Abstract Interpretation of Constructive Type Theory

Alastair J. Telford and Simon J. Thompson

The Computing Laboratory, The University,
Canterbury, Kent, CT2 7NF, UK

{A.J.Telford,S.J.Thompson}@ukc.ac.uk

October 1996

Abstract

Constructive type theories, such as that of Martin-Löf, allow program construction and verification to take place within a single system: proofs may be read as programs and propositions as types. However, parts of proofs may be seen to be irrelevant from a computational viewpoint. We show how a form of abstract interpretation may be used to detect computational redundancy in a functional language based upon Martin-Löf's type theory. Thus, without making any alteration to the system of type theory itself, we present an *automatic* way of discovering and removing such redundancy. We also note that the strong normalisation property of type theory means that proofs of correctness of the abstract interpretation are simpler, being based upon a set-theoretic rather than a domain-theoretic semantics.

Keywords: Type theory, functional programming, computational redundancy, abstract interpretation.

Contents

Introduction

Over the last fifteen years constructive type theories such as Martin-Löf type theory [29] and the calculus of constructions [8] have been widely investigated by computer scientists. These theories are worth examining for a number of reasons.

- Constructive type theories are simultaneously (constructive) logics and (functional) programming languages and so they allow program construction and verification to take place in a single system.
- A proof constructed interactively can be interpreted as giving rise to a program 'witnessing' the theorem proved.
- Since propositions can be read as types, the type system of the programming language is considerably richer than in conventional functional languages such as Haskell [43]. For instance, any property (such as being a prime number) over a type defines a subtype of the type in question (in this case the natural numbers).

Despite the identification of propositions and types and of proofs and programs (the so-called *Curry-Howard isomorphism*) the activities of proving and programming have different purposes. Typically in proving a theorem we are interested in constructing a proof object, while in writing a program we are not only interested in constructing the program (or proof object) but also in executing the program, that is reducing expressions involving the program to normal form. In other words, proof construction is concerned with the *static* behaviour of the language: does this term have this type? Programming is concerned both with this and with the *dynamic* behaviour of expressions: what value does this expression have?

When we examine the reduction behaviour of expressions in constructive type theory we discover a disadvantage of identifying proofs and programs; the static and the dynamic are intimately linked, and particularly we find that static parts of a program can affect its dynamic behaviour. We consider two cases now.

- The existential type

  $$(\exists y : B)C(y)$$

  can be used to represent the subtype of B consisting of those elements b with the property $C(b)$. Note that we have used the term 'represent'; the elements of $(\exists y : B)C(y)$ are in fact pairs (b, p) where $b : B$ and p is a proof that b has the property C, that is $p : C(b)$. Elements of the existential type thus consist of pairs of data values and proof terms.

  Suppose, as suggested by [31] , that we take the proposition

  $$(\forall x : A)(\exists y : B)C(x, y) \tag{1}$$

as a specification of a function from A to B. Elements of the type (1) will in fact be functions returning pairs of values, the second component of which contains proof information. Using (1) as a specification of a function thus introduces information *irrelevant* to the computation itself.

- Given that the type system is more expressive than those in traditional languages we can give more accurate types to functions. We can, for example, express the fact that the *head* function on lists should only be applied to non-empty lists by giving it the type

  $$(\forall l : [A])(nonempty(l) \Rightarrow A)$$

  This means that the *head* function takes two arguments: a list and a proof that the list is non-empty. (The *nonempty* function is defined in Section 4.) In a well-formed application of the function,

  $$head\ e\ p \tag{2}$$

  the presence of the p ensures that the list e has a head – in other words it ensures that the expression (2) is well-typed – but the proof term is not needed in the calculation of the head of e itself. Again we can see here that the proof information which is important for the static checking of the application is nonetheless irrelevant to the computation of the result.

How are we to deal with this proof irrelevance?

One approach is to modify the types of the language to include the so-called 'information loss' types [36, 1] from which the witnessing information is removed. While this approach appears neat, it is unfortunately weaker than might be expected [36, 30] and so unworkable in practice.

Some separation of proof and data values can be given by judicious use of the axiom of choice, which is valid in Martin-Löf's type theory. We investigated this Skolemization process in [42].

A third approach, which can be seen in systems such as Coq [7] and a later re-working of Martin-Löf's theory [31], is to set aside the identification of propositions and types. While this has the desired effect of allowing the separation of the computationally relevant (inhabitants of types) and irrelevant (inhabitants of propositions) it has two drawbacks.

- The Curry-Howard isomorphism is an important principle, which allows the user of a type theory to think in two quite distinct ways about the same problem, and thus to give him or herself a greater chance of succeeding in finding its solution. Moreover, losing the identification means that there is substantial duplication of effort, as constructions have to be performed in both the universe of types and of propositions.

- When writing a specification or making a construction a user is forced to decide between using a proposition or a type: between making something computationally irrelevant or not. We suggest that this sort of choice is difficult to make; an incorrect decision can result in substantial effort having to be expended in reconstruction. Moreover, we see such a choice as being better made analytically than in an *ad hoc* way. We turn to this approach now.

In this paper we suggest two complementary approaches based on experience of implementation of mainstream functional programming languages [34].

First we observe that there are different strategies for reducing expressions to normal form. In applicative order reduction of a function application

$$f\ a\ b \tag{3}$$

the arguments a and b are fully evaluated before the application itself, while in normal-order reduction a and b are substituted unevaluated for the formal parameters of f, and are only reduced when and to the extent that it is necessary during the evaluation of the normal form of the whole expression (3). An optimisation of normal-order reduction is *lazy* evaluation, under which each argument a and b is evaluated at most once: this is achieved by substituting pointers to a and b for the formal parameters of f, and thereby results in the evaluation of a graph rather than an expression tree to normal form.

Consider an application of the form (2) above; the proof p will not be required in computing the head of e, and so under lazy evaluation will not be evaluated. We therefore avoid evaluating computationally irrelevant terms by evaluating expressions lazily. Nevertheless, there is an overhead in handling proof terms such as p during expression evaluation. They will, for instance, occupy considerable amounts of heap space during evaluation, and therefore slow down execution.

Our second insight is to analyse programs such as *head* to try to discover when arguments will and will not be needed. The paper shows how in cases such as *head* and many others we can use techniques of static analysis or abstract interpretation [9, 25, 5] to discover when arguments are not evaluated and therefore computationally irrelevant. We also show how these techniques can be combined with more traditional analyses, such as strictness analysis, which are seen in production-quality compilers of lazy functional languages [35].

A crucial part of any abstract interpretation is a proof that the analysis we perform is sound: in this case we show that any argument asserted to be unnecessary is indeed so. We characterise this in the set-theoretic semantics for the type theory as follows. f is said to be independent of its argument if for all possible arguments a and b

$$[\![\, f\ a \,]\!] = [\![\, f\ b \,]\!]$$

that is, the result is independent of the value of the argument.

What is novel about this approach? The simple set-theoretic semantics makes proofs considerably easier than in the case of a programming language containing unrestricted recursion and therefore requiring a domain-theoretic semantics.

Secondly, we stress that this method is used at compile time, and requires no change to the source language; it is therefore invisible to the programmer who works in a type theory having the Curry-Howard identification between propositions and types.

Finally, the analysis can be performed in conjunction with other techniques such as strictness analysis which form part of the traditional repertoire of the implementor.

Overview of the paper

The remainder of this paper is organised as follows. In Section 1 we describe the main features of type theory and how computationally irrelevant proof objects may occur. Section 2 describes the basic ideas of abstract interpretation, focussing upon backwards analysis. We show how this method is used to detect computational redundancy in the TT system in Section 3. Type theory allows types to be the inputs and results of functions. We consequently describe how types may be analysed in Section 4. Section 5 then gives some examples of the application of backwards analysis to type theory. In Section 6 we discuss the semantics of type theory as a prelude to Section 7 in which we demonstrate how the backwards analysis of type theory may be proved correct. This shows that our analysis cannot erroneously detect a parameter that will be needed by the computation as redundant. Our system has been implemented within a prototype compiler for a prototype functional system based upon type theory. This is described briefly in Section 8. Finally in Section 9 we describe related work and in Section 10 we present our conclusions.

Syntactic conventions

We now give the conventions to indicate different classes of syntactic objects in this paper:-

`Teletype` font is used for actual program fragments in a functional programming language such as Miranda[1] or Haskell.

Italic font is used to denote type theoretic program fragments. These include function names such as *index* and variables such as x and the selectors and constructors of the theory such as *lrec* and *Succ*.

Bold face is used to indicate abstract constants such as **A**, abstract variables such as **c** and **v** and abstract functions such as **f** and **g**.

1 Type theory

We describe the structure of type theory and how certain proof objects may be seen as computationally irrelevant. This is particularly evident when certain function definitions in a functional system based upon type theory are compared with their counterparts in a language such as Haskell.

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

1.1 Rules of TT

Type theory is an intuitionistic logic with explicit proof objects which are terms in an extension of the typed lambda calculus. The types of the theory are given meaning as intuitionistic sets and, by the Curry-Howard correspondence [22], may also be seen as both logical formulas and specifications of programs. We purposely adhere to this correspondence in this paper, identifying, for example, $\forall$ formulas with Π sets. We thus use the TT system presented in [41]. This approach is taken as we argue that it is not necessary to separate the types (sets) from logical formulas in order to obtain computational efficiency.

Each type is presented completely by four classes of rules: formation, introduction, elimination and computation. The first three classes use a natural deduction style whilst the last class, computation, uses the reduction rule style of lambda calculus.

Formation rules describe how types are formed from other types. The base types of the system such as *bool* and N may be formed without reference to other types and so may be seen as constants within U_0, the lowest in a hierarchy of type universes. For example, we have:

$$\frac{}{bool : U_0} \quad (bool\ Form)$$

Some types, such as $\exists$ types which represent dependent products, will require other types as premises in their formation:

$$\frac{A : U_m \qquad \begin{matrix}[x : A] \\ \vdots \\ P : U_n\end{matrix}}{(\exists x : A).P : U_{max(m,n)}} \quad (\exists\ Form)$$

Type theory also allows the formation of types dependent on values from other types, as is illustrated by the formation rule for equality types:

$$\frac{A : U_n \qquad a : A \qquad b : A}{I(A, a, b) : U_n} \quad (Equality\ Form)$$

Introduction rules specify how canonical terms of each type may be formed. For example, we have,

$$\frac{}{True : bool} \quad (bool\ Intro_t) \qquad \frac{}{False : bool} \quad (bool\ Intro_f)$$

The introduction of a pair relies upon two premises; the type of the second part of the pair relies upon the value of the first:

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

Elimination rules indicate how other terms may be defined using a term of the type in question. For example, an element of the *bool* type gives the following elimination rule:

$$\frac{tr : bool \qquad l : C[True/x] \qquad d : C[False/x]}{if\ tr\ then\ l\ else\ d : C[tr/x]} \quad (bool\ Elim)$$

Similarly, either the first or second part of a pair may be extracted:

$$\frac{p : (\exists x : A).P}{Fst\ p : A} \quad (\exists\ Elim_f) \qquad\qquad \frac{p : (\exists x : A).P}{Snd\ p : P[Fst\ p/x]} \quad (\exists\ Elim_s)$$

Computation rules show how the terms formed by the elimination rules will be reduced. Together, the set of computation rules for TT show how terms may be reduced to normal form. For example, the *if-then-else* construct of TT has the following computation rules:-

$$\begin{aligned} if\ True\ then\ l\ else\ d &\rightarrow l \\ if\ False\ then\ l\ else\ d &\rightarrow d \end{aligned}$$

The terms resulting from the elimination of products have the following computation rules:-

$$\begin{aligned} Fst(a,b) &\rightarrow a \\ Snd(a,b) &\rightarrow b \end{aligned}$$

There are is the special case of the $\perp$ type which does not contain any elements. However, if we have a pathological proof object of this type we can eliminate it to produce an arbitrary element of any type that we choose:

$$\frac{p : \perp}{abort_A\ p : A} \quad (\perp\ Elim)$$

$\perp$ does not have computation rules (it does not make sense to try to reduce a proof object that should not exist) and the *abort* term is defined to be in normal form.

Finally there are structural rules by which assumptions can be introduced, substitutions of convertible terms can be made and so on. We refer to [41] for details of these.

1.2 Dealing with computational irrelevancy

Computationally irrelevant parameters may be included within functions during the application of the relevant introduction rules. This is because we only want the functions to be applied to arguments of the correct form, such as non-empty lists in the case of the *head* of list function discussed in the introduction. We illustrate this by comparing the functions which index a list in Haskell (denoted by `!!`) and in TT. The Haskell definition is:

```
(!!)                       :: [a] -> Int -> a
(x:_)  !! 0                =  x
(_:xs) !! n | n > 0        =  xs !! (n-1)
(_:_)  !! _                =  error "PreludeList.!!: negative index"
[]     !! _                =  error "PreludeList.!!: index too large"
```

However, in type theory, the function is defined as follows (using the informal presentation style of type theoretic functions that comes from [41] — typing is denoted by ':' and list construction by '::' i.e. the reverse of the Haskell convention), where A denotes an arbitrary type.

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

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

As can be seen, the TT version has an extra parameter. In the Haskell version, if an out of range index is required from a list then an error occurs with an appropriate message. This is equivalent to the computation resulting in the undefined object of the type of the components of the input list. Thus such functions in Haskell are partial and each type contains an undefined element. In contrast, in TT, *all functions are total* and undefined terms do not exist. The *abort* clause in TT indicates that a pathological error has occurred during program development (when applying the *index* function) and the p is a nonsensical witness to the $\perp$ type, which should be empty. Such clauses will never be evaluated at run-time, assuming that we evaluate closed terms (i.e. terms containing no free variables) and that the TT system is consistent. The latter is expressed by $\perp$ containing no closed terms. During correct program development, however, the proof object p should be of type $\top$. p will then indicate that an appropriate index is being used with the given list.

Objects such as p are not, however, of computational interest since they do not form part of the results in which we are interested, such as the third element of a list. Moreover, they are formed from other inputs as part of the proof obligations in applying a function such as *index*. Whilst lazy evaluation will ensure that such parameters will not have to be calculated when computing the normal form of the output, they still a space burden during execution.

Instead of altering the type theory and weakening the Curry-Howard correspondence, as is the case with the subset theory, we use abstract interpretation to detect automatically computational redundancy at compile-time. Abstract interpretation will act upon the syntax of terms in TT to determine abstract semantic properties such as computational redundancy. Given such information we should be able to transform the TT version of *index* into one where the third parameter is eliminated.

2 Backwards analysis

2.1 Abstract interpretation

The idea of abstract interpretation is to discover, without executing a program, *partial information* about the parameters of the 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.

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.

2.2 Basic ideas of backwards analysis

The abstract values of backwards analysis, which we shall refer to as contexts, may be seen as sets of continuations [23] 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 [23, 24, 25] has linear complexity in relation to the number of arguments. This, as Hughes notes, is unlike forwards analysis, which in the form given in [6] is NP-complete. The linear complexity of backwards analysis 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 [12], 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. 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 [24] in order to gain non-trivial information.

With regard to data structures, the work of Hughes and Launchbury [26] 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. This advantage is of particular relevance to type theory since we will be dealing with large amounts of structured data with some components being purely proof theoretic. For example, some products will represent a function together with a proof that it meets some specification.

The use of backwards analysis can also be justified by the fact that the flow of information is naturally backwards in the case of the semantic properties in which we are interested. We know that the results of our programs are needed but we wish to deduce which parameters of functions may be discarded.

2.3 Contexts and lattices

We assume a knowledge of basic lattice theory, including the Knaster-Tarski fixpoint theorem, as may be found in [11].

Definition 1 (Context lattice)
A **context lattice** is a finite lattice (that is a set partially ordered by a relation $\sqsubseteq$ where each pair of elements has a greatest lower bound, called the meet, and a least upper bound, called the join) 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.
The least element of the lattice, **CONTRA** is a zero. We also denote the context join as *contor* ($\sqcup$) for which **CONTRA** is the identity.

2.3.1 ABSENT and CONTRA

We shall first discuss the two contexts which must be present in any context lattice. They may not, however, be necessarily distinct from other contexts in the lattice or each other. 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 applies 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:

$$\begin{array}{rcl} hd & : & (\forall l : [A]).((nonempty\, l) \Rightarrow A) \\ hd\, []\, p & \equiv_{df} & abort_A\, p \\ hd\, (a :: x)\, p & \equiv_{df} & a \end{array} \tag{4}$$

   nonempty is a function over lists which returns a type that depends upon whether the given list is empty or not. We give its definition in Section 4. In (4), 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 by a closed term if the theory is consistent.

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, **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. 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 abbreviate **ABSENT** by **AB** and **CONTRA** by **CR**.

2.3.2 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 conjunction. 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, for instance, to combine contexts resulting from different actual parameters to a function application. This is discussed in Section 3.6. There are the following restrictions upon the definition of the & operator:

1. The **ABSENT** context 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, 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.

2.3.3 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 over structured data.

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

Definition 2

$$\mathbf{strict\ c} = \mathit{inf}\{\mathbf{c}' \mid \mathbf{c}' \sqcup \mathbf{ABSENT} = \mathbf{c} \sqcup \mathbf{ABSENT}\}$$

This defines $\mathbf{strict\,c}$ uniquely since the set above must contain $\mathbf{c}$ at least and so must have an *inf*.

For example, in the sharing analysis lattice, 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 **strict** is simply the identity function as $\mathbf{AB} = \mathbf{CR}$.

2.4 Lattices for the analysis of TT

We now present the context lattices used to analyse programs in TT.

2.4.1 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. 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 example above, 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, we have to assign the **N** context as the abstract value of the parameter x 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 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 corresponds to $\subseteq$ on sets.

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.

Once we detect a parameter has having the context **U**, we can then remove it from the transformed version of the function.

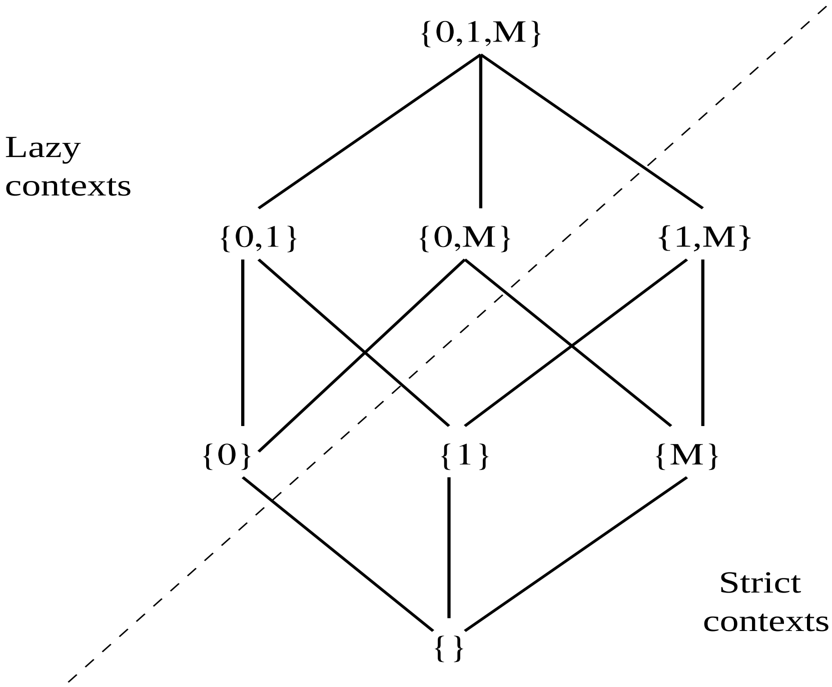

Figure 1: The sharing analysis lattice.

2.4.2 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).
- 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 [16].

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 1. $0, 1, M$ 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.

The following illustrates the contexts that we would expect to derive for a single call of *condshar*.

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

Here, the context of b should be $\{1\}$ as it is used precisely once. The context of x should be $\{1, M\}$ since x may be used once or a multiple of times 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 2 that on the sharing analysis lattice:

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

& 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 3

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 Structured data and contexts

2.5.1 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. 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 and so we have types of contexts in one-to-one correspondence with the types of the language. Each of these structured contexts has constructors akin to those in the syntax of type theory.

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**. If a type has more than one constructor then it is necessary to enumerate the different cases e.g. the empty and non-empty cases for lists.

2.5.2 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.)

Note that nullary context constructors such as $\mathbf{0}$ 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 [39, Section 2.14].

2.5.3 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** to denote the atomic and structured parts of a context, respectively.

2.5.4 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, if we have the list context,

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

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 & 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 which allows the strict part of input contexts to be propagated to the structured components.

2.5.5 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,

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

and

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

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

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

The results of & are approximated as detailed below.

2.5.6 Recursive data structures and context approximation

A problem arises with backwards analysis 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 $\mathbf{e}$ *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 results will in general be as follows:

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

The repetition in the result is guaranteed by the restriction we have placed upon the form of structured contexts. 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:

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

We use the symbol $\overset{\sqsubseteq}{\sim}$ to denote the fact that this approximating process has been applied and that the resulting context is safely ordered above the original one. That is, 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:

$$\mathbf{e} \overset{\sqsubseteq}{\sim} \mathbf{e}^{'}$$

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 process is an example of a widening operation [9, 10] in that it serves to find a fixpoint more readily, even if in general it will not be the least one.

We focus in this paper upon the recursive type of lists. The approximation techniques given above generalise to other types such as binary trees, as is discussed in [39].

2.6 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 parameters of TT functions 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 Section 3.

Context functions may be evaluated over atomic contexts so that 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. 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.

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

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.

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

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

The above properties were stipulated by Hughes in [24] for *concrete context domains*. The sharing lattice is an example of a concrete context domain, where the **ABSENT** and **CONTRA** elements are made distinct from the other elements of the lattice.

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.

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

In the above, $\mathbf{c'} = \mathbf{strict\ c}$ (**strict** is defined in Section 2.3.3). **E** is the defining expression of the context function, $\mathbf{f_i}$.

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.

3 Detecting computational redundancy

We now show how computational redundancy may be defined in our backwards analysis. Suppose then that we have some closed expression E, a variable, x, and some *initial context*, $\mathbf{c}$. $\mathbf{c}$ 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*. It is described by the following notation:

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

This represents a context that depends upon the value of the input context, $\mathbf{c}$.

We give an inductive definition of the formation of such context expressions. Amongst the base types, we shall focus particularly upon the $\perp$, $\top$ and equality types. It is these types that are one source of computational redundancy.

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

$$t \twoheadrightarrow \mathit{Triv}$$

since $\top$ is inhabited by one element only. As is stated in [1],

> 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.

We build upon the definitions of the base cases by showing how abstract values may be propagated throughout a type theoretic program. Consequently, we can detect automatically the computational irrelevancy of function parameters.

3.1 The type $\perp$

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.)

Abort constructs an arbitrary object of a type A, eliminating $p : \perp$, thus:

$$\frac{p : \perp}{\mathit{abort}_A\, p : A} \quad (\perp\ \mathit{Elim})$$

The term $\mathit{abort}_A\, p$ 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 $\mathit{abort}_A\, p$, where $p : \perp$, may not be reduced further, any parameter must be computationally irrelevant with respect to such closed expressions. Since we have a consistent theory, it will be impossible to construct a closed expression p of type $\perp$.

Definition 4

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

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

Here **ABSENT** will be reduced to the **AB** context of the type corresponding to x.

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 5.

3.2 The type $\top$

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:

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

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

We assume that we are dealing 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 reducing the term p is unnecessary since we know that it may be of one form only and we make the following definition:

Definition 5

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

for any variable, x.

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

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

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.

3.3 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 rule is

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

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. The computation rule states that

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

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 proved to be convertible (see [41, 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*. We can avoid computing the equality witness due to it being the sole inhabitant of its type.

Consequently, for closed terms, we state that it is not necessary to evaluate $c : I(A, a, b)$ (*cf* p.62 of [31] 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 6

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

for any variable x.

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

3.4 Variables

Computational redundancy may be introduced even more simply when an expression does not contain the parameter in question.

Definition 7

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 8

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

where x and y are distinct variables.

3.5 Conditionals

As noted in Section 2.3.2, the **contand** operation, $\&$, is used to combine together contexts in a manner similar to logical conjunctions. Also, the **contor** operation, $\sqcup$, is used to denote uncertainty and joins contexts together like disjunctions. These operations are useful in defining context propagation with respect to Booleans and selection over finite types, since we know that exactly one sub-expression must be evaluated in order to determine which other sub-expression will be the result of the conditional.

Boolean conditionals are thus handled by the following definition:

Definition 9

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

The above definitions capture the intuition that we must evaluate the boolean expression in an *if-then-else* expression before evaluating one of the branches. However, we cannot know in advance which branch will be evaluated. We can extend the above definition to any finite type.

3.6 Applications

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

$$\mathbf{f_i}\ \mathbf{c}$$

We may form an expression, given in terms of $\mathbf{c}$, for $\mathbf{f_i}\,\mathbf{c}$, as described below. As mentioned above, $\mathbf{f_i}\ \mathbf{c}$ 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:

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

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, $(\mathbf{f_i}\ \mathbf{c}) \xrightarrow{E_i} x$. Thus we have:

Definition 10

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

3.7 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, context types may be formed in correspondence to the types of the concrete semantics. Thus if we have a function f where:

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

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

$$\mathbf{f_i} : \mathbf{C}_T \to \mathbf{C}_{T_i}$$

Here, $\mathbf{C}_T$ is the type of contexts corresponding to the output type, T, of the original function. $\mathbf{C}_{T_i}$ is the type of contexts corresponding to the type of the ith input. In general, the basic procedure to find an expression for a context function is defined as follows.

Definition 11

If f is defined in the following way,

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

then we have,

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

where E is the expression over which f is abstracted.

This method can easily be extended to functions defined by pattern matching, as is shown in [39]. This will also be apparent in our examples.

3.8 Primitive recursive functions

It is essential to analyse the primitive recursive functions, such as *lrec*, which operate over types with inductive definitions such as lists. However, these functions are higher-order, as is shown by the computation rules for primitive recursion over natural numbers:

$$\begin{array}{rcl} prim\; 0\, c\, f & \to & c \\ prim\; (Succ\, n)\, c\, f & \to & f\, n\, (prim\, n\, c\, f) \end{array}$$

As can be seen, the third argument is applied to the predecessor of the second and the result of recursively applying *prim*. It follows that the contexts of the first and second arguments must be dependent on the context functions that will correspond to the parameter, f. For example, if f is absent in its second argument then this will mean that the expression c may not necessarily be evaluated. The theory that we shall outline will parameterise the context expressions that we derive with respect to a list of actual context functions. The contexts that will be derived for each parameter will thus depend not only on the input context but also on the context functions that are used.

Further details can be found in [39], where it is shown that the method can be adapted to cope with functions defined by partial application but cannot provide useful abstract values with regard to functions extracted from data structures.

4 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 [41]):

$$\begin{array}{rcl} nonempty & : & [A] \Rightarrow U_0 \\ nonempty\,[] & \equiv_{df} & \bot \\ nonempty\,(a :: x) & \equiv_{df} & \top \end{array}$$

This definition may be taken a step further. The function ranges over a type variable A. We may quantify over this variable as follows:

$$\begin{array}{rcl} nonempty' & : & (\forall A : U_0).[A] \Rightarrow U_0 \\ nonempty'\,A\,[] & \equiv_{df} & \bot \\ nonempty'\,A\,(a :: x) & \equiv_{df} & \top \end{array}$$

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 and so forth. For example,

$$\mathbf{nonempty'_1 c} = (\mathbf{c} \xrightarrow{\bot} A) \sqcup (\mathbf{c} \xrightarrow{\top} A)$$

Types such as $\top$ or N (natural numbers) may be viewed as constants so that, using the previous theory, we may see that the above will evaluate to **AB** and consequently we can optimize $nonempty'$ so that it is in a form with implicit rather than explicit polymorphism i.e. with the first parameter removed.

Those types which are formed from a number of components, such as $\forall$, $\exists$, $\vee$ may be analysed by taking the combination of the contexts propagated from each component. For instance, for equality types we have:

$$\mathbf{c} \xrightarrow{I(A,a,b)} x = (\mathbf{c} \xrightarrow{A} x) \,\&\, (\mathbf{c} \xrightarrow{a} x) \,\&\, (\mathbf{c} \xrightarrow{b} x)$$

The remainder of the analysis of type expressions is constructed in an analogous way to that for terms given earlier. We refer the reader to [39] for a fuller description.

5 Analysing 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, and which indeed does not involve this proof term at all.

5.1 Definition of the function in TT

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

$$
\begin{aligned}
index\ []\ n\ p &\equiv_{df} abort_A\ p && (5)\\
index\ (a{::}x)\ 0\ p &\equiv_{df} a && (6)\\
index\ (a{::}x)\ (n+1)\ p &\equiv_{df} index\ x\ n\ p && (7)
\end{aligned}
$$

Here the $\#$ function gives the length of a list, whilst the $<$ operator is a function which produces a type, either $\top$ (denoting a true proposition) or $\bot$ (an absurd proposition). The definition of $<$ is as follows:

$$
\begin{aligned}
< \quad &: \quad N \Rightarrow N \Rightarrow U_0\\
m < 0 &\equiv_{df} \bot\\
0 < (n+1) &\equiv_{df} \top\\
(m+1) < (n+1) &\equiv_{df} m < n
\end{aligned}
$$

5.2 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 (5) 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 give 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*:

$$\mathbf{index_1\ c} = \mathbf{at(c)}_{[], (\mathbf{c} \xrightarrow{(6),(7)} a) :: (\mathbf{c} \xrightarrow{(6),(7)} x)}$$

The head and tail contexts may, as they refer to clauses (6) and (7) 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:

$$\begin{aligned}\mathbf{c} \xrightarrow{(6),(7)} a &= (\mathbf{c} \xrightarrow{a} a) \sqcup (\mathbf{c} \xrightarrow{index\ x\ n\ p} a) \\ &= \mathbf{c} \sqcup \mathbf{AB}\end{aligned}$$

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

$$\begin{aligned}\mathbf{c} \xrightarrow{(6),(7)} x &= (\mathbf{c} \xrightarrow{a} x) \sqcup (\mathbf{c} \xrightarrow{index\ x\ n\ p} x) \\ &= \mathbf{AB} \sqcup \mathbf{index_1 c}\end{aligned}$$

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 zeroth approximation to the fixpoint is defined to be **CR** whilst the $(n+1)$st approximation is formed by substituting the nth approximation to the fixpoint for every occurrence of ($\mathbf{index_1\ c}$). As shown below, the third in a series of fixpoint iterations gives the result.

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

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

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.

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

5.3 Analysis of the second argument

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

The solution to this is again found by a fixpoint iteration and we find that:

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

This illustrates the fact that the second argument is lazy and even if it is used it may not be fully evaluated. However, it should be noted that the second parameter will be fully evaluated in the two non-pathological cases.

5.4 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 straightforward:

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

The least fixpoint solution of the above is:

$$\mathbf{index_3\,c} = \mathbf{AB}$$

We then have that:

$$\mathbf{index_3}\,\{1\} = \{0\}$$

We thus conclude that the third argument is not necessary when computing an application of *index* to normal form.

5.5 Functions defined using *index*

The value of the approach that we have taken is emphasised when we analyse function which are defined in terms of others such as *index*. For example, suppose that we have the function *third* which is a specialisation of *index* in that it extracts the third element from a list. To be defined in TT such a function requires an extra argument which proves that a given list has at least three elements. A suitable definition of *third* would be:

$$\begin{array}{rcl} third & : & (\forall l : [A]).((2 < \#l) \Rightarrow A) \\ third\, l\, p & \equiv_{df} & index\, l\, 2\, p \end{array}$$

Here,

$$\mathbf{third_3\ c = index_3\ c}$$

It follows that since the analysis will detect the third argument of *index* as being unused, the second argument of *third* will also be unused.

This applies to other expressions that include applications of the *index* function. For example, we may have:

$$(index\ [1,2,4]\ 2\ p) + (index\ [1,2]\ 1\ q)$$

where p and q are proofs that the indices make sense for each list. A backwards analysis of this expression should again show that both p and q may be eliminated from the transformed code.

6 A set theoretic semantics for TT

Martin-Löf type theory can be given a naive set-theoretic semantics, since we know from meta-theoretic results that all expressions are defined (see [41] for further references). In this section we explain the semantics which we will be using in later sections to show the safety of our analysis of neededness of function arguments. The set theory we use is standard; a good introduction is provided by [13], while wider and deeper coverage is given by [15].

The base types of the system are interpreted by the following sets

$$\begin{array}{rcl} [\![\bot]\!] & = & \emptyset \\ [\![\top]\!] & = & \{0\} \\ [\![N_n]\!] & = & \{0, 1, \ldots (n-1)\} \\ [\![N]\!] & = & \omega \\ [\![bool]\!] & = & \{True, False\} \end{array}$$

while the type constructors are given by

$$
\begin{aligned}
[\![A \vee B]\!] &= [\![A]\!] \,\dot{\cup}\, [\![B]\!] \\
[\![A \wedge B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A \rightarrow B]\!] &= [\![A]\!] \Rightarrow [\![B]\!] \\
[\![\mathit{list}\ A]\!] &= [\![A]\!]^{\star}
\end{aligned}
$$

where we use $\dot{\cup}$ for disjoint union, $\times$ for Cartesian product, $\Rightarrow$ for the set of (total) functions from the domain to the range and a superscript $\star$ for the set of finite sequences of elements of a given set.

In interpreting the quantifiers by means of families of sets we need environments (such as e) which record the bindings of free variables to sets; a binding (such as $(x \mapsto a)$) is added to the environment by the $\oplus$ operator.

$$
\begin{aligned}
[\![(\exists x : A)B]\!]_e &= \{(a, p) \mid a \in [\![A]\!]_e ; p \in [\![B]\!]_{e \oplus (x \mapsto a)}\} \\
[\![(\forall x : A)B]\!]_e &= \{f \in A \rightarrow \bigcup_{a \in [\![A]\!]_e} [\![B]\!]_{e \oplus (x \mapsto a)} \mid f(a) \in [\![B]\!]_{e \oplus (x \mapsto a)} \text{ for all } a \in [\![A]\!]\}
\end{aligned}
$$

Note that these interpretations are those of $A \wedge B$ and $A \rightarrow B$ when B does not contain x as a free variable.

The interpretations of the canonical constructors for these types can be read off from the interpretations of the types; non-canonical elements are constructed by the appropriate elimination operations over the sets. For instance, functions are interpreted as set-theoretic functions, that is as relations:

$$
[\![(\lambda x : A)b]\!]_e = \{(a, [\![b]\!]_{e \oplus (x \mapsto a)}) \mid a \in [\![A]\!]\}
$$

That functions can be defined by primitive recursion over ω is a theorem of ZFC, and using this theorem we can interpret functions over N; similarly structural recursion over $[\![A]\!]^{\star}$ is a consequence of the set-theoretic result that well-founded recursion is derivable in ZF.

Since our theory is intensional we interpret identity types as follows

$$
\begin{aligned}
&[\![I(A, a, b)]\!]_e \\
&\quad = \emptyset \qquad \text{if } \mathit{nf}(a) \neq \mathit{nf}(b) \\
&\quad = \{0\} \qquad \text{if } \mathit{nf}(a) = \mathit{nf}(b)
\end{aligned}
$$

where $\mathit{nf}(a)$ is the normal form of the expression a which is guaranteed to exist since the system is strongly normalising [ML73]; an extensional interpretation is given by

$$
\begin{aligned}
&[\![I(A, a, b)]\!]_e^{\mathrm{ext}} \\
&\quad = \emptyset \qquad \text{if } [\![a]\!]_e \neq [\![b]\!]_e \\
&\quad = \{0\} \qquad \text{if } [\![a]\!]_e = [\![b]\!]_e
\end{aligned}
$$

In what follows we shall omit the environment subscript when either it is empty or it can easily be inferred from the context.

The construction given here interprets types in the first universe, U_0, in Zermelo-Frankel set theory with the Axiom of Choice (ZFC). To interpret the theory with a hierarchy of universes $(U_n)_{n\in\omega}$ we augment ZFC with ω inaccessible cardinals, $(\kappa_n)_{n\in\omega}$ which gives a cumulative hierarchy of inner models of ZFC, V_{κ_n}, which can be used to interpret the universes U_n.

Theorem 1

If x is not free in the term b then for all a and a' in $[\![A]\!]$ and all environments e,

$$[\![(\lambda x : A)b]\!]_e\, a \quad = \quad [\![(\lambda x : A)b]\!]_e\, a'$$

Proof

$$[\![b]\!]_{e\oplus(x\mapsto a)} \quad = \quad [\![b]\!]_{e\oplus(x\mapsto a')}$$

for all expressions b and a, a' and e, by induction over the construction of the term b.
□

7 Correctness

We now give a characterisation of neededness of expressions in the theory. In particular, it is shown how the backwards analysis that has been developed may be demonstrated to be *safe* with respect to neededness, that is to be consistent with the semantics of type theory presented in Section 6.

We need to be able to show that the analysis that has been developed is correct since if it is not then there will be potentially catastrophic 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, in general, 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 definition of a computationally absent parameter has to be modified.

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 need to 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 Haskell, making a function strict in an argument cannot affect the semantics of a type theoretic program. In this sense, therefore, strictness analysis must be safe with respect to the semantics of type theory, which may be formulated in terms of simple set theory. Similarly, the only optimisations that can be performed based upon sharing information are ones to do with the allocation of storage in the machine that will run the resulting transformed program.

In the discussion that follows we shall simply be concerned with the contexts **N** and **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 lattice: we need to perform an abstraction on the contexts of this lattice to give contexts in $\{\mathbf{N}, \mathbf{U}\}$. These **abstractions of the context lattices** are as follows for atomic values:

$$\mathbf{abscxt\ c} = \begin{cases} \mathbf{U}, & \text{if } \mathbf{c} \sqsubseteq \mathbf{AB} \\ \mathbf{N}, & \text{otherwise} \end{cases}$$

So, for instance, **abscxt** $\{0\} = \mathbf{U}$ and **abscxt** $\{0, 1\} = \mathbf{N}$. The idea of abstractions of context domains/lattices comes from [24].

7.1 Definitions of safety

Definition 12

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. $[\![B]\!]_{(x \mapsto a)} = [\![B]\!]_{(x \mapsto b)}$
2. $[\![f]\!]\, a = [\![f]\!]\, b$

for any a,b of type $[\![A]\!]$. We also say that in this case the first parameter of f 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. Also, the definition of independence ensures that if a parameter is unused then we will be dealing with the non-dependent function space. 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.

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, and the atomic parts of its components, are all **N**.

An abstraction map, **abstr**, from the definition of (in)dependent parameters in the set theoretic semantics to the neededness context lattice is defined as follows:

Definition 13

$$\mathbf{abstr}\, f j = \begin{cases} \mathbf{N}, & \text{if } f \text{ is dependent on its } j\text{th parameter} \\ \mathbf{U}, & \text{if } f \text{ is independent of its } j\text{th parameter} \end{cases}$$

Theorem 2 (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,

$$\mathbf{abstr}\, f\, j \sqsubseteq \mathbf{at}(\mathbf{f_j}\, \mathcal{N})$$

where $\mathbf{f_j}$ is the context function of f's jth parameter.
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*.

7.2 Proof outline

We prove correctness for the terms of TT by employing the method of structural induction over the types and expressions of TT, just as in the definition of Section 6. In certain cases it is necessary to prove correctness for both atomic objects and their components should they have them. We present, as illustration of the correctness argument, three cases. Two of these deal with types exhibiting computational redundancy, whilst the third, lists shows how correctness is proven for components of a structured type.

7.2.1 Empty type

$\perp$, the absurd proposition, is interpreted by the empty set

$$[\![\perp]\!] = \emptyset$$

and in its elimination rule:

$$\frac{p : \perp}{abort_A\; p : A}\;(\perp\; Elim)$$

$abort_A$ is interpreted by an inhabitant of the *empty function type* with domain $\emptyset$; in other words, given the consistency of ZFC the type is uninhabited.

Since we have no interpretation of $abort_A$ we can say that it is independent of any closed term argument of type $\perp$.

Thus the safety condition must also hold as

$$\mathbf{abstr}\; f\; j = \mathbf{U}$$

where the jth parameter is of type $\perp$.

7.2.2 Single Element type

The type $\top$, which may be viewed as the "true" proposition, contains just one element, $Triv$, and this type is interpreted as the set $\{0\}$ in the set-theoretic semantics. Its formation, introduction, elimination and computation rules are as follows:

$$\frac{}{\top : U_0}\;(\top\; Form) \qquad \frac{}{Triv : \top}\;(\top\; Intro)$$

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

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

Since the semantics specifies that there is only one inhabitant of $[\![\top]\!]$,

$$[\![f]\!]\; a = [\![f]\!]\; b$$

for any elements $a, b \in [\![\top]\!]$.

This analysis applies particularly to the first argument of *case* 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:

$$\mathbf{at}(\mathbf{case_2}\; \mathcal{N}) = \mathbf{N}$$

and so it must follow that the safety condition is met i.e.

$$\mathbf{abstr}\; case\; 2 \sqsubseteq \mathbf{at}(\mathbf{case_2}\; \mathcal{N})$$

7.2.3 Functions

In analysing a function $(\lambda x : A)b$ we deduce that the variable x is unused if it is not free in the body of the function b. By the semantic Theorem 1 in such a case the interpretation of the function will be a constant set-theoretic function, and so by Definition 12 our analysis is safe.

7.2.4 Lists

The computation rules for *lrec* show how primitive recursion over lists is reduced.

$$
\begin{aligned}
lrec\,[]\,s\,f &\rightarrow s \\
lrec\,(a :: l)\,s\,f &\rightarrow f\,a\,l\,(lrec\,l\,s\ f)
\end{aligned}
$$

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 14

If $f : [A] \Rightarrow C$ and for any $a, b : [\![A]\!]$ and $l : [\![A]\!]^\star$

$$f\,(a \hat{::} l) = f\,(b \hat{::} l)$$

(where $\hat{::}$ is the interpretation of the cons operation, $::$) then we say that f **is independent of the head component of its argument**. We can define independence of the tail component in a similar way.

An examination of the semantic cases shows that our analysis is safe in these component cases as well as in the case of the complete argument.

Corresponding to the above definitions there is an extension to the **abstr** mapping so that safety may be characterised by the following equations:

$$\mathbf{abstr}_{hd}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{hd} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N} \tag{8}$$

$$\mathbf{abstr}_{tl}\, f\, j \sqsubseteq (\mathbf{at} \circ \mathbf{tl} \circ \mathbf{str} \circ \mathbf{f_j})\, \mathcal{N} \tag{9}$$

(Naturally, the above only apply when the argument of list type reduces to a non-empty value. Also, in (8), **hd** projects from the structured part of $\mathbf{C}_{[A]}$ to $\mathbf{C}_A$ and, in (9), **tl** is a projection from the structured part of $\mathbf{C}_{[A]}$ to $\mathbf{C}_{[A]}$.)

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 $\mathbf{c}'$ and $\mathbf{c}''$ are arbitrary contexts of the appropriate types. It follows that:

$$\begin{aligned}
\mathbf{abstr}_{tl}\, lrec\, 1 &\sqsubseteq (\mathbf{abstr}\, f\, 2) \sqcup (\mathbf{abstr}\, f\, 3) \\
&\sqsubseteq \mathbf{at}(\mathbf{AB} \sqcup (\mathbf{f_2}\, \mathcal{N} \sqcup \mathbf{f_3}\, (\mathbf{Fix}\, (\mathbf{lrec_1}\, [\mathbf{f}]\, \mathcal{N})))) \\
&= (\mathbf{at} \circ \mathbf{tl} \circ \mathbf{str} \circ (\mathbf{lrec_1}\, [\mathbf{f}]))\, \mathcal{N}
\end{aligned}$$

In the above, $\mathbf{lrec_1}\, [\mathbf{f}]$ denotes the context function of the first parameter of *lrec*, relative to the context functions that are deduced from the input function f. **Fix** gives the least fixpoint solution of the recursive context function definition of $\mathbf{lrec_1}$. Hence it has been proven that the analysis is safe with respect to list recursion and tail components of lists.

7.2.5 Other cases

Other cases in the analysis are examined in a similar way, with arguments for both the atomic and structured parts. It is clear in each case that the semantic description from Section 6 matches the intuition behind our earlier definitions and thus ensures the safety of the earlier analysis.

8 Implementation

This scheme of backwards analysis has been implemented within a prototype compiler for a functional language Ferdinand, based upon type theory [14]. Due to the fact that the compiler produces FLIC code [28] it was not possible to make optimisations based upon sharing analysis information. However, the code produced was optimised according to both neededness and strictness information. These experimental results showed that neededness analysis information could produce significant increases in speed in itself, although, as would be expected, best results were obtained from combining strictness with neededness information. For example, with a permutation sort program, a speed up of 9.4% was observed with neededness optimisation whilst code enhanced using both strictness and neededness information produced a speed increase of 132.4% with respect to unoptimised code. Strictness optimisation alone produced a speed up of 106.1%. We believe that further optimisations can be made with respect to neededness information since we have implemented *monovariant specialisation* of functions. This has meant that we use only one set of backwards analysis results for each function. We could instead produce different versions of object code for each function according to the different contexts in which the function may be called. This idea of *polyvariant* specialisation is explained further with regard to partial evaluation in [27]. It should also be mentioned that the FLIC compiler that we used, fc, was specially optimised to deal with the strictness information that was supplied [40].

Further information on the implementation of backwards analysis within the Ferdinand compiler may be found in [39, Chapter 4].

9 Related work

Paulin-Mohring [32] has presented a method of extracting programs, with computationally irrelevant material removed, from proofs in the calculus of constructions [8] 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 [38] 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 [37]. Berardi and Boerio [2, 4] built upon this by casting a lambda expression as a tree to be pruned. This was improved upon in [3] where a notion of subtyping was developed to produce a simplification relation which allowed optimized λ-terms to be deduced. Their algorithm detects and removes "useless computations" from a λ-calculus system based upon Gödel's system $\mathcal{T}$ [18]. 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.

Systems based upon Feferman's theory of types [17] may also be contrasted with this work. Such systems (e.g. TK [21] and PX [19]) 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. Of particular note is the paper by Henson [20] which discusses how the realizability process removes computationally redundant proof objects.

Turner has proposed a particular paradigm of functional programming whereby termination will be guaranteed [44]. This system of *elementary strong functional programming*, which is the subject of work in progress, has the advantage of not requiring the programmer to develop any computationally irrelevant proof objects in order to guarantee termination: strong normalization is guaranteed by syntactic restrictions upon the forms of recursion that are permitted. However, such a system may be seen to have 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 type theory.

10 Conclusion

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 [31], are unnecessary and we may adhere to a type theory based upon the original ideas of Martin-Löf [29, 41] which identifies logical propositions and types.

One of the important properties of type theory is strong normalisation. This has the effect of simplifying the semantics of terms in the theory and consequently makes proofs of correctness of the abstract interpretation more straightforward. This work therefore has both improved the viability of future functional programming systems based upon type theory and demonstrated the theory's ability to incorporate both programming and formal reasoning.

Acknowledgements

We would like to offer our thanks to various members of the Theoretical Computer Science group at the University of Kent at Canterbury who have offered help and advice, particularly Andy King and Andrew Douglas. The comments and criticisms of Ray Turner of Essex University have also been invaluable in helping to prepare this work. The first author is also grateful to the Engineering and Physical Sciences Research Council for a studentship award.

References

[1] Roland C. Backhouse et al. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, January-March 1989.

[2] Stefano Berardi. Pruning simply typed λ-terms. Technical report, Department of Computer Science, University of Turin, 1993.

[3] 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.

[4] Luca Boerio. Extending pruning techniques to polymorphic second-order λ-calculus. In Donald Sannella, editor, *ESOP 94*, volume 788 of *Lecture Notes in Computer Science*, pages 120–134. Springer-Verlag, 1994. 5th European Symposium on Programming. Edinburgh, Scotland, April 1994.

[5] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research monographs in parallel and distributed computing. Pitman in association with MIT press, 1991.

[6] Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. The theory of strictness analysis for higher-order functions. In 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.

[7] The Coq project. World Wide Web page by INRIA and CNRS, France, 1996. URL: `http://pauillac.inria.fr/~coq/coq-eng.html`.

[8] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Control*, 76, 1988.

[9] 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.

[10] 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.

[11] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[12] Kei Davis and Philip Wadler. Strictness analysis in 4D. In Peyton Jones et al. [33], pages 23–43.

[13] Keith J. Devlin. *Sets, functions and logic.* Chapman and Hall, 1992.

[14] 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.

[15] Frank R. Drake. *Set Theory.* North-Holland, 1974.

[16] 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.

[17] 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.

[18] Kurt Gödel. On a hitherto unutilized extension of the finitary standpoint. *Dialectica*, 12:280–287, 1958.

[19] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic.* MIT Press, 1988.

[20] 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.

[21] 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.

[22] William A. Howard. *The formulae-as-types notion of construction.* Academic Press, 1980. Originally an unpublished manuscript from 1969.

[23] R. John M. Hughes. Analysing strictness by abstract interpretation of continuations. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd, 1987.

[24] 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.

[25] R. John M. Hughes. Compile-time analysis of functional programs. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 117–155. Addison-Wesley, 1990.

[26] R. John M. Hughes and John Launchbury. Towards relating forwards and backwards analyses. In Peyton Jones et al. [33], pages 101–113.

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[28] Michael S. Joy and Simon L. Peyton Jones. FLIC - a functional language intermediate code. Technical report, University of Warwick, June 1990.

[29] 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.

[30] Bengt Nordström et al. Discussion: "Problems of viewing proofs as programs". In 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. Discussion about computational relevance, the subset type and the subset theory.

[31] 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.

[32] 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.

[33] Simon L. Peyton Jones et al., editors. *Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool.* Springer-Verlag, 1991.

[34] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[35] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In John T. O'Donnell and Kevin Hammond, editors, *Functional Programming, Glasgow 1993*, pages 201–220. Springer-Verlag, 1993.

[36] 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.

[37] 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.

[38] Yukihide Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.

[39] Alastair J. Telford. *Static Analysis of Martin-Löf's Intuitionistic Type Theory*. PhD thesis, University of Kent at Canterbury, 1995.

[40] Stephen P. Thomas. *The Pragmatics of Closure Reduction*. PhD thesis, University of Kent at Canterbury, September 1993.

[41] Simon J. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[42] 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.

[43] Simon J. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[44] 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.