in order to determine strictness. This technique involves applying the reduction rules of a functional system using abstract values. (The flow of information is therefore forwards.) Abstract reduction deals with recursion by analysing the abstract reduction sequences (termed *reduction path analysis*) rather than by solving a fixpoint equation. Since abstract reduction has shown good results with both higher-order functions and data structures, this would appear to be a fruitful technique to apply to type theory. In particular, it would be interesting to discover whether the strong normalisation property of type theory aids the reduction path analysis method.

Such static analysis methods would all be employed at the compile-time level. We speculate below whether a form of static analysis is feasible during *program development*.

Analyses in logical form

It has been observed that it is possible to perform strictness analysis using a Hindley-Milner type inference system [62, 101] with *abstract types* which denote strictness [151, 10, 80] and Hankin and Le Métayer have given a technique for deriving efficient static analysis algorithms from type inference systems [55]. Consequently, each inference of a type of a term during compilation may thus allow the strictness of the term to be inferred.

It has been intended that the type inference method of deducing strictness properties would be applied at compile-time, during the type checking phase. However, if we have a theorem-proving system such as ALF [4], in which the programs of type theory may be derived, then for each deduction made (which will prove some type in the theory) we speculate that it would be possible for this to entail the automatic deduction of an abstract type by the theorem prover.

There are, however, some problems that may be foreseen with this idea. Firstly, the system of types in TT is richer (including dependent types, for instance) than for the systems proposed by Benton [10] and others. Consequently, it would be necessary to make a substantial augmentation to the existing program logics. Secondly, the underlying models of the abstract properties deduced by these systems are based upon Scott-closed sets and it has been shown by Kamin [87] that finite combinations of these are inadequate for expressing the property of head strictness of a list, for example[1]

Nevertheless, the development of such a logic for type theory would provide an interesting contrast with both the system of backwards analysis that we have presented and the alterations to the type theory that have been proposed in order to eliminate computational redundancy. Such a system of deducing abstract properties via types would operate during program development rather than compile-time but, unlike the additions to type theory such as subsets, or the system of Paulin-Mohring for marking redundant proof objects [115], the process would not be visible to the programmer.

6.4 Concluding remarks

We have shown that static analysis techniques, in particular the backwards analysis form of abstract interpretation, may be used to optimise type theoretic programs. Specifically, we have developed an analysis which is capable of providing an *automatic* means of detecting both computational redundancy *and* properties used to perform optimisations on lazy functional languages such as Haskell. Consequently we conclude that modifications to the theory in order to remove computational redundancy, such as the subset type and the subset theory of [114], are unnecessary and we may adhere to a type theory based upon the original ideas of Martin-Löf [95, 134] which identifies logical propositions and types.

Our theory of backwards analysis has been applied to the whole of the TT system, including lists, described in [134]. The abstract interpretation theory has been directed towards obtaining a method for a practical implementation within a compiler for a language based upon type theory. With our proof of the soundness of the analysis in Chapter 3, we believe that we have satisfied Nielson's dictum [106] that an abstract interpretation should consist of an induced (i.e. purely theoretically driven) analysis, an implementable analysis and correctness

[1] Hankin and Le Métayer have used their *lazy type inference* technique [56] to emulate Wadler's four-point strictness analysis for lists [144] but this does not capture head strictness.

verification.

We conclude that the type theory of Martin-Löf not only provides a system for the integration of proofs and program development but that static analysis techniques may be developed that automatically improve the efficiency of programs written in the formalism.

Appendix A

Examples of backwards analysis

A.1 Introduction to the examples

In this part of the report we present several examples of the application of backwards analysis to type theory. Most of the examples are functions used in the definition of quicksort, which is analysed in Section 2.13. The syntax is that used in [134].

Section A.2 covers numerical functions in type theory. The *lesseq* function (a boolean function ordering the natural numbers) covers many aspects of the way we perform sharing analysis upon functions. It covers the two distinct methods for calculating fixpoints of our context functions: the first method is to find fixpoints for *particular* contexts and the second is to find fixpoints for *arbitrary* contexts where we perform algebra to find the solution for an arbitrary initial value. We naturally find the same results by both methods. Only the first method is considered in [72]. We suggest that the first method will be better for mechanical evaluation in practice but that the second method is more efficient (and elegant) when we are finding results for "simple" functions (such as the basic selector functions of type theory) as a prelude to mechanical evaluation of more complicated functions. In this example we also show how we can improve upon the information we gain from backwards sharing analysis by distinguishing between contexts corresponding to different nullary constructors. This is again distinct from Hughes's approach.

Section A.3 gives analyses of functions upon lists in type theory. The particular example to note here is that of *append*: the generalised results that we obtain may again be compared to the point-by-point analyses of the same function given in [72]. Included also is the higher-order function, *map*.

A.2 Numerical functions

A.2.1 Backwards analysis of the *lesseq* and *greater* functions in TT

Introduction

This subsection gives a sharing analysis of the *lesseq* function in type theory. The analysis first shows that the first argument must be used although its predecessor may not necessarily be fully evaluated. In addition, the second argument may not necessarily be evaluated at all and even if it is it may not be fully evaluated. Further analysis is done to indicate the mutual dependency between the two arguments and a method is developed in which contexts depend upon the boolean results of *lesseq*. The analysis of the *greater* function upon natural numbers is subsequently presented: it is defined in terms of the *lesseq* function.

Definition of the function

$$\mathit{lesseq} \quad : \quad (N \Rightarrow N \Rightarrow \mathit{bool})$$

$$\mathit{lesseq}\, 0\, x \equiv_{df} \mathit{True} \tag{61}$$

$$\mathit{lesseq}\,(n+1)\,0 \equiv_{df} \mathit{False} \tag{62}$$

$$\mathit{lesseq}\,(n+1)\,(m+1) \equiv_{df} \mathit{lesseq}\, n\, m \tag{63}$$

Analysis of the first argument

We first formulate the context function of the first argument of *lesseq* for an arbitrary initial context $\mathbf{c}$. In order to simplify the process of analysis it is useful to assume that we are dealing only with strict contexts and that, since *lesseq* is defined by pattern matching, we will not be altering the atomic part of the context we begin with, $\mathbf{c}$. (It should be emphasised that these assumptions do not make the analysis invalid. The first assumption is valid by $\mathbf{f}\,(\mathbf{AB} \sqcup \mathbf{c}) = \mathbf{AB} \sqcup \mathbf{f}(\mathbf{c})$: this equation allows us to use **atst** below. The second follows from the definition of context propagation with regard to pattern matching given in Section 2.8.5.) To emphasise this we use the notation, $\mathbf{atst}(\mathbf{c})$ (the atomic, strict part of $\mathbf{c}$), for the atomic part of the resulting context. The *lesseq* function may be divided into two parts: the first which deals with the case that the first argument evaluates to 0 (clause (61) of the *lesseq* function) and the second which deals with a non-zero first argument. We, naturally, do not know which of these parts will apply in the actual execution of the function: this uncertainty is shown in the sharing analysis by the $\sqcup$ operator. In other words, we are joining together the contexts which result from each of the possible two parts. The subscripted contexts below reflect the fact that we are dealing with these two cases. One context has, as a subscripted context, the context constructor $\mathbf{0}$: this is the context corresponding to a zero natural number. (We may also view this as saying that an argument has to be *fully evaluated* in order to have such a context.) The context from the other part uses the **Succ** context constructor as its structured part, reflecting the fact that we are dealing with a non-zero natural number argument. Here the structured part has a context variable which has to be evaluated, namely the argument of **Succ**: this context variable gives us information about the predecessor value of the argument. (This helps us discover the level to which a natural number argument may have to be evaluated.) We may thus form the following expression for the context function of the first argument of *lesseq*:

$$\mathbf{lesseq_1 c} = \mathbf{atst(c)}_{\mathbf{0},\,\mathbf{Succ}(\mathbf{c} \xrightarrow{(62),(63)} n)}$$

The subscripted part of the second of this disjunction of contexts may, as it refers to clauses (62) and (63) of *lesseq*, be split into two parts. Here the two cases arise from the form of the second argument which may also be zero or not. For this part of the analysis we thus have:

$$\begin{aligned} \mathbf{c} \xrightarrow{(62),(63)} n &= \mathbf{c} \xrightarrow{False} n \sqcup \mathbf{c} \xrightarrow{lesseq\,n\,m} n \\ &= \mathbf{AB} \sqcup \mathbf{lesseq_1 c} \end{aligned}$$

The above follows from n (the numerical predecessor to the first argument) not being present in the boolean expression *False* and the second context follows directly from our definition of the analysis of function applications. For the sake of notational convenience, we shall leave out the first part (with the **0** context constructor: this part is solved as **0** is nullary.) We have thus to solve the following recursive equation:

$$\mathbf{lesseq_1 c} = \mathbf{atst(c)}_{\mathbf{Succ(AB} \sqcup \mathbf{lesseq_1 c)}}$$

This can be solved via a fixpoint iteration. We can guarantee that we will find a fixpoint due to the fact that the context functions are computable and that we are dealing with finite lattices of contexts. As shown below, the third in a series of fixpoint iterations gives the fixpoint. Note that we have to make approximations to the resulting context after the second iteration since the context has more than one level of subscripting: we assume that natural number contexts have either the subscript **0** or the subscript **Succ(d)**. The latter case means that we assume that **d** itself is of the form **Succ(d)**.

$$\begin{aligned} (\mathbf{lesseq_1\, c})^0 &= \mathbf{CONTRA} \\ (\mathbf{lesseq_1 c})^1 &= \mathbf{atst(c)}_{\mathbf{Succ(AB)}} \\ (\mathbf{lesseq_1 c})^2 &= \mathbf{atst(c)}_{\mathbf{Succ(AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ(AB)}})} \\ &\precsim \mathbf{atst(c)}_{\mathbf{Succ(AB\ \sqcup\ atst(c))}} \end{aligned}$$

$\mathbf{c}$	$\{1\}$	$\{1, M\}$
$(\mathbf{lesseq_1 c})^0$	$\mathbf{CR}$	$\mathbf{CR}$
$(\mathbf{lesseq_1 c})^1$	$\{1\}_{\mathbf{Succ}(\{0\})}$	$\{1, M\}_{\mathbf{Succ}(\{0\})}$
$(\mathbf{lesseq_1 c})^2$	$\{1\}_{\mathbf{Succ}(\{0,1\})}$	$\{1, M\}_{\mathbf{Succ}(\{0,1,M\})}$
$(\mathbf{lesseq_1 c})^3$	$\{1\}_{\mathbf{Succ}(\{0,1\})}$	$\{1, M\}_{\mathbf{Succ}(\{0,1,M\})}$
$\mathbf{lesseq_1 c}$	$\{1\}_{\mathbf{Succ}(\{0,1\})}$	$\{1, M\}_{\mathbf{Succ}(\{0,1,M\})}$

Table 10: Iteration using specific elements in the context domain.

$$\begin{aligned}
(\mathbf{lesseq_1 c})^3 &\precsim \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)})})} \\
&= \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)})}
\end{aligned}$$

The above illustrates how a fixpoint solution may be found by using purely *algebraic* manipulation upon an arbitrary argument $\mathbf{c}$. Mechanically, however, this may not be straightforward in more complicated cases. Therefore, in practice, results for particular contexts will be obtained. This is illustrated in Table 10. Note that these results agree with those obtained for the algebraic method for these contexts (given in Table 11).

Analysis of the second argument

$$\begin{aligned}
\mathbf{lesseq_2 c} &= \mathbf{c} \xrightarrow{True} x \\
&\quad \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{c} \xrightarrow{lesseq\ n\ m} m)} \\
&= \mathbf{AB} \\
&\quad \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{lesseq_2 c})}
\end{aligned}$$

A fixpoint iteration shows that the second argument may not be used, with any predecessor not necessarily being used even if the argument is.

$$\begin{aligned}
(\mathbf{lesseq_2\ c})^0 &= \mathbf{CR} \\
(\mathbf{lesseq_2\ c})^1 &= \mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ(CR)}} \\
(\mathbf{lesseq_2\ c})^2 &= \mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ(CR)}})} \\
&\precsim \mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)})}
\end{aligned}$$

c	$\{1\}$	$\{1, M\}$
$\mathbf{lesseq_1\ c}$	$\{1\}_{\mathbf{Succ}(\{0,1\})}$	$\{1, M\}_{\mathbf{Succ}(\{0,1,M\})}$
$\mathbf{lesseq_2\ c}$	$\{0, 1\}_{\mathbf{Succ}(\{0,1\})}$	$\{0, 1, M\}_{\mathbf{Succ}(\{0,1,M\})}$

Table 11: Table of results for the context functions of *lesseq*.

$$
\begin{aligned}
(\mathbf{lesseq_2 c})^3 &\stackrel{\sqsubseteq}{\sim} \mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)})})} \\
&= \mathbf{AB} \sqcup \mathbf{atst(c)}_{\mathbf{Succ}(\mathbf{AB} \sqcup \mathbf{atst(c)})}
\end{aligned}
$$

Values given by the context functions

Whilst the atomic values are as might be expected from an inspection of the definition of the function, the subscripted contexts provide us with something of an anomaly: they indicate that in no case is either argument *necessarily* fully evaluated. We can show, however, that one or other of the arguments must be fully evaluated by applying the following *forwards analysis* where **1** stands for "may not be fully evaluated" and **0** indicates "must be fully evaluated". Naturally, we wish the result of *lesseq* to be fully evaluated i.e. it should be **0**.

Let n' and m' represent the two variables.

$$
\begin{aligned}
\mathit{lesseq}^* \, [n']\rho \, [m']\rho &= (\mathit{PM_1}^* \, [n']\rho \, [m']\rho) \\
&\quad \sqcap ((\mathit{PM_1}^* \, [n']\rho \, [m']\rho) \\
&\qquad \sqcap (\mathit{True}^* \sqcup \mathit{False}^* \\
&\qquad\quad \sqcup \mathit{lesseq}^* \, [n']\rho \, [m']\rho)) \\
&= [n']\rho \\
&\quad \sqcap ([m']\rho \sqcap \\
&\qquad\quad (\mathbf{1} \sqcup \mathbf{1} \sqcup \mathit{lesseq}^* \, [n']\rho \, [m']\rho)) \\
&= [n']\rho \sqcap ([m']\rho \sqcap \mathbf{1}) \\
&= [n']\rho \sqcap [m']\rho
\end{aligned}
$$

(Above the asterisked names indicate the appropriate abstract functions. In particular $\mathit{PM_1}^*$ is the abstract function formed from pattern matching upon the first argument.)

So,

$$\mathit{lesseq}^* \, \mathbf{1} \, \mathbf{1} = \mathbf{1}$$

We thus have a contradiction to our assumption about the result of *lesseq* being fully evaluated if we also assume that neither argument need be fully evaluated. This means that at least one argument must be fully evaluated.

$\mathbf{c}$	$\{1\}$	$\{1, M\}$
$\mathbf{lesseq}_1\ \mathbf{c}$	$\{1\}_{[\mathbf{T}]\,\mathbf{0} \sqcup [\mathbf{F}]\,\mathbf{Succ}(\{0,1\})}$	$\{1, M\}_{[\mathbf{T}]\,\mathbf{0} \sqcup [\mathbf{F}]\,\mathbf{Succ}(\{0,1,M\})}$
$\mathbf{lesseq}_2\ \mathbf{c}$	$\{0, 1\}_{[\mathbf{F}]\,\mathbf{0} \sqcup [\mathbf{T}]\,\mathbf{Succ}(\{0,1\})}$	$\{0, 1, M\}_{[\mathbf{F}]\,\mathbf{0} \sqcup [\mathbf{T}]\,\mathbf{Succ}(\{0,1,M\})}$

Table 12: Modified table of results for the context functions of *lesseq*.

Reanalysing *Lesseq*

To remedy the original analysis we simply distinguish between the two possible results of lesseq i.e. $\mathbf{c}_{\mathbf{True}}$ and $\mathbf{c}_{\mathbf{False}}$.

We then obtain the following:

$$\begin{aligned} \mathbf{lesseq}_1\ \mathbf{c}_{\mathbf{True}} &= \mathbf{at}(\mathbf{c}_{\mathbf{True}})_{\mathbf{0}} \\ &= (\mathbf{strict}\ \mathbf{c})_{\mathbf{0}} \end{aligned}$$

The context function of the second argument of *lesseq* is found similarly in the other possible case:

$$\mathbf{lesseq}_2\ \mathbf{c}_{\mathbf{False}} = (\mathbf{strict}\ \mathbf{c})_{\mathbf{0}}$$

We thus conclude that the first argument of *lesseq* should be fully evaluated to 0 if the result of the function is *True* but otherwise only the second argument need be fully evaluated (again to 0).

It might appear that such an analysis is rather irrelevant (and rather trivial) since we cannot tell at compile time what will be the result of the function. However, such information may well be useful to analyse functions which call lesseq.

Nevertheless, such an analysis is only practical with the finite types of low orders (2 or 3) as we otherwise run into the levels of computational complexity that occurred with forward analysis. As, however, the booleans are probably amongst the most frequently used of the finite types, this approach has some merit. Moreover, this teaches us to be more careful when combining contexts: we may label the results so that we may be clear as to which **Nat** context constructor has been substituted for which **Bool** context constructor. This is shown in Table 12.

The *Greater* function

The *greater* function is defined using *not* and *lesseq* and hence its analysis is relatively simple.

Definition of the function

$$\begin{aligned} greater &\;:\; (N \Rightarrow N \Rightarrow bool) \\ greater\,x\,y &\equiv_{df} not\,(lesseq\,x\,y) \end{aligned}$$

Definition of the *not* function

$$\begin{aligned} not &\;:\; (bool \Rightarrow bool) \\ not\ True &\equiv_{df} False \\ not\ False &\equiv_{df} True \end{aligned}$$

Analysis of the *not* function

$$\begin{aligned} \mathbf{not}_1\,\mathbf{c}_{\mathbf{False}} &= \mathbf{c}_{\mathbf{True}} \\ \mathbf{not}_1\,\mathbf{c}_{\mathbf{True}} &= \mathbf{c}_{\mathbf{False}} \end{aligned}$$

Analysis of the first argument

$$\begin{aligned} \mathbf{greater}_1\,\mathbf{c} &= \mathbf{c} \xrightarrow{not\,(lesseq\,x\,y)} x \\ &= \mathbf{not}_1\,\mathbf{c} \xrightarrow{lesseq\,x\,y} x \\ &= \mathbf{lesseq}_1\,(\mathbf{not}_1\,\mathbf{c}) \end{aligned}$$

(Note that the context function $\mathbf{not}_1$ does not alter the atomic part of $\mathbf{c}$ and only "negates" the boolean context constructor.)

Analysis of the second argument

Similarly,

$$\mathbf{greater}_2\,\mathbf{c} = \mathbf{lesseq}_2(\mathbf{not}_1\,\mathbf{c})$$

A.2.2 Simon Thompson's example

Introduction

We perform sharing analysis upon one of the functions given by Simon Thompson at a UKC Theoretical Computer Science seminar. This analysis shows that the second argument is not needed and that the first argument is "strict", may be shared, and will be fully evaluated.

Definition of the function

The function is defined as follows:

$$
\begin{array}{lll}
g\ a\ b & \equiv_{df}\ \ g\ (a-1)\ (b+a) & ,\ \ \textbf{if}\ a > 0 \\
 & \equiv_{df}\ \ 17 & ,\ \ \textbf{otherwise}
\end{array}
$$

Basic analysis of the first argument

$$
\begin{array}{lll}
\mathbf{g_1\ c} & = & ((\mathbf{greater_1}\ (\mathbf{at}\ (\mathbf{c})_{\mathbf{True}})) \\
 & & \ \&\ (\mathbf{c} \xrightarrow{g\ (a-1)\ (b+a)} a)) \\
 & \sqcup & \\
 & & ((\mathbf{greater_1}\ (\mathbf{at}\ (\mathbf{c})_{\mathbf{False}})) \\
 & & \ \&\ (\mathbf{c} \xrightarrow{17} a)) \\
 & & \\
 & = & (\mathbf{at}\ (\mathbf{c})_{\mathbf{Succ}\,(\mathbf{AB}\ \sqcup\ \mathbf{at}\ (\mathbf{c}))} \\
 & & \ \&\ (\mathbf{g_1\ c})) \\
 & \sqcup & \\
 & & ((\mathbf{at}\ (\mathbf{c}_{\mathbf{0}}))\ \&\ \mathbf{AB}) \\
 & & \\
 & = & (\mathbf{at}\ (\mathbf{c})_{\mathbf{Succ}\,(\mathbf{AB}\ \sqcup\ \mathbf{at}\ (\mathbf{c}))} \\
 & & \ \&\ (\mathbf{g_1\ c})) \\
 & \sqcup & \\
 & & (\mathbf{at}\ (\mathbf{c}_{\mathbf{0}}))
\end{array}
$$

The above is due to the following results:

$$
\begin{array}{rcl}
\mathbf{greater_1}\ (\mathbf{c}_{\mathbf{True}}) & = & \mathbf{c}_{\mathbf{Succ(AB}\ \sqcup\ \mathbf{c)}} \\
\mathbf{greater_1}\ (\mathbf{c}_{\mathbf{False}}) & = & (\mathbf{c}_{\mathbf{0}}) \\
-_1\ (\mathbf{g_1\ c}) & = & (\mathbf{g_1\ c}) \\
+_2\ (\mathbf{g_2\ c}) & = & (\mathbf{g_2\ c}) \\
\mathbf{g_2\ c} & = & \mathbf{AB}
\end{array}
$$

The result for $\mathbf{greater_1}$ follows from that of $\mathbf{lesseq_1}$. The result for $\mathbf{g_2\ c}$ is explained on page 181. Note also that

$$\mathbf{AB} = \{0\}_{\mathbf{CR}}$$

Fixpoint iteration to find the first context function

If we let $F = \mathbf{g_1}(\{1\}_{\mathbf{0},\ \mathbf{Succ}(\{1\})})$ then we can determine F by a fixpoint iteration, using the above equation to define successive approximations.

The results of iteration to find the least fixpoint are summarised in the table below:

F^0	**CR**
F^1	$\{1\}_{\mathbf{0}}$
F^2	$\{1\}_{\mathbf{0},\ \mathbf{Succ}(\{0,1\})}$
F^3	$\{1, M\}_{\mathbf{0},\ \mathbf{Succ}(\{1,M\})}$
F^4	$\{1, M\}_{\mathbf{0},\ \mathbf{Succ}(\{1,M\})}$
F	$\{1, M\}_{\mathbf{0},\ \mathbf{Succ}(\{1,M\})}$

This shows that the function is strict in its first argument and that the first argument may be shared. In addition, this shows that it has to be fully evaluated.

Details of the fixpoint iteration

The details of the fixpoint iteration to find the first context function are quite interesting and are given below:

$$\begin{aligned}
F^1 &= \quad (\{1\}_{\mathbf{Succ}(\{0,1\})}) \\
&\sqcup \\
&\quad \{1\}_{\mathbf{0}} \\
&= \{1\}_{\mathbf{0}}
\end{aligned}$$

$$\begin{aligned}
F^2 &= \quad ((\{1\}_{\mathbf{Succ}(\{0,1\})}) \\
&\quad \& \ \{1\}_{\mathbf{0}}) \\
&\sqcup \\
&\quad \{1\}_{\mathbf{0}} \\
&= \quad \{1\}_{\mathbf{0},\ \mathbf{Succ}(\{0,1\})} \\
&\sqcup \\
&\quad \{1\}_{\mathbf{0}} \\
&= \{1\}_{\mathbf{0},\ \mathbf{Succ}(\{0,1\})}
\end{aligned}$$

$$
\begin{aligned}
F^3 &= \quad ((\{1\}_{\mathbf{Succ}(\{0,1\})}) \\
&\qquad \& \ (\{1\}_{\mathbf{0},\,\mathbf{Succ}(\{0,1\})})) \\
&= \{M\}_{\mathbf{0},\,\mathbf{Succ}(\{1,M\})} \sqcup \{1\}_{\mathbf{0}} \\
&= \{1, M\}_{\mathbf{0},\,\mathbf{Succ}(\{1,M\})}
\end{aligned}
$$

$$
\begin{aligned}
F^4 &= \quad ((\{1\}_{\mathbf{Succ}(\{0,1\})}) \\
&\qquad \& \ \{1, M\}_{\mathbf{0},\,\mathbf{Succ}(\{1,M\})}) \\
&\quad \sqcup \ \{1\}_{\mathbf{0}} \\
&= \{M\}_{\mathbf{0},\,\mathbf{Succ}(\{1,M\})} \sqcup \{1\}_{\mathbf{0}} \\
&= \{1, M\}_{\mathbf{0},\,\mathbf{Succ}(\{1,M\})}
\end{aligned}
$$

N.B. In order to keep the notation relatively concise the contexts belonging to the **0** and **Succ** parts have been merged. This abuse of notation should be treated with caution, however, since clearly the first argument is not shared if it evaluates to 0.

Analysis of the second argument

The analysis of the second argument is as follows:

$$
\begin{aligned}
\mathbf{g_2\ c} &= \mathbf{AB} \\
&\quad \& \\
&\quad (\mathbf{AB} \sqcup (\mathbf{c} \xrightarrow{g\ (a-1)\ (b+a)} b)) \\
&= \mathbf{AB} \sqcup (\mathbf{c} \xrightarrow{g\ (a-1)\ (b+a)} b) \\
&= \mathbf{AB} \sqcup ((\mathbf{g_2\ c}) \xrightarrow{(b+a)} b) \\
&= \mathbf{AB} \sqcup (+_1(\mathbf{g_2\ c})) \\
&= \mathbf{AB} \sqcup \mathbf{g_2\ c}
\end{aligned}
$$

It may readily be seen that the fixpoint of this is **AB** for all **c**. This means that the second argument is never used.

A.3 Analysis of list functions

A.3.1 The *tail* of list function

Definition of the function

$$
\begin{array}{rcl}
tail & : & (\exists l : [A]).(nonempty\, l) \Rightarrow [A] \\
 & & \\
tail\,([], r) & \equiv_{df} & abort_{[A]}\; r \\
tail\,((a :: x), r) & \equiv_{df} & x
\end{array}
$$

Backwards Analysis of *tail's* argument

$$\mathbf{tail_1(c)} = \mathbf{atst(c)}_{((\mathbf{mkpair_1}(\mathbf{tail_1}(\mathbf{c}))),(\mathbf{mkpair_2}(\mathbf{tail_1}(\mathbf{c}))))}$$

$$
\begin{array}{rcl}
\mathbf{mkpair_1(tail_1(c))} & = & \mathbf{atst(c)}_{[],\, ((\mathbf{at(c)} \xrightarrow{x} a) :: (\mathbf{c} \xrightarrow{x} x))} \\
 & = & \mathbf{atst(c)}_{[],\, (\mathbf{AB} :: \mathbf{c})}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{mkpair_2(tail_1(c))} & = & (\mathbf{c} \xrightarrow{abort_{[A]}\; r} r) \sqcup (\mathbf{c} \xrightarrow{x} r) \\
 & = & \mathbf{AB} \sqcup \mathbf{AB} \\
 & = & \mathbf{AB}
\end{array}
$$

Hence,

$$\mathbf{tail_1(c)} = \mathbf{atst(c)}_{((\mathbf{atst(c)}_{[],\, (\mathbf{AB} :: \mathbf{c})}),\mathbf{AB})}$$

Since $\mathbf{c}$ is an arbitrary list context (and is therefore structured), we approximate the above as follows:

$$\mathbf{atst(c)}_{((\mathbf{atst(c)}_{[],\, (\mathbf{AB} \,\sqcup\, (\mathbf{mkpair_1}(\mathbf{c})) :: (\mathbf{at}(\mathbf{c}) \,\sqcup (\mathbf{mkpair_2}(\mathbf{c}))))}),\mathbf{AB})}$$

A.3.2 The *head* of list function

Definition of the *head* function

$$
\begin{array}{rcl}
head & : & (\exists l : [A]).(nonempty\, l) \Rightarrow [A] \\
 & & \\
head\,([], r) & \equiv_{df} & abort_{[A]}\; r \\
head\,((a :: x), r) & \equiv_{df} & a
\end{array}
$$

Analysis of the context propagated to the first argument of *head*

$$\mathbf{head_1(c)} = \mathbf{atst(c)}_{((\mathbf{mkpair_1(head_1(c))}),(\mathbf{mkpair_2(tail_1(c))}))}$$

$$\begin{aligned}
\mathbf{mkpair_1(head_1(c))} &= \mathbf{atst(c)}_{[],\ ((\mathbf{at(c)} \xrightarrow{a} a) :: (\mathbf{c} \xrightarrow{a} x))} \\
&= \mathbf{atst(c)}_{[],\ (\mathbf{at(c)} :: \mathbf{AB})}
\end{aligned}$$

$$\begin{aligned}
\mathbf{mkpair_2(head_1(c))} &= (\mathbf{c} \xrightarrow{abort_{[A]}\ r} r) \sqcup (\mathbf{c} \xrightarrow{x} r) \\
&= \mathbf{AB} \sqcup \mathbf{AB} \\
&= \mathbf{AB}
\end{aligned}$$

Hence,

$$\mathbf{head_1(c)} = \mathbf{atst(c)}_{((\mathbf{atst(c)}_{[],\ (\mathbf{at(c)} :: \mathbf{AB})}),\mathbf{AB})}$$

A.3.3 The *length* function over lists

Definition of the *length* (#) function

$$\# \quad : \quad [A] \Rightarrow N$$

$$\begin{aligned}
\#[] &\equiv_{df} 0 \\
\#(a :: x) &\equiv_{df} \#x + 1
\end{aligned}$$

Sharing analysis of the length of list function (#)

$$\begin{aligned}
\#_1\mathbf{c} &= \mathbf{atst(c)}_{[],\ (\mathbf{at(c)} \xrightarrow{(\#x)+1} a) :: (\mathbf{at(c)} \xrightarrow{(\#x)+1} x)} \\
&= \mathbf{atst(c)}_{[],\ \mathbf{AB} :: (\#_1\mathbf{c})}
\end{aligned}$$

We thus perform a fixpoint iteration:

$$\begin{aligned}
(\#_1\mathbf{c})^0 &= \mathbf{CR} \\
(\#_1\mathbf{c})^1 &= \mathbf{atst(c)}_{[],\ \mathbf{AB} :: \mathbf{CR}} \\
(\#_1\mathbf{c})^2 &= \mathbf{atst(c)}_{[],\ \mathbf{AB} :: (\mathbf{atst(c)}_{[],\ \mathbf{AB} :: \mathbf{CR}})} \\
&\precsim \mathbf{atst(c)}_{[],\ \mathbf{AB} :: \mathbf{atst(c)}}
\end{aligned}$$

This is the fixpoint.
Hence we deduce that the tail of the list is used "as much" as the whole list.

A.3.4 The *append* (++) function

The following is a backwards analysis upon the list append (++) function in type theory.

Definition of the function

$$
\begin{array}{rcl}
+\!\!+ & : & [A] \Rightarrow [A] \Rightarrow [A] \\
\\
[]+\!\!+y & \equiv_{df} & y \\
(a :: x)+\!\!+y & \equiv_{df} & a :: (x+\!\!+y)
\end{array}
$$

Analysis of the first argument

$$+\!\!+_{\mathbf{1}}\mathbf{c} = \mathbf{atst}(\mathbf{c})_{(::_1(\mathbf{c}))\, ::\, +\!\!+_{\mathbf{1}}(::_2(\mathbf{c}))}$$

$$+\!\!+_{\mathbf{1}}(::_2(\mathbf{c})) = \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, +\!\!+_{\mathbf{1}}(::_2(\mathbf{c}))}$$

As, by our notion of repeating contexts in data structures,

$$
\begin{array}{rcl}
::_1(::_2(\mathbf{c})) & = & ::_1(\mathbf{c}) \\
::_2(::_2(\mathbf{c})) & = & ::_2(\mathbf{c})
\end{array}
$$

A solution to this is achieved by performing the following fixpoint iteration:

$$
\begin{array}{rcl}
(+\!\!+_{\mathbf{1}}(::_2(\mathbf{c})))^0 & = & \mathbf{CR} \\
(+\!\!+_{\mathbf{1}}(::_2(\mathbf{c})))^1 & = & \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, \mathbf{CR}} \\
(+\!\!+_{\mathbf{1}}(::_2(\mathbf{c})))^2 & = & \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, (\mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, \mathbf{CR}})} \\
 & \precsim & \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, \mathbf{atst}(::_2(\mathbf{c}))} \\
(+\!\!+_{\mathbf{1}}(::_2(\mathbf{c})))^3 & = & \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, \mathbf{atst}(::_2(\mathbf{c}))}
\end{array}
$$

So, we have,

$$
\begin{array}{rcl}
+\!\!+_{\mathbf{1}}\mathbf{c} & = & \mathbf{atst}(\mathbf{c})_{(::_1(\mathbf{c}))\, ::\, \mathbf{atst}(::_2(\mathbf{c}))_{(::_1(\mathbf{c}))\, ::\, \mathbf{atst}(::_2(\mathbf{c}))}} \\
 & = & \mathbf{atst}(\mathbf{c})_{(::_1(\mathbf{c}))\, ::\, \mathbf{atst}(::_2(\mathbf{c}))} \\
 & = & \mathbf{c}
\end{array}
$$

Analysis of the second argument

$$\begin{aligned}
+\!\!+_2\mathbf{c} &= \mathbf{c} \xrightarrow{y} y \sqcup \mathbf{c} \xrightarrow{a::(x+\!\!+y)} y \\
&= \mathbf{c} \sqcup (::_2(\mathbf{c}) \xrightarrow{x+\!\!+y} y) \\
&= \mathbf{c} \sqcup +\!\!+_2(::_2(\mathbf{c}))
\end{aligned}$$

$$+\!\!+_2(::_2(\mathbf{c})) = ::_2(\mathbf{c}) \sqcup +\!\!+_2(::_2(\mathbf{c}))$$

It is easily shown (via iteration) that we therefore have:

$$+\!\!+_2(::_2(\mathbf{c})) = ::_2(\mathbf{c})$$

Hence,

$$+\!\!+_2\mathbf{c} = \mathbf{c} \sqcup ::_2(\mathbf{c})$$

(Note that only the atomic part of the second argument's context will be distinct from **c**.)

A.3.5 The *map* function

Definition of the function

$$map \quad : \quad (A \Rightarrow B) \Rightarrow [A] \Rightarrow [B]$$

$$\begin{aligned}
map\ f\ [] &\equiv_{df} [] \\
map\ f\ (a::x) &\equiv_{df} (f\ a)::(map\ f\ x)
\end{aligned}$$

Analysis of the first argument

$$\mathbf{map}_1\ \mathbf{c} = \mathbf{AB} \sqcup \mathbf{c} \xrightarrow{(f\ a)::(map\ f\ x)} f$$

$$\begin{aligned}
\mathbf{c} \xrightarrow{(f\ a)::(map\ f\ x)} f &= (\mathbf{c} \xrightarrow{f\ a} f) \,\&\, (\mathbf{c} \xrightarrow{map\ f\ x} f) \\
&= \mathbf{ap}_1\ \mathbf{c} \,\&\, \mathbf{map}_1\ \mathbf{c}
\end{aligned}$$

In the case of sharing analysis,

$$\mathbf{ap}_1\ \mathbf{c} = \mathbf{at}(\mathbf{c})$$

The above may be solved by performing an iteration to find the fixpoint.

$$\begin{aligned}
(\mathbf{map}_1\ \mathbf{c})^0 &= \mathbf{CR} \\
(\mathbf{map}_1\ \mathbf{c})^1 &= \mathbf{AB} \\
(\mathbf{map}_1\ \mathbf{c})^2 &= \mathbf{AB} \sqcup \mathbf{at}(\mathbf{c})
\end{aligned}$$

$$
\begin{aligned}
(\mathbf{map}_1\,\mathbf{c})^3 &= \mathbf{AB} \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, (\mathbf{AB} \sqcup \mathbf{at}(\mathbf{c}))) \\
&= \mathbf{AB} \sqcup (\mathbf{at}(\mathbf{c}) \sqcup (\mathbf{at}(\mathbf{c}) \sqcup \mathbf{at}(\mathbf{c}))) \\
(\mathbf{map}_1\,\mathbf{c})^4 &= \mathbf{AB} \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, (\mathbf{AB} \sqcup (\mathbf{at}(\mathbf{c}) \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, \mathbf{at}(\mathbf{c}))))) \\
&= \mathbf{AB} \\
&\quad \sqcup \\
&\quad (\mathbf{at}(\mathbf{c}) \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, (\mathbf{at}(\mathbf{c}) \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, \mathbf{at}(\mathbf{c}))))) \\
&= \mathbf{AB} \sqcup (\mathbf{at}(\mathbf{c}) \sqcup (\mathbf{at}(\mathbf{c}) \,\&\, \mathbf{at}(\mathbf{c})))
\end{aligned}
$$

(Note that the last two transformations only apply to sharing analysis.)
Thus we have found the fixpoint.

Analysis of the second argument

$$
\begin{aligned}
\mathbf{map_2}\,[f]\,\mathbf{c} &= \mathbf{at}(\mathbf{c})_{\mathbf{0}} \sqcup \mathbf{at}(\mathbf{c})_{(\mathbf{c} \xrightarrow{f\,a} a) :: (\mathbf{c} \xrightarrow{f\,a} x)} \\
\mathbf{c} \xrightarrow{f\,a} a &= \mathbf{map_2}\,[f]\,\mathbf{c}
\end{aligned}
$$

(Note that f may be a partial application e.g. $lesseq\, a$, which will affect subscripting.)

$$
\mathbf{c} \xrightarrow{map\ f\ x} x = \mathbf{map_2}\,[f]\,\mathbf{c}
$$

So,

$$
\mathbf{map_2}\,[f]\,\mathbf{c} = \mathbf{at}(\mathbf{c})_{\mathbf{0} \sqcup (\mathbf{map_2}\,[f]\,\mathbf{c}) :: (\mathbf{map_2}\,[f]\,\mathbf{c})}
$$

A solution to this may be found through an iteration.

$$
\begin{aligned}
(\mathbf{map_2}\,[f]\,\mathbf{c})^0 &= \mathbf{CR} \\
(\mathbf{map_2}\,[f]\,\mathbf{c})^1 &= \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{CR}} \\
(\mathbf{map_2}\,[f]\,\mathbf{c})^2 &= \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: (\mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{CR}})} \\
&\underset{\sim}{\sqsubset} \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{at}(\mathbf{c})} \\
(\mathbf{map_2}\,[f]\,\mathbf{c})^3 &= \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{at}(\mathbf{c})}} \\
&\underset{\sim}{\sqsubset} \mathbf{at}(\mathbf{c})_{(\mathbf{map_2}\,[f]\,\mathbf{c}) :: \mathbf{at}(\mathbf{c})}
\end{aligned}
$$

The above is the fixpoint.

Appendix B

Further documentation on Ferdinand

B.1 Source code

B.1.1 The top-level module `main`

```
%nolist

||==============================================================||
||                                                              ||
|| main.m -- The main script for analysed translations.         ||
||                                                              ||
|| Author:              Alastair J. Telford                     ||
|| Place:               University of Kent at Canterbury        ||
|| Last modified:       18th February 1995                      ||
||                                                              ||
|| Description:         Coordinates all the analysis and        ||
||                      translation phases.                     ||
||                                                              ||
||==============================================================||

||==============================================================||

||--- Exports and Imports---||

%export
    analysed_translation
    ||
    analysis_type
    ||
    fnDefn_Env
    environment
    empty_env
    add_env
    ||
    new_flictrans
    ||
```

```
    "../Flic_Syntax/Flic"

%include
    "../typecheck_type"
%include
    "../Translate/new_translate"
%include
    "../Eager_translate/eager_translate"
%include
    "../Nd_translate/nd_new_translate"
%include
    "../St_translate/st_new_translate"
%include
    "../Stab_translate/stab_translate"
%include
    "../Gen_translate/gen_translate"
%include
    "../Context_expressions/context_expr.m"
%include
    "../Structured/context_calc"
%include
    "../FnParam_Env/fnParam_Env"
%include
    "../Environment/environment"
%include
    "../Index_table/index_table"
%include
    "../Values/values"
%include
    "../Flic_Syntax/Flic"

||==============================================================||

|| fn_name is the type of function names.

fn_name
==
    [char]

|| fnDefn_Env is the type of the function definition environment.

fnDefn_Env
==
    environment tc_Elem


||==============================================================||

|| Functions

||==============================================================||
||                                                              ||
|| analysed_translation analyses the top-level expression and   ||
|| the combinators and translates them into their appropriate   ||
```

```
|| FLIC representations.                                        ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The type of the analysis.                           ||
||      (2) The top level expression of the program and the set ||
||      of function definitions.                                ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

analysed_translation ::
    analysis_type ->                                            || (1)
        (tc_Elem, fnDefn_Env) ->                                || (2)
            (flic_simple_part, flic_program)

analysed_translation antype (top_expr,fn_names_defs)
=
    (flic_simple_tl_Expr, flic_simple_bindings)
    , if is_simp_trans antype
=
    (flic_antrans_tl_expr, flic_antrans_bindings)
    , otherwise
    where
        ||
        || flic_simple_tl_Expr is the FLIC translation of the top-level
        || expression. The translation does not use any abstract
        || interpretation information.
        || flic_simple_tl_Expr :: flic_simple_part
        flic_simple_tl_Expr
        =
            simp_translator antype top_expr
        ||
        || flic_simple_bindings is the FLIC translation of the function
        || definitions. The translation does not use any abstract
        || interpretation information.
        || flic_simple_bindings :: flic_program
        flic_simple_bindings
        =
            all_simp_translate antype fn_names_defs fn_names
        ||
        || flic_antrans_tl_expr is the FLIC translation of the top-level
        || expression using abstract interpretation information.
        || (N.B. The top-level expression has an empty set of parameters
        || and contexts.)
        || flic_antrans_tl_expr :: flic_simple_part
        flic_antrans_tl_expr
        =
            func_anno_trans
                antype empty_idx_table context_environ top_expr
        ||
        || flic_antrans_bindings is the FLIC translation of the function
        || definitions using abstract interpretation information.
        || flic_antrans_bindings :: flic_program
        flic_antrans_bindings
```

```
        =
            all_anno_translate
                antype context_environ fn_names_defs fn_names
        ||
        || context_environ is the environment of contexts found by
        || evaluating the context expressions.
        || context_environ :: context_Env
        context_environ
        =
            all_Context_Expr_Val antype cxt_Expr_Env fn_names
        ||
        || cxt_Expr_Env is the context expression environment produced
        || from the function definitions.
        || context_Expr_Env :: context_Expr_Env
        cxt_Expr_Env
        =
            all_Context_Prop_Expr fn_names_defs fn_names
        ||
        || fn_names is the list of function names.
        || fn_names :: [[char]]
        fn_names
        =
            env_keys fn_names_defs


||================================================================||
||                                                                ||
|| all_simp_translate is the translation to FLIC code of a set    ||
|| of function definitions without using any abstract             ||
|| interpretation information.                                    ||
||                                                                ||
|| Parameters:                                                    ||
||      (1) The type of the analysis being performed.             ||
||      (2) The environment of function definitions.              ||
||      (3) The list of function names.                           ||
|| (See the type declaration for a key to the above.)             ||
||                                                                ||
||================================================================||

all_simp_translate ::
    analysis_type ->                                               || (1)
        fnDefn_Env ->                                              || (2)
            [fn_name] ->                                           || (3)
                flic_program

all_simp_translate antype fnDefs fns
=
    map (simp_bind_trans antype fnDefs) fns


||================================================================||
||                                                                ||
|| simp_bind_trans produces a FLIC binding for a function: the    ||
|| translation will not use any abstract interpretation           ||
```

```
|| information.                                                  ||
||                                                               ||
|| Parameters:                                                   ||
||      (1) The type of the analysis.                            ||
||      (2) The function definition environment.                 ||
||      (3) The function name.                                   ||
|| (See the type declaration for a key to the above.)            ||
||                                                               ||
||===============================================================||

simp_bind_trans ::
    analysis_type ->                                             || (1)
        fnDefn_Env ->                                            || (2)
            fn_name ->                                           || (3)
                flic_binding

simp_bind_trans antype fnDefs fn
=
    (fn, simp_translator antype fnDef)
    where
        || fnDef is the function definition of the function named fn.
        || fnDef :: tc_Elem
        fnDef
        =
            fnDefn "simp_bind_trans:" fnDefs fn


||===============================================================||
||                                                               ||
|| simp_translator performs translation for a specific function ||
|| but with no use of abstract interpretation information made. ||
||                                                               ||
|| Parameters:                                                   ||
||      (1) The type of the analysis.                            ||
||      (2) The function definition.                             ||
|| (See the type declaration for a key to the above.)            ||
||                                                               ||
||===============================================================||

simp_translator ::
    analysis_type ->                                             || (1)
        tc_Elem ->                                               || (2)
            flic_simple_part

simp_translator antype fnDef
=
    new_flictrans fnDef
    , if antype = NONE
=
    eager_flictrans fnDef
    , if antype = Eager
=
    error ("\n\t" ++
            "Unknown or unsuitable analysis for simple translation.")
```

```
    , otherwise


||==============================================================||
||                                                              ||
|| all_anno_translate is the translation to FLIC code of a set  ||
|| of function definitions and using corresponding abstract     ||
|| information.                                                 ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The type of the analysis being performed.           ||
||      (2) The environment of context information.             ||
||      (3) The environment of function definitions.            ||
||      (4) The list of function names.                         ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

all_anno_translate ::
    analysis_type ->                                            || (1)
        context_Env ->                                          || (2)
            fnDefn_Env ->                                       || (3)
                [fn_name] ->                                    || (4)
                    flic_program

all_anno_translate antype cEnv fnDefs fns
=
    map (anno_translator antype cEnv fnDefs) fns


||==============================================================||
||                                                              ||
|| anno_translator performs translation for a specific function.||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The type of the analysis.                           ||
||      (2) The context environment.                            ||
||      (3) The function definition environment.                ||
||      (4) The function name.                                  ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

anno_translator ::
    analysis_type ->                                            || (1)
        context_Env ->                                          || (2)
            fnDefn_Env ->                                       || (3)
                fn_name ->                                      || (4)
                    flic_binding

anno_translator antype cEnv fnDefs fname
=
    (fname, trans_func)
||
```

```
    where
        || trans_func is the FLIC translation of the function.
        || trans_func :: flic_simple_part
        trans_func
        =
            func_anno_trans antype paramAnnos cEnv fnDef
        ||
        || paramAnnos is the set of formal parameters and their
        || contexts found from a lookup via the function name.
        || paramAnnos :: idx_table structured_context
        paramAnnos
        =
            value_elim cxt_lkup_err (param_idx_table cEnv fname)
        ||
        || fnDef is the function definition, found from a lookup via
        || the function name in the definition environment.
        || fnDef :: tc_Elem
        fnDef
        =
            fnDefn "anno_translator:" fnDefs fname
        ||
        || cxt_lkup_err is the main error message produced when an error
        || in the lookup of parameters and contexts occurs.
        || cxt_lkup_err :: [char]
        cxt_lkup_err
        =
            "anno_translator: Error in lookup of contexts."


||=============================================================||
||                                                             ||
|| func_anno_trans translates a function definition into its   ||
|| corresponding FLIC code, using the context information      ||
|| obtained for each parameter and, possibly context information||
|| from other functions.                                       ||
||                                                             ||
|| Parameters:                                                 ||
||      (1) The kind of analysis being performed.              ||
||      (2) An indexed table of formal parameters and their    ||
||      contexts.                                              ||
||      (3) The environment of contexts for all parameters of  ||
||      every function.                                        ||
||      (4) The function definition to be translated.          ||
|| (See the type declaration for a key to the above.)          ||
||                                                             ||
||=============================================================||

func_anno_trans ::
    analysis_type ->                                            || (1)
        idx_table structured_context ->                         || (2)
            context_Env ->                                      || (3)
                tc_Elem ->                                      || (4)
                    flic_simple_part
```

```
func_anno_trans antype paramAnnos cEnv fnDef
=
    nd_flictrans paramAnnos cEnv fnDef
    , if antype = Neededness
=
    st_flictrans paramAnnos fnDef
    , if antype = Strictness
=
    stab_flictrans paramAnnos cEnv fnDef
    , if antype = Strabsence
=
    error "func_anno_trans: Unknown analysis type"
    , otherwise


||===============================================================||
||                                                               ||
|| all_Context_Expr_Val is the solution of all context           ||
|| expressions so as to form an environment of structured        ||
|| contexts.                                                     ||
||                                                               ||
|| Parameters:                                                   ||
||      (1) The type of the analysis.                            ||
||      (2) The context expression environment.                  ||
||      (3) The list of function names.                          ||
|| (See the type declaration for a key to the above.)            ||
||                                                               ||
||===============================================================||

all_Context_Expr_Val ::
    analysis_type ->                                             || (1)
        context_Exp_Env ->                                       || (2)
            [fn_name] ->                                         || (3)
                context_Env

all_Context_Expr_Val antype cExp_Env fn_names
=
    foldr (cxt_evaluator antype cExp_Env) empty_Param_Env fn_names


||===============================================================||
||                                                               ||
|| cxt_evaluator adds to a given context environment by          ||
|| evaluating the context expressions associated with the        ||
|| parameters of a function.                                     ||
||                                                               ||
||                                                               ||
|| Parameters:                                                   ||
||      (1) The type of analysis being performed.                ||
||      (2) The context expression environment.                  ||
||      (3) The name of a function.                              ||
||      (4) The current context environment.                     ||
|| (See the type declaration for a key to the above.)            ||
||                                                               ||
```

```
||==============================================================||

cxt_evaluator ::
    analysis_type ->                                            || (1)
        context_Exp_Env ->                                      || (2)
            fn_name ->                                          || (3)
                context_Env ->                                  || (4)
                    context_Env

cxt_evaluator antype cExpEnv fnm cEnv
=
    add_Param_Env cEnv fnm params str_cxts
    where
        || params is the list of parameters for the function.
        || params :: [[char]]
        || cExps is the corresponding list of context expressions.
        || cExps :: [context_Expression]
        (params, cExps)
        =
            value_elim param_lkup_err params_exps
        ||
        || param_lkup_err is the main error message when an exception
        || occurs when looking up the function in the context expression
        || environment.
        || param_lkup_err :: [char]
        param_lkup_err
        =
            "cxt_evaluator: Error in parameters/expressions lookup."
        ||
        || params_exps is the (params, cExps) pair above, but tagged
        || to indicate whether it has been retrieved successfully.
        || params_exps :: value ([[char]],[context_Expression])
        params_exps
        =
            param_vrs_vls cExpEnv fnm
        ||
        || str_cxts is the list of structured contexts resulting
        || from the evaluation of the context expressions.
        || str_cxts :: [structured_Context]
        str_cxts
        =
           [
            context_Evaluation (i+1) (cExps!i) |
                i <- index cExps
           ]
        ||
        || context_Evaluation finds the context that is the result of
        || evaluating a context expression with a given initial context.
        || context_Evaluation ::
        ||      num ->  context_Expression -> structured_Context
        context_Evaluation ind
        =
            context_Expr_Evaluation
                antype initial_cxt cExpEnv fnm ind []
```

```
        ||
        || initial_cxt is the initial context supplied as input to the
        || evaluation of each context expression.
        || initial_cxt :: structured_Context
        initial_cxt
        =
            lattice_SCINIT antype


||==============================================================||
||                                                              ||
|| all_Context_Prop_Expr produces all the context expressions   ||
|| which correspond to a set of function definitions.           ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The environment of function definitions.            ||
||      (2) The list of function names.                         ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

all_Context_Prop_Expr ::
    fnDefn_Env ->                                               || (1)
        [fn_name] ->                                            || (2)
            context_Exp_Env


all_Context_Prop_Expr fn_defs fn_names
=
    foldr (cxt_expr_form fn_defs) initial_cxt_Exp_Env fn_names


||==============================================================||
||                                                              ||
|| cxt_expr_form formulates a set of context expressions, one   ||
|| for each parameter of a given function, and thus modifies    ||
|| the context expression environment.                          ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The function definition environment.                ||
||      (2) The given function.                                 ||
||      (3) The current context expression environment.         ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

cxt_expr_form ::
    fnDefn_Env ->                                               || (1)
        fn_name ->                                              || (2)
            context_Exp_Env ->                                  || (3)
                context_Exp_Env

cxt_expr_form fn_def_env fnm cExpEnv
=
```

```
    add_Param_Env cExpEnv fnm params cExps
    where
        || fndef is the function definition found.
        || fndef :: tc_Elem
        fndef
        =
            fnDefn "cxt_expr_form:" fn_def_env fnm
        ||
        || params is the list of parameters for the function.
        || params :: [[char]]
        || defn_exp is the defining expression for the function.
        || defn_exp :: tc_Elem
        (params, defn_exp)
        =
            func_def_split fndef
        ||
        || cExps finds the list of context expressions corresponding to
        || each parameter for a function.
        || cExps :: [context_Expression]
        cExps
        =
            map (expr_Formulation params fndef) params
        ||
        || expr_Formulation formulates a context expression for
        || a particular parameter of a function.
        || expr_Formulation ::
        ||      [[char]] ->
        ||          tc_Elem ->
        ||              [char] ->
        ||                  context_Expression
        expr_Formulation ps fdef p
        =
            context_Prop_Expr (mk_ord_par p) ps fdef


|| mk_ord_par converts a variable name to its prop_Var equivalent.

mk_ord_par v
=
    Ord v


||============================================================||
||                                                            ||
|| fnDefn is the function definition corresponding to a given ||
|| name. The definition is found from an environment lookup.  ||
||                                                            ||
|| Parameters:                                                ||
||      (1) A message to be used in case the lookup of the    ||
||      function definition fails.                            ||
||      (2) The function definition environment.              ||
||      (3) The function name.                                ||
|| (See the type declaration for a key to the above.)         ||
||                                                            ||
```

```
||=============================================================||

fnDefn ::
    [char] ->                                                   || (1)
        fnDefn_Env ->                                           || (2)
            [char] ->                                           || (3)
                tc_Elem

fnDefn message fnDefs fn
=
    value_elim lkup_errmess (env_lkup fnDefs fn)
    where
        || lkup_errmess is the error message produced if the lookup of
        || a function's definition goes wrong.
        || lkup_errmess :: [char]
        lkup_errmess
        =
            message ++ "\nFunction definition not found for " ++ fn
```

B.1.2 Neededness optimisation functions

```
||=============================================================||
||                                                             ||
|| ndd_flic_appl_expr is a FLIC simple part which represents a ||
|| function application. The translation to FLIC is done with  ||
|| respect to neededness analysis information.                 ||
||                                                             ||
|| Parameters:                                                 ||
||      (1) A translator for subexpressions.                   ||
||      (2) An environment of contexts.                        ||
||      (3) The tc_Elem expression to be translated.           ||
|| (See the type declaration for a key to the above.)          ||
||                                                             ||
||=============================================================||

ndd_flic_appl_expr ::
    (context_Env -> tc_Elem -> flic_simple_part) ->             || (1)
        context_Env ->                                          || (2)
            tc_Elem ->                                          || (3)
                flic_simple_part

ndd_flic_appl_expr fltrans cEnv appl
=
    flic_appl_expr fltrans' flic_fun orig_acts
    , if fn_is_formal
=
    ndd_opt_appl_trans fltrans' fn_par_cxts flic_fun fn_arity orig_acts
    , otherwise
||
    where
        || fn_is_formal indicates whether the leftmost and innermost
        || function being applied is a formal parameter.
```

```
|| fn_is_formal :: bool
fn_is_formal
=
    is_Assump inner_fun
||
|| inner_fun is the leftmost and innermost function in the
|| application.
|| inner_fun :: tc_Elem
inner_fun
=
    hd applic_list
||
|| orig_acts is the original list of actual parameters in the
|| application.
|| orig_acts :: [tc_Elem]
orig_acts
=
    tl applic_list
||
|| applic_list is the list of all components of the application.
|| applic_list :: [tc_Elem]
applic_list
=
    appl_list appl
||
|| fltrans' is the actual translation function formed using
|| the context environment.
fltrans'
=
    fltrans cEnv
||
|| flic_fun is the FLIC translation of inner_fun.
|| flic_fun :: flic_simple_part
flic_fun
=
    Fl_Name fn_name
||
|| fn_name is the name of the function being applied.
|| fn_name :: [char]
fn_name
=
    tc_Elem_name inner_fun
||
|| fn_par_cxts is the table of parameters and their contexts
|| associated with the given function.
|| fn_par_cxts :: idx_table structured_context
fn_par_cxts
=
    value_elim lkup_errmess (param_idx_table cEnv fn_name)
||
|| lkup_errmess is the error message produced by an incorrect
|| lookup of the function name in the context environment.
|| lkup_errmess :: [char]
lkup_errmess
```

```
        =
            "ndd_flic_appl_expr: Error in lookup of contexts of " ++ fn_name
        ||
        || fn_arity is the arity (i.e. the number of parameters) of the
        || function.
        || fn_arity :: num
        fn_arity
        =
            value_elim arity_lkup_errmess (fn_size cEnv fn_name)
        ||
        || arity_lkup_errmess is the error message produced if a fault
        || occurs during the lookup of the function's arity.
        || arity_lkup_errmess :: [char]
        arity_lkup_errmess
        =
            "ndd_flic_appl_expr: Error in lookup of arity of " ++ fn_name


||==============================================================||
||                                                              ||
|| ndd_opt_appl_trans is the optimised translation of an        ||
|| application to FLIC. The leftmost and innermost function is  ||
|| the one to be applied to the applicands and it is the        ||
|| characteristics of this function which determine how the     ||
|| application may be optimised.                                ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) A translator from tc_Elem objects to FLIC for       ||
||      the sub-expressions.                                    ||
||      (2) An index table of contexts for the parameters of the||
||      functions being applied.                                ||
||      (3) The function being applied in FLIC form.            ||
||      (4) The size (arity) of the function being applied.     ||
||      (5) The applicands of the application, in tc_Elem form. ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

ndd_opt_appl_trans ::
    (tc_Elem -> flic_simple_part) ->                            || (1)
        idx_table structured_context ->                         || (2)
            flic_simple_part ->                                 || (3)
                num ->                                          || (4)
                    [tc_Elem] ->                                || (5)
                        flic_simple_part

ndd_opt_appl_trans fltrans par_cxts flic_fun fsize act_list
=
    dummy_abs_make dummy_appl_expr dummy_no
    , if dummies_reqd
=
    ord_appl_expr
    , otherwise
||
```

```
where
    || dummies_reqd denotes whether any dummy variables have to be
    || added to the application.
    || dummies_reqd :: bool
    dummies_reqd
    =
        #ndd_unapp_nos < dummy_no
    ||
    || ndd_unapp_nos is a list of the indices of the parameters of
    || the function which are needed but which do not have any
    || actual counterparts i.e. they are unapplied.
    || ndd_unapp_nos :: [num]
    ndd_unapp_nos
    =
        filter (>= num_acts) ndd_indices
    ||
    || num_acts is the number of actual parameters.
    || num_acts :: num
    num_acts
    =
        #act_list
    ||
    || ndd_indices is a list of all of the parameter indices of the function
    || which are needed.
    || ndd_indices :: [num]
    ndd_indices
    =
        needed_par_nos par_cxts fsize
    ||
    || dummy_no is the number of parameters of the function which
    || are "unapplied".
    || dummy_no :: num
    dummy_no
    =
        fsize - num_acts
    ||
    || dummy_appl_expr is a FLIC application expression which includes
    || dummy variables.
    || dummy_appl_expr :: flic_simple_expr
    dummy_appl_expr
    =
        flic_appl_expr fltrans flic_fun (ndd_acts ++ dummy_acts)
    ||
    || ndd_acts is a list of all the actual parameters which are
    || needed by the function.
    || ndd_acts :: [tc_Elem]
    ndd_acts
    =
        select ndd_indices act_list
    ||
    || dummy_acts is a list of dummy actual parameters which stand
    || for needed parameters of the function which are not included  in
    || the application.
    dummy_acts
```

```
        =
            dummy_var_list ndd_unapp_nos
        ||
        || ord_appl_expr is a FLIC application expression which does not
        || include dummy variables.
        || ord_appl_expr :: flic_simple_expr
        ord_appl_expr
        =
            flic_appl_expr fltrans flic_fun (ndd_acts ++ extra_acts)
        ||
        || extra_acts is a list of the actual parameters which are
        || superfluous to the function being applied as the function's
        || arity is smaller than the number of applicands.
        || extra_acts :: [tc_Elem]
        extra_acts
        =
            drop fsize act_list


||==============================================================||
||                                                              ||
|| ndd_flic_name_expr is the translation of a function name to  ||
|| FLIC with regard to the neededness of the function's         ||
|| parameters. We are assuming that the function name occurs as ||
|| a "null" application.                                        ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The environment of context information.             ||
||      (2) The name of the function.                           ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

ndd_flic_name_expr ::
    context_Env ->                                              || (1)
        [char] ->                                               || (2)
            flic_simple_part

ndd_flic_name_expr cEnv nm
=
    dummy_abs_make dummy_appl_expr fn_arity
    , if dummies_reqd
=
    name_trans
    , otherwise
||
    where
        || dummies_reqd denotes whether any dummy variables have to be
        || added to the application.
        || dummies_reqd :: bool
        dummies_reqd
        =
            ndd_no < fn_arity
        ||
```

```
|| ndd_no is the number of needed parameters of the function.
ndd_no
=
    #ndd_indices
||
|| ndd_indices is a list of all of the parameter indices of the function
|| which are needed.
|| ndd_indices :: [num]
ndd_indices
=
    needed_par_nos fn_par_cxts fn_arity
||
|| dummy_appl_expr is a FLIC application expression which includes
|| dummy variables.
|| dummy_appl_expr :: flic_simple_expr
dummy_appl_expr
=
    Expr (appls (Simple name_trans) dummy_acts)
||
|| dummy_acts is a list of dummy actual parameters which stand
|| for needed parameters of the function.
|| dummy_acts :: [flic_simple_part]
dummy_acts
=
    dummy_simples ndd_indices
||
|| name_trans is the FLIC translation of the given function name.
|| name_trans :: flic_simple_expr
name_trans
=
    Fl_Name nm
||
|| fn_par_cxts is the table of parameters and their contexts
|| associated with the given function.
|| fn_par_cxts :: idx_table structured_context
fn_par_cxts
=
    value_elim lkup_errmess (param_idx_table cEnv nm)
||
|| lkup_errmess is the error message produced by an incorrect
|| lookup of the function name in the context environment.
|| lkup_errmess :: [char]
lkup_errmess
=
    "ndd_flic_name_expr: Error in lookup of contexts of " ++ nm
||
|| fn_arity is the arity (i.e. the number of parameters) of the
|| function.
|| fn_arity :: num
fn_arity
=
    value_elim arity_lkup_errmess (fn_size cEnv nm)
||
|| arity_lkup_errmess is the error message produced if a fault
```

```
        || occurs during the lookup of the function's arity.
        || arity_lkup_errmess :: [char]
        arity_lkup_errmess
        =
            "ndd_flic_name_expr: Error in lookup of arity of " ++ nm


||==============================================================||
||                                                              ||
|| needed_par_nos is the list of parameter numbers which are    ||
|| needed for a function. The "neededness" of each parameter    ||
|| is found from an index table of structured contexts.         ||
|| There are size parameters to be examined in all.             ||
|| N.B. The indices produced are numbered from 0.               ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The index table of structured contexts for the      ||
||      function.                                               ||
||      (2) The size of the function.                           ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

needed_par_nos ::
    idx_table structured_context ->                             || (1)
        num ->                                                  || (2)
            [num]

needed_par_nos par_cxts size
=
    [(i-1) |
        i <- [1..size];
        ilkup_pred is_Ndd par_cxts i
    ]


||==============================================================||
||                                                              ||
|| dummy_abs_make adds "dummy" abstractions to the front of a   ||
|| FLIC defining expression.                                    ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The FLIC defining expression.                       ||
||      (2) The number of dummy abstractions to be added.       ||
||      (3) The list of indices of which variables are needed.  ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

dummy_abs_make ::
    flic_simple_part ->                                         || (1)
        num ->                                                  || (2)
            flic_simple_part
```

```
dummy_abs_make flExp n
=
    foldr abst_form flExp (dummy_names n)


||=============================================================||
||                                                             ||
|| dummy_names is a list of a given number of dummy variable   ||
|| names.                                                      ||
||                                                             ||
|| Parameters:                                                 ||
||      (1) The number of dummy names to be generated.         ||
|| (See the type declaration for a key to the above.)          ||
||                                                             ||
||=============================================================||

dummy_names ::
    num ->                                                     || (1)
        [param]

dummy_names n
=
    map dummy_var_name [1..n]


||=============================================================||
||                                                             ||
|| dummy_var_list is a list of tc_Elem dummy variables derived ||
|| from a list of numbers. (c.f. dummy_simples)                ||
||                                                             ||
|| Parameters:                                                 ||
||      [(1)] The list of numbers from whence the list of dummy ||
||      variables is derived.                                  ||
|| (See the type declaration for a key to the above.)          ||
||                                                             ||
||=============================================================||

dummy_var_list ::
    [num] ->                                                   || [(1)]
        [tc_Elem]

dummy_var_list
=
    map (tc_var_expr . dummy_var_name . (+1))


||=============================================================||
||                                                             ||
|| dummy_var_name is a dummy parameter name derived from a given||
|| natural number.                                             ||
||                                                             ||
|| Parameters:                                                 ||
||      (1) The number being used to derive a name.            ||
|| (See the type declaration for a key to the above.)          ||
```

```
||                                                              ||
||==============================================================||

dummy_var_name ::
    num ->                                                      || (1)
        param

dummy_var_name n
=
    "dummy_" ++ (shownum n)


||==============================================================||
||                                                              ||
|| dummy_simples is a list of FLIC simple parts corresponding to||
|| a list of dummy names. The number of dummy names results from||
|| a list of natural numbers. (c.f. dummy_var_list)             ||
||                                                              ||
|| Parameters:                                                  ||
||      [(1)] The given natural number.                         ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

dummy_simples ::
    [num] ->
        [flic_simple_part]

dummy_simples
=
    map (dummy_simple_part . (+1))


||==============================================================||
||                                                              ||
|| dummy_simple_part is a FLIC simple part corresponding to a   ||
|| dummy name. The dummy name is derived from a natural number. ||
||                                                              ||
|| Parameters:                                                  ||
||      (1) The given natural number.                           ||
|| (See the type declaration for a key to the above.)           ||
||                                                              ||
||==============================================================||

dummy_simple_part ::
    num ->                                                      || (1)
        flic_simple_part

dummy_simple_part n
=
    Fl_Name (dummy_var_name n)


||==============================================================||
```

```
||                                                             ||
|| abst_form is a FLIC abstraction formed from a given variable ||
|| name and a FLIC simple part (representing the defining       ||
|| expression).                                                 ||
||                                                             ||
|| Parameters:                                                 ||
||      (1) The variable name.                                  ||
||      (2) The FLIC simple part.                               ||
|| (See the type declaration for a key to the above.)          ||
||                                                             ||
||=============================================================||

abst_form ::
    param ->                                                   || (1)
        flic_simple_part ->                                    || (2)
            flic_simple_part

abst_form v def_exp
=
    Abs (Single_Abs v (Simple def_exp))
```

B.2 Results produced from indextest

B.2.1 Sample of the tc_Elem expressions

```
        Defining functions.
             Environment:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
==============================
Name: f13
Value:
Lambda "_Var25" Triv (Lrec (Assume "_Var25" (List (UAssume "a" (U 0))))
(Name "f4") (Name "f12"))
==============================

==============================
Name: f12
Value:
Lambda "_Var17" Triv (App (Name "f11") (Assume "_Var17" Triv))
==============================

==============================
Name: f11
Value:
Lambda "_Var17" Triv (Lambda "_Var18" Triv (App (Name "f10") (Assume
"_Var17" Triv)))
==============================

==============================
Name: f10
Value:
```

```
Lambda "_Var17" Triv (Lambda "%@hyp_Var18" Triv (App (App (Name "f9")
(Assume "_Var17" Triv)) (Assume "%@hyp_Var18" Triv)))
==============================
```

B.2.2 Examples of the context expressions formed

```
        Context expressions.
             Environment:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
==============================
Name: f13
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var25
Value:
CONTAND
First op.:
    CXT-APPL
    Applied fn:
        INITIAL
    Input expression:
        CXT-APPL
        Applied fn:
            CXT-FUNC Lrec 1
            Actuals:
            CXT-LAMBDA
            Input Param #: 1            Function index: 0
            Actuals:
            (No actuals)

            ABSENT
            ABSENT

        Input expression:
            INITIAL
Second op.:
    CONTAND
    First op.:
        CXT-APPL
        Applied fn:
            ABSENT
        Input expression:
            CXT-APPL
            Applied fn:
                CXT-FUNC Lrec 2
                Actuals:
                CXT-LAMBDA
                Input Param #: 1                Function index: 0
                Actuals:
                (No actuals)

                ABSENT
```

```
                ABSENT

            Input expression:
                INITIAL
    Second op.:
        CXT-APPL
        Applied fn:
            ABSENT
        Input expression:
            CXT-APPL
            Applied fn:
                CXT-FUNC Lrec 3
                Actuals:
                CXT-LAMBDA
                Input Param #: 1                Function index: 0
                Actuals:
                (No actuals)

                ABSENT
                ABSENT

            Input expression:
                INITIAL
----------
++++++++++++++++++++
==============================

==============================
Name: f12
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
CXT-APPL
Applied fn:
    INITIAL
Input expression:
    CXT-APPL
    Applied fn:
        CXT-FUNC f11 1
        Actuals:
        CXT-LAMBDA
        Input Param #: 1        Function index: 0
        Actuals:
        (No actuals)


    Input expression:
        INITIAL
----------
++++++++++++++++++++
==============================
```

```
==============================
Name: f11
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
CXT-APPL
Applied fn:
    INITIAL
Input expression:
    CXT-APPL
    Applied fn:
        CXT-FUNC f10 1
        Actuals:
        CXT-LAMBDA
        Input Param #: 1        Function index: 0
        Actuals:
        (No actuals)


    Input expression:
        INITIAL
----------
2.      Name: _Var18
Value:
CXT-APPL
Applied fn:
    ABSENT
Input expression:
    CXT-APPL
    Applied fn:
        CXT-FUNC f10 1
        Actuals:
        CXT-LAMBDA
        Input Param #: 1        Function index: 0
        Actuals:
        (No actuals)


    Input expression:
        INITIAL
----------
++++++++++++++++++++
==============================

==============================
Name: f10
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
CONTAND
```

```
First op.:
    CXT-APPL
    Applied fn:
        INITIAL
    Input expression:
        CXT-APPL
        Applied fn:
            CXT-FUNC f9 1
            Actuals:
            CXT-LAMBDA
            Input Param #: 1            Function index: 0
            Actuals:
            (No actuals)

            CXT-LAMBDA
            Input Param #: 2            Function index: 0
            Actuals:
            (No actuals)


        Input expression:
            INITIAL
Second op.:
    CXT-APPL
    Applied fn:
        ABSENT
    Input expression:
        CXT-APPL
        Applied fn:
            CXT-FUNC f9 2
            Actuals:
            CXT-LAMBDA
            Input Param #: 1            Function index: 0
            Actuals:
            (No actuals)

            CXT-LAMBDA
            Input Param #: 2            Function index: 0
            Actuals:
            (No actuals)


        Input expression:
            INITIAL
----------
2.      Name: %@hyp_Var18
Value:
CONTAND
First op.:
    CXT-APPL
    Applied fn:
        ABSENT
    Input expression:
        CXT-APPL
```

```
        Applied fn:
            CXT-FUNC f9 1
            Actuals:
            CXT-LAMBDA
            Input Param #: 1            Function index: 0
            Actuals:
            (No actuals)

            CXT-LAMBDA
            Input Param #: 2            Function index: 0
            Actuals:
            (No actuals)


        Input expression:
            INITIAL
Second op.:
    CXT-APPL
    Applied fn:
        INITIAL
    Input expression:
        CXT-APPL
        Applied fn:
            CXT-FUNC f9 2
            Actuals:
            CXT-LAMBDA
            Input Param #: 1            Function index: 0
            Actuals:
            (No actuals)

            CXT-LAMBDA
            Input Param #: 2            Function index: 0
            Actuals:
            (No actuals)


        Input expression:
            INITIAL
----------
++++++++++++++++++++
==============================
```

B.2.3 The resulting structured contexts

```
        Context environment.
             Environment:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
==============================
Name: f13
Value:
    Table:
++++++++++++++++++++
```

```
1.      Name: _Var25
Value:
(Strict,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f12
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
(Lazy,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f11
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
(Lazy,
    END)
----------
2.      Name: _Var18
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f10
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
(Lazy,
    END)
----------
2.      Name: %@hyp_Var18
Value:
(Lazy,
    END)
----------
```

```
+++++++++++++++++++
==============================

==============================
Name: f9
Value:
    Table:
+++++++++++++++++++
1.      Name: _Var17
Value:
(Lazy,
    END)
----------
2.      Name: %@hyp_Var18
Value:
(Lazy,
    END)
----------
3.      Name: _Var19
Value:
(Strict,
    END)
----------
+++++++++++++++++++
==============================

==============================
Name: f8
Value:
    Table:
+++++++++++++++++++
1.      Name: %@hyp_Var18
Value:
(Strict,
    END)
----------
2.      Name: _Var14
Value:
(Lazy,
    END)
----------
+++++++++++++++++++
==============================

==============================
Name: f7
Value:
    Table:
+++++++++++++++++++
1.      Name: %@hyp_Var18
Value:
(Strict,
    END)
----------
```

```
2.      Name: _Var14
Value:
(Lazy,
    END)
----------
3.      Name: %@hyp_Var14
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f6
Value:
    Table:
++++++++++++++++++++
1.      Name: %@hyp_Var18
Value:
(Strict,
    END)
----------
2.      Name: _Var14
Value:
(Lazy,
    END)
----------
3.      Name: _Var15
Value:
(Lazy,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f5
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var17
Value:
(Strict,
    INIT)
----------
2.      Name: _Var20
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================
```

```
==============================
Name: f4
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var26
Value:
(Strict,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f3
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var23
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f2
Value:
    Table:
++++++++++++++++++++
1.      Name: %@hyp_Var23
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f0
Value:
    Table:
++++++++++++++++++++
1.      Name: _Var27
Value:
(Absent,
    END)
----------
++++++++++++++++++++
==============================

==============================
Name: f14
```

```
Value:
EMPTY table
==============================


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

B.3 Test scripts

B.3.1 acker

The function below is an implementation of Ackermann's function.

```
ack :: nat -> nat -> nat
ack 0 n = (n + 1)
ack (m+1) 0 = 1
ack (m+1) (n+1) = ack m (ack (m + 1) n)

main :: nat
main = ack 12 12
```

B.3.2 polymap

Polymap is a definition of map with explicit polymorphism.

```
map :: (a;b :: Un 0) -> (a -> b) -> (List a) -> (List b)
map a b f [] = []
map a b f (x:y) = (f x):(map a b f y)

gen :: nat -> (List nat)
gen 0 = []
gen (n+1) = ((n+1):(gen n))

main :: List nat
main = map nat nat g (gen 10000)
       where
       g :: nat -> nat
       g z = z + 1
```

B.3.3 mergesort

```
msort :: (List nat) -> (List nat)
msort [] = []
msort [a] = [a]
msort (a:b:c) = merge [a] (msort (b:c))
                where
                merge :: (List nat) -> (List nat) -> (List nat)
                merge [] b = b
                merge a [] = a
                merge (a:x) (b:y) = a:(merge x (b:y)) , if ( a <= b )
                                  = b:(merge (a:x) y) , otherwise
```

```
gen :: nat -> (List nat)
gen 0 = []
gen (n+1) = ((n+1):(gen n))

reverse :: (List nat) -> (List nat)
reverse [] = []
reverse (a:x) = append (reverse x) [a]

append :: (List nat) -> (List nat) -> (List nat)
append [] y = y
append (a:x) y = (a:(append x y))

take :: nat -> (List nat) -> (List nat)
take 0 x = []
take (n+1) [] = []
take (n+1) (a:x) = (a:(take n x))

drop :: nat -> (List nat) -> (List nat)
drop 0 y = y
drop (n+1) [] = []
drop (n+1) (a:x) = drop n x

len :: (List nat) -> nat
len [] = 0
len (a:x) = 1 + (len x)

main :: (List nat)
main = msort (append (drop (div (len x) 2 (convert (2 >= 0) True)) x)
                     (take (div (len x) 2 (convert (2 >= 0) True)) x))
       where
        x :: (List nat)
        x = (gen 1000)
```

B.3.4 bubblesort

```
%insert "identityproofs2.fe"
%insert "msort.fe"



 bsort :: (List nat) -> (List nat)

 bsort l = passes (#l) l



 passes :: nat -> (List nat) -> (List nat)

 passes 0 x
        = x
 passes (n+1) x
        = pass (#pv) pv (nateqrefl (#pv))
          where
```

```
          pv :: (List nat)
          pv = (passes n x)

          nateqrefl :: (n::nat) -> (Id bool True (n=n))
          nateqrefl 0
                = identify True
          nateqrefl (n+1)
                = nateqrefl n
```

```
pass :: (n::nat) -> (l::(List nat)) -> (Id bool True ((#l)=n)) -> (List nat)

pass 0 l p
               = l
pass (n+1) [] p
               = abort (feqtimpbot p) (List nat)
pass (n+1) [a] p
               = [a]
pass (n+1) (a:b:x) p
               = (a:(pass n (b:x) newp))
                       , if a<=b
               = (b:(pass n (a:x) q))
                       , otherwise
                 where
                 q :: (Id bool True ((#(a:x))=n))
                 newp :: (Id bool True ((#(b:x))=n))
```

```
main :: (List nat)
main = bsort (reverse (gen 15))
```

B.3.5 permsort

```
compose :: (ca;cb;cc::Un 0) = (cb -> cc) -> (ca -> cb) -> ca -> cc
compose f g x = f (g x)

include :: (a :: Un 0) = a -> (List (List a)) -> (List (List a))
include x s
    = concat (map (compose (map smallins) mkpairs) s)
    where
        smallins :: ((List a) # (List a)) -> (List a)
        smallins (t1,t2) = t1 ++ [x] ++ t2

        mkpairs :: (List a) -> (List ((List a) # (List a)))
        mkpairs [] = [([],[])]
        mkpairs (h:t) = ([],(h:t)) : (map acons (mkpairs t))
                        where
                          acons :: ((List a) # (List a)) -> ((List a) # (List a))
                          acons (r,s) = (h:r, s)
```

```
perms :: (a :: Un 0) => (List a) -> (List (List a))
perms l = foldr include [[]] l

sorted :: (List nat) -> bool
sorted [] = True
sorted [a] = True
sorted (a:b:c) = sorted (b:c) , if a <= b
               = False        , otherwise



psort :: (List nat) -> (List (List nat))
psort l = take 1 (filter sorted (perms l))

main :: List (List nat)
main = psort [14,55,2,7,3,4,1,1,14]
```

B.3.6 treesort

Some functions are used below whose definitions appear in the mergesort program.

```
flatten :: (Tree nat) -> (List nat)
flatten NilT = []
flatten (Node x b c) = (flatten b) ++ [x] ++ (flatten c)

mktree :: (List nat) -> (Tree nat)
mktree [] = NilT
mktree (a:x) = place a (mktree x)

place :: nat -> (Tree nat) -> (Tree nat)
place a NilT = Node a NilT NilT
place a (Node b p q) = Node b (place a p) q , if a < b
                     = Node b p (place a q) , otherwise

tsort :: (List nat) -> (List nat)
tsort = compose flatten mktree

main :: (List nat)
main = tsort (append (drop (div (len x) 2 (convert (2 >= 0) True)) x)
               (take (div (len x) 2 (convert (2 >= 0) True)) x))
       where
            x :: (List nat)
            x = (gen 1000)
```

B.4 The UNIX manual page

The manual page for the scripts that act as interfaces to the Ferdinand system is shown on the following pages.

NAME

ferd2, mferd2 - interfaces to the Ferdinand functional programming system

SYNOPSIS

ferd2 [-cfhl] [-a *analysis*] *filename*

mferd2 [-cfhl] *filename...*

DESCRIPTION

Ferdinand is a lazy functional programming language, based upon intuitionistic type theory, which was developed by Andrew Douglas at the University of Kent at Canterbury.

ferd2 compiles a Ferdinand language input file (which should have the .fe extension) to produce FLIC (Functional Language Intermediate Code). Also, by default, the FLIC code produced is subsequently compiled (using **fc**) to produce an executable program.

Ferdinand scripts consist of a set of function definitions, one of which should be called *main*: this will give the topmost expression to be evaluated. The normal form of this expression should be the result given by the executable program. Ferdinand syntax is quite similar to that of Miranda (see **mira**(1)). The Ferdinand language is described fully in Andrew Douglas's PhD thesis (see below).

Since the Ferdinand compiler is written in Miranda, the **ferd2** script actually invokes the Miranda system, evaluating the *ferdinand* compilation function for a given input file and a given set of options. The set of options depends upon those flags given to **ferd2** (see below).

The Ferdinand compiler has been modified by Alastair Telford at UKC so that certain optimisations based upon abstract interpretation analyses may be performed. The **mferd2** script simply performs batches of compilations: each possible analysis/optimisation (including no optimisation) is invoked for every input file. Consequently, the **-a** flag is not valid for **mferd2**. The resulting FLIC files and executable code files are placed in a different directory for each input file. These files are given appropriate suffixes which indicate the optimisation technique which produced them. In addition, a file giving a record of the compilation process is produced.

OPTIONS

-c The Ferdinand script in *filename* is simply compiled, with no FLIC code being produced. (This means that the **-f** flag is set as well as there will be no FLIC code to compile.)

-f By default, not only is a FLIC code file produced but the FLIC file is then compiled to produce an executable object file. The **-f** flag is used to avoid the compilation of the FLIC code to an executable. Consequently, if this flag is used then only FLIC files will be produced.

-h Displays a help message.

-l The Ferdinand script in *filename* is a *literate* script.

-a *analysis* (This option is not allowed with the **mferd2** script.) An optimisation technique, *analysis*, is used when compiling the Ferdinand program. This will normally be an abstract interpretation, the results of which will affect the FLIC code produced. The analysis type which follows the **-a** option may be one of **Eager**, **Neededness**, **Strictness** and **Strabsence** (strictness and absence) or other valid forms of those names. Valid abbreviations are:

e,E,eager and EAGER
for making the evaluation totally eager (strict);

n,N,need and NEED
for performing neededness analysis and optimisation;

s,S, strict and STRICT
for performing strictness analysis and optimisation;

a,A,strab and STRAB
for performing strictness-and-absence analysis and optimisation.

FILES

Below, *basename* refers to the name of the Ferdinand file with the .fe extension stripped off.

basename.fl The FLIC file produced by **ferd2**.

basename The executable file produced by compiling *basename*.fl.

*basename*_d/*basename*_*analysis*.fl The FLIC files produced by **mferd2** for each different analysis (including no optimisation, which is signified by NONE).

*basename*_d/*basename*_*analysis* The executable files produced by **mferd2** from each of the above FLIC files.

*basename*_d/*basename*_out A record of the compilation process produced by **mferd2**.

SEE ALSO

A Compiled Functional Language With A Martin-Löf Type System, Andrew M. Douglas, PhD thesis, University of Kent at Canterbury, 1994;

mira(1);

Type Theory and Functional Programming, Simon J. Thompson, Addison Wesley, 1991;

FLIC - a Functional Language Intermediate Code, Simon L. Peyton Jones and M.S. Joy, Computing Science Departmental Report, University of Glasgow, 1990;

The Tempest, W. Shakespeare, 1611.

DIAGNOSTICS

The following errors and warning messages may be generated by the **ferd2** and **mferd2** scripts. Other errors may be given by the Ferdinand compiler itself. When appropriate a help message is given in addition to one of the diagnostics below.

"Need filename"

"Too many arguments" One and only one filename should be given to ferd2. (However, the latter error will not occur with mferd2 since that will take more than one filename.)

"...does not exist"

"...is not readable"

"...is empty" The given file cannot be found/is not readable/is empty, respectively.

"Unknown option..." An unrecognised option has been given.

WARNING MESSAGES

There are also some (non-fatal) warning messages:

"Extra analysis option ignored..." The **-a** option can be used just once. If it is used more than once then its first usage will determine the analysis used.

"Unrecognised type of analysis ignored."

BUGS

Please email any bug reports/suggestions to either amd2@ukc.ac.uk or ajt1@ukc.ac.uk.

AUTHORS

Andrew M. Douglas, amd2@ukc.ac.uk, who wrote the main Ferdinand system.

Alastair J. Telford, ajt1@ukc.ac.uk, who wrote the analysis/optimisation phase, the **ferd2** and **mferd2** scripts and this manual page.

Stephen P. Thomas, spt@cs.nott.ac.uk wrote the **fc** compiler.

COPYRIGHT

TRADEMARK NOTICE

Bibliography

[1] Samson Abramsky. Strictness analysis and polymorphic invariance. In Ganzinger and Jones [51].

[2] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd, 1987.

[3] Lennart A. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 218–227. ACM Press, 1984.

[4] Lennart A. Augustsson, Thierry Coquand, and Bengt Nordström. A short description of Another Logical Framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[5] Roland C. Backhouse et al. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, January-March 1989.

[6] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[7] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[8] Henk P. Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1(2):229–334, April 1991.

[9] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[10] Peter Nicholas Benton. Strictness analysis of lazy functional programs. Technical Report 309, University of Cambridge Computer Laboratory, August 1993. Version of the author's PhD thesis.

[11] Stefano Berardi. Pruning simply typed λ-terms. Technical report, Department of Computer Science, University of Turin, 1993.

[12] Stefano Berardi and Luca Boerio. Using subtyping in program optimization. In Mariangiola Dezani-Ciancagliani and Gordon Plotkin, editors, *TLCA 95*,

volume 902 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 1995. Typed Lambda Calculi and Applications. Edinburgh, Scotland, April 10–12, 1995.

[13] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[14] Bror Bjerner. *Time Complexity of Programs in Type Theory.* PhD thesis, University of Goteborg, 1989. Thesis for the Licentiate degree.

[15] Luca Boerio. Extending pruning techniques to polymorphic second-order λ-calculus. In Sannella [125], pages 120–134. 5th European Symposium on Programming. Edinburgh, Scotland, April 1994.

[16] Simon P. Booth and Simon B. Jones. A screen editor written in the Miranda functional programming language. Technical Report TR-116, University of Stirling, September 1994.

[17] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*, volume 97 of *London Mathematical Society Lecture Notes Series.* Cambridge University Press, 1987.

[18] A.W. Burks, H.H. Goldstine, and John von Neumann. *Preliminary discussion of the logical design of an electronic computing instrument.* Ablex Publishing Corporation, second edition, 1989. Paper was originally published in 1946.

[19] Geoffrey L. Burn. A relationship between abstract interpretation and projection analysis. In *Principles of Programming Languages*, pages 151–156. ACM Press, January 1990.

[20] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research monographs in parallel and distributed computing. Pitman in association with MIT press, 1991.

[21] Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. The theory of strictness analysis for higher-order functions. In Ganzinger and Jones [51].

[22] Geoffrey L. Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on a global program analysis. In *Proceedings of the Fifth International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, volume 714 of *Lecture Notes in Computer Science*, pages 346–364, Tallinn, Estonia, August 1993. Springer-Verlag.

[23] Rod M. Burstall. An approach to program specification and development in constructions. In Dybjer et al. [47].

[24] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[25] Rod M. Burstall, David B. MacQueen, and Don T. Sanella. Hope: an experimental applicative language. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 136–143. ACM press, 1981.

[26] Rod M. Burstall and James H. McKinna. Deliverables: an approach to program development in constructions. Technical Report ECS-LFCS-91-133, University of Edinburgh, 1991.

[27] Paul Chisholm. Derivation of a parsing algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[28] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[29] Chris Clack and Simon L. Peyton Jones. Strictness analysis—a practical approach. In *Functional Programming & Computer Architecture*, 1985.

[30] Robert L. Constable et al. *Implementing Mathematics with the NuPrL Proof Development System.* Prentice-Hall Inc., 1986.

[31] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Control*, 76, 1988.

[32] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming, Paris, France*, 1976.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference record of the Fourth ACM Symposium on Principles of Programming Languages.* ACM, 1977.

[34] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92: Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992. Proceedings of the Fourth International Symposium, Leuven, Belgium, 13–17 August 1992.

[35] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[36] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, pages 1–19, 1995.

[37] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[38] Kei Davis and Philip Wadler. Strictness analysis: Proved and improved. In Kei Davis and Philip Wadler, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing. Springer-Verlag, 1990.

[39] Kei Davis and Philip Wadler. Strictness analysis in 4D. In Peyton Jones et al. [118], pages 23–43.

[40] Nicolaas G. de Bruijn. *A survey of the project AUTOMATH.* In Seldin and Hindley [127], 1980.

[41] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[42] Antoni Diller. *Z: An introduction to Formal Methods.* Wiley, 1990.

[43] Andrew M. Douglas. The development of a functional programming language with a Martin-Löf type system. Research report, 1992.

[44] Andrew M. Douglas. *A Compiled Functional Programming Language with a Martin-Löf Type System.* PhD thesis, University of Kent at Canterbury, 1994. Currently being revised.

[45] Peter Dybjer. Computing inverse images. In *Proceedings of the International Conference on Automata, Languages and Programming*, 1987.

[46] Peter Dybjer. Inductively defined sets in Martin-Löf's set theory. In *Proceedings of the Workshop on General Logic, Edinburgh, February 1988.* Laboratory for the Foundations of Computer Science, University of Edinburgh, 1988. Proceedings published in technical report form: report number ECS-LFCS-88-52.

[47] Peter Dybjer et al., editors. *Proceedings of the Workshop on Programming Logic*, number 54 in Programming Methodology Group Report. University of Göteborg and Chalmers University of Technology, May 1989.

[48] Jon Fairbairn and Stuart C. Wray. TIM: A simple lazy abstract machine to execute super-combinators. In Gilles Kahn, editor, *Proceedings of the IFIP Symposium on Functional Programming Languages and Computer Architecture, Portland, Or., 14-16 September 1987*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[49] Solomon Feferman. Constructive theories of functions and classes. In N. Boffa, D. Van Dalen, and K. McAloon, editors, *Logic Colloquium '78*, North-Holland studies in logic and the foundations of mathematics, pages 159–224, 1979.

[50] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[51] Harald Ganzinger and Neil D. Jones, editors. *Proceedings of the Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[52] Kurt Gödel. On a hitherto unutilized extension of the finitary standpoint. *Dialectica*, 12:280–287, 1958.

[53] Chris L. Hankin. *Lambda Calculii*, volume 3 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[54] Chris L. Hankin and L. Sebastian Hunt. Approximate fixed points in abstract interpretation. *Science of Computer Programming*, 22(3):283–306, June 1994.

[55] Chris L. Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: application to strictness analysis. In *Proceedings of POPL'94*. ACM Press, 1994. Proceedings of the 21st ACM Symposium on Principles of Programming Languages.

[56] Chris L. Hankin and Daniel Le Métayer. Lazy type inference for the strictness analysis of lists. In Sannella [125], pages 257–271. 5th European Symposium on Programming. Edinburgh, Scotland, April 1994.

[57] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[58] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. MIT Press, 1988.

[59] Martin C. Henson. Information loss in the programming logic TK. In B. Möller, editor, *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods, Pacific Grove, CA, USA, 13-16 May 1991*. North-Holland, 1991.

[60] Martin C. Henson and Raymond Turner. A constructive set theory for program development. In *Proceedings of the 8th Conference on FST and TCS*, volume 338 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[61] Arend Heyting, editor. *Collected works of L.E.J. Brouwer*, volume I. North-Holland, 1975.

[62] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[63] C. Anthony R. Hoare and John C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[64] Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic parallelization of lazy functional programs. In G. Goos and J. Hartmanis, editors, *ESOP 92*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. 4th European Symposium on Programming. Rennes, France, February 26-28, 1992.

[65] William A. Howard. *The formulae-as-types notion of construction*. In Seldin and Hindley [127], 1980. Originally an unpublished manuscript from 1969.

[66] Paul Hudak et al. Report on the programming language Haskell (version 1.2). *ACM SigPlan Notices*, 27(5):1–164, 1992.

[67] R. John M. Hughes. Strictness detection in non-flat domains. In Ganzinger and Jones [51].

[68] R. John M. Hughes. Analysing strictness by abstract interpretation of continuations. In Abramsky and Hankin [2].

[69] R. John M. Hughes. Backwards analysis of functional programs. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208. Elsevier Science Publishers B.V. (North-Holland), 1988.

[70] R. John M. Hughes. Projections for polymorphic strictness analysis. In David H. Pitt et al., editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1989.

[71] R. John M. Hughes. Abstract interpretation of first-order polymorphic functions. In *Functional Programming: Proceedings of the 1988 Glasgow Workshop*, Workshops in Computing. Springer-Verlag, 1990. Workshop in Rothesay, 2-5 August 1988.

[72] R. John M. Hughes. Compile-time analysis of functional programs. In Turner [140], pages 117–155.

[73] R. John M. Hughes. Why functional programming matters. In Turner [140], pages 17–42.

[74] R. John M. Hughes and John Launchbury. Towards relating forwards and backwards analyses. In Peyton Jones et al. [118], pages 101–113.

[75] R. John M. Hughes and John Launchbury. Reversing abstract interpretations. *Science of Computer Programming*, 22(3):307–326, June 1994.

[76] L. Sebastian Hunt. Frontiers and open sets in abstract interpretation. *Functional Programming Languages and Computer Architecture*, pages 1–11, 1989.

[77] L. Sebastian Hunt. PERs generalise projections for strictness analysis. In Peyton Jones et al. [118].

[78] L. Sebastian Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, October 91.

[79] L. Sebastian Hunt and Chris L. Hankin. Fixed points and frontiers: a new perspective. *Journal of Functional Programming*, 1(1):91–120, 1991.

[80] Thomas P. Jensen. Disjunctive strictness analysis. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1992.

[81] Thomas Johnsson. Detecting when call-by-value can be used instead of call-by-need. Memo PMG-14, Programming Methodology Group, Institutionen för Informationsbehandling, Chalmers Tekniska Högskola, Göteborg, 1981.

[82] Mark P. Jones. Computing with lattices: an application of type classes. *Journal of Functional Programming*, 2(4):475–503, October 1992.

[83] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[84] Jesper Jorgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992.

[85] Jesper Jorgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 258–268. ACM, January 1992.

[86] Michael S. Joy and Simon L. Peyton Jones. FLIC - a functional language intermediate code. Technical report, University of Warwick, June 1990.

[87] Sam Kamin. Head-strictness is not a monotonic abstract property. *Information Processing Letters*, 41(4):195–198, 1992.

[88] John Launchbury. Strictness and binding-time analyses: two for the price of one. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 80–91. ACM Press, 1991.

[89] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[90] Zhaohui Luo. *Computation and Reasoning*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.

[91] Zhaohui Luo and Randy A. Pollack. LEGO proof development system: User's manual. Technical report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[92] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM press, 1986.

[93] Chris C. Martin and Chris L. Hankin. *Finding fixed points in finite lattices*, volume 274 of *Lecture Notes in Computer Science*, pages 426–445. Springer-Verlag, 1987.

[94] Per Martin-Löf. A theory of types. Technical Report 71-3, Department of Mathematics, University of Stockholm, 1971.

[95] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings of the Logic Colloquium, Bristol, July 1973*. North Holland, 1975.

[96] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North-Holland, 1982.

[97] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[98] Per Martin-Löf. *Constructive Mathematics and Computer Programming*. In Hoare and Shepherdson [63], 1985.

[99] James H. McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, November 1992. Available as technical report ECS-LFCS-92-247.

[100] Paul F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1987. Cornell technical report TR 87-870.

[101] Robin A. Milner. Theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[102] Alan J. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[103] Alan J. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, University of Edinburgh, 1981.

[104] Alan J. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Conference on Programming*, volume 167 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[105] Alan J. Mycroft and Arthur C. Norman. Optimising compilation, part I: Classical imperative languages. In *Proceedings of SOFSEM 92*, 1992.

[106] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1986.

[107] Eric G.J.M.H. Nöcker. Strictness analysis using abstract reduction. In *Proceedings of Conference on Functional Programming Languages and Computer Architectures.* ACM Press, 1993.

[108] Eric G.J.M.H. Nöcker et al. Concurrent Clean. In *Proceedings of Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, 1991.

[109] Bengt Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.

[110] Bengt Nordström et al. Discussion: "Problems of viewing proofs as programs". In Dybjer et al. [47]. Discussion about computational relevance, the subset type and the subset theory.

[111] Bengt Nordström and Kent Petersson. Types and specifications. In R. E. A. Mason, editor, *Information processing 83*, pages 915–920. IFIP 9th World Computer Congress, North-Holland, 1983.

[112] Bengt Nordström and Kent Petersson. The semantics of module specifications in Martin-Löf's type theory. Technical Report 36, University of Göteborg and Chalmers University of Technology, 1985.

[113] Bengt Nordström, Kent Petersson, and Jan M. Smith. An introduction to Martin-Löf's theory of types. Technical report, University of Göteborg and Chalmers University of Technology, 1986.

[114] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International series of monographs on computer science.* Oxford Science Publications, 1990.

[115] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages.* ACM press, 1989.

[116] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[117] Simon L. Peyton Jones. FLIC – a Functional Language Intermediate Code. Technical Report 2048, Department of Computer Science, University College London, April 1988.

[118] Simon L. Peyton Jones et al., editors. *Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool.* Springer-Verlag, 1991.

[119] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

[120] Grzegorz Rozenberg and Arto Salomaa. *Cornerstones of Undecidability*. Prentice Hall, 1994.

[121] Erik H. Saaman and Grant R. Malcolm. Well-founded recursion in type theory. Technical Report CS 8710, Department of Mathematics and Computer Science, University of Groningen, 1987.

[122] Anne Salvesen. On specifications, subset types and interpretation of propositions in type theory. *BIT*, 32(1):84–101, 1992.

[123] Anne Salvesen and Jan M. Smith. The strength of the subset type in intuitionistic type theory. In Dybjer et al. [47].

[124] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1989.

[125] Donald Sannella, editor. *ESOP 94*, volume 788 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. 5th European Symposium on Programming. Edinburgh, Scotland, April 1994.

[126] Peter Schroeder-Heister. The completeness of intuitionistic logic with respect to a validity concept based on an inversion principle. *Journal of Philosophical Logic*, (12), 1983.

[127] Jonathan P. Seldin and J. Roger Hindley, editors. *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, 1980.

[128] William W. Tait. Intensional interpretation of functionals of finite type, I. *Journal of Symbolic Logic*, 32, 1967.

[129] Yukihide Takayama. QPC: QJ-based proof compiler – simple examples and analysis. In Harald Ganzinger, editor, *ESOP 88*, volume 300 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988. 2nd European Symposium on Programming. Nancy, France, March 1988.

[130] Yukihide Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.

[131] Stephen P. Thomas. OPTIM/PG-TIM compilation system (including fc manual). Describes the usage of fc together with the organisation of the PG-TIM system in general, 1993.

[132] Stephen P. Thomas. *The Pragmatics of Closure Reduction*. PhD thesis, University of Kent at Canterbury, September 1993.

[133] Simon J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1(4):339–365, 1989.

[134] Simon J. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[135] Simon J. Thompson. Are subsets necessary in Martin-Löf's type theory? In J.P. Myers and M.J. O'Donnell, editors, *Constructivity in computer science: summer symposium, San Antonio, TX, June 19-22, 1991*, volume 613 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, 1992.

[136] Simon J. Thompson. Formulating Haskell. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer-Verlag, 1993.

[137] Simon J. Thompson. A logic for Miranda, revisited. *Formal Aspects of Computing*, 7(4):412–429, 1995.

[138] David A. Turner. *Functional Languages as Executable Specifications*, pages 29–50. In Hoare and Shepherdson [63], 1985.

[139] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1985.

[140] David A. Turner, editor. *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[141] David A. Turner. Elementary strong functional programming. In Peter Hartel and Rinus Plasmeijer, editors, *FPLE 95*, volume 1022 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 1st International Symposium on Functional Programming Languages in Education. Nijmegen, Netherlands, December 4–6, 1995.

[142] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw-Hill, 1991.

[143] Raymond Turner. Formalising Bishop in Type Theory. Talk given at the UKC-Essex day seminar series, May 1995.

[144] Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Abramsky and Hankin [2].

[145] Philip Wadler. Strictness analysis aids time analysis. In *Principles of Programming Languages*, San Diego, California, January 1988.

[146] Philip Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, August 1992.

[147] Philip Wadler and R. John M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987. Proceedings of Conference in Portland, Oregon, in September 1987.

[148] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), 1975.

[149] Stuart C. Wray. A new strictness detection algorithm. In L. Augustsson et al., editors, *Proceedings of the workshop on implementations of functional languages*, volume Report 17, Göteborg, 1985.

[150] Stuart C. Wray. *Implementation and Programming Techniques for Functional Languages*. PhD thesis, University of Cambridge Computer Laboratory, 1986.

[151] David A. Wright. A new technique for strictness analysis. In S. Abramsky and T.S.E. Maibaum, editors, *Theory and Practice of Software Development*, number 494 in Lecture Notes in Computer Science, pages 235–258. Springer-Verlag, 1991.