UKC Computing TR 14–97

Ensuring the Productivity of Infinite Structures[1]

Alastair Telford[2] David Turner

September 1997
Revised March 1998

[1]This work was supported by the UK Engineering and Physical Sciences Research Council grant number GR/L03279. We would also like to thank members of the Theoretical Computer Science group at the University of Kent at Canterbury for their discussions in connection with this work, particularly Andy King, Erik Poll and Simon Thompson. Eduardo Giménez, of INRIA, France, has also been most helpful in explaining his ideas and how they have been implemented within the Coq system.
[2]*E-Mail*: `A.J.Telford@ukc.ac.uk`. *Tel*: +44 1227 827590. *Fax*: +44 1227 762811. *ESFP webpage*: `http://www.cs.ukc.ac.uk/people/staff/ajt/ESFP/`

Abstract

It is our aim to develop an elementary strong functional programming (ESFP) system. To be useful, ESFP should include structures such as streams which can be computationally unwound infinitely often. We describe a syntactic analysis to ensure that infinitely proceeding structures, which we shall term *codata*, are productive. This analysis is an extension of the check for *guardedness* that has been used with definitions over coinductive types in Martin-Löf's type theory and in the calculus of constructions. Our analysis is presented as a form of abstract interpretation that allows a wider syntactic class of corecursive definitions to be recognised as productive than in previous work. Thus programmers will have fewer restrictions on their use of infinite streams within a strongly normalizing functional language.

Contents

List of Tables

List of Figures

1 Introduction

We aim to develop an *Elementary Strong Functional Programming* (ESFP) system. That is, we wish to exhibit a language that has the strong normalization (every program terminates) and Church-Rosser (reduction strategies converge) properties whilst avoiding the complexities (such as dependent types, computationally irrelevant proof objects) of Martin-Löf's type theory [18, 26]. We would like our language to have a type system straightforwardly based on that of Hindley-Milner [11, 21] and to be similar in usage to a language such as Miranda[1] [28]. The case for such a language is set out in [31] — briefly, we believe that such a language will allow direct equational reasoning whilst being sufficiently elementary to be used for programming at the undergraduate level.

For such a language to be generally useful, it must be capable of programming input/output and, more generally, interprocess communication. The methods of doing this in Miranda, Haskell [27] etc., typically involve infinite lists (or *streams*), or other non-well-founded structures.

However, in languages such as Miranda, the presence of infinite objects depends upon the use of the *lazy evaluation* strategy in that terms are only evaluated as far as is necessary to obtain the result of a program. In those languages, infinite objects are syntactically undifferentiated from their finite counterparts and, indeed, are of the same type. For example, in Miranda, the lists `[1]` and `[1..]` both have the type `[num]`, despite the fact that the latter is an infinite list (of all the positive integers).

It is apparent that such structures pose problems if we wish to construct a language that is strongly Church-Rosser. Firstly, how can we ensure that our programs reach a normal form? Secondly, how do we do so without relying on a particular evaluation method, as is the case with Miranda etc.? Finally, should infinite objects have the same type as their finite counterparts?

We have argued in [31] that infinite structures, which we call *codata*, should be kept in a separate class of types from the finite ones (*data*), reflecting the fact that they are duals of one another, semantically. We have formulated rules for codata in an elementary term language in [30]. These rules ensure that programs involving codata and corecursion will be strongly Church-Rosser. However, we would like the ESFP source language to permit more free-wheeling definitions, which it should then be possible to translate into the intermediate language. We now need a compile-time check to ensure that these definitions are well-formed in the sense that the extraction of any piece of data from the codata structure will terminate. This means that, for example, the heads of infinite lists must be well-defined. Or, to put it another way, there is a continuous "flow" of data from the stream. Coquand [2] in Type Theory, and Giménez [8], in the Calculus of (Inductive) Constructions, have used the idea of *guardedness*, first proposed by Milner in the area of process algebras [22], to produce methods for checking whether corecursive terms are normalizable.

We argue that their notion of guardedness is too restrictive for programming practice in that it precludes definitions such as:

$$evens \stackrel{def}{=} 2 \diamondsuit (comap\,(+2)\; evens) \tag{1}$$

Here, $\diamondsuit$ is the *coconstructor* for infinite lists and *comap* is the mapping function

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

over infinite lists. Clearly, we can extract the nth positive even number from such a list, yet *evens* is unguarded according to the definitions used by Coquand and Giménez. Their notions of guardedness would appear to be sufficient for their purpose of *reasoning* about infinite objects, particularly within the Coq system [1], but are too limiting for programming in practice.

We have extended the idea of guardedness so that applications to the recursive call will not necessarily mean that they will be rejected as being ill-defined. To do this we have formulated the guardedness detection algorithm as an *abstract interpretation*. In particular, definitions of the form of (1) will be detected as being guarded. Conversely, our analysis is *sound* in that it will disallow definitions such as:

$$bh \stackrel{def}{=} 1 \diamondsuit (cotl\, bh)$$

Here *cotl* is the tail function over infinite lists.

Hughes, Pareto and Sabry have developed a type inference system [15] that can be used to determine whether corecursive definitions are productive. We shall exhibit an example of a productive definition that cannot be accepted by their system but which is accepted by ours.

Whilst it is undecidable whether a corecursive function is well-defined the extension to guardedness that we present here makes programming with infinite objects more straightforward in a strongly normalizing functional language.

Overview of this Paper. In Section 2 we give a summary of the theory behind infinite objects in strongly normalizing systems. We then show in Section 3 how the idea of guardedness can be extended by using an abstract interpretation. Examples of how the analysis detects whether a corecursive function is well-defined are given in Section 4. We shall give examples of how our analysis can accept productive definitions that are rejected by the Coquand/Giménez guardedness analysis and also by the type inference system of Hughes, Pareto and Sabry. This is followed in Section 5 by a proof that our analysis is sound. We complement this in Section 6 where we demonstrate that our analysis is complete with respect to that of Coquand. We also examine other properties of our analysis and discuss its advantages over the Hughes, Pareto and Sabry type inference system. Finally, in Section 7, we present our conclusions and suggestions for future work.

2 Infinite Objects

In this section we summarise how infinite objects have been represented in functional programming languages such as Miranda and Haskell and in systems based upon type theory. In general, infinite objects may be seen as the greatest fixed points of monotonic type operators. This, together with more details on the relationship between data and codata can be found in [24]. Here, however, we seek a concrete form of infinite data structures which does not rely upon the greatest fixpoint model and, moreover, does not rely on either a particular evaluation strategy or a type-theoretic proof system to have a sound semantics. We describe how we propose to represent infinite objects in an elementary strong functional language and why this requires the automatic syntactic check upon infinite recursive definitions that we present in the following sections.

2.1 Functional Programming and Infinite Data

Functional programming languages, such as Miranda, have exploited the idea of *lazy evaluation* to introduce the idea of infinite data structures. Hughes has pointed out the programming advantages of infinite lists in [14]. The disadvantages of these methods is that they rely upon a fixed evaluation strategy. In Miranda, definitions such as

```
ones = 1 : ones
```

only produce useful results with a lazy evaluation strategy (i.e. based upon call-by-name): a strict evaluation strategy (based upon call-by-value) would produce an undefined ("bottom") result for an evaluation of such a definition. There is also no guarantee that the streams will generate an arbitrary number of objects. For example, the following is a legal definition in Miranda:

```
ones' = 1 : tl ones'
```

However, it is only possible to evaluate the head of this list, whilst the rest is undefined. We have argued, in [31], that the existence of such partial objects greatly complicates the process of reasoning about infinite objects.

2.2 Guarded Infinite Objects

Coquand [2] in Type Theory and Giménez [8] in the Calculus of Constructions produced syntactic checks upon the definitions of infinite data structures which they called *guardedness*. (Giménez makes additional restrictions in order to cope with difficulties arising from impredicative types in the Calculus of Constructions.) The idea is similar to that formulated by Milner [22] for process algebras in that a check is made that recursive calls only occur beneath constructors. However, the work of both Coquand and Giménez is intended only to produce definitions of infinite structures that can be used within a proof system such as Coq [1] in order to prove coinductive propositions i.e. types of infinite structures. Their definitions of guardedness are, however, insufficient for a practical programming system. For example, we would not be allowed the following:

```
ints = 1 : map (+1) ints
```

This is due to the application of `map` to `ints`.

Hughes, Pareto and Sabry have developed a type inference system, not based directly on the idea of guardedness, for determining whether definitions are productive. In Section 4.3 we give a productive definition of the list of Fibonacci numbers which is rejected by their system but which is accepted by the guardedness analysis that we define in Section 3. In Section 6.2 we give reasons for why we believe that our system has advantages over theirs.

Conversely, the reasoning system of Sijtsma [25], being purely semantics-based, is not implementable as an automatic means of detecting whether a codata definition is productive.

Introduction rule

$$\frac{s :: S;\ \{y :: S,\ x :: S \Rightarrow\uparrow T\ \vdash\ X :: T\}}{\mathsf{Fix}\,(y = s)\, x.\, X ::\uparrow T}$$

Side condition: X must be purely introductory with regard to x.

Write $\mathsf{Fix}\, y\, x.\, X$ for $\lambda y'\,.\,\mathsf{Fix}\,(y = y')\, x.\, X$

Elimination rule

$$a ::\uparrow A\ \vdash \downarrow a :: A$$

Computation rule

$$\downarrow(\mathsf{Fix}\,(y = s)\, x.\, X) \to X[s/y,\ (\mathsf{Fix}\, y\, x.\, X)/x]$$

Normal form

$$\mathsf{Fix}\, s'\, F' ::\uparrow T$$

where s' and F' are both normal forms.

Table 1: Rules for codata.

2.3 Infinite Objects in ESFP

In ESFP, unlike in functional programming languages such as Haskell, we separate finite structures (*data*) from their infinite counterparts (*codata*). This is due to the fact that we cannot rely upon a lazy evaluation strategy to provide a computationally useful semantics for infinite structures. Indeed we seek *reduction transparency*. It is claimed that pure functional languages have the advantage of *referential transparency* over their imperative counterparts in that the meaning of expressions is independent of context. Reduction transparency goes further in that the semantics of expressions is independent of reduction order.

As in Coquand's approach for type theory [2], we have maintained the pivotal role of constructors in introducing codata. Thus, although we have separated codata from data, we have maintained similar syntactic forms to that of Haskell and Miranda. For example, the following is the type of infinite lists:

$$\mathsf{codata}\ \mathit{Colist}\, a \ \stackrel{def}{=}\ a \diamondsuit \mathit{Colist}\, a$$

Functions upon codata use *corecursion*: that is they recurse on their results rather than their inputs.

We need to check that an ESFP program will type check according to a set of rules that also serve to define an intermediate term language into which the top-level language may be translated. These rules, given in natural deduction style, are shown in Table 1 and were first given in [30]. They are derived from those of Mendler and others [20] for the Nuprl system, a variant of type theory.

Briefly, recursive occurrences of a type are replaced with their *suspension* (denoted with a $\uparrow$). This terminology comes from the fact that each layer of the structure lies dormant ("in suspension") until the function is applied. We keep separate reductions upon elements of an infinite structure from the structure's construction. Data or codata used to construct parts of the structure is *state* information. An infinite data structure will consist of:

- The data at its topmost level.
- A function to generate the next level of the structure, given some state information.
  This is the suspended part of the structure.

Parts of a suspended structure can only be obtained by applying the *unwind* function ($\downarrow$) to produce a normal form of a type T, $C\,e_1 \ldots e_n$, where each e_i is in normal form. Typically, some of the e_i will be the normal forms of suspensions of type T, $\uparrow T$. We have, in effect, made the lazy evaluation strategy that was implicit in the Haskell definition above, explicit in our approach. This method thus is also similar to simulations of lazy evaluation that have been produced for strict languages such as ML, as may be seen in [23].

It is the problem of guaranteeing the side condition of "X must be purely introductory with regard to x" in the introduction rule that will concern us in the rest of this paper. Indeed, it is this condition that determines whether our codata definitions are "productive" or not in the sense that normal forms can be produced when they are unwound. In [30] the restriction is a purely syntactic one — only constructors and no destructors are permitted. This is similar to Coquand's definition of guardedness. It would be more convenient to extend this in a way that is driven by semantic considerations. Formally, we have the following definition:

Definition 2.1 *Suppose that we have, $f :: A_1 \to \ldots \to A_n \to \uparrow T$, where $n \geq 0$, and that T is a sum of product types (i.e. $T \stackrel{def}{=} \sum_{i=1}^{i=m} (\prod_{j=1}^{j=N(i)} T_{i,j})$, where $N(i) \geq 0$). Then f is* ***productive****, written $\mathbf{Pr}(f)$, if and only if*

$$(\forall a_1 :: A_1^r \ldots a_n :: A_n^r)\ ((\downarrow (f\ a_1 \ldots a_n)) \twoheadrightarrow C_i\ e_{i,1} \ldots e_{i,N(i)})$$

where C_i is a constructor of type T, $\twoheadrightarrow$ is the reflexive, transitive finite-step closure of $\beta\eta$-reduction and each $e_{i,j}$ is in normal form. Here, A_i^r denotes all the reducible elements of type A_i (see Definition 2.2 below). In addition, each $e_{i,j}$ is reducible.

This definition of productivity can be extended to expressions in the obvious way where we form a combinator abstraction over the expression, e. We write $\mathbf{Pr}(e)$.

In tandem with the above, we have a definition of what it means for an expression to be reducible.

Definition 2.2 *An expression, e, is* ***reducible****, written $\mathbf{Rd}(e)$, if one of the following applies:-*

1. *e is data and is normalizable i.e. is convertible to normal form.*
2. *e is codata and is productive.*

We assert that expressions $e_1 \ldots e_n$ are reducible by using the notation, $\mathbf{Rd}(e_1, \ldots e_n)$

We shall assume here that all data is strongly normalizing. We ensure productivity (which is a property of the term model semantics of the ESFP rules) by defining an extension of Coquand and Giménez's idea of guardedness. This will serve as an abstraction of the property of productivity which is clearly undecidable.

3 Guardedness Analysis

In this section we define an abstract interpretation to detect whether a function definition is guarded. Rather than work with a *concrete* semantics[2] of infinite data structures (which may be expressed via our unwind function, for instance), we use a simpler, *abstract* semantics, whereby the meaning of a stream is given as a single ordinal. We do this by a form of *backwards analysis* which Hughes and others[3] have used to detect properties such as strictness within lazy functional programs. The point of a backwards analysis is that abstract properties, such as the guardedness levels that we shall define below, flow from the outputs of programs to the inputs. This reflects the intuitive way we think about infinite streams: the resulting list, *produced* rather than *analysed* by the function, is neither guarded nor is it split up into its component parts. Therefore we know that the guardedness level of the *result* is 0. We thus use 0 as an input to our guardedness functions in order to determine whether the recursive call(s) is guarded. If it is safely guarded by a constructor then the resulting guardedness level will be greater than 0.

3.1 The Abstract Guardedness Domain, A

The abstract guardedness domain, $\mathbf{A}$, is a complete lattice defined as the set, $\mathbb{Z} \cup \{-\omega, \omega\}$, where $-\omega$ and ω are the bottom and top points of the lattice, respectively. The usual ordering on $\mathbb{Z}$ applies to the rest of the lattice. We refer to elements of the lattice as **guardedness levels** and we call the greatest lower bound operator (which is necessarily both associative and commutative), min.

The guardedness levels represent the depth at which recursion occurs in the program graph. $-\omega$ indicates an unlimited or unknown number of destructions, whilst ω indicates that an infinite number of constructors will occur before a recursive call is encountered. No one program will use the whole lattice of guardedness levels since we will only have strictly finitary definitions in our source language.

We also have an associative and commutative addition operation, which is used to combine guardedness levels:

$$\begin{aligned} \omega +_{\mathbf{A}} x &\stackrel{def}{=} \omega & \\ x +_{\mathbf{A}} -\omega &\stackrel{def}{=} -\omega & \qquad (x \in \mathbb{Z} \cup \{-\omega\}) \\ x +_{\mathbf{A}} y &\stackrel{def}{=} x +_{\mathbb{Z}} y & \qquad (x, y \in \mathbb{Z}) \end{aligned}$$

This addition is used in calculating the resulting guardedness levels of applications and this is why ω takes precedence. In suspended computations if, as a

[2]The Cousots [4] have shown how different semantic views of infinite structures may be related through abstract interpretation.

[3][13] gives a good summary of abstract interpretation and backwards analysis in particular and [12] gives further details of backwards analysis.

result of a substitution, ω is the resulting guardedness level, then any corecursive calls in either the function being applied or the actual parameters must be irrelevant. This is so as the resulting substitution cannot contain a corecursive call.

3.2 Guardedness Functions

We define mappings, called *guardedness functions*, which transform guardedness levels. This transformation is based upon the syntax of a function definition in the source language. We assume that codata in our source-level language is based upon a sugaring of the following abstract syntax of expressions:

$$e ::= x \mid c \mid \lambda x.e \mid C e_1 \ldots e_n \mid f\, e \mid \mathsf{case}\ e\ \mathsf{of}\ (p_1 \to e_1) \ldots (p_n \to e_n)$$

Each c is a primitive constant and each p_i is a pattern match. Each source function definition will give rise to a number of guardedness functions. These functions are defined via an abstract semantic operator, $\mathcal{G}$, which maps from expressions to $\mathbf{A}$.

Definition 3.1 *Assume that a function definition has the form,* $f\ x_1 \ldots x_n \stackrel{def}{=} E$*. Then the* ***guardedness functions*** *of f are defined, relative to a vector $\boldsymbol{h}$ of actual parameter functions, as follows:*

$$\begin{array}{lll} f_0^{\#}\ \boldsymbol{h}\ 0 \stackrel{def}{=} \mathcal{G}(f, E, \boldsymbol{h}) & \\ f_i^{\#}\ \boldsymbol{h}\ 0 \stackrel{def}{=} \mathcal{G}(x_i, E, \boldsymbol{h}) & (i > 0) \\ f_i^{\#}\ \boldsymbol{h}\ g \stackrel{def}{=} g +_{\mathbf{A}} f_i^{\#}\ \boldsymbol{h}\ 0 & (g \neq 0, i \geq 0) \end{array}$$

In the above, $f_0^{\#}$ is the *principal* (or *zeroth*) guardedness function of f. It measures the degree to which the recursive call of f is guarded by constructors within its own definition. The addition of the guardedness level of a non-zero input to the result of the guardedness function upon a zero input reflects the fact that we are interested in the guardedness of the resulting substitution.

Definition 3.2 *We say that a function f is* ***guarded*** *(relative to a vector, $\boldsymbol{h}$, of actual parameter functions) if and only if*

$$f_0^{\#}\ \boldsymbol{h}\ 0 >_{\mathbf{A}} 0$$

The other guardedness functions, $f_i^{\#}$, where $i > 0$, reflect the extent to which the parameters of f are guarded within its definition. These *auxiliary* guardedness functions are important in that they allow us to determine whether functions passed as parameters to f will be guarded within f. It is by this mechanism of auxiliary guardedness functions that we can determine whether functions of the form, $f \ldots \stackrel{def}{=} \ldots (comap \ldots f) \ldots$, are guarded.

The set of guardedness functions thus produced will in general be recursive. Since the guardedness functions are *continuous*, as we shall prove in Lemma 3.1, below, and, since they operate upon a complete lattice, $\mathbf{A}$, their *greatest fixed point* exists and is found by forming a descending Kleene chain[4]. The continuity property is guaranteed by the following result:

[4]This contrasts with most abstract interpretations which deal with *least* fixed points and *ascending* chains. However, we have used the definitions here to retain compatibility with Coquand's approach. The definitions are also compatible with the fact that we are dealing with the greatest fixed points of coinductive type definitions.

$$\mathcal{G}(f, f, \boldsymbol{h}) \stackrel{def}{=} 0 \tag{2}$$
$$\mathcal{G}(f, c, \boldsymbol{h}) \stackrel{def}{=} \omega \tag{3}$$
$$\mathcal{G}(f, x, \boldsymbol{h}) \stackrel{def}{=} \omega \tag{4}$$
$$\mathcal{G}(f, \mathit{fname}, \boldsymbol{h}) \stackrel{def}{=} \mathcal{S}(f, \mathit{fname}, \langle\rangle) \tag{5}$$
$$\mathcal{G}(f, \lambda x.E, \boldsymbol{h}) \stackrel{def}{=} \mathcal{G}(f, E, \boldsymbol{h}) \tag{6}$$
$$\mathcal{G}(f, C\, a_1 \ldots a_n, \boldsymbol{h}) \stackrel{def}{=} 1 + \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) \tag{7}$$
$$\mathcal{G}(f, F\, a, \boldsymbol{h}) \stackrel{def}{=} \mathcal{F}(f, F, 1, \langle a\rangle, \boldsymbol{h}) \tag{8}$$
$$\mathcal{G}(f, (\mathsf{case}\ s\ \mathsf{of}\ \langle p_1, e_1\rangle \ldots \langle p_n, e_n\rangle), \boldsymbol{h}) \stackrel{def}{=} \min(\min_{i=1}^{i=n} \min(\mathcal{G}(f, e_i, \boldsymbol{h}), \mathcal{P}\,(p_i, e_i)\,\boldsymbol{h}\,g), g) \quad \text{where } g = \mathcal{G}(f, s, \boldsymbol{h}) \tag{9}$$

Table 2: Definition of the $\mathcal{G}$ operator.

Lemma 3.1 *The guardedness functions that we form are* ***continuous****, that is they are both monotonic (so that* $(x \leq y) \Rightarrow (f^{\#}\, \boldsymbol{h}\, x \leq f^{\#}\, \boldsymbol{h}\, y)$*) and preserve greatest lower bounds (so that* $f^{\#}\boldsymbol{h}\, \min(x, y) = \min(f^{\#}\, \boldsymbol{h}\, x, f^{\#}\, \boldsymbol{h}\, y)$*).*

Proof. The continuity of the guardedness functions follows immediately from $+_{\mathbf{A}}$ being continuous, monotonic and distributive over min; and the definition of guardedness functions on non-zero, non-omega guardedness levels i.e. that $f^{\#}\, \boldsymbol{h}\, x \stackrel{def}{=} x + f^{\#}\, \boldsymbol{h}\, 0$. In the case of the input guardedness level being ω then if $x \leq y$ and either or both of x and y is ω then it must follow that $f\, y = \omega$. Similarly, for $x = \omega$ and any y,

$$f\ \min(\omega, y) = f\, y = \min(\omega, f\, y) = \min(f\, x, f\, y)$$

□

The $\mathcal{G}$ operator is used to define the guardedness functions over the syntactic form of expressions in the source language. In defining this operator, we also need, in general, a vector of actual parameter functions, $\boldsymbol{h}$. This reflects the fact that our function definitions may be *higher-order*, as is the case with *comap* which applies a function to every element of a list. In practice, however, we shall often omit this vector where it is inessential or empty.

Definition 3.3 (The $\mathcal{G}$ operator) *Suppose that we have a named entity, f, which may be either a function or a variable name. We define the $\mathcal{G}$ operator, which produces the guardedness level of f relative to an expression in the source language, E, and a vector of actual parameter functions, $\boldsymbol{h}$, in Table 2. The definition of $\mathcal{G}$ involves the auxiliary operators, $\mathcal{S}$, $\mathcal{F}$ and $\mathcal{P}$, described below.*

3.2.1 Commentary on the $\mathcal{G}$ Operator Definition.

Clauses (8) and (9) extend the definitions of Guardedness given by Coquand and Giménez. (8) permits a function F (which may possibly be f itself) to be applied to an expression involving f. Furthermore, the function under analysis may be called as an actual parameter to itself and still may be guarded. (9) allows the possibility of corecursion occurring within the switch expression of a case.

$$\mathcal{F}(f, f, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \min(0, f_i^{\#}\, \boldsymbol{b}\, \mathcal{G}(f, a_i, \boldsymbol{h})) \tag{10}$$

In the above, $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{x}]$ (Component by component substitution.)

$$\mathcal{F}(x_j, x_j, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \min(0, \mathcal{F}(x_j, h_j, i, \boldsymbol{a}, \boldsymbol{h})) \tag{11}$$

$$\mathcal{F}(m_k, m_k, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \min(0, nom^{\#}\, \mathcal{G}(m_k, a_i, \boldsymbol{h})) \tag{12}$$

$$\mathcal{F}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \min(\mathcal{S}(f, \mathit{fname}, \boldsymbol{b}), \mathcal{N}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h})) \tag{13}$$

Here, $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{x}]$.

$$\mathcal{F}(f, x_j, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \mathcal{F}(f, h_j, i, \boldsymbol{a}, \boldsymbol{h}) \tag{14}$$

$$\mathcal{F}(f, p_k, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} nom^{\#}\, \mathcal{G}(f, a_i, \boldsymbol{h}) \tag{15}$$

$$\mathcal{F}(f, F'\, b, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \min(g', g'') \tag{16}$$

Here, $g' = \mathcal{F}(f, F', i, \boldsymbol{c}, \boldsymbol{h})$; $g'' = \mathcal{F}(f, F', i+1, \boldsymbol{c}, \boldsymbol{h})$; $\boldsymbol{c} = \langle b, a_1 \ldots a_n \rangle$

Table 3: Definition of the $\mathcal{F}$ operator.

Function applications. In clause (8) $\mathcal{F}$ is the *guardedness function applicator*: it is a function which constructs a guardedness function application from the corresponding application in the source program. Table 3 gives the definition of $\mathcal{F}$ which produces applications of guardedness functions from applications in the source syntax.

In the definition, $\mathit{fname} \in$ **FnNames**, the syntactic domain of function names; $x_i \in$ **ParNames**, the syntactic domain of formal parameter variables, not including pattern matching variables; $m_i \in$ **MatchVar**, the syntactic domain of pattern matching variables. The $\mathcal{S}$ operator, which calculates the guardedness level of a function name within the body of another function, is described below. The auxiliary function, $\mathcal{N}$, produces the guardedness level of the application of a named function:

$$\mathcal{N}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \begin{cases} \mathit{fname}_i^{\#}\, \boldsymbol{b}\, g & \text{if } i \leq \textbf{Arity}(\mathit{fname}) \\ nom^{\#}\, g & \textit{otherwise} \end{cases}$$

Here, $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{x}]$ and $g = \mathcal{G}(f, a_i, \boldsymbol{h})$.

The basic idea is that the ith auxiliary guardedness function is applied to the guardedness level of the ith actual parameter. To do this it uses the form of the function being applied, F, together with the index i of the actual parameter, a_i, and a new list of actual parameter functions formed by appending all the actual parameters of F to the vector $\boldsymbol{h}$.

If F is a variable, for example, an abstraction will be constructed so that when one of the actual parameter functions, $\boldsymbol{h}$, is substituted the correct guardedness function application will result. Where the ith auxiliary guardedness function does not exist, due to applications which return a function as their result, we must instead safely approximate using the $nom^{\#}$ function.

It should be noted that f *can* be applied to a call of itself and still be guarded, provided that its auxiliary guardedness functions return appropriate results on the guardedness levels of the actual parameters.

Where we do not know the actual parameter functions that comprise $\boldsymbol{h}$, an abstraction will be constructed over $\boldsymbol{h}$.

Note that lambda lifting upon our definitions is required and that lambda abstractions should be treated as named functions when they are applied.

The definition gives us the following for named function applications and we can derive similar results for other applicative forms.

Lemma 3.2

$$\mathcal{G}(f, \mathit{fname}\; a_1 \ldots a_n, \boldsymbol{h}) = \min(\mathcal{S}(f, \mathit{fname}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{N}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h}))$$

$\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{x}]$ where $\boldsymbol{x}$ consists of the formal parameters.

Proof. By induction on the number of actual parameters. □

Examples of this will be seen in Sect. 4 where the second argument of *comap* is applied in the definition of the Hamming function. This method of dealing with general applications, including higher-order constructs, comes from [12].

The $\mathcal{S}$ operator. In the above, $\mathcal{S}$ is the *substituted guardedness level* of f in F. It is intended to ensure that functions are guarded within mutually recursive definitions. If $\mathit{fname}\; y_1 \ldots y_m \stackrel{\mathit{def}}{=} E$ then

$$\mathcal{S}(f, \mathit{fname}, \boldsymbol{h}) \stackrel{\mathit{def}}{=} \mathcal{G}(f, E, \boldsymbol{h})$$

Here the idea is to produce the guardedness level of the function being analysed relative to its actual occurrences within another function's definition.

Case expressions. Clause (9) extends the class of definitions that are allowed in that the recursive call may conceivably occur in the switch, s, of the case expression. This means that the guardedness of s, relative to the recursive call is paramount when considering the guardedness of the whole expression: the case expression cannot be productive if the switch is not productive. This is why the resulting guardedness level is the minimum of the guardedness level of the switch together with the guardedness level of the rest of the components of the case expression. Even if the switch is productive, we have to ensure that each part of the structure that may be split up by this pattern matching process is in turn guarded. This is done by defining the *pattern guardedness function*, $\mathcal{P}$, for every pattern, expression pair in the case statement. $\mathcal{P}$ is defined as follows:

$$\mathcal{P}\;(p_i, e_i)\;\boldsymbol{h}\;0 \stackrel{\mathit{def}}{=} \min_{j=1}^{j=N(i)} (\mathcal{G}(v_i^j, e_i, \boldsymbol{h}) - \mathcal{D}(v_i^j, p_i))$$

Here, $\mathcal{D}$ is the *level of destruction* function of the infinite object, f i.e. the depth of a pattern matching variable where depth is measured by the number of constructors. It is defined as follows:

$$\begin{aligned} \mathcal{D}(v, v) &\stackrel{\mathit{def}}{=} 0 \\ \mathcal{D}(v, x) &\stackrel{\mathit{def}}{=} -\omega \\ \mathcal{D}(v, C\; q_1 \ldots q_n) &\stackrel{\mathit{def}}{=} 1 + \max_{i=1}^{i=n} \mathcal{D}(v, q_i) \end{aligned}$$

Here, max and $-$ are the dual operations to min and $+$, respectively. In the definition of $\mathcal{P}$, above, $v_i^j \in \mathrm{Var}(p_i)$ where $\mathrm{Var}(p_i)$ is the set of variables in the pattern, p_i. In addition, $N(i) \stackrel{def}{=} |\mathrm{Var}(p_i)|$. If analyse over our intermediate language there is no need for the $\mathcal{D}$ function since there the patterns can only be one-level deep: in order to get to refer to the third element of an infinite list, say, we would have to apply the unwind function ($\downarrow$) three times. Thus, in this case, we may simply subtract one from the guardedness level for each application of the unwind function.

It may be noted that, in the definition of the $\mathcal{F}$ operator it is possible that terms in the vector $\boldsymbol{h}$ may contain pattern matching variables. To avoid pattern-matching variable capture, therefore, it is necessary that α-conversions are performed. When any pattern-matching variable is applied, however, the $nom^{\#}$ guardedness function will be its abstract interpretation. This reflects the fact that we cannot determine, in general, the guardedness properties of an arbitrary function that has been projected from a data structure.

An alternative approach to finding the guardedness level of each pattern matching variable would be to substitute the projection, $\pi_{i,j}\ s$ for each occurrence of v_i^j in e_i and then calculate the guardedness level of the function f in the resulting expression. The projection function[5], $\pi_{i,j}$, would be defined thus:

$$\pi_{i,j}\ t \stackrel{def}{=} \mathsf{case}\ s\ \mathsf{of}\ p_i \to v_i^j \tag{17}$$

This would produce $-\mathcal{D}(v_i^j, p_i)$ as the result of its auxiliary guardedness function, where p_i is exactly the same pattern as in the original case expression.

4 Examples of Guardedness Analysis

In this section we show how guardedness functions may be used to detect whether certain streams are well-defined or not. As a substantial example, we look at the Hamming function which, in the form that we give, cannot be detected as being guarded by the definitions of Coquand [2] or Giménez [8]. In another example we show that a form of the list of Fibonacci numbers, which the type inference method of Hughes et al. [15] will not accept, is guarded according to our analysis.

In the analyses that follow we shall assume that the guardedness functions of purely recursive functions such as *compare* will be the identity guardedness function. We shall omit the vector of actual parameter functions except where necessary. We shall also refer to larger expressions by E, E', E'' etc. We shall also assume that definition via pattern matching is a sugaring of nested case expressions. The type *Colist* here consists of the streams of integers.

4.1 Detecting Non-Productive Definitions

Consider the definition:

$$ones \stackrel{def}{=} 1 \Diamond\ (cotl\ ones)$$

where *cotl* gives the tail of an infinite list:

$$cotl\ (h \Diamond t) \stackrel{def}{=} t$$

[5] $\pi_{i,j}$ is not total over the type of t but it *is* total in the context that it is used within the case expression. That is, we are assured that t is of the subtype of terms that begin with the constructor C_i. This form of subtyping is used in [19].

$$
\begin{array}{l}
\mathit{ham} :: \mathit{Colist} \\
\mathit{ham} \stackrel{def}{=} 1\Diamond(\mathit{comerge}\ (\mathit{comap}\ (\times 2)\ \mathit{ham})\ (\mathit{comap}\ (\times 3)\ \mathit{ham})) \\[4pt]
\mathit{comap} :: (\mathit{Int} \longrightarrow \mathit{Int}) \longrightarrow \mathit{Colist} \longrightarrow \mathit{Colist} \\
\mathit{comap}\ f\ (a\Diamond y) \stackrel{def}{=} (f\ a)\Diamond(\mathit{comap}\ f\ y) \\[4pt]
\mathit{comerge} :: \mathit{Colist} \longrightarrow \mathit{Colist} \longrightarrow \mathit{Colist} \\
\mathit{comerge}\ l@(a\Diamond x)\ m@(b\Diamond y) \stackrel{def}{=} \\
\quad \mathsf{case}\ \mathit{compare}\ a\ b\ \mathsf{of} \\
\qquad \mathsf{LT} \rightarrow a\Diamond(\mathit{comerge}\ x\ m) \\
\qquad \mathsf{EQ} \rightarrow a\Diamond(\mathit{comerge}\ x\ y) \\
\qquad \mathsf{GT} \rightarrow b\Diamond(\mathit{comerge}\ l\ y)
\end{array}
$$

Figure 1: Definition of the Hamming function.

The definition of *ones* is obviously non-productive since its tail consists of an irresolvable circularity. This is detected as follows:

$$
\begin{aligned}
\mathit{ones}_0^{\#}\ 0 &= \mathcal{G}(\mathit{ones}, 1\Diamond \mathit{cotl}(\mathit{ones})) \\
&= 1 + \mathcal{G}(\mathit{ones}, \mathit{cotl}(\mathit{ones})) \\
&= 1 + \mathit{cotl}_1^{\#}\ \mathcal{G}(\mathit{ones}, \mathit{ones}) \\
&= 1 + \mathit{cotl}_1^{\#}\ 0
\end{aligned}
$$

Now,

$$
\begin{aligned}
\mathit{cotl}_1^{\#}\ 0 &= \mathcal{G}(l, \mathsf{case}\ l \mathsf{of} (h\Diamond t) \rightarrow t) \\
&= \mathcal{P}\ ((h\Diamond t), t)\ \mathcal{G}(l, l) \\
&= \min(\mathcal{G}(h,t) - 1, \mathcal{G}(t,t) - 1) \\
&= \min(\omega - 1, 0 - 1) \\
&= -1
\end{aligned}
$$

Hence,

$$\mathcal{G}(\mathit{ones}, 1\Diamond \mathit{cotl}(\mathit{ones})) = 0$$

Thus the definition of *ones* is not guarded.

4.2 The Hamming Function

The Hamming function, *ham* is defined as the list of positive integers that have only 2 and 3 as their prime factors — further details on such a function can be found in [6]. It and functions used in its definition are given in a Haskell-like syntax in Figure 1.

In the analyses that follow we shall assume that the guardedness functions of purely recursive functions such as *compare* will be the identity guardedness function. We shall omit the vector of actual parameter functions except where necessary and refer to larger expressions by E, E', E'' etc. We shall also assume that definition via pattern matching is a sugaring of nested case statements.

4.2.1 Analysis of Guardedness of the *comap* and *comerge* Functions.

$$\begin{array}{rcl}
comerge^{\#}_0\ 0 & = & \mathcal{G}(comerge, \textsf{case}\ l\ \textsf{of}\ (a\diamondsuit x) \to E') \\
& = & \mathcal{G}(comerge, E') \\
& = & \mathcal{G}(comerge, \textsf{case}\ m\ \textsf{of}\ (b\diamondsuit y) \to E'') \\
& = & \mathcal{G}(comerge, E'') \\
& = & \mathcal{G}(comerge, \textsf{case}\ compare\ a\, b\ \textsf{of}\ E''') \\
& = & \mathcal{G}(comerge, E''') \\
& = & \min(\mathcal{G}(comerge, a\diamondsuit(comerge\ x\, m)), \\
& & \quad \mathcal{G}(comerge, a\diamondsuit(comerge\ x\, y)), \\
& & \quad \mathcal{G}(comerge, b\diamondsuit(comerge\ l\, y))) \\
& = & \min(1 + \mathcal{G}(comerge, comerge\ x\, m), \\
& & \quad 1 + \mathcal{G}(comerge, comerge\ x\, y), \\
& & \quad 1 + \mathcal{G}(comerge, comerge\ l\, y)) \\
& = & \min(1, 1, 1) = 1
\end{array}$$

Therefore, *comerge* is guarded.

$$\begin{array}{rcl}
comap^{\#}_0\ 0 & = & \mathcal{G}(comap, (f\, a)\diamondsuit(comap\ f\, y)) \\
& = & 1 + \mathcal{G}(comap, comap\ f\, y) \\
& = & 1 + 0 = 1
\end{array}$$

Therefore, *comap* is guarded.

4.2.2 Analysis of Auxiliary Guardedness Functions of *comap* and *comerge*.

In order to analyse the *ham* function we shall need to know the level of guardedness of the second argument of *comap* and of both of the two arguments of *comerge*.

$$\begin{array}{rcl}
comap^{\#}_2\ \langle h\rangle\, 0 & = & \mathcal{G}(l, \textsf{case}\ l\ \textsf{of}\ (a\diamondsuit y) \to E') \\
& = & \min(\mathcal{G}(l, E', \langle h\rangle), \mathcal{P}\ (a\diamondsuit y, E')\ \langle h\rangle\, 0, 0) \\
\mathcal{G}(l, E', \langle h\rangle) & = & \mathcal{G}(l, (f\, a)\diamondsuit(comap\ f\, y), \langle h\rangle) = \omega \\
\mathcal{P}\ (a\diamondsuit y, E')\ \langle h\rangle\, 0 & = & \min(\mathcal{G}(a, E', \langle h\rangle) - 1, \mathcal{G}(y, E', \langle h\rangle) - 1) \\
\mathcal{G}(a, E', \langle h\rangle) & = & 1 + \mathcal{F}(a, f, 1, \langle a\rangle, \langle h\rangle) \\
& = & 1 + h^{\#}_1\ 0 \\
\mathcal{G}(y, E', \langle h\rangle) & = & 1 + comap^{\#}_2\ \langle h\rangle\, 0
\end{array}$$

It follows that,

$$comap^{\#}_2\ \langle h\rangle\, 0 \quad = \quad \min(h^{\#}_1\ 0, comap^{\#}_2\ \langle h\rangle\, 0, 0)$$

Note that $comap^{\#}_2$ depends upon the form of the actual parameter, h. Typically, the stream consists of *data* elements and so the function being applied by *comap* will have a guardedness function equivalent to the identity. However, it is possible that a corecursive function may be applied in the case where the stream consists of a list of codata. In such a case, the application of the guardedness function, h, will ensure that the stream itself is productive only if each of its tributaries, so to speak, is productive.

The analysis of the auxiliary functions of *comerge* proceeds as follows.

$$\begin{array}{rcl} comerge_1^{\#}\ 0 & = & \mathcal{G}(l, \mathsf{case}\ l\ \mathsf{of}\ (a \Diamond x) \to E') \\ & = & \min(\mathcal{G}(l, E'), \mathcal{P}\,(a \Diamond x, E')\,0, 0) \\ \mathcal{G}(l, E') & = & 1 + comerge_1^{\#}\ 0 \\ \mathcal{P}\,(a \Diamond x, E')\,0 & = & \min(\mathcal{G}(a, E') - 1, \mathcal{G}(x, E') - 1) \\ \mathcal{G}(a, E') & = & \mathcal{G}(a, \mathsf{case}\ m\ \mathsf{of}\ (b \Diamond y) \to E'') \\ & = & \mathcal{G}(a, \mathsf{case}\ compare\ a\, b\ \mathsf{of}\ E''') \\ & = & \min(1 + \mathcal{G}(a, a), 1 + \mathcal{G}(a, a), \omega) = 1 \\ \mathcal{G}(x, E') & = & \min(1 + comerge_1^{\#}\ 0, 1 + comerge_1^{\#}\ 0, \omega) \end{array}$$

Thus,

$$\begin{array}{rcl} comerge_1^{\#}\ 0 & = & \min(1 - 1, \min(1 + comerge_1^{\#}\ 0, 1 + comerge_1^{\#}\ 0, \omega) - 1, 0) \\ & = & \min(0, comerge_1^{\#}\ 0, 0) \end{array}$$

The greatest fixpoint of the functional corresponding to this equation is 0.

Likewise, $comerge_2^{\#}\ 0 = \min(\mathcal{G}(b, E'') - 1, \mathcal{G}(y, E'') - 1, 0)$, and the solution to this is also 0.

4.2.3 Analysis of the Main Function, *ham*.

$$\begin{array}{rcl} ham_0^{\#}\ 0 & = & 1 + \mathcal{G}(ham, comerge\ (comap\ (\times 2)\ ham)\ (comap\ (\times 3)\ ham)) \\ & = & 1 + \min(\mathcal{S}(ham, comerge), \\ & & \qquad (comerge_1^{\#}\ \mathcal{G}(ham, (comap\ (\times 2)\ ham))), \\ & & \qquad (comerge_2^{\#}\ \mathcal{G}(ham, (comap\ (\times 3)\ ham)))) \\ & = & 1 + \min(\omega, \mathcal{G}(ham, comap\ (\times 2)\ ham), \mathcal{G}(ham, comap\ (\times 3)\ ham)) \end{array}$$

(The above follows since $comerge_1^{\#}$ and $comerge_2^{\#}$ both give 0 when applied to 0 and *ham* does not occur within the definition of *comerge* or any functions called through *comerge*.)

$$\mathcal{G}(ham, comap\ (\times 2)\ ham) = comap_2^{\#}\ \langle(\times 2)\rangle\, 0 = \mathbf{GFP}\ F^{\#}$$

where $F^{\#} = \lambda f.(\min((\times 2)_1^{\#}\ 0, f, 0))$. Now, $\mathbf{GFP}\ F^{\#} = 0$, since $(\times 2)_1^{\#}\ 0 = 0$, and so $\mathcal{G}(ham, comap\ (\times 2)\ ham) = 0$. Similarly, $\mathcal{G}(ham, comap\ (\times 3)\ ham) = 0$, and thus we obtain,

$$ham_0^{\#}\ 0 = 1 + \min(\omega, 0, 0) = 1$$

Therefore, *ham* is guarded.

$$
\begin{array}{|l|}
\hline
\mathit{fib} :: \mathit{Colist} \\
\mathit{fib} \stackrel{def}{=} 0\Diamond(1\Diamond(\mathit{cosuml}\ \mathit{fib}\ (\mathit{cotl}\ \mathit{fib}))) \\
\mathit{cosuml} :: \mathit{Colist} \longrightarrow \mathit{Colist} \longrightarrow \mathit{Colist} \\
\mathit{cosuml}\ x\ y \stackrel{def}{=} \mathit{zipWith}\ (+)\ x\ y \\
\mathit{zipWith} :: (\mathit{Int} \longrightarrow \mathit{Int}) \longrightarrow \mathit{Colist} \longrightarrow \mathit{Colist} \longrightarrow \mathit{Colist} \\
\mathit{zipWith}\ f\ (a\Diamond x)\ (b\Diamond y) \stackrel{def}{=} (f\ a\ b)\Diamond(\mathit{zipWith}\ f\ x\ y) \\
\hline
\end{array}
$$

Figure 2: Definition of the Fibonacci list function.

4.3 The Fibonacci List

In this section we analyse a function that produces the list of Fibonacci numbers. The function *fib*, given in Figure 2, calculates this list and is productive. However, the type inference method of Hughes, Pareto and Sabry will not accept this algorithm since:

> ...the system cannot prove that the application (*cotl fib*) will succeed. This is because the structure of the definition does not match the structure of the termination proof for *fib*.
>
> (Section 7.1 of [15])

(We have altered their notation slightly so that it is consistent with ours.) However, we shall show that our analysis detects *fib* as being guarded and therefore acceptable.

4.3.1 Analysis of *fib*.

To analyse *fib*, we first produce expressions for the auxiliary guardedness functions of *cosuml* and *zipWith*. We take the guardedness functions of + to be the identity.

$$
\begin{array}{rcl}
\mathit{cosuml}_1^{\#}\ 0 & = & \mathit{zipWith}_2^{\#}\ \langle(+)\rangle\ 0 \\
\mathit{zipWith}_2^{\#}\ \langle(+)\rangle\ 0 & = & \min((+)_1^{\#}\ 0, \mathit{zipWith}_2^{\#}\ \langle(+)\rangle\ 0) \\
 & = & \min(0, \mathit{zipWith}_2^{\#}\ \langle(+)\rangle\ 0)
\end{array}
$$

The greatest fixed point solution to the above is 0. Similarly, $\mathit{cosuml}_2^{\#}\ 0 = 0$.

It then follows that:

$$
\begin{array}{rcl}
\mathit{fib}_0^{\#}\ 0 & = & 2 + \min(\mathit{cosuml}_1^{\#}\ 0, \mathit{cosuml}_2^{\#}\ (\mathit{cotl}_1^{\#}\ 0)) \\
 & = & 2 + \min(0, -1 + 0) \\
 & = & 1
\end{array}
$$

Consequently, *fib* is guarded.

5 Proof of Soundness

It is necessary to show that any function that is detected as being guarded by our abstract interpretation will indeed be productive in the sense that it will be possible to obtain the normal form of any element of the structure within a finite time — the intuitive meaning of Definition 2.1.

Precisely, we are claiming that the following class of functions, those that are *entirely guarded*, are productive.

Definition 5.1 *If a (defining) expression, e, is guarded wrt a function name, f, of type A, and e contains only reducible constants apart from f then the expression e and its function f are called* ***entirely guarded****, written* $\mathbf{EG}(e, f, A)$.

The following result does indeed show that our analysis is *sound*.

Theorem 5.1 (Due to Coquand, 1993) *If we assume that all data terms are reducible then a codata function, f, will be productive for any set of inputs if it is entirely guarded.*

Structure of the Proof. Our proof of Theorem 5.1 proceeds as follows. We need to translate our source language into the formal language in which suspensions (using a Fix constructor) and unwinds are used to introduce and eliminate codata, respectively. This enables us to relate our formal definition of productivity (Definition 2.1) to the abstract interpretation that we have described. We then give a result (Lemma 5.2) which establishes the guardedness level that results from a substitution. As in the typed lambda calculus (see, for example, [10] for a proof by Girard of strong normalization which formed the model of our proof[6]), we need to prove productivity by induction over types by establishing a stronger criterion of *stability* (see Definition 5.3) for all guarded definitions. We also introduce the idea of *neutral* terms (as proposed by Girard) in order to simplify the structure of our proof. We show in Lemma 5.3 that all stable codata expressions are productive. We show in Lemmas 5.5, 5.6 and 5.7 stability results for products, abstractions and case expressions, respectively. In Lemma 5.9 we show that all substitution instances (see Definition 5.5) of entirely guarded definitions are stable. This then leads to Lemma 5.10 which states that all entirely guarded expressions are stable. This then allows us to prove Theorem 5.1.

Note that although our proof of correctness is on the assumption that we have a monomorphic language, we believe that our analysis is also applicable to systems with shallow polymorphism (i.e. the polymorphism of Hindley-Milner type inference). This belief is based on the fact that the analysis simply relies upon the program being well-typed rather than monomorphically typed.

5.1 Translation of the Source Language

The translation that we shall make is to treat suspension and its associated unwinding as a monad. We need to make a translation which, effectively, treats data in a call-by-value way and codata in a call-by-name way. The background to such a monadic translation has been given by Wadler in [32].

[6] It should be noted, however, that Girard's use of the term *reducibility* differs from ours. His idea of reducibility corresponds to our one of *stability* (which is also used in [26]). Our definition of reducibility comes from that of Coquand [2].

$$
\begin{aligned}
&mapS :: (t \longrightarrow u) \longrightarrow \uparrow t \longrightarrow \uparrow u \\
&mapS\, f\, l \stackrel{def}{=} \mathsf{Fix}\, l\, (\lambda s \rightarrow \lambda g \rightarrow (f(\downarrow s))) \\
&apBindS :: (t \longrightarrow \uparrow u) \longrightarrow \uparrow t \longrightarrow \uparrow u \\
&apBindS\, f\, l \stackrel{def}{=} \mathsf{Fix}\, l\, (\lambda s \rightarrow \lambda g \rightarrow \downarrow (f(\downarrow s))) \\
&compBindS :: \uparrow (t \longrightarrow u) \longrightarrow \uparrow t \longrightarrow \uparrow u \\
&compBindS\, f\, l \stackrel{def}{=} \mathsf{Fix}\, l\, (\lambda s \rightarrow \lambda g \rightarrow (\downarrow f)(\downarrow s)) \\
&compMapS :: \uparrow (t \longrightarrow u) \longrightarrow t \longrightarrow \uparrow u \\
&compMapS\, f\, l \stackrel{def}{=} \mathsf{Fix}\, l\, (\lambda s \rightarrow \lambda g \rightarrow (\downarrow f) s)
\end{aligned}
$$

Figure 3: The Suspend & Unwind Monadic Combinators.

1. If $F^* :: T \longrightarrow U$ then
   (a) If a^* is not suspended then $(Fa)^* = F^* a^*$
   (b) If a^* is suspended then $(Fa)^* = mapS\, F^* a^*$
2. If $F^* :: T \longrightarrow \uparrow U$ then
   (a) If a^* is not suspended then $(Fa)^* = F^* a^*$
   (b) If a^* is suspended then $(Fa)^* = apBindS\, F^* a^*$
3. If $F^* :: \uparrow (T \longrightarrow U)$ then
   (a) If a^* is not suspended then $(Fa)^* = compMapS\, F^* a^*$
   (b) If a^* is suspended then $(Fa)^* = compBindS\, F^* a^*$

Figure 4: The Translation of Source Applications, Fa.

Definition 5.2 *We make the following translation from* ESFP, *our source level programming language, to* $\mathrm{ESFP^{FC}}$, *in which codata is formalised by suspensions and unwinds.*

Types. *Suppose that we have codata type definitions of the form:-*

$$\mathsf{codata}\ T\, t_1 \ldots t_n \stackrel{def}{=} \ldots T \ldots$$

(Here $t_1 \ldots t_n$ are type variables.) Then each occurrence of T on the right-hand side of the type definition should be replaced by $\uparrow T$. If T occurs as the result type of a function then that result type becomes $\uparrow T$.

Expressions. *The change to expressions relates purely to applications. All applications in expressions and sub-expressions are translated using the monadic combinators in Fig. 3. The translation algorithm, to produce the translation of an application, Fa which we denote $(Fa)^*$ is given in Fig. 4. If the result of this translation has a suspended type but the original application did not have a suspended type as its result and the original application was not an argument to another application then we unwind the translated application i.e. we get $\downarrow (Fa)^*$.*

As should be expected, we have the following result.

Lemma 5.1 *The guardedness levels of corresponding functions and parameters are preserved by the translation described in Definition 5.2.*

Proof. By structural induction on the expressions of the language: the basic structure of functions has not changed in that we have only added monadic combinators in the place of simple application. □

For the sake of brevity, in the sequel we shall take the reduction of codata expressions to mean the unwinding of the corresponding monadic applications.

5.2 Guardedness Levels of Substitutions

We now show how the guardedness levels of substitutions relate to those of applications.

Lemma 5.2 *If fname* $x_1 \ldots x_n \stackrel{def}{=} E$ *and we have the application, fname* $a_1 \ldots a_n$ *where elements from a vector* $\boldsymbol{s}$ *of free variables may occur in the* a_i *then, if* $\boldsymbol{h}$ *is the vector of actual parameters to be substituted for* $\boldsymbol{s}$*,*

$$\mathcal{G}(f, E[\boldsymbol{a}/\boldsymbol{x}], \boldsymbol{h}) \geq \min(\mathcal{S}(f, \mathit{fname}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathit{fname}_i^{\#}\ \boldsymbol{b}\, g_i)$$

Here, $g_i = \mathcal{G}(f, a_i, \boldsymbol{h})$ *and* $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{s}]$.

Equivalently, we have:

$$\mathcal{G}(f, E[\boldsymbol{a}/\boldsymbol{x}], \boldsymbol{h}) \geq \min(\mathcal{G}(f, E, \boldsymbol{h}), \min_{i=1}^{i=n} g_i + \mathcal{G}(x_i, E, \boldsymbol{h}))$$

Proof. The proof is by structural induction over the forms of defining expressions: defining expressions must all be of finite length and have only a finite number of forms. For the sake of brevity, we shall use E' to denote $E[\boldsymbol{a}/\boldsymbol{x}]$ and, similarly, $\boldsymbol{a}'$ to denote $\langle a_1[\boldsymbol{a}/\boldsymbol{x}] \ldots a_p[\boldsymbol{a}/\boldsymbol{x}]\rangle$.

Base cases

1. For recursive occurrences, where no application is involved, both sides are equal to 0 unless g is $-\omega$ in which case the inequality holds.

2. For constants, the LHS is ω, as is the RHS unless g is $-\omega$ in which case the inequality holds.

3. If the variable is x_i then $E[\boldsymbol{a}/x_i] = a_i$. In addition,

   $$\mathcal{G}(x_i, x_i, \boldsymbol{h}) = 0$$

   and so the RHS becomes,

   $$\min_{i=1}^{i=n}(\mathcal{G}(f, x_i, \boldsymbol{h}), \mathcal{G}(f, a_i, \boldsymbol{h})) = \mathcal{G}(f, a_i, \boldsymbol{h})$$

   and thus the LHS equals the RHS.

Inductive cases

1. **Abstractions**. This case follows immediately by induction from the definition of the $\mathcal{G}$ operator.

2. **Constructor expressions**. We have:

$$\begin{array}{rl} \multicolumn{2}{l}{\mathcal{G}(f, E[\boldsymbol{a}/\boldsymbol{x}], \boldsymbol{h}) =} \\ & \mathcal{G}(f, C\, e_1[\boldsymbol{a}/\boldsymbol{x}] \ldots e_n[\boldsymbol{a}/\boldsymbol{x}], \boldsymbol{h}) \\ = & 1 + \min\limits_{j=1}^{j=n} \mathcal{G}(f, e_j[\boldsymbol{a}/\boldsymbol{x}], \boldsymbol{h}) \\ \geq & \{\text{By the induction hypothesis.}\} \\ & 1 + \min\limits_{j=1}^{j=n}(\min(\mathcal{G}(f, e_j, \boldsymbol{b}), \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, e_j, \boldsymbol{b}))) \\ = & \min(1 + \min\limits_{j=1}^{j=n} \mathcal{G}(f, e_j, \boldsymbol{b}), 1 + \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \min\limits_{j=1}^{j=n} \mathcal{G}(x_i, e_j, \boldsymbol{b})) \\ = & \min(\mathcal{G}(f, C\, e_1 \ldots e_n, \boldsymbol{b}), \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, C\, e_1 \ldots e_n, \boldsymbol{b})) \end{array}$$

3. case **expressions**. For the sake of brevity, we shall denote the original and substituted expressions as follows:

$$\begin{array}{rcl} CE & \stackrel{def}{=} & \mathsf{case}\ S\ \mathsf{of}\ \langle p_1, e_1\rangle, \ldots \langle p_n, e_n\rangle \\ CE' & \stackrel{def}{=} & \mathsf{case}\ S'\ \mathsf{of}\ \langle p_1, e'_1\rangle, \ldots \langle p_n, e'_n\rangle \end{array}$$

Now,

$$\mathcal{G}(f, CE', \boldsymbol{h}) = \min(\min_{j=1}^{j=n} \min(\mathcal{G}(f, e'_j, \boldsymbol{h}), \mathcal{P}\,(p_j, e'_j)\,\boldsymbol{h}\,g'), g')$$

Here, $g' = \mathcal{G}(f, S', \boldsymbol{h})$ and we have that,

$$\mathcal{P}\,(p_j, e'_j)\,\boldsymbol{h}\,0 = \min_{k=1}^{k=N(i)} (\mathcal{G}(v_j^k, e'_j, \boldsymbol{h}) - \mathcal{D}(v_j^k, p_j))$$

Now, $\mathcal{D}(v_j^k, p_j)$ is the same for both e_j and e'_j. Moreover, since v_j^k does not occur in a_k, it follows that,

$$\mathcal{P}\,(p_j, e'_j)\,\boldsymbol{h}\,0 = \mathcal{P}\,(p_j, e_j)\,\boldsymbol{h}\,0$$

By the induction hypothesis, we have,

$$\begin{array}{rcl} \mathcal{G}(f, e'_j, \boldsymbol{h}) & \geq & \min(\mathcal{G}(f, e_j, \boldsymbol{b}), \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, e_j, \boldsymbol{b})) \\ g' & \geq & \min(\mathcal{G}(f, S, \boldsymbol{b}), \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, S, \boldsymbol{b})) \end{array}$$

By using the associativity and commutativity of min and that + distributes over min, we have,

$$\begin{array}{l} \mathcal{G}(f, CE', \boldsymbol{h}) \geq \\ \quad \min(\min\limits_{j=1}^{j=n}(\min(\min(\mathcal{G}(f, e_j, \boldsymbol{b}), \mathcal{P}\,(p_j, e_j)\,\boldsymbol{b}\,\mathcal{G}(f, S, \boldsymbol{b})), \\ \qquad\quad \min\limits_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \min(\mathcal{G}(x_i, e_j, \boldsymbol{b}), \mathcal{P}\,(p_j, e_j)\,\boldsymbol{b}\,\mathcal{G}(x_i, S, \boldsymbol{b})))), g') \end{array}$$

We can apply the associativity, commutativity and distributivity properties again to obtain the required result. That is,

$$\mathcal{G}(f, CE', \boldsymbol{h}) \geq \min(\mathcal{G}(f, CE, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, CE, \boldsymbol{b}))$$

4. **Applications**. To prove the statement for applications we shall also use induction on the number of applications in the expression.

   Firstly, we make the following definitions, where $\boldsymbol{c}$ is the actual parameter list in the application in the expression E:

$$\begin{aligned} \boldsymbol{c}' &= \boldsymbol{c}[\boldsymbol{a}/\boldsymbol{x}] \\ \boldsymbol{d} &= \boldsymbol{c}'[\boldsymbol{h}/\boldsymbol{s}] = \boldsymbol{c}[\boldsymbol{a}/\boldsymbol{x}][\boldsymbol{h}/\boldsymbol{s}] = \boldsymbol{c}[\boldsymbol{b}/\boldsymbol{x}] \end{aligned}$$

   The following are our base cases in our induction over the number of applications:

   (a)

$$\begin{aligned} &\mathcal{F}(f, f, j, \boldsymbol{c}', \boldsymbol{h}) = \\ &\quad \min(0, f_j^{\#}\, \boldsymbol{d}\, \mathcal{G}(f, c_j', \boldsymbol{h})) \\ &\geq \{\text{By the structural IH for } c_j'.\} \\ &\quad \min(0, \min(f_j^{\#}\, \boldsymbol{d}\, \min(\mathcal{G}(f, c_j, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, c_j, \boldsymbol{b}))) \\ &= \{\text{By continuity.}\} \\ &\quad \min(0, \min(f_j^{\#}\, \boldsymbol{d}\, \mathcal{G}(f, c_j, \boldsymbol{b}), f_j^{\#}\, \boldsymbol{d} \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, c_j, \boldsymbol{b})))) \\ &= \{\text{By associativity of min and continuity.}\} \\ &\quad \min(\min(0, f_j^{\#}\, \boldsymbol{d}\, \mathcal{G}(f, c_j, \boldsymbol{b})), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + f_j^{\#}\, \boldsymbol{d}\, \mathcal{G}(x_i, c_j, \boldsymbol{b})))) \\ &= \min(\mathcal{F}(f, f, j, \boldsymbol{c}, \boldsymbol{b}), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{F}(x_i, f, j, \boldsymbol{c}, \boldsymbol{b}))) \end{aligned}$$

   Consequently, we have that:

$$\min_{j=1}^{j=p} \mathcal{F}(f, f, j, \boldsymbol{c}', \boldsymbol{h}) \geq \min(\mathcal{G}(f, E, \boldsymbol{b}), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, E, \boldsymbol{b})))$$

   (b)

$$\mathcal{F}(x_l, x_l, j, \boldsymbol{c}', \boldsymbol{h}) = \min(0, \mathcal{F}(x_l, a_l, j, \boldsymbol{c}', \boldsymbol{h}))$$

   Since no free variables may occur in a_l, the result will follow by induction if and only if it holds for all other expressions that may replace a_l.

   (c)

$$\mathcal{F}(m_x, m_x, j, \boldsymbol{c}', \boldsymbol{h}) = \min(0, nom^{\#}\, \mathcal{G}(m_x, \boldsymbol{c}', \boldsymbol{h}))$$

   This case follows similarly to that for (4a) above: we use the continuity of the guardedness function $nom^{\#}$ in conjunction with the associativity of min.

(d) For $\mathcal{F}(f, fn, j, \boldsymbol{c}', \boldsymbol{h})$, if $j > \mathbf{Arity}(fn)$ then the result holds trivially since by definition the guardedness level must be $-\omega$. If $j \leq \mathbf{Arity}(fn)$ then

$$\begin{aligned}
&\mathcal{F}(f, fn, j, \boldsymbol{c}', \boldsymbol{h}) = \\
&\quad \min(\mathcal{S}(f, fn, \boldsymbol{d}), \mathcal{N}(f, fn, j, \boldsymbol{c}', \boldsymbol{h})) \\
&= \quad \min(\mathcal{S}(f, fn, \boldsymbol{d}), fn_j^{\#}\ \boldsymbol{d}\, \mathcal{G}(f, c_j', \boldsymbol{h})) \\
&\geq \quad \{\text{By the structural IH.}\} \\
&\quad \min(\mathcal{S}(f, fn, \boldsymbol{b}), fn_j^{\#}\ \boldsymbol{d}\ \min(\mathcal{G}(f, c_j, \boldsymbol{b}), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, c_j, \boldsymbol{b})))) \\
&= \quad \{\text{By continuity and the associativity of min.}\} \\
&\quad \min(\min(\mathcal{S}(f, fn, \boldsymbol{b}), \mathcal{N}(f, fn, j, \boldsymbol{c}, \boldsymbol{b})), \\
&\qquad \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{N}(x_i, fn, j, \boldsymbol{c}, \boldsymbol{b})))
\end{aligned}$$

Thus we have that,

$$\min_{j=1}^{j=p} \mathcal{F}(f, fn, j, \boldsymbol{c}', \boldsymbol{h}) \geq \min(\mathcal{G}(f, E, \boldsymbol{b}), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{G}(x_i, E, \boldsymbol{b})))$$

(Note that the above argument is valid also for recursive calls of *fname*.)

(e) $\mathcal{F}(f, x_j, i, \boldsymbol{c}', \boldsymbol{h}) = \mathcal{F}(f, h_j, i, \boldsymbol{c}', \boldsymbol{h})$ and so the result follows by induction if we assume that it is true for h_j.

(f) $\mathcal{F}(f, m_k, i, \boldsymbol{c}', \boldsymbol{h}) = nom^{\#}\, \mathcal{G}(f, c_i', \boldsymbol{h})$ and so the result follows as in (4c) above.

The inductive case, for more than one application, is as follows.

$$\mathcal{F}(f, F'\, b', j, \boldsymbol{c}', \boldsymbol{h}) = \min(g', g'')$$

Here, $g' = \mathcal{F}(f, F', j, \boldsymbol{c}', \boldsymbol{h})$ and $g'' = \mathcal{F}(f, F', j+1, \boldsymbol{c}', \boldsymbol{h})$ where, if $\boldsymbol{c}' = \langle c_1' \ldots c_m' \rangle$, then $\boldsymbol{e}' = \langle b', c_1' \ldots c_m' \rangle$.

Since both g' and g'' operate over F', the induction hypothesis applies in both cases as the number of applications in the expression has been reduced by one. We thus have:

$$\begin{aligned}
g' &= \min(\mathcal{F}(f, F, j, \boldsymbol{c}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{F}(x_i, F, j, \boldsymbol{c}, \boldsymbol{b})) \\
g'' &= \min(\mathcal{F}(f, F, i+1, \boldsymbol{c}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{F}(x_i, F, j+1, \boldsymbol{c}, \boldsymbol{b}))
\end{aligned}$$

It follows from the associativity of min and the distributivity of + over min that,

$$\begin{aligned}
&\mathcal{F}(f, F'\, b', j, \boldsymbol{c}', \boldsymbol{h}) \geq \\
&\quad \min(\min(\mathcal{F}(f, F, j, \boldsymbol{c}, \boldsymbol{b}), \mathcal{F}(f, F, j+1, \boldsymbol{c}, \boldsymbol{b})), \\
&\qquad \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \min(\mathcal{F}(x_i, F, j, \boldsymbol{c}, \boldsymbol{b}), \mathcal{F}(x_i, F, j+1, \boldsymbol{c}, \boldsymbol{b})))) \\
&= \quad \min(\mathcal{F}(f, F\, b, j, \boldsymbol{c}, \boldsymbol{b}), \min_{i=1}^{i=n}(\mathcal{G}(f, a_i, \boldsymbol{h}) + \mathcal{F}(x_k, F\, b, j, \boldsymbol{c}, \boldsymbol{b})))
\end{aligned}$$

□

5.3 Stability

We now introduce the idea of *stability* (using the terminology of [26] for the typed lambda calculus). Our base types for our system include the integers, *Int*, and all finite types including the booleans, *Bool*, and characters, *Char*.

Definition 5.3 *An expression e of type A is* ***stable****, written* $\mathbf{St}(e, A)$

- *If e is of base type and e is reducible (see Definition 2.2) then* $\mathbf{St}(e, A)$*. (Note that this means in the case of base types that the expression must be strongly normalising.)*
- *If e is a sum of products type i.e.* $A \equiv \sum_{i=1}^{i=m}(\prod_{j=1}^{j=n} A_{i,j})$ *then* $\mathbf{St}(e, A)$ *if and only if* $\exists i\,.\,\forall j\,.\,\mathbf{St}(\pi_{i,j}\, e, A)$ *where $\pi_{i,j}$ is the relevant projection function.*
- *If e is of a functional type i.e.* $A \equiv B \longrightarrow C$ *then* $\mathbf{St}(e, A)$ *iff* $\forall b.\mathbf{St}(b, B) \Rightarrow \mathbf{St}(eb, C)$*.*

In order to make the structure of the proof clearer, we introduce the idea of *neutrality.*

Definition 5.4 *An expression, e, is* ***neutral*** *if and only if it is a variable, a pre-defined constant or an application i.e. it is neither a constructor expression (i.e. e is of the form $C\, e_1 \ldots e_n$) nor is it an abstraction (i.e. e is of the form $\lambda x.E$) and nor is it a* `case` *expression.*

We now show that stable codata expressions are productive.

Lemma 5.3 *Where A is a codata type we have:*

1. *If* $\mathbf{St}(a, A)$ *then* $\mathbf{Pr}(a)$
2. *If* $\mathbf{St}(a, A)$ *and* $a \twoheadrightarrow a'$ *then* $\mathbf{St}(a', A)$
3. *If* $\mathbf{Ne}(a)$ *and if* $\forall t_i.(a \rightarrow t_i) \wedge \mathbf{St}(t_i, T) \Rightarrow \mathbf{St}(t_i, T)$ *then* $\mathbf{St}(a, T)$

Proof. The proof for all clauses is by simultaneous induction over the type A.

Base Types. We do not have to examine base types since they are all in the data class and we assume that all data is strongly normalising.

Sum of Products.

1. Suppose that a is stable of type $\sum_{i=1}^{i=m}(\prod_{j=1}^{j=n} A_{i,j})$. Then, for some i and any j, $\pi_{i,j}\, a$ is stable (by definition) and by the induction hypothesis (IH) for $A_{i,j}$ is reducible. Now, $\pi_{i,j}\, a \twoheadrightarrow \pi_{i,j}\, a' \twoheadrightarrow e_{i,j}$ where $e_{i,j}$ is reducible and a' is of the form, $C_i\, e_{i,1} \ldots e_{i,m}$. Consequently, a is reducible.
2. If $a \twoheadrightarrow a'$ then for some i and any j, $\pi_{i,j}\, a \twoheadrightarrow \pi_{i,j}\, a'$. a is stable and so is each $\pi_{i,j}\, a'$ by definition. Thus, by the IH for $A_{i,j}$, $\pi_{i,j}\, a'$ is stable and, consequently, a' is stable.
3. Suppose that a is neutral and that $\mathbf{St}(a', A)$ where $a \rightarrow a'$. By the assumption of neutrality, (i.e. a is not of the form, $C_i\, e_{i,1} \ldots e_{i,n}$),

$$\pi_{i,j}\, a \twoheadrightarrow \pi_{i,j}\, a'$$

for some i. Now, by definition, since a' is stable so is $\pi_{i,j}\, a'$. In addition, $\pi_{i,j}\, a$ is neutral and so, by the IH, $\pi_{i,j}\, a$ is stable. Thus a is stable.

Function Spaces.

1. If $\mathbf{St}(a, B \longrightarrow C)$, let x be a variable of type B. By the IH, part 3, for A, x is stable. Hence, ax is stable by definition. Now, the IH, part 1, for C guarantees that ax is productive. However, ax is η-equivalent to a when abstracting over x and thus a must be productive.
2. If $a \twoheadrightarrow a'$ and $\mathbf{St}(a', B \longrightarrow C)$, take b such that $\mathbf{St}(b, B)$. Then ab is stable and $ab \twoheadrightarrow a'b$. By the IH for C, $a'b$ is stable and thus a' is stable.
3. Suppose that a of type $B \longrightarrow C$ is neutral and that if $a \to a'$ then a' is stable. Let b be a stable expression of type B. By the IH, part 1, for B, b is reducible. We now argue by induction on the size of the reduction path of b that the neutral expression ab reduces in one step into stable terms only.

   In one step, ab converts to one of the following. (There are no other possibilities since a is assumed to be neutral.)

   - $a'b$ with a' one step from a. As a' is stable and b is, $a'b$ is stable.
   - ab' with b' one step from b. b' is stable by IH, part 2, for B. Since the reduction path for b' is of smaller size (and we can only have a finite number of unwindings) than that of b, we have by induction that ab' is stable.

   Thus the IH, part 3, allows us to conclude that ab is stable and so a is stable.

□

The following can be proved by induction on the number of reduction steps.

Lemma 5.4 *Suppose that e is neutral and that $e \twoheadrightarrow e'$ and e' is stable. If every intermediate expression (i.e. those apart from e and e') on any reduction path from e to e' is neutral then e is stable.*

We now show in the following three lemmas how stability is propagated through non-neutral terms.

Lemma 5.5 *If,*

$$\exists i \,.\, \forall j \,.\, \mathbf{St}(e_{i,j}, A_{i,j})$$

then

$$\exists i \,.\, \mathbf{St}(C_i\, e_{i,1} \ldots e_{i,m}, A)$$

Proof. Since for some i and any j, $\mathbf{St}(e_{i,j}, A_{i,j})$, $e_{i,j}$ is reducible by Lemma 5.3, part 1, we can argue by induction on the sum of the sizes of the reduction paths of the $e_{i,j}$ that $\pi_{i,j}\,(C_i e_{i,1} \ldots e_{i,n})$ is stable. This converts to one of the following:-

- $e_{i,j}$ which is stable.
- $\pi_{i,j}\,(C_i e_{i,1} \ldots e'_{i,k} \ldots e_{i,n})$ — here $e'_{i,k}$ is one step from $e_{i,k}$. By Lemma 5.3, part 2, $e'_{i,k}$ is stable and as $e'_{i,k}$ has a shorter reduction path than $e_{i,k}$, it follows by induction that the resulting expression is also stable.

Thus $\pi_{i,j}\, a$ converts in one step to stable terms only and so by Lemma 5.3, part 3, it is stable. Hence a is stable. □

Lemma 5.6 *If for all stable b of type B, $c[b/x]$ is stable then so is $\lambda x.c$.*

Proof. We need to show that $(\lambda x.c)b$ is reducible for all reducible b. We reason by induction on the sum of the sizes of the reduction paths of c and b.

$(\lambda x.c)b$ converts in one step to one of the following

- $c[b/x]$ which is stable by assumption.
- $(\lambda x.c')b$ with c' one step from c. Thus c' is stable by Lemma 5.3, part 2, as c itself must be stable by the assumption (which includes null substitutions). Thus by induction on the size of the reduction paths, $(\lambda x.c')b$ must also be reducible.
- $(\lambda x.c)b'$ with b' one step from b. This follows similarly to the above case.

So $(\lambda x.c)b$ (which is neutral) converts in one step to a stable expression and so $(\lambda x.c)b$ and thus $(\lambda x.c)$ are stable by Lemma 5.3, part 3. □

Lemma 5.7 *If S is stable and each $e_i[\pi_{i,j}\, S/v_i^j]$ is stable then*

$$\mathsf{case}\ S\ \mathsf{of}\langle p_1, e_1\rangle \ldots \langle p_1, e_n\rangle$$

is stable.

Proof. We reason by induction on the sum of the sizes of the reduction paths of S and all the e_i. The expression

$$\mathsf{case}\ S\ \mathsf{of}\langle p_1, e_1\rangle \ldots \langle p_1, e_n\rangle$$

converts in one step to one of the following.

- $$\mathsf{case}\ S'\ \mathsf{of}\langle p_1, e_1\rangle \ldots \langle p_1, e_n\rangle$$

  with S' one step from S. By Lemma 5.3, part 2, S' is stable and, as the sum of the sizes of the reduction paths has decreased we have by the IH that the whole expression is stable.
- $$e_i[\pi_{i,j}\, S/v_i^j]$$

  This is stable by assumption.

□

We then have the following one-step conversion lemma.

Lemma 5.8 *If $e[\boldsymbol{h}/\boldsymbol{s}] \to e'[\boldsymbol{h}/\boldsymbol{s}]$ where none of the s_i is bound in e, each h_i is stable and $e'[\boldsymbol{h}/\boldsymbol{s}]$ must be stable then $e[\boldsymbol{h}/\boldsymbol{s}]$ is stable.*

Proof. We give a sketch of the proof which is very similar to those of Lemmas 5.5, 5.6 and 5.7. Lemma 5.3, part 3 means that we only have to deal, by structural induction, with the cases of non-neutral terms.

- e is $\lambda x.E$. Then the relevant case is where

  $$e'[\boldsymbol{h}/\boldsymbol{s}] = \lambda x.E'[\boldsymbol{h}/\boldsymbol{s}]$$

  Now, for any stable term a of the correct type,

  $$E'[\boldsymbol{h}/\boldsymbol{s}][a/x]$$

  is stable. In addition,

  $$(\lambda x.E[\boldsymbol{h}/\boldsymbol{s}])\, a = E[a/x, \boldsymbol{h}/\boldsymbol{s}]$$

  We can then argue, as in Lemma 5.6, by induction on the sum of the sizes of the reduction paths of E and a to show that $E[a/x, \boldsymbol{h}/\boldsymbol{s}]$ is stable. It then follows from Lemma 5.6 that e is stable.

- e is $C_i e_{i,1} \ldots e_{i,p}$. The argument is similar to that of Lemma 5.5.

- e is a case expression. The argument is similar to that of Lemma 5.7.

□

5.4 Stability of Entirely Guarded Expressions

We have the following definition that names the possible expressions that may arise by substitution of stable expressions for variables.

Definition 5.5 *An **s-instance** e' of an expression e is a substitution instance,*

$$e' \equiv e[g_1/x_1, \ldots, g_r/x_r]$$

where each g_i is stable.

The crucial lemma is then as follows:

Lemma 5.9 *If e is an entirely guarded (for some function f) expression then all s-instances, e', of expression, e, are stable.*

Proof. The proof is by structural induction over the forms of defining expressions: defining expressions must all be of finite length and have only a finite number of forms. We label the vector of substituting expressions $\boldsymbol{h}$ and the vector of parameters that they are replacing, $\boldsymbol{s}$.

Base cases

1. **Constants**. No substitutions are possible within a constant c and c must be guarded. Now c is neutral and, by assumption, productive and therefore by Lemma 5.3, part 3 it is stable.

2. **Variables**. By assumption, each input, h_i that will replace the variable is stable and so the result follows.

3. **Single Recursive Occurrences**. By the assumption of guardedness, these cannot occur at the top level of the entirely guarded expression.

Inductive cases

1. **Abstractions**. A lambda abstraction, of the form $\lambda y.E$, of type $A \longrightarrow B$ is guarded *iff* E is guarded. By the induction hypothesis (IH), therefore, $E[\boldsymbol{h}/\boldsymbol{s}]$ is stable for any stable a of type A. It follows by Lemma 5.6 that $\lambda y.E[\boldsymbol{h}/\boldsymbol{s}]$ is stable.

2. **Constructor expressions**.

$$(C\, e_1 \ldots e_p)[\boldsymbol{h}/\boldsymbol{s}] = C\, b_1 \ldots b_p$$

   Where $b_i = e_i[\boldsymbol{h}/\boldsymbol{s}]$.

   If $f_0^{\#}\, \boldsymbol{h}\, 0 \geq 2$ then $\mathcal{G}(f, e_i, \boldsymbol{h}) \geq 1$ for each i and therefore the result follows immediately by induction and Lemma 5.5.

   Suppose then that $f_0^{\#}\, \boldsymbol{h}\, 0 = 1$. By the definition of unwinding, each occurrence of f in F is replaced by f^u in the reduct where

$$f^u = \lambda \boldsymbol{s}.C\, e_1 \ldots e_p$$

   Then we can observe that the guardedness level of f within $Cb_1[f^u/f] \ldots b_p[f^u/f]$ must be 2 since each occurrence of f is guarded by an additional constructor. (This comes from Lemma 5.2.) In addition, the guardedness level of f within each $b_i[f^u/f]$ must be at least 1. Hence, each $b_i[f^u/f]$ is stable by the IH. Now, since we do not have any substitutions for bound variables in b_i or $b_i[f^u/f]$ and

$$b_i \rightarrow b_i[f^u/f]$$

   it follows from Lemma 5.8 that each b_i is stable. Consequently, it follows by Lemma 5.5 that $C\, b_1 \ldots b_p$ must be a stable expression.

3. **Function applications**. First we observe that, due to the definition of guardedness, corecursive applications of the form,

$$f\, a_1 \ldots a_n$$

   where $n \geq 0$ cannot occur at the top level in our definitions. Such terms can only be detected as being guarded if they occur within a constructor expression, as covered in case (2) above.

   We argue over the remainder of the possible forms by induction on the number of applications involved.

   Named function application. Here we examine the case of the application of a named function. We assume that we have an application of the form:

$$\mathit{fname}\, a_1 \ldots a_n$$

   where $\mathit{fname}\, x_1 \ldots x_n \stackrel{def}{=} E$. In what follows, $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{s}]$ where $\boldsymbol{a}$ is the vector of actual parameters to *fname*.
   Since the application is guarded, then in the case where $n \leq \mathbf{Arity}(\mathit{fname})$

$$\begin{array}{rcl} 0 & < & \mathcal{G}(f, \mathit{fname}\, a_1 \ldots a_n, \boldsymbol{h}) \\ & = & \{\text{By Lemma 3.2.}\} \end{array}$$

$$
\begin{aligned}
& \min(\mathcal{S}(f, \mathit{fname}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{N}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h})) \\
\leq\ & \{\text{By Lemma 5.2.}\} \\
& \mathcal{G}(f, E[a_1/x_1 \ldots a_n/x_n], \boldsymbol{h})
\end{aligned}
$$

Now, suppose that we replace the original application, $\mathit{fname} a_1 \ldots a_n$, with $\mathit{fname}'\, s_1 \ldots s_p$ where $s_1 \ldots s_p$ are the bound variables of F and

$$\mathit{fname}'\, s_1 \ldots s_p \stackrel{def}{=} E[a_1/x_1 \ldots a_n/x_n]$$

Then, since f does not occur in any s_i,

$$\mathcal{G}(f, \mathit{fname}'\, s_1 \ldots s_p, \boldsymbol{h}) = \mathcal{G}(f, E[a_1/x_1 \ldots a_n/x_n], \boldsymbol{h})$$

Therefore, by the IH over fname', fname' is stable. In addition, each s_i is stable and thus by definition, $\mathit{fname}'\, s_1 \ldots s_p$ is stable. Furthermore,

$$(\mathit{fname}'\, s_1 \ldots s_p)[\boldsymbol{h}/\boldsymbol{s}] \twoheadrightarrow E[a_1/x_1 \ldots a_n/x_n][\boldsymbol{h}/\boldsymbol{s}]$$

and so $E[a_1/x_1 \ldots a_n/x_n][\boldsymbol{h}/\boldsymbol{s}]$ is stable by Lemma 5.3. In addition,

$$(\mathit{fname}\ a_1 \ldots a_n)[\boldsymbol{h}/\boldsymbol{s}] \twoheadrightarrow E[a_1/x_1 \ldots a_n/x_n][\boldsymbol{h}/\boldsymbol{s}]$$

and any reduction path involves only neutral expressions. It then follows by Lemma 5.4 that any s-instance of the application $\mathit{fname}\, a_1 \ldots a_n$ must be stable.

Now, if $n > m$, where $m = \mathbf{Arity}(\mathit{fname})$, then, since the application is guarded, for all $1 \leq i \leq n - m$,

$$\mathit{nom}^{\#}\, \mathcal{G}(f, a_{m+i}, \boldsymbol{h}) = \omega$$

Thus, $\mathcal{G}(f, a_{m+i}, \boldsymbol{h}) = \omega$ and it follows that for any G, where $G\, x \stackrel{def}{=} (\mathit{fname}\, b_1 \ldots b_m\, b_{m+1} \ldots b_{m+i})\, x$, with $\boldsymbol{b}$ as above, $G_1^{\#}$ must produce ω on this input too. It then follows from Lemma 5.2 and induction that $G\, a_{m+i}$ is stable and so $\mathit{fname}\, a_1 \ldots a_n$ is stable.

Variable applications. In addition, in the case of the application of ordinary variables, the application is stable *iff* guardedness implies stability in all other cases. This is due to the fact that a component of the vector, $\boldsymbol{h}$, is substituted for the variable. Furthermore, no such variables may occur within $\boldsymbol{h}$ as $\boldsymbol{h}$ may only be constructed by an application of the $\mathcal{S}$ operator. In that case, the vector used, $\boldsymbol{b}$, makes substitutions for each variable in the argument vector, $\boldsymbol{a}$. (In other words, variable applications will reduce to one of the other cases.)

Pattern matching variable application. The definition of guardedness means that applications involving pattern matching variables cannot be detected as being guarded, where the guardedness level of the parameter is not ω, since the resulting guardedness level is $-\omega$.

The case where the guardedness level of the parameter is ω is similar to that for named function applications where the number of actual parameters is greater than the arity of the named function. Here, since for any i, $nom^{\#}\ \mathcal{G}(f, a_i, \boldsymbol{h}) = \omega$ then for any function f_m that it is matched with the variable, m, the application $f_m\, a_1 \ldots a_k$ must be stable since the guardedness functions in each case must return ω. Moreover, f_m must exist since for the application of a pattern matching variable to be guarded it must be enclosed within a case expression which itself must be guarded. This in turn ensures that the switch of the case must be guarded and so the switch expression must be stable and therefore reducible. The typing constraints (as we have assumed that we are working within a Hindley-Milner type system) then ensure that the pattern must be matched.

4. **Case expressions**. We assume here, without loss of generality, that each case expression only contains one level of destruction: to obtain substructures requires applications of case expressions upon pattern-matching variables.

   It follows directly from the assumption of guardedness that the switch S must be guarded for f (with respect to $\boldsymbol{h}$), where S is the switch expression in

   $$\mathsf{case}\ S\ \mathsf{of} \langle p_1, e_1 \rangle \ldots \langle p_n, e_n \rangle$$

   and, therefore, by the induction hypothesis, $S[\boldsymbol{h}/\boldsymbol{s}]$ must be stable and therefore reducible and so that it should reduce to some pattern p_j.

   It therefore only remains to show that each $e_i[\pi_{i,j}\, S/v_i^j]$ is guarded for f with $\boldsymbol{h}$ as inputs since it will then follow from our inductive assumption that each $e_i[\pi_{i,j}\, S/v_i^j]$ is stable. Let us substitute $\pi_{i,j}\, S$for v_i^j in e_i, Then, by Lemma 5.2 we get,

   $$\begin{array}{l} \min\limits_{j=1}^{j=N(i)} \mathcal{G}(f, e_i[\pi_{i,j}\, S/v_i^j], \boldsymbol{h}) \geq \\ \qquad \min\limits_{j=1}^{j=N(i)} \min(\mathcal{G}(f, e_i, \boldsymbol{h}), \mathcal{G}(f, \pi_{i,j}\, S, \boldsymbol{h}) + \mathcal{G}(v_i^j, e_i, \boldsymbol{h})) \end{array}$$

   Here, $N(i)$ is the number of variables in the pattern p_j. Now, the auxiliary guardedness function of $\pi_{i,j}$ produces -1 on an input of 0 and so $\mathcal{G}(f, \pi_{i,j}\, S, \boldsymbol{h})$ is equal to $\mathcal{G}(f, S, \boldsymbol{h}) - 1$. Then,

   $$\begin{array}{rcl} \mathcal{G}(f, \pi_{i,j}\, S, \boldsymbol{h}) + \mathcal{G}(v_i^j, e_i, \boldsymbol{h}) & = & \mathcal{G}(f, S, \boldsymbol{h}) + \mathcal{G}(v_i^j, e_i, \boldsymbol{h}) - 1 \\ & = & \mathcal{P}\,(p_j, e_i)\, \boldsymbol{h}\, \mathcal{G}(f, S, \boldsymbol{h}) \\ & > & 0 \end{array}$$

   (The last inequality follows by the assumption of guardedness.) Since $\mathcal{G}(f, e_i, \boldsymbol{h})$ must also be greater than 0 it follows that f must be guarded in each $e_i[\pi_{i,j}\, S/v_i^j]$. It follows that each $e_i[\pi_{i,j}\, S/v_i^j]$ is stable and therefore the entire case expression is stable by Lemma 5.7.

□

5.5 Proofs of the Main Results

The above result allows us a straightforward proof of the following.

Lemma 5.10 *All entirely guarded expressions, defining some function f, are stable.*

Proof. Each expression, e, of type A, is the trivial s-instance of itself and so by Lemma 5.9, $\mathbf{EG}(e, f, A) \Rightarrow \mathbf{St}(e, A)$. □

Thus we are able to prove our main theorem of soundness, Theorem 5.1.
Proof. By Lemma 5.10 all entirely guarded defining expressions are stable and by Lemma 5.3 all stable codata expressions are productive. □

Theorem 5.1 is a safety criterion for our abstraction interpretation. This means that a function can be seen to have a guardedness level greater than or equal to that given by the analysis. To formalise this we make a mapping from expressions into $\mathbf{A}$ as follows:

Definition 5.6

$$\mathbf{abs}\,(f\ a_1 \ldots a_n) \stackrel{def}{=} \begin{cases} \omega & \textit{if } f \textit{ is productive} \\ 0 & \textit{otherwise} \end{cases}$$

We consequently have the following corollary to Theorem 5.1.

Corollary 5.1 *The following holds for any function, f, for arbitrary well-formed and well-typed, reducible actual parameters, $\boldsymbol{a}$. We assume that all data terms are normalizable:*

$$\mathbf{abs}\,(f\ a_1 \ldots a_n) \geq_{\mathbf{A}} f_0^{\#}\ \boldsymbol{a}\, 0$$

6 Properties of Guardedness Analysis

6.1 Completeness of the Analysis

We can formalise Coquand's definition of guardedness [2] over our abstract domain, $\mathbf{A}$, and this gives the $\mathcal{G}^C$ operator shown in Table 6.1. The $\mathcal{S}^C$ operator is the counterpart to the $\mathcal{S}$ operator and is defined in a parallel way. Note that it is implicit in Coquand's definition of guardedness that a function is guarded *iff* it is guarded across all function definitions. (This is the check made in the Coq system [1, 9].)

Our system for detecting productivity is stronger than Coquand's, due to the Hamming function example and the following completeness result.

Theorem 6.1 (Completeness) *For corresponding expressions in* ESFP *and Coquand's type theory [2],*

$$\mathcal{G}^C(f, E) \leq \mathcal{G}(f, E, \boldsymbol{h})$$

where $\boldsymbol{h}$ is any set of well-formed expressions not containing free variables and where each h_i will have the same type as the corresponding free variable x_i in E.

Proof. Our proof is by structural induction over expressions.

Base cases

$$\begin{aligned} \mathcal{G}^C(f, f) = &\quad 0 &= \mathcal{G}(f, f, \boldsymbol{h}) \\ \mathcal{G}^C(f, x) = &\quad \omega &= \mathcal{G}(f, x, \boldsymbol{h}) \\ \mathcal{G}^C(f, c) = &\quad \omega &= \mathcal{G}(f, c, \boldsymbol{h}) \end{aligned}$$

$$\mathcal{G}^C(f, f) \stackrel{def}{=} 0 \quad (18)$$

$$\mathcal{G}^C(f, c) \stackrel{def}{=} \omega \quad (19)$$

$$\mathcal{G}^C(f, x) \stackrel{def}{=} \omega \quad (20)$$

$$\mathcal{G}^C(f, \lambda x.E) \stackrel{def}{=} \mathcal{G}^C(f, E) \quad (21)$$

$$\mathcal{G}^C(f, C\, a_1 \ldots a_n) \stackrel{def}{=} 1 + \min_{i=1}^{i=n} \mathcal{G}^C(f, a_i) \quad (22)$$

$$\mathcal{G}^C(f, (\mathsf{case}\ s\ \mathsf{of}\ \langle p_1, e_1\rangle \ldots \langle p_n, e_n\rangle)) \stackrel{def}{=} \min(\min_{i=1}^{i=n} \mathcal{G}^C(f, e_i), nom^{\#}\, \mathcal{G}^C(f, s)) \quad (23)$$

$$\mathcal{G}^C(f, f\, a_1 \ldots a_n) \stackrel{def}{=} \min(0, \min_{i=1}^{i=n} nom^{\#}\, \mathcal{G}^C(f, a_i)) \quad (24)$$

$$\mathcal{G}^C(f, \mathit{fname}\, a_1 \ldots a_n) \stackrel{def}{=} \min(\mathcal{S}^C(f, \mathit{fname}), \min_{i=1}^{i=n} nom^{\#}\, \mathcal{G}^C(f, a_i)) \quad (25)$$

$$\mathcal{G}^C(f, x_i\, a_1 \ldots a_n) \stackrel{def}{=} \min(\min_{F \in \mathbf{FnNames}\setminus\{f\}} (\mathcal{S}^C(f, F)), \min_{i=1}^{i=n} \mathcal{G}^C(f, a_i)) \quad (26)$$

$$\mathcal{G}^C(f, p_i\, a_1 \ldots a_n) \stackrel{def}{=} \min_{i=1}^{i=n} nom^{\#}\, \mathcal{G}^C(f, a_i) \quad (27)$$

Table 4: Definition of the $\mathcal{G}^C$ operator.

Inductive cases

1. Using the induction hypothesis we have that:

$$\mathcal{G}^C(f, \lambda x.E) = \mathcal{G}^C(f, E) \leq \mathcal{G}(f, E, \boldsymbol{h}) = \mathcal{G}(f, \lambda x.E, \boldsymbol{h})$$

2.

$$\begin{aligned}
\mathcal{G}^C(f, C\, a_1 \ldots a_n) &= 1 + \min_{i=1}^{i=n} \mathcal{G}^C(f, a_i) \\
&\leq \{\text{By the induction hypothesis.}\} \\
& 1 + \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) \\
&= \mathcal{G}(f, C\, a_1 \ldots a_n, \boldsymbol{h})
\end{aligned}$$

3.

$$\mathcal{G}^C(f, f\, a_1 \ldots a_n) = \min(0, \min_{i=1}^{i=n} nom^{\#}\, \mathcal{G}^C(f, a_i))$$

If $\mathcal{G}^C(f, a_i) = \omega$ for each i then

$$\mathcal{G}^C(f, f\, a_1 \ldots a_n) = 0 = \mathcal{G}(f, f\, a_1 \ldots a_n, \boldsymbol{h})$$

since, for each i, $f_i^{\#}\, \boldsymbol{a}\, \omega = \omega$. If $\mathcal{G}^C(f, a_i) \neq \omega$ then $-\omega$ results and so the inequality must be satisfied.

4.

$$\mathcal{G}^C(f, \mathit{fname}\, a_1 \ldots a_n) = \min(\mathcal{S}^C(f, \mathit{fname}), \min_{i=1}^{i=n} nom^{\#}\, \mathcal{G}^C(f, a_i))$$

If, for any i, $\mathcal{G}^C(f, a_i) \neq \omega$ then the result is $-\omega$ and so the inequality holds.

In the case that ω results then, by the induction hypothesis, $\mathcal{G}(f, a_i, \boldsymbol{h}) = \omega$ and so ω will result from the application of any guardedness function, in particular $\mathit{fname}_i^{\#}$ for any i and also for $\mathit{nom}^{\#}$. In addition, it also follows from the induction hypothesis that $\mathcal{S}^C(f, \mathit{fname}) \leq \mathcal{S}(f, \mathit{fname}, \boldsymbol{b})$. It thus follows that the required inequality holds.

5. Consider the application,

$$x_k\ a_1 \ldots a_n$$

and suppose that for each i, $\mathcal{G}^C(f, a_i) = \omega$ (since otherwise the inequality will hold trivially). Now, if

$$\min_{F \in \mathbf{FnNames} \backslash \{f\}} (\mathcal{S}^C(f, F)) \neq -\omega$$

then it must be the case, due to the definition of $\mathcal{G}^C$, that f does not occur as the actual parameter to any function. In particular, for any application of the form

$$f\ a_1 \ldots a_n$$

in any function F, then f does not occur in any a_i. Moreover, if a formal parameter, x_j, of F occurs in a_i then x_j cannot be replaced by any expression containing f since otherwise there would be an application of F with f in one of the actual parameters. It follows that any term h_k that would replace x_k when evaluating $\mathcal{F}$ must have either a named function or a pattern matching variable at its head. In the case of the former,

$$\min_{F \in \mathbf{FnNames} \backslash \{f\}} (\mathcal{S}^C(f, F)) \leq \mathcal{S}^C(f, \mathit{fname}) \leq \mathcal{S}(f, \mathit{fname}, \boldsymbol{h})$$

with the second inequality being the induction hypothesis. Then, by assuming the induction hypothesis on each actual parameter, the overall result holds.

In the case of a pattern matching variable being the head function, the result holds due to the induction hypothesis on each actual parameter.

6. The case for the application of pattern matching variables follows immediately from the induction hypothesis for each actual parameter, a_i.

6.2 Comparison with Hughes, Pareto and Sabry Type Inference

Hughes, Pareto and Sabry have developed a type inference system for deducing the correctness of reactive systems, where the latter are modelled as functional programs upon streams [15]. Their approach consists of developing a system of *sized types*. Sized types consist of standard types tagged with either an integer or ω. This ordinal tag, i, is meant to indicate that the stream has at least i elements. Unification of types and terms then includes a constraint-solving phase in which a system of inequalities over ordinal variables is solved.

We do not know precisely how our analysis relates to the type inference system of [15], despite the fact that any type inference system can be viewed as an abstract interpretation [3]. However, we have shown in Section 4.3 that our analysis allows a productive definition of the list of Fibonacci numbers which is not accepted by their system. Moreover, we believe that our system is less complicated than theirs in that it requires only one phase, the calculation of guardedness functions, after Hindley-Milner type inference. The Hughes, Pareto and Sabry system, on the other hand, requires that a constraint solving system is used to ensure that the sized types (that have been inferred in a sequel to standard type inference) are consistent.

Their system, as it has been implemented, is also only semi-automatic, unlike ours: in order not to compromise the strength of the resulting constraint language they have actually produced a *type checker* which "requires all let-bound variables of a program to be annotated with sized type signatures, but infers the types for all other expressions". Their decision to use type checking rather than inference was also influenced by the complexity of solving letrec-expressions. By contrast, our system is completely automatic and does not require any preliminaries aside from standard Hindley-Milner type inference. Furthermore, we do not have to concern ourselves with the power of the available constraint solving systems: the only limitations (as discussed in Section 6.3 below) are built into the guardedness analysis system itself.

6.3 Limitations of the Analysis

We now look briefly at the ways in which the analysis is limited in that there are certain productive functions which will not be detected as being guarded. The limitations arise from two directions. Firstly, our abstract interpretation has been chosen to be easily implementable and of practical complexity. However, more sophisticated analyses may be able to detect a wider class of functions as being guarded. Secondly, the basic idea of a definition being guarded, being taken from process algebras, would also appear to impose a constraint on the class of definitions that can be admitted.

In either case we believe that such restrictions can be justified in a teaching context and can be summarised by some intuitive rules for the construction of corecursive definitions.

6.3.1 Limitations due to the Abstract Interpretation.

The main limitations on the class of corecursive algorithms admitted is due to the fact that our analysis is really just "first order and a little bit" as Hughes phrased it [12]. The analysis cannot, for instance, detect guardedness when a pattern matching variable is applied to a recursive call since we do not have any way of knowing the auxiliary guardedness functions of any term that will be matched by the variable. The same problem applies when a function is returned as a result by an expression and this resulting function then applied to a recursive call. This is why the $nom^{\#}$ guardedness function, which returns $-\omega$ on all results apart from ω, is used in the definition of our analysis. For example, the following function is productive but will not be detected as being guarded:

$$f \stackrel{def}{=} 1 \Diamond (fst\ cofnpair\ f)$$

Here, $cofnpair \stackrel{def}{=} (coid, coid)$ where $coid$ is the identity function over codata.

It is unclear how this restriction could be overcome without greatly increasing the complexity of the analysis, as is the case with Hughes's truly higher-order analysis in [12].

This restriction could probably be justified to students by using the above argument (that we do not know what the function that is extracted will do to the codata being corecursively defined) and we may summarise the restriction by saying that "No indirect applications to corecursive calls are allowed".

6.3.2 Limitations due to Guardedness.

The origin of the idea of guardedness in process algebra provides a more subtle restriction on the algorithms that will be allowed. This is due to the fact that a guarded process, as defined in [22], for example, involves sequential composition so that the process X defined by $a \cdot t$ can only be guarded if X does not occur in a. That is, forward references may not be made to parts of the tail of the process, t.

This means that, similarly, corecursive calls may not occur in the head of a colist, say. For example, the following *is* productive according to our unwinding rules but it is not guarded:

$$f \stackrel{def}{=} cohd(cotl f)\Diamond(1\Diamond f)$$

In general, we cannot have functions of the form:

$$f \stackrel{def}{=} (cohd\, cl_1)\Diamond cl_2$$

Here, cl_1 and cl_2 are some functions involving f.

It is unclear whether this would be a significant restriction to the programmer. Moreover, disallowing such occurrences of f is in keeping with our intuition that certain infinite structures represent sequences of values where each value may depend upon previous values in the sequence but not latter ones. It should be straightforward, therefore, to justify this restriction pedagogically e.g. "we cannot refer back to the whole structure until some elements have been defined".

6.4 Complexity of the Analysis

Here we describe the computational complexity of the analysis that we have given. This is naturally of concern to use since we seek to produce a useful functional programming system in which definitions may be checked for reducibility without imposing an intolerable burden on the compilation process.

It is straightforward to see from the definition of $\mathcal{G}$ that the guardedness analysis of an expression in the language will have linear complexity with respect to the size of the expression, if we assume that the primitive abstract domain functions, min, and $+_{\mathbf{A}}$ have linear complexity. In addition, as discussed in [5], where the analysis does not depend on any actual parameters (i.e. the function definition does not include any applications of the formal parameters), the principal guardedness function will be completely determined by, *at worst*, $n|\mathbf{A}|$ combinations of argument values, where n is the number of arguments to the function. This is so since there are $|\mathbf{A}|$ possibilities in determining the fixpoint of each auxiliary guardedness function. It is important to realise that whilst $|\mathbf{A}|$ is $\aleph_0$, we only use a small subset, $\{-\omega, -2, -1, 0, 1, 2, \omega\}$ in practice. We also emphasise the fact that this is the worst case complexity: we will only have to compute the auxiliary guardedness functions for those parameters which will correspond to a recursive call. This is unlike strictness analysis, for example, where we wish to determine whether each and every parameter is strict.

Complexity increases, however, in the higher-order case where the guardedness functions depend upon actual parameters. Here the principal guardedness function will be completely determined by, at worst, $|\mathbf{A}|^n$ combinations of argument values, since the guardedness of each parameter may depend on that of all the others.

This potentially exponential worst-case complexity is mitigated by the following factors:-

- The number of actual parameters in corecursive function definitions which are applied to corecursive calls of the function is typically two at the most.
- The size of the abstract domain that is used in practice has less than ten values. It should also be noted that we are using a simple extension of the integers and so the domain operations should be efficient. Moreover, we are not having to deal with a structured abstract domain where we seek to determine the abstract properties of the tail of a list, for example.
- Hindley-Milner type inference itself has potentially exponential worst-case complexity — see [17].

Since we have an infinite domain, we have to guard against the possibility of an infinitely descending fixpoint computation as in the following example:

$$
\begin{array}{l}
g\,s\,l \stackrel{def}{=} \\
\quad \mathsf{if}\ s\ \mathsf{then} \\
\quad\quad cotl\,l \\
\quad \mathsf{else} \\
\quad\quad g\,s\,(cotl\,l)
\end{array}
$$

This requires the solution of $\lambda f.\min(-1, f-1)$. However, we can easily detect such computations and make the result $-\omega$. The detection can be done by seeing whether a pre-fixpoint has a lower value than that with 0 substituted for f.

7 Conclusions and Future Work

We have demonstrated that a form of abstract interpretation, which may be shown to be sound, can be used to extend the notion of guardedness for infinite data structures. Such a method can be incorporated within a compiler for an elementary strong functional programming language to detect whether infinite objects are productive or not. We have suggested that the overhead of performing this analysis should be polynomial in practice and so should not impact badly upon any future compiler for an elementary strong functional language.

We would expect to be able to perform a similar analysis for *data* i.e. the least fixed points of inductive type definitions. This would naturally follow since Giménez [8] defined the dual notion of *guarded by destructors* for recursive function definitions over data. Consequently, we would expect to be performing the dual analysis (with least fixed points rather than greatest fixed points) over the *same* abstract domain, **A**. It would also be worth comparing such an approach to that of Walther recursion where a decidable test for a broader class of definitions than primitive recursion has been established [19]. Similarly, it would appear worthwhile to investigate the link to work by Giesl on automated termination proofs for nested and mutually recursive functional programs [7].

Another avenue for future research would be to investigate the meta-theoretic properties of this analysis. We have employed a backwards analysis in the style of Hughes [12] and it is unclear whether a forwards analysis would be sufficient to obtain the same results. A reason why forwards analysis may be inadequate for guardedness detection is that, for certain definitions, we have to determine whether the head of a *Colist* is guarded. It is known that, using a standard forward analysis, it is not possible to detect head-strictness of lists [16].

We conclude that a syntactic check for productivity in a simply-typed yet expressive functional language is made feasible by the work presented.

References

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

[2] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[3] P. Cousot. Types as abstract interpretations. In *24th ACM Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press.

[4] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 83–94. ACM press, 1992.

[5] K. Davis and P. Wadler. Strictness analysis in 4D. In S. L. Peyton Jones et al., editors, *Functional Programming, Glasgow 1990*, pages 23–43. Springer-Verlag, 1991.

[6] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[7] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, August 1997.

[8] E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs (TYPES '94)*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, 1995. International workshop, TYPES '94 held in June 1994.

[9] E. Giménez. Guardedness algorithm for co-inductive types. Coq club mailing list (`coq-club@pauillac.inria.fr`) correspondence, April 1997.

[10] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[11] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[12] R.J.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.

[13] R.J.M. Hughes. Compile-time analysis of functional programs. In Turner [29], pages 117–155.

[14] R.J.M. Hughes. Why functional programming matters. In Turner [29], pages 17–42.

[15] R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM Symposium on Principles of Programming Languages*, St Petersburg, Florida, January 1996. ACM Press.

[16] S. Kamin. Head-strictness is not a monotonic abstract property. *Information Processing Letters*, 41(4):195–198, 1992.

[17] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL '90*, pages 382–401. ACM Press, January 1990.

[18] P. 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.

[19] D. McAllester and K. Arkoudas. Walther recursion. In M.A. Robbie and J.K. Slaney, editors, *13th Conference on Automated Deduction (CADE 13)*, volume 1104 of *Lecture Notes in Computer Science*, pages 643–657. Springer-Verlag, 1996.

[20] P.F. Mendler, P. Panangaden, and R.L. Constable. Infinite objects in type theory. Technical Report TR 86-743, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1987.

[21] A.J.R.G. Milner. Theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[22] A.J.R.G. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[23] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, July 1996.

[24] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. Technical Report CS-R9652, CWI, Netherlands, CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands, 1996.

[25] B.A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

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

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

[28] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1985.

[29] D.A. Turner, editor. *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[30] D.A. Turner. Codata. Unpublished technical note (longer article in preparation), February 1995.

[31] D.A. Turner. Elementary strong functional programming. In P. Hartel and R. 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.

[32] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.