Ensuring Streams Flow*

Alastair Telford and David Turner

The Computing Laboratory, The University,
Canterbury, Kent, CT2 7NF, UK
E-Mail: A.J.Telford@ukc.ac.uk
Tel: +44 1227 827590 *Fax*: +44 1227 762811
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.

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 [11,20]. We would like our language to have a type system straightforwardly based on that of Hindley-Milner [6,14] and to be similar in usage to a language such as Miranda[1] [22]. The case for such a language is set out in [25] — 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

* 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.

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

of doing this in Miranda, Haskell [21] 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 [25] 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 [24]. 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 [5], in the Calculus of (Inductive) Constructions, have used the idea of *guardedness*, first proposed by Milner in the area of process algebras [15], 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 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.

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 Sect. 2 we give a summary of the theory behind infinite objects in strongly normalizing systems. We then show in Sect. 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 Sect. 4. This is followed in Sect. 5 by a proof that our analysis is sound and in Sect. 6 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 [17]. 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 [9]. 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 [25], 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 [5] 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 [15] 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`.

Conversely, the reasoning system of Sijtsma [18], being purely semantics-based, is not implementable as an automatic means of detecting whether a codata definition is productive.

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:

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

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

Introduction rule

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

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

$$a :: \&A\ \vdash\downarrow a :: A$$

Computation rule

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

Normal form

$$\mathsf{Fix}\ s'\ F'\ ::\ \&T$$

where s' and F' are both normal forms.

Fig. 1. Rules for codata.

top-level language may be translated. These rules, given in natural deduction style, are shown in Fig. 1 and were first given in [24]. They are derived from those of Mendler and others [13] for the Nuprl system, a variant of type theory.

Briefly, recursive occurrences of a type are replaced with their *suspension* (denoted with a &). 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, $\&T$. 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 [16].

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 [24] 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 1. Suppose that we have, $f :: A_1 \to \ldots \to A_n \to \&T$, where $n \geq 0$, and that T is a sum of product types (i.e. $T \stackrel{def}{=} \sum_{i=1}^{i=m} C_i\, T_i^1 \ldots T_i^{N(i)}$, where $N(i) \geq 0$). Then f is **productive** 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 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 below). In addition, each e_i^j is reducible.

This definition of productivity can be extended to closed expressions in the obvious way.

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

Definition 2. An expression, e, is **reducible** 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 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 Detecting Guardedness by Abstract Interpretation

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

[2] The Cousots [3] have shown how different semantic views of infinite structures may be related through abstract interpretation.

[3] [8] gives a good summary of abstract interpretation and backwards analysis in particular and [7] gives further details of backwards analysis.

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, **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:

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

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 **A**.

Definition 3. 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:

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

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.

Definition 4. 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. However, since these functions operate upon a complete lattice, $\mathbf{A}$, and can be shown to be continuous (see [19]), their *greatest fixed point* exists. This is found by forming a descending Kleene chain[4].

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 5 (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 Fig. 2. The definition of $\mathcal{G}$ involves the auxiliary operators, $\mathcal{S}$, $\mathcal{F}$ and $\mathcal{P}$, described below.

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. (9) allows the possibility of corecursion occurring within the switch expression of a case.

[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.

$$\mathcal{G}(f, f, \boldsymbol{h}) \stackrel{def}{=} 0 \tag{2}$$

$$\mathcal{G}(f, c, \boldsymbol{h}) \stackrel{def}{=} \omega \tag{3}$$

$$\mathcal{G}(f, x, \boldsymbol{h}) \stackrel{def}{=} \omega \tag{4}$$

$$\mathcal{G}(f, \mathit{fname}, \boldsymbol{h}) \stackrel{def}{=} \mathcal{S}(f, \mathit{fname}, \langle\rangle) \tag{5}$$

$$\mathcal{G}(f, \lambda x.E, \boldsymbol{h}) \stackrel{def}{=} \mathcal{G}(f, E, \boldsymbol{h}) \tag{6}$$

$$\mathcal{G}(f, C\ a_1 \ldots a_n, \boldsymbol{h}) \stackrel{def}{=} 1 + \min_{i=1}^{i=n} \mathcal{G}(f, a_i, \boldsymbol{h}) \tag{7}$$

$$\mathcal{G}(f, F\ a, \boldsymbol{h}) \stackrel{def}{=} \mathcal{F}(f, F, 1, \langle a\rangle, \boldsymbol{h}) \tag{8}$$

$$\mathcal{G}(f, \mathsf{case}\ s\ \mathsf{of}\ \langle p_1, e_1\rangle \ldots \langle p_n, e_n\rangle, \boldsymbol{h}) \stackrel{def}{=} \min(\min_{i=1}^{i=n} \min(\mathcal{G}(f, e_i, \boldsymbol{h}), \mathcal{P}\ (p_i, e_i)\ \boldsymbol{h}\ g), g) \tag{9}$$
$$\text{where } g = \mathcal{G}(f, s, \boldsymbol{h})$$

Fig. 2. Definition of the $\mathcal{G}$ 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. The basic idea is that the ith auxiliary guardedness function is applied to the guardedness level of the ith actual parameter. 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. This will return $-\omega$ on all inputs apart from ω.

We must also consider the possibility that the function, f, whose guardedness we are investigating, may occur in the body of the function F being applied. We thus have another operator, $\mathcal{S}$, the *substituted guardedness level* of f in F. It is intended to ensure that functions are guarded within mutually recursive definitions. If, $F\ y_1 \ldots y_p \stackrel{def}{=} E'$ then $\mathcal{S}(f, F, \boldsymbol{a}) \stackrel{def}{=} \mathcal{G}(f, E', \boldsymbol{a})$. Thus with the application of a named function, *fname*, say, we obtain the following:

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

Here, $\boldsymbol{b} = \boldsymbol{a}[\boldsymbol{h}/\boldsymbol{x}]$ and the auxiliary function, $\mathcal{N}$, produces the guardedness level of the application of a named function to a parameter:

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

Here, $g = \mathcal{G}(f, a_i, \boldsymbol{h})$. The substitution required to produce $\boldsymbol{b}$ consists of substituting actual parameters for their formal counterparts.

Similarly, we may obtain for corecursive applications:

$$\mathcal{G}(f, f\ a_1 \ldots a_n, \boldsymbol{h}) = \min(0, \min_{i=1}^{i=n} \mathcal{N}(f, f, i, \boldsymbol{a}, \boldsymbol{h}))$$

This means 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.

In higher-order functions, the function applied may be one of the parameters to the function. This is dealt with by substituting the corresponding element of $\boldsymbol{h}$ for the variable, so that we have $\mathcal{F}(f, x_j, i, \boldsymbol{a}, \boldsymbol{h}) \stackrel{def}{=} \mathcal{F}(f, h_j, i, \boldsymbol{a}, \boldsymbol{h})$. Where we do not know the actual parameter functions that comprise $\boldsymbol{h}$, an abstraction will be constructed over $\boldsymbol{h}$. 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 [7] and is explained further in [19].

`case` *expressions.* (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:

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

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:

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

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, $N(i) \stackrel{def}{=} |\mathrm{Var}(p_i)|$.

4 Example 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 [5]. 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 [4]. It and functions used in its definition are given in a Haskell-like syntax in Fig. 3. The type *Colist* here consists of the streams of integers. Further examples of guardedness analysis, including a demonstration that both *comap* and *comerge* are guarded, may be found in [19].

$$
\begin{array}{l}
\mathit{ham} :: \mathit{Colist} \\
\mathit{ham} \stackrel{def}{=} 1 \Diamond (\mathit{comerge}\ (\mathit{comap}\ (\times 2)\ \mathit{ham})\ (\mathit{comap}\ (\times 3)\ \mathit{ham})) \\[6pt]
\mathit{comap} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Colist} \to \mathit{Colist} \\
\mathit{comap}\ f\ (a \Diamond y) \stackrel{def}{=} (f\ a) \Diamond (\mathit{comap}\ f\ y) \\[6pt]
\mathit{comerge} :: \mathit{Colist} \to \mathit{Colist} \to \mathit{Colist} \\
\mathit{comerge}\ l@(a \Diamond x)\ m@(b \Diamond y) \stackrel{def}{=} \\
\quad \mathsf{case}\ \mathit{compare}\ a\, b\ \mathsf{of} \\
\quad\quad \mathsf{LT} \to a \Diamond (\mathit{comerge}\ x\, m) \\
\quad\quad \mathsf{EQ} \to a \Diamond (\mathit{comerge}\ x\, y) \\
\quad\quad \mathsf{GT} \to b \Diamond (\mathit{comerge}\ l\, y)
\end{array}
$$

Fig. 3. Definition of the Hamming function.

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.

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*.

$$
\begin{aligned}
\mathit{comap}_2^{\#}\ \langle h \rangle\, 0 &= \mathcal{G}(l, \mathsf{case}\ l\ \mathsf{of}\ (a \Diamond y) \to E') \\
&= \min(\mathcal{G}(l, E', \langle h \rangle), \mathcal{P}\ (a \Diamond y, E')\ \langle h \rangle\, 0, 0) \\
\mathcal{G}(l, E', \langle h \rangle) &= \mathcal{G}(l, (f a) \Diamond (\mathit{comap}\ f\ y), \langle h \rangle) = \omega \\
\mathcal{P}\ (a \Diamond y, E')\ \langle h \rangle\, 0 &= \min(\mathcal{G}(a, E', \langle h \rangle) - 1, \mathcal{G}(y, E', \langle h \rangle) - 1) \\
\mathcal{G}(a, E', \langle h \rangle) &= 1 + \mathcal{F}(a, f, 1, \langle a \rangle, \langle h \rangle) \\
&= 1 + h_1^{\#}\ 0 \\
\mathcal{G}(y, E', \langle h \rangle) &= 1 + \mathit{comap}_2^{\#}\ \langle h \rangle\, 0
\end{aligned}
$$

It follows that,

$$comap_2^{\#}\ \langle h \rangle\, 0 = \min(h_1^{\#}\, 0, comap_2^{\#}\ \langle h \rangle\, 0, 0)$$

$$\begin{aligned}
comerge_1^{\#}\ 0 &= \mathcal{G}(l, \mathsf{case}\ l\ \mathsf{of}\ (a \diamondsuit x) \to E') \\
&= \min(\mathcal{P}\ (a \diamondsuit x, E')\, 0, 0)
\end{aligned}$$

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

Thus,

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

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.

Analysis of the Main Function, *ham*.

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

(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**.

5 Soundness and Completeness

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

Theorem 6 (Due to Coquand, 1993). *If we assume that all data terms are normalizable then a codata function, f, will be productive for any set of inputs if it is guarded and its definition includes only reducible functions apart from f.*

Proof. The proof is by structural induction over the forms of defining expressions. We shall give a sketch of part of the proof — further details are in [19].

The base cases over primitive constants and variables are trivial, as is the abstraction case given clause (6) in the definition of $\mathcal{G}$.

As an example of one of the extensions, we take the case of (named) function applications. Since the application is guarded, if $n \leq \mathbf{Arity}(\mathit{fname})$,

$$\begin{aligned} 0 &< \mathcal{G}(f, \mathit{fname}\ a_1 \ldots a_n, \boldsymbol{h}) \\ &= \min(\mathcal{S}(f, \mathit{fname}, \boldsymbol{b}), \min_{i=1}^{i=n} \mathcal{N}(f, \mathit{fname}, i, \boldsymbol{a}, \boldsymbol{h})) \\ &\leq \mathcal{G}(f, E[b_1/x_1 \ldots b_n/x_n], \boldsymbol{b}) \end{aligned} \tag{10}$$

Here, $\mathit{fname}\ x_1 \ldots x_n \stackrel{def}{=} E$, and $\boldsymbol{b}$ consists of $\boldsymbol{a}$ with the components of $\boldsymbol{h}$ substituted for corresponding free variables. The last inequality (10) is proved in [19]. Since $E[b_1/x_1 \ldots b_n/x_n]$ must, by assumption, include only reducible terms (including possibly *fname*) apart from f, $E[b_1/x_1 \ldots b_n/x_n]$ must be productive by the induction hypothesis. Consequently, the application must be productive.

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, $\mathcal{G}(f, a_{m+i}, \boldsymbol{h}) = \omega$. It follows that for any G, where we add the definition, $G x \stackrel{def}{=} (\mathit{fname}\, b_1 \ldots b_i)\, x$, with $\boldsymbol{b}$ as above, $G_1^{\#}$ must produce ω on this input too. It then follows similarly to the inequality (10) that $G b_{i+1}$ is reducible and so $\mathit{fname}\, a_1 \ldots a_n$ is productive.

Our Hamming function example showed that our analysis could detect a productive definition as being guarded which would not fulfil the Coquand definition. The following result shows that our analysis is a complete extension of Coquand's work.

Theorem 7 (Completeness). *For corresponding definitions in* ESFP *and Coquand's type theory [2], if the definitions are guarded by Coquand's algorithm then they will be detected as being guarded by our abstract interpretation.*

Proof. Coquand's definition of guardedness can be formalised as an abstract interpretation, mapping from expressions to the abstract semantic domain, **A**. We can show, by structural induction over expressions that the abstract value produced by Coquand's analysis will always be less than or equal to that of ours. Full details are given in [19].

6 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 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 [5] 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 [12].

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 [7] 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 [10].

It also remains to show precisely the complexity of this abstract interpretation process. We have suggested in [19] 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 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 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.
4. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
5. 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.
6. J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

7. 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.
8. R.J.M. Hughes. Compile-time analysis of functional programs. In Turner [23], pages 117–155.
9. R.J.M. Hughes. Why functional programming matters. In Turner [23], pages 17–42.
10. S. Kamin. Head-strictness is not a monotonic abstract property. *Information Processing Letters*, 41(4):195–198, 1992.
11. 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.
12. 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.
13. 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.
14. A.J.R.G. Milner. Theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
15. A.J.R.G. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
16. L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, July 1996.
17. 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.
18. B.A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.
19. A.J. Telford and D.A. Turner. Ensuring the productivity of infinite structures. Technical report, University of Kent at Canterbury, 1997.
20. S.J. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
21. S.J. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.
22. 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.
23. D.A. Turner, editor. *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.
24. D.A. Turner. Codata. Unpublished technical note (longer article in preparation), February 1995.
25. 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.