Journal of Universal Computer Science, vol. 6, no. 4 (2000), 474-488
submitted: 30/4/99, accepted: 28/1/00, appeared: 28/4/00 © Springer Pub. Co.

Ensuring Termination in ESFP

Alastair Telford
(The Computing Laboratory, The University,
Canterbury, Kent, CT2 7NF, UK
A.J.Telford@ukc.ac.uk)

David Turner
(The Computing Laboratory, The University,
Canterbury, Kent, CT2 7NF, UK)

Abstract: In previous papers we have proposed an elementary discipline of *strong* functional programming (ESFP), in which all computations terminate. A key feature of the discipline is that we introduce a type distinction between *data* which is known to be finite, and *codata* which is (potentially) infinite. To ensure termination, recursion over data must be well-founded, and corecursion (the definition schema for codata) must be productive, and both of these restrictions must be enforced automatically by the compiler. In our previous work we used abstract interpretation to establish the productivity of corecursive definitions in an elementary strong functional language. We show here that similar ideas can be applied in the dual case to check whether recursive function definitions are strongly normalising. We thus exhibit a powerful termination analysis technique which we demonstrate can be extended to partial functions.

Key Words: Functional programming, termination analysis, abstract interpretation.
Category: D.1.1

1 Introduction

We are interested in the development of 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 (all reduction strategies converge) properties whilst avoiding the complexities (such as dependent types, computationally irrelevant proof objects) of Martin-Löf's type theory [13]. We would like our language to have a type system straightforwardly based on that of Hindley-Milner [15] and to be similar in usage to a language such as Miranda[1] [25]. The full case for such a language is set out in [26] but we recap its main potential benefits here:

- Such a language will allow both direct equational reasoning and simple induction principles — we do not have to worry about undefined elements when verifying properties.
- There is no dichotomy between lazy and strict evaluation as we shall have the Church-Rosser property and strong normalisation. This means that we have *evaluation transparency*, or what may be termed *true referential transparency.* We believe that this has the added benefits of making program optimisation, debugging and parallelisation easier to achieve.
- Since it does not have the complexities of type theory it is sufficiently elementary to be used for programming at the undergraduate level. Moreover, it is

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

more satisfactory from the pedagogical point of view: typically undergraduates are given step-by-step evaluations to perform which are done strictly in the recursive case, even in a lazy language such as Haskell (see [19]). Then, infinite structures, *with the same syntax and types*, are evaluated lazily.

In ESFP we make a clear distinction between *data* (finite structures — initial algebras) and *codata* (infinite structures — final coalgebras). We have described the characteristics of the latter in [22] and have extended syntactic checks devised by Coquand [4] in Type Theory, and Giménez [11], in the Calculus of (Inductive) Constructions, to check whether corecursive definitions are well-formed.

In this paper we apply the dual ideas to the dual structures, data. This extends the Giménez work [11] in the area of recursion. In particular, our analysis allows some non-primitive recursive algorithms which has been achieved by formulating a size descent detection algorithm as an *abstract interpretation*. The key point of using the abstract interpretation method is that it allows us to determine the level of destruction of an actual parameter when a function is applied *within* a recursive call.

We also extend our analysis to cope with partial functions using a simple subtyping mechanism. Furthermore, this extension allows a wider class of total algorithms to be accepted. As an illustration of the power of our analysis, we show how it can accept Euclid's *gcd* algorithm, which is undefined for two zero inputs.

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

2 An ESFP Language

We now present the characteristics of types and terms in an ESFP language and describe the restrictions that we make to ensure strong normalisation.

2.1 Data and codata

Firstly, we make a distinction between *data* (finite structures of inductive types) and *codata* (infinite structures of coinductive types). The reason for doing this is that functions acting upon data should perform a computation whilst recursively descending through a structure whilst those producing codata will be building a structure, possibly using some inputs. The semantic issues for infinite data structures, in which we explain what it means for codata functions to be productive and Church-Rosser, are explored further in [23].

2.2 Types

Algebraic data type definitions are basically as they appear in Haskell and each type constructor should occur only once in all the type definitions. In our abstract syntax, each type constructor is labelled C_i, where i is a natural number. There are the following added restrictions on algebraic type definitions:

1. Only strictly positive occurrences are allowed in the inductive definition of types. This means that in the definition of a type, T, say, T may not occur within the domain of any function space in the definition of T. For example, the following would not be allowed:

$$\textbf{\textit{data}}\ \mathit{ilist} \overset{def}{=} C\,(\mathit{ilist} \longrightarrow \mathit{Int})$$

2. T may not be defined via polymorphic type U where T occurs as an instantiation of U. For example, we would not allow rosetrees which can be given the following definition:

$$\textbf{\textit{data}}\ \mathit{Rosetree}\,a \overset{def}{=} \mathit{Leaf}\ a \,|\, \mathit{Node}\,[\mathit{Rosetree}\,a]$$

3. T may not be defined via a type U which is transitively defined using T.

4. T must have a *base case* i.e. one with no recursive occurrences of T.

We use the standard notion of ground types i.e. types which do not contain in their definition any function types.

2.3 Expressions

The basic typing system for expressions is that of Hindley-Milner [15]. Again, as in languages such as Miranda and Haskell, the same constructors that appear in type definitions appear in the same form within expressions in the language. We use $\mathrm{T}(e)$ to denote the type of expression e and $\mathrm{Unify}(e_1, e_2)$ to indicate that the types of expressions e_1 and e_2 unify.

The abstract syntax and applicative order operational semantics of data within our language is given in Table 1. The reduction relation, $\twoheadrightarrow_{\mathsf{Env}(E)}$, is a "big-step" one, relative to the environment $\mathsf{Env}(E)$ which binds closed expressions to free variables. A script in the language consists of a set of function definitions, for elements, f_i (where i is an integer) from the syntactic domain of function names, $\mathbb{F}$. Each function f_i has formal parameters labelled $x_{i,j}$.

Normal forms within the language are either lambda abstractions or constructor expressions of the form $C_i c_{i,1} \ldots c_{i,r}$ where all the $c_{i,j}$ are in normal form. The fact that an expression c is in normal form is denoted $\mathrm{nf}(c)$.

Note that we have specified an applicative order reduction sequence in which expressions are reduced to weak normal form [20], which is similar to the reduction strategy and notion of normal form used in strict functional languages such as SML [16]. This does not mean that ESFP programs *must* be evaluated strictly: we simply use this reduction strategy for data to demonstrate that our analysis will ensure termination in this case, hence guaranteeing strong normalisation. The fact that we only reduce as far as weak normal form is also unproblematical since we assume that lambda abstractions only occur as part of top-level definitions. Since a program may only be accepted if all the top-level definitions are accepted by the analysis, this, as we shall demonstrate, means that each function must terminate for any inputs i.e. strong normalisation will be ensured. We use $\mathrm{Ar}(f_i)$ to denote the arity of function f_i. $\mathrm{FT}(e)$ is used to indicate that an expression is of non-ground type.

Syntax

$$d \in \mathbb{D} \qquad f_i \in \mathbb{F}$$
$$x_{i,j} \in \mathbb{H} \qquad e_i \in \mathbb{E}$$
$$C_i \in \mathbb{C} \qquad p_i \in \mathbb{G}$$
$$v_{i,r} \in \mathbb{M}$$

$$\begin{aligned}
d &::= f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i \\
e &::= x_{i,j} \mid f_i \mid C_i\, e_1 \ldots e_r \mid e_1 e_2 \mid \textbf{\textit{case}}\; e_s\; \textbf{\textit{of}}\, \langle p_1, e_1\rangle \ldots \langle p_r, e_r\rangle \\
p &::= C_i\, v_{i,1} \ldots v_{i,r}
\end{aligned}$$

Operational Semantics

$$\frac{x_{i,j} \in \mathrm{Dom}(\mathsf{Env}(E)) \quad \mathsf{Env}(E)(x_{i,j}) \twoheadrightarrow_{\mathsf{Env}(E)} c}{x_{i,j} \twoheadrightarrow_{\mathsf{Env}(E)} c}\;(Vars)$$

$$\frac{\forall i \in \{1 \ldots (j-1)\}.\mathrm{nf}(e_i) \quad e_j \twoheadrightarrow_{\mathsf{Env}(E)} c_j \quad (\mathrm{nf}(c_j))}{C_i e_1 \ldots e_r \twoheadrightarrow_{\mathsf{Env}(E)} C_i e_1 \ldots e_{j-1} c_j e_{j+1} \ldots e_r}\;(Constr)$$

$$\frac{f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.E_i}{f_i \twoheadrightarrow_{\mathsf{Env}(E)} \lambda x_{i,1} \ldots x_{i,n}.E_i}\;(Func) \qquad \frac{e_1 \twoheadrightarrow_{\mathsf{Env}(E)} \lambda x.E;\; e \twoheadrightarrow_{\mathsf{Env}(E)} c \quad (\mathrm{nf}(c))}{e_1 e_2 \twoheadrightarrow_{\mathsf{Env}(E)} E[c/x]}\;(Appl)$$

$$\frac{\begin{array}{c}(\exists_1 i.e_s \twoheadrightarrow_{\mathsf{Env}(E)} C_i e_{i,1} \ldots e_{i,n}) \quad (p_i \equiv C_i v_{i,1} \ldots v_{i,n}); \\ \forall j.e_{i,j} \twoheadrightarrow_{\mathsf{Env}(E)} c_{i,j} \quad (\mathrm{nf}(c_{i,j}))\end{array}}{\textbf{\textit{case}}\; e_s\; \textbf{\textit{of}}\, \langle p_1, e_1\rangle \ldots \langle p_r, e_r\rangle \twoheadrightarrow_{\mathsf{Env}(E)} e_i[c_{i,1}/v_{i,1} \ldots c_{i,n}/v_{i,n}]}\;(Case)$$

Table 1: The Syntax and Semantics of Data in ESFP

Pattern matching over an input to a function will be taken to mean the application of a ***case*** expression to an input. Furthermore, nested patterns will be unsugared as nested ***case*** expressions. In order to help ensure termination, we stipulate that ***case*** expressions must be *exhaustive* over the patterns of the type:

Definition 1. A ***case*** expression, of the form, ***case*** *s* ***of*** $\langle p_1, e_1\rangle \ldots \langle p_r, e_n\rangle$ is **exhaustive** over the patterns (of the type of s) iff for every constructor of the type of s occurs within at the head of the patterns, p_i. Furthermore, patterns nested within a pattern must themselves be representable as exhaustive ***case*** expressions upon a simple variable.

We also assume that super-combinator abstraction has been applied to the original program so that we simply have a set of top-level definitions and that there are no definitions by partial application.

3 A Termination Condition

We now exhibit a termination condition based upon abstracting the sizes of terms in the language. The termination condition is based upon a semantic, undecidable property of actual parameter expressions. The property is, basically, that there is some well-founded descent upon *some* lexicographical ordering of the arguments for any recursive call of the function. We shall call this the *monotonic descent property.* The termination analysis that we shall develop in later sections will be a safe approximation to this condition. The theory here is closely based on that given in [2].

3.1 The Monotonic Descent Property

Definition 2. The **recursive sub-components** of a closed algebraic expression e, is defined as $\mathrm{Rec}(e) \triangleq \{e_j | e \twoheadrightarrow C_i\, e_1 \ldots e_r \wedge \mathrm{T}(e) \equiv \mathrm{T}(e_j)\}$

Definition 3. The **size** of a closed expression[2], e, is defined as follows:

– If e is not an algebraic type or if e does not have a normal form then $|e| = \omega$.

– If e is of algebraic type and normalises then,

$$|e| \triangleq \begin{cases} 0 & \text{if } \mathrm{Rec}(e) = \{\} \\ 1 + \sum_{e' \in \mathrm{Rec}(e)} |e'| & \text{otherwise} \end{cases}$$

In producing a condition for strong normalisation, we need to distinguish between each call of a function in the program text and, in addition, each call within the evaluation of a function upon some arguments.

Definition 4. Let P be a program i.e. a set of function definitions. Within P there are finitely many calls of each function, f, which we can label with positive integers to get labelled calls of the form f^k. We call k a **static label**.

Similarly, there are countably many recursive calls of each f^k that occur in the reduction path of some initial expression, $f\, t_1 \ldots t_n$. We label these, $f^{k,1}, f^{k,2} \ldots$

The arguments of each $f^{k,i}$ will be labelled $e_1^{k,i} \ldots e_n^{k,i}$.

The above labelling enables us to give a characterisation of the distinct (in terms of points in the program text) recursive calls of a function that are encountered during an evaluation.

Definition 5. Let $\mathrm{Calls}(f\, t_1 \ldots t_n)$ be the set of static label-distinct calls of f that are redexes within an applicative-order reduction of $f\, t_1 \ldots t_n$ where $t_1 \ldots t_n$ are closed terms.

Definition 6. The jth argument of a function f is termed **monotonic descending** for $F \equiv \mathrm{Calls}(f\, t_1 \ldots t_n)$, written $\mathrm{MonDesc}(f, j, F)$, iff

$$(\forall k . \forall i . |e_j^{k,i}| \leq |t_j|) \wedge (\exists f^m \in F . \forall i . |e_j^{m,i}| < |t_j|)$$

Definition 7. Let f be a function defined on n arguments and let $F \equiv \mathrm{Calls}(f\, t_1 \ldots t_n)$ (where $t_1 \ldots t_n$ are closed terms that are well-typed but otherwise arbitrary).

Then f has the **monotonic descent property** (written $\mathrm{MDP}(f, F)$) iff $F \equiv \{\} \vee (\exists j . \mathrm{MonDesc}(f, j, F) \wedge \mathrm{MDP}(f, F'))$. Here, $F' \equiv F \backslash F_j^{desc}$ and $F_j^{desc} \triangleq \{f^k \mid f^k \in F \wedge \forall i . |e_j^{k,i}| < |t_j|\}$

[2] We can also give the size of an open expression, when evaluating with respect to an environment $\mathsf{Env}(E)$, and denote this $|e|_{\mathsf{Env}(E)}$

$$\begin{array}{lll}
\mathcal{A}_{i,j}\,[\![\,x\,]\!]_{\rho,\sigma} & \triangleq & \begin{cases} 0 & \text{if } x \equiv x_{i,j} \\ -\omega & \text{if } x \equiv x_{i,k} \\ \mathcal{A}_{i,j}\,[\![\,t\,]\!]_{\rho,\sigma} - 1 & \text{if } \sigma(x) = \{t\} \wedge \text{Unify}(x,t) \\ \omega & \text{otherwise} \end{cases} \quad (1) \\
\mathcal{A}_{i,j}\,[\![\,f_k\,]\!]_{\rho,\sigma} & \triangleq & \begin{cases} f^a_{k,0}\,\{\} & \text{if } \text{Ar}(f_k) = 0 \wedge j = 0 \\ -\omega & \text{otherwise} \end{cases} \quad (2) \\
\mathcal{A}_{i,j}\,[\![\,C_t\,a_1 \ldots a_r\,]\!]_{\rho,\sigma} & \triangleq & \text{cs}(\text{Rec}(E), i, j, \rho, \sigma) \quad (3) \\
\mathcal{A}_{i,j}\,[\![\,\textbf{\textit{case}}\; s\; \textbf{\textit{of}}\; \langle p_r, e_r\rangle\,]\!]_{\rho,\sigma} & \triangleq & \max_{k=1}^{k=r} \mathcal{A}_{i,j}\,[\![\,e_k\,]\!]_{\rho,\sigma_k} \quad (4) \\
\mathcal{A}_{i,j}\,[\![\,F\,a\,]\!]_{\rho,\sigma} & \triangleq & \max\{\text{ap}^a(f,i,j,\boldsymbol{a},\rho,\sigma) \mid (f,\boldsymbol{a}) \in \mathcal{C}\,[\![\,F\,]\!]_{\rho,\sigma}\,\langle a\rangle\} \quad (5)
\end{array}$$

In (4),

$$\sigma_k = \bigcup_{l=1}^{l=|p_k|} B(p_{k,l}, s, \sigma) \text{ and } B(p_{k,l}, s, \sigma) = \begin{cases} \sigma\{p_{k,l} := \{s\}\} & \text{if}\,\text{Unify}(p_{k,l}, s) \\ \sigma & \text{otherwise} \end{cases}$$

Table 2: The definition of $\mathcal{A}_{i,j}\,[\![\,E\,]\!]_{\rho,\sigma}$

The above says that there must be some argument, j, of f which is both descending at some recursive call point in the program and, moreover, must not be ascending at any other recursive call point. Furthermore, f must have the monotonic descent property at all recursive call points where j is not descending.

Theorem 8. *Suppose the following about the definition of a function f:*

- *Apart from recursive calls of f (which may indirectly occur in functions called by f), the definition of f comprises only strongly normalising constants and functions.*
- *The typing rules of ESFP are followed.*
- ***case** expressions are exhaustive.*
- *f has the monotonic descent property.*

Then f is strongly normalising (SN) on all inputs, $t_1 \ldots t_{\text{Ar}(f)}$.

Proof. By induction on the size of F — see [24]. □

4 Termination Analysis By Abstract Interpretation

In this section we define an *abstract interpretation*[3] to detect whether a recursive function definition has the monotonic descent property. Here, due to space considerations, we only give the main definitions in our methodology and omit proofs of soundness and a demonstration of how the abstract interpretation can be constructed via successive approximations. Full details appear in [24].

[3] See [5] for an overview of abstract interpretation.

$$\mathcal{G}_{i[j]}[\![\, x \,]\!]_{\rho,\sigma} \triangleq \langle\rangle \tag{6}$$

$$\mathcal{G}_{i[j]}[\![\, f_k \,]\!]_{\rho,\sigma} \triangleq \begin{cases} f^g_{k[j]}\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k \neq j \\ \langle \Omega \rangle & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k = j \\ \langle\rangle & \text{otherwise} \end{cases} \tag{7}$$

$$\mathcal{G}_{i[j]}[\![\, C_t\, a_1 \ldots a_r \,]\!]_{\rho,\sigma} \triangleq \biguplus_{k=1}^{k=r} \mathcal{G}_{i[j]}[\![\, a_k \,]\!]_{\rho,\sigma} \tag{8}$$

$$\mathcal{G}_{i[j]}[\![\, \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r \rangle \,]\!]_{\rho,\sigma} \triangleq \biguplus(\mathcal{G}_{i[j]}[\![\, s \,]\!]_{\rho,\sigma}, (\biguplus_{k=1}^{k=r}(\mathcal{G}_{i[j]}[\![\, e_k \,]\!]_{\rho,\sigma_k}))) \tag{9}$$

$$\mathcal{G}_{i[j]}[\![\, F\, a \,]\!]_{\rho,\sigma} \triangleq \biguplus_{\substack{(f,\boldsymbol{a})\in \\ c[\![\, F \,]\!]_{\rho,\sigma}\langle a \rangle}} (\mathrm{ap}^g(f,i,j,\boldsymbol{a},\rho,\sigma) \uplus (\biguplus_{i=1}^{i=|\boldsymbol{a}|} \mathcal{G}_{i[j]}[\![\, a_i \,]\!]_{\rho,\sigma})) \tag{10}$$

In (9), σ_k is as for $\mathcal{A}$.

Table 3: The definition of $\mathcal{G}_{i[j]}[\![\, E \,]\!]_{\rho,\sigma}$

4.1 Static semantics

Starting from our basic, operational semantics we wish to obtain a series of abstract approximations to the idea of size of an expression and its size relative to a given parameter. Each successive approximation will be an *abstract semantics* of the preceding *concrete semantics*. Following the Cousots' approach [7], we wish to obtain an adjoint relationship between each abstract and concrete semantics. The maps, are *abstraction*, denoted α, which maps from a concrete to an abstract semantics, and *concretisation*, denoted γ, mapping in the opposite direction. To do so, we need to define a *static semantics* based upon our operational semantics. This will form our initial concrete semantics.

Definition 9. The **static semantics** of basic ESFP expressions is defined as follows: $\mathcal{O}[\![\, e \,]\!] \triangleq \{\lambda \mathsf{Env}(E). \begin{cases} c & \textit{if}(e \twoheadrightarrow_{\mathsf{Env}(E)} c) \wedge \mathrm{nf}(c) \\ \bot & \textit{otherwise} \end{cases} \}$

4.2 Relative size semantics

We require that the sizes of expressions are in fact *relative* to some given input.

Definition 10. The **relative size** domain, R, is the complete lattice, $\mathbb{Z} \cup \{\omega, -\omega\}$ (where $\top = \omega$ and $\bot = \omega$), with lub operator max and the following additive and multiplicative operations:

$$\begin{array}{lll} \omega + s = s + \omega = \omega & & -\omega * s = s * -\omega = -\omega \\ -\omega + s = s + (-\omega) = s & & s_1 * s_2 = s_1 + s_2 \qquad (s_1, s_2 \in \mathsf{R} \backslash \{-\omega\}) \\ s_1 + s_2 = s_1 +_{\mathbb{Z}} s_2 & (s_1, s_2 \in \mathbb{Z}) & s_1 - s_2 = s_1 + (-s_2) \end{array}$$

Definition 11. The **relative size semantics** of an expression, e, with respect to a parameter x, is defined as: $\mathcal{R}[\![\, e \,]\!]_x \triangleq \max\{\lambda \mathsf{Env}(E).|e|_{\mathsf{Env}(E)} - |\mathsf{Env}(E)(x)|\}$

4.3 Abstract interpretation of relative size

Before we present our abstract interpretation that approximates the idea of relative size, we mention an auxiliary analysis that allows us to abstract higher-order applications. This *closure analysis*, which is based on that given in [12] for Scheme, takes an application, $F\,a$ and produces a set of pairs of the form $f_i, \boldsymbol{a}$, where f_i is a function label and $\boldsymbol{a}$ is an actual parameter sequence and $\mathrm{Ar}(f_i) \geq |\boldsymbol{a}|$, where $|\boldsymbol{a}|$ is the length of $\boldsymbol{a}$. This set can be shown to be a sound approximation of the closures that may result from $F\,a$. The top of the abstract sub-domain of possible functions which may be applied is denoted $\top_{\mathsf{F}}$ and is used in the case where an indeterminate function is being applied, which occurs as the closure analysis cannot be complete. The bottom of the sub-domain, $\{\}$, is used when a variable is applied but there is no corresponding binding within ρ. Full details of this closure analysis semantic operator, $\mathcal{C}\,[\![\, e \,]\!]_{\rho,\sigma}\, \boldsymbol{a}$, are in [24].

4.3.1 Development of the abstract interpretation

We require an abstraction that can be used to compute an approximation of the relative size semantics of an expression. To do this, we calculate the contribution to the size of an expression made by each formal parameter in the current scope. For example, in the expression, $1 + x$, the parameter x makes a contribution to the size of the result. In addition, there is a constant factor, due to literal parts of expressions. In the previous example, there is a constant size factor of 1 as a consequence of the literal 1.

Definition 12. The $\mathcal{A}$ operator is defined over the structure of expressions in Table 2. In the definition, ρ is an environment binding function type expressions to variables, whilst σ is an environment binding pattern matching variables of algebraic types to expressions. i is a function index whilst $0 \leq j \leq \mathrm{Ar}(f_i)$. The above requires the auxiliary definitions given below (Defns 13–15).

The key clause in the definition is (1).There the size result depends upon whether a match is made with the parameter with respect to which we are analysing. In the case of pattern matching variables then if the variable is in σ it must be a recursive sub-component of the value that it is bound to. Otherwise, its relative size cannot be determined and so this must be approximated by ω.

Definition 13. We define the **constructor abstract size** function, cs, which appears in (3) in Table 2, as follows:

$$\begin{array}{lcl}
\mathrm{cs}(\{\}, i, 0, \rho, \sigma) & \triangleq & 0 \\
\mathrm{cs}(\{\}, i, j, \rho, \sigma) & \triangleq & -\omega \\
\mathrm{cs}(R, i, 0, \rho, \sigma) & \triangleq & 1 + \mathrm{sv} \\
\mathrm{cs}(R, i, j, \rho, \sigma) & \triangleq & \begin{cases} \omega & \text{if } \exists s_{k_1}, s_{k_2} \in S. \\ & \quad (k_1 \neq k_2) \wedge (s_{k_1} > -\omega) \wedge (s_{k_2} > -\omega) \\ -\omega & \text{if sv} = -\omega \\ 1 + \mathrm{sv} & \text{otherwise} \end{cases}
\end{array}$$

In the above,

$$S \;\triangleq\; \mathrm{Map}\,(\mathcal{A}_{i,j}\,[\![\, \bullet \,]\!]_{\rho,\sigma})\, R$$

$$\text{sv} \stackrel{\triangle}{=} \sum_{s_k \in S} s_k$$

Here Map is the mapping functor, defined in the standard way, over sequences.

Definition 14. We define the **abstract applicator for size analysis**, ap^a, which is used in (5) in Table 2, as follows.

$$\begin{aligned}
\text{ap}^a(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \rho, \sigma) &\stackrel{\triangle}{=} \omega \\
\text{ap}^a(\{\}, i, j, \boldsymbol{a}, \rho, \sigma) &\stackrel{\triangle}{=} \omega \\
\text{ap}^a(\{f_k\}, i, j, \boldsymbol{a}, \rho, \sigma) &\stackrel{\triangle}{=} (\boldsymbol{f_k^a} * \boldsymbol{a^a}) + v_j
\end{aligned}$$

In the above, $\boldsymbol{f_k^a} \stackrel{\triangle}{=} [f_{k,1}^a\,\rho' \ldots f_{k,\text{Ar}(f_k)}^a\,\rho']$ and $\boldsymbol{a^a} \stackrel{\triangle}{=} [\mathcal{A}_{i,j}\,[\![\, a_1 \,]\!]_{\rho,\sigma} \ldots \mathcal{A}_{i,j}\,[\![\, a_{|\boldsymbol{a}|} \,]\!]_{\rho,\sigma}]$.
$v_j \stackrel{\triangle}{=} \begin{cases} f_{k,0}^a\,\rho' & \text{if } j = 0 \\ -\omega & \text{otherwise} \end{cases}$ If $\boldsymbol{x}_i$ are the formal parameters of f_i,
$\rho' = \{(x_{i,j} \mapsto a_j[\rho]) \mid j \in \{1 \ldots \text{Ar}(f_i)\}, \text{FT}(a_i)\}$ where $a_j[\rho]$ is the simultaneous substitution, $a_j[\rho(x)/x]_{x \in \text{FV}(a_j) \wedge x \in \text{Dom}(\rho)}$.

Definition 15. The **abstract size function** of a function, $f_i \stackrel{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$, **relative to parameter** j is defined for a given environment of non-ground expressions ρ, $f_{i,j}^a\,\rho \stackrel{\triangle}{=} \mathcal{A}_{i,j}\,[\![\, e_i \,]\!]_{\{\},\rho}$

Performing the abstract interpretation with $j = 0$ gives the constant size factor of the expression. Each expression thus has $\text{Ar}(f_i) + 1$ interpretations under the $\mathcal{A}$ operator.

Definition 16. The **abstract size vector** of an expression e, with respect to environments ρ and σ, is defined as follows:

$$\boldsymbol{s}(e, i, \rho, \sigma) \stackrel{\triangle}{=} \begin{bmatrix} \mathcal{A}_{i,1}\,[\![\, e \,]\!]_{\sigma,\rho} \\ \vdots \\ \mathcal{A}_{i,\text{Ar}(f_i)}\,[\![\, e \,]\!]_{\sigma,\rho} \end{bmatrix}$$

We need to aggregate the elements of an abstract size vector so that the result is greater or equal to the size of the expression relative to one particular parameter. To do this we note that we cannot, of course, determine the value of $|\mathsf{Env}(E)(x_{i,j})| - |\mathsf{Env}(E)(x_{i,k})|$ for $j \neq k$ in general. Consequently, if $\mathcal{A}_{i,j}\,[\![\, e \,]\!]_{\sigma,\rho}$ is not $-\omega$ for $j \neq k$ then $\mathcal{R}\,[\![\, e \,]\!]_{x_{i,k}}$ is unknown in general. In such a situation we can must safely approximate with the ω value, which leads to the following definitions.

Definition 17. The j**th weighting vector** is a vector with a 0 in the j position and ω in all other positions.

Definition 18. The *abstract interpretation* of relative sizes over expressions is defined by the **component size semantics** of an expression, e, with respect to a parameter, $x_{i,j}$, is defined: $\mathcal{R}^{\#}\,[\![\, e \,]\!]_{i,j} \stackrel{\triangle}{=} \lambda\mathsf{Env}(E).(\boldsymbol{w_j}\,\boldsymbol{s}(e, i, \rho(\mathsf{Env}(E)), \{\}))$ where $\rho(\mathsf{Env}(E))$ is the subset of $\mathsf{Env}(E)$ of non-ground bindings and juxtaposition indicates vector product.

4.3.2 Determining least fixpoints

We form abstract size functions which may be recursive. These recursive equations may be solved by calculating the ascending Kleene chain, $F(-\omega)^i$, for $0 \leq i$, of the corresponding functional F. Since the abstract domain is infinite, however, convergence is not guaranteed within a finite number of steps. The simple chain structure of our domain however ensures the following, which is proved in [24]:

Lemma 19. *Let F be a functional corresponding to an abstract recursion equation formed from our abstract interpretation of relative sizes. Then either lfp $F = F(-\omega)^2$ or lfp $F = \omega$.*

Consequently, we can modify our least fixed point iteration method so that if the second iteration is not a fixpoint then ω is given as the result. This is an example of a *widening* process [8].

4.4 Detecting recursive calls

For a function, f_i, we need to perform an analysis of the definition of f_i which produces a representation of all potential recursive calls. Each recursive call will be represented by an *component size transformation.*

Definition 20. The **constant factors vector** and the **variable factors matrix** for a sequence of expressions, $\boldsymbol{e}$, and with respect to the parameters of function f_i and environments, ρ and σ, are denoted $\boldsymbol{k}(i,e,\rho,\sigma)$ and $\mathbf{V}(i,e,\rho,\sigma)$, respectively, and defined as follows:

$$\boldsymbol{k}(i,e,\rho,\sigma) \triangleq \begin{bmatrix} \mathcal{A}_{i,0} \llbracket e_1 \rrbracket_{\sigma,\rho} \\ \vdots \\ \mathcal{A}_{i,0} \llbracket e_{|\boldsymbol{e}|} \rrbracket_{\sigma,\rho} \end{bmatrix} \quad \mathbf{V}(i,e,\rho,\sigma) \triangleq \begin{bmatrix} \mathcal{A}_{i,1} \llbracket e_1 \rrbracket_{\sigma,\rho} \cdots \mathcal{A}_{i,\mathrm{Ar}(f_i)} \llbracket e_1 \rrbracket_{\sigma,\rho} \\ \vdots \qquad\qquad \vdots \\ \mathcal{A}_{i,1} \llbracket e_n \rrbracket_{\sigma,\rho} \cdots \mathcal{A}_{i,\mathrm{Ar}(f_i)} \llbracket e_n \rrbracket_{\sigma,\rho} \end{bmatrix}$$

Definition 21. The **component size transformation** (CST) and for a sequence of expressions, $\boldsymbol{e}$, and with respect to the parameters of function f_i and environments, ρ and σ, is defined: $\mathbf{T}(i,e,\rho,\sigma) \triangleq (\mathbf{V}(i,e,\rho,\sigma), \boldsymbol{k}(i,e,\rho,\sigma))$. The top of the domain of CSTs, Ω, consists of a transformation with all ω components.

If $(\mathbf{V_1}, \boldsymbol{k_1}), (\mathbf{V_2}, \boldsymbol{k_2})$ are CSTs then $(\mathbf{V_1}, \boldsymbol{k_1}) \star (\mathbf{V_2}, \boldsymbol{k_2}) \triangleq (\mathbf{V_1}\mathbf{V_2}, (\mathbf{V_1}\boldsymbol{k_2} + \boldsymbol{k_1}))$ if the relevant matrix multiplications are defined.

We again use an abstract interpretation process to discover all the component size transformations that correspond to the actual parameters of a recursive call of function f_j within function f_i.

Definition 22. $\mathcal{G}$ is the **abstract calls operator** and is defined over the structure of expressions in Table 3. $\mathcal{G}$ produces a sequence of CSTs. In the definition, $\uplus$, denotes the concatenation of sequences of CSTs and other auxiliary definitions follow below. We also have, for each function, a family of **abstract calls functions** which give the CSTs for the recursive calls of function f_j within the definition of function f_i: $f^g_{i[j]}\, \rho \triangleq \mathcal{G}_{i[j]} \llbracket e_i \rrbracket_{\rho,\sigma}$

In the definition of $\mathcal{G}$, the significant clause is (10). There a test for a recursive call is made. Note also that mutual recursion is dealt with by composing CSTs produced by the recursive call and the actual parameters. There is also a widening process that is in correspondence with that for size analysis.

Definition 23. We define the **abstract applicator for calls analysis**, ap^g which is used in (10) in Table 3, as follows

$$\begin{array}{lcl}
\mathrm{ap}^g(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \rho, \sigma) & \triangleq & \langle \Omega \rangle \\
\mathrm{ap}^g(\{\}, i, j, \boldsymbol{a}, \rho, \sigma) & \triangleq & \langle\rangle \\
\mathrm{ap}^g(\{f_k\}, i, j, \boldsymbol{a}, \rho, \sigma) & \triangleq & \begin{cases} \langle\rangle & \text{if } |\boldsymbol{a}| < \mathrm{Ar}(f_k) \\ \langle \mathbf{T}(i, \boldsymbol{a}, \rho, \sigma) \rangle & \text{if } f_k \equiv f_j \\ \biguplus_{\mathbf{T}' \in f^g_{k[j]} \rho'} (\mathrm{Map}\, (\star \mathbf{T}(i, \boldsymbol{a}, \rho, \sigma))\, \mathbf{T}') & \text{if } f_k \not\equiv f_j \end{cases}
\end{array}$$

In the above, Map is the standard mapping functor from the category of sets to that of sequences and $(\star \mathbf{T}(i, \boldsymbol{a}, \rho, \sigma))$ denotes right transformation multiplication. ρ' is as that given in Defn 14.

Definition 24. The **abstract calls matrix** of recursive calls of function f_i is defined thus:

$$\mathbf{ACM}(i) \triangleq \{\boldsymbol{r} \mid (\mathbf{v}, \boldsymbol{c}) \in f^g_{i[i]}\, \{\}\}$$

where, if $x_{i,j}$ is an algebraic argument, $r_j \triangleq \boldsymbol{w_j} \mathbf{v}_j + c_j$, $\boldsymbol{w_j}$ is the jth weighting vector and $\mathbf{v}_j$ is the jth column of $\mathbf{v}$. If $x_{i,j}$ is non-algebraic then $r_j \triangleq \omega$.

Analogously to the monotonic descent property, defined over $\mathrm{Calls}(f_i\, t_1 \ldots t_n)$ we may define the **abstract descent property** over $\mathbf{ACM}(i)$, written $\mathrm{ADP}(f_i, \mathbf{ACM}(i))$, and, as we show in [24]:

Theorem 25. $\mathrm{ADP}(f_i, \mathbf{ACM}(i)) \Rightarrow \mathrm{MDP}(f_i, \mathrm{Calls}(f_i\, t_1 \ldots t_n))$ *for any well-typed closed terms,* $t_1 \ldots t_n$.

5 Adding Subtyping

The above analysis is powerful enough to show that, for example, quicksort terminates. However, the class of functions admitted is still inadequate for the purposes of ESFP since we cannot, for example, make definitions via a head of list function (or any similar projection) since such a function is only partial. Moreover, the operational behaviour of certain total functions depends upon the form of the input e.g. whether the input is greater than zero. We would like to have a method of extending the analysis to partial functions (which may have ***error*** expressions for some clauses of ***case*** expressions) so that there is a well-defined sub-domain over which they are total and so that they are only ever applied over expressions within this sub-domain.

To do this we use a simple notion of subtyping, using sets of constructors of an algebraic type. That is, constructor C_i is within the subtype of a if and only if $a \twoheadrightarrow C_i\, e_1 \ldots e_{\mathrm{Ar}(C_i)}$ for some expressions e_j.

Note that we do not have any notion of subtyping of functions: this is because we are restricting attention to expressions of algebraic type.

We sketch how the analysis is modified, with full details given in [24].

- Each of the abstract semantic operator (and correspondingly each abstract function) has extra parameters, representing environments binding subtype sets to variables of algebraic types. Thus the modified operators are, $\mathcal{A}^1_{i,j}[\![\, i \,]\!]^{\phi}_{\rho,\sigma} e$ and $\mathcal{G}^1_{i[j,\phi_j]}[\![\, i \,]\!]^{\phi}_{\rho,\sigma} e$. In each case, ϕ is an environment of subtypes, whilst in the latter case, ϕ_j is the environment of subtypes that f_j was called with i.e. we no longer match simply on the function label but the subtyping environments must match too.
- The analyses will also need to determine whether functions that may be called have the abstract descent property for the subtyping environment implied by the actual parameters of the call. If this is not the case then the top of the relevant abstract domain (ω for size analysis, Ω for calls analysis) must result.
- The main change, and the point of this method, is at ***case*** expressions: instead of analysing all possible expressions that may result we only analyse those that match the subtype of the switch expression s. For example,

$$\mathcal{G}^1_{i[j,\phi_j]}[\![\, \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_i, e_i\rangle \,]\!]^{\phi}_{\rho,\sigma} \overset{def}{=} \biguplus(\mathcal{G}^1_{i[j,\phi_j]}[\![\, s \,]\!]^{\phi}_{\rho,\sigma}, (\biguplus_{k=1}^{k=r} G_k))$$

  where $G_k = \begin{cases} \mathcal{G}^1_{i[j,\phi_j]}[\![\, e_k \,]\!]^{\phi_k}_{\sigma_k,\rho} & \text{if } H(p_k) \in \mathcal{S}[\![\, s \,]\!]^{\phi_k}_{\sigma,\rho} \\ \{\} & \text{otherwise} \end{cases}$. Here, $H(p_k)$ is the leading constructor of the pattern p_k and $\mathcal{S}[\![\, s \,]\!]^{\phi_k}_{\sigma,\rho}$, which is also defined by abstract interpretation, gives an approximation to the subtype of the switch, s. ϕ_k is formed by adding the possible subtypes of the pattern matching variables to the environment, ϕ.
- Subtype environments need to be *partitioned* into the possible combinations of singleton sets when a function is encountered. For example, suppose we have the environment $\{m := \{0, S\}, n := \{S\}\}$ (where 0 and S are the constructors for the naturals) then this gives rise to two environments, $\{m := \{0\}, n := \{S\}\}$ and $\{m := \{S\}, n := \{S\}\}$.
- The weighting vectors can also be refined since, for a base case constructor, the size of the expression must be 0. Thus, if x_j has a base constructor subtype then it represents size descent from an inductive case constructor.

6 Example

An ESFP encoding of Euclid's *gcd* algorithm, which is not defined for two zero inputs, is as follows

```
gcd m n ≝
    case m of
        0 → error; (Succ n′) → n
        (Succ m′) →
            case compare m n of
                EQ → m;
                LT → gcd m (n − m);
                GT → gcd (m − n) n
```

0 - b $\overset{def}{=}$0; (***Succ*** a') - 0 $\overset{def}{=}$(***Succ*** a'); (***Succ*** a') - (***Succ*** b') $\overset{def}{=}$a' - b'

The analysis of the function, showing that *gcd* terminates for two non-zero inputs, proceeds as follows for $\phi = \{m := \{S\}, n := \{S\}\}$:

$$\mathcal{G}^1_{gcd[gcd,\phi]} [\![E_{gcd}]\!]^{\{\}}_{\phi,\{\}} = \mathcal{G}^1_{gcd[gcd,\phi]} [\![\textbf{case}\ compare\ m\,n\ \textbf{of}\ E']\!]^{\sigma}_{\phi',\{\}}$$

$$\phi' = \phi\{m' := \{0, S\}, n' := \{0, S\}\}; \sigma = \{m' := m, n' := n\}$$

$$= \{\} \cup \mathcal{G}^1_{gcd[gcd,\phi]} [\![gcd\ (m-n)\ n]\!]^{\sigma}_{\phi',\{\}} \cup \mathcal{G}^1_{gcd[gcd,\phi]} [\![gcd\ m\ (n-m)]\!]^{\sigma}_{\phi',\{\}}$$

$$\mathcal{G}^1_{gcd[gcd,\phi]} [\![gcd\ (m-n)\ n]\!]^{\sigma}_{\phi',\{\}} =$$
$$\Bigg\{ gcd^{g_1}_{[gcd,\{m:=\{0\},n:=\{S\}\}]}\ \{\},$$
$$\left(\begin{bmatrix} \mathcal{A}^1_{gcd,m} [\![m-n]\!]^{\sigma}_{\phi',\{\}} & \mathcal{A}^1_{gcd,n} [\![m-n]\!]^{\sigma}_{\phi',\{\}} \\ \mathcal{A}^1_{gcd,m} [\![n]\!]^{\sigma}_{\phi',\{\}} & \mathcal{A}^1_{gcd,n} [\![n]\!]^{\sigma}_{\phi',\{\}} \end{bmatrix}, \begin{bmatrix} \mathcal{A}^1_{gcd,0} [\![m-n]\!]^{\sigma}_{\phi',\{\}} \\ \mathcal{A}^1_{gcd,0} [\![n]\!]^{\sigma}_{\phi',\{\}} \end{bmatrix} \right) \Bigg\}$$

$$-^{a_1}_{1}\{a := \{S\}, b := \{S\}\}\ \{\} = -^{a_1}_{1}\{a := \{0, S\}, b := \{0, S\}\}\ \{\} * -1$$

$$-^{a_1}_{1}\{a := \{0, S\}, b := \{0, S\}\}\ \{\} = \max(-\omega, 0, -^{a_1}_{1}\{a := \{0, S\}, b := \{0, S\}\}\ \{\})$$

The least fixpoint of the above is 0 and thus,

$$-^{a_1}_{1}\{a := \{S\}, b := \{S\}\}\ \{\} = -1$$

$$-^{a_1}_{2}\{a := \{S\}, b := \{S\}\}\ \{\} = -^{a_1}_{2}\{a := \{0, S\}, b := \{0, S\}\}\ \{\} * 0$$

$$-^{a_1}_{2}\{a := \{0, S\}, b := \{0, S\}\}\ \{\} = \max(-\omega, -\omega, -^{a_1}_{2}\{a := \{0, S\}, b := \{0, S\}\}\ \{\})$$

Thus, $-^{a_1}_{1}\{a := \{S\}, b := \{S\}\}\ \{\} = -\omega$ and
$\mathcal{G}^1_{gcd[gcd,\phi]} [\![gcd\ (m-n)\ n]\!]^{\sigma}_{\phi',\{\}} = \left\{ \left(\begin{bmatrix} 0 & -1 \\ -\omega & -\omega \end{bmatrix}, \begin{bmatrix} -\omega \\ 0 \end{bmatrix} \right) \right\}$

Thus the **ACM** *gcd* with the subtyping environment ϕ is: $\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$

Hence, *gcd* has the abstract descent property for inputs with subtypes of the form indicated by ϕ.

7 Related Work

The general area of term rewriting has covered many aspects of general termination problems with work by Zantema of particular note (e.g. [27]). Most of this work does not address the issue of fully automated termination checks for programs, with [9] being an exception. In more specific programming areas, Giesl has worked on automated termination proofs for nested, mutually recursive and partial functional programs [10, 3]. This, like the work of Slind on TFL [21], are based on synthesising term orderings and termination predicates within a theorem-proving environment. In the more specific area of functional programming, a decidable test for a broader class of definitions than primitive recursion has been established for Walther recursion [14]. However, whilst ours is higher-order and polymorphic, theirs is first-order and monomorphic. Moreover, the discipline requires a programmer to provide different versions of functions for each algebraic subtype: our subtyping mechanism does this automatically. The TEA system [18] has used Nöcker's abstract reduction technique (whereby the standard evaluation of a program is replicated with abstract values; [17]) as a termination analyser. Their method detects whether a program terminates under a normal order evaluation scheme — it would have to be adapted for strict evaluation so as to detect strong normalisation. TEA does not deal with error expressions as we have done in our strongly normalising discipline in that it "usually treats errors as termination". Abel has also recently produced a termination checker, the Foetus system based on analysing call graphs [1]. This system only deals with simple syntactic descent at present.

8 Conclusions and Future Work

We have demonstrated that abstract interpretation can be used as an effective method for determining whether recursive functions terminate. The analysis is derived from the semantics of the language and uses the same domain of values employed to analyse the dual, corecursive case. The method can be incorporated within a compiler for an elementary strong functional programming language.

Recent work in this area has involved extending the basic domain so as to cope with types such as *Rosetree* and increasing the sophistication of the subtyping domain by including a representation of the possible size orderings between inputs. The purpose of this is to allow algorithms such as mergesort which divides a list in half.

An advantage of our abstract interpretation approach is that it may be possible to integrate our algorithm with Cousot's abstract interpretation rendering of Hindley-Milner type inference [6]. Thus we would have a single system which would ensure that type correctness meant that the program would have to be strongly normalising. Furthermore, analyses used for optimisation, such as binding-time analysis [12], may be integrated into this mechanism. In conclusion, we believe that this work gives an extensible and modular framework for broadening the class of algorithms that can be admitted by a syntactic analysis.

Acknowledgement

This work was supported by the UK Engineering and Physical Sciences Research Council grant number GR/L03279.

References

[1] A. Abel. Foetus - termination checker for simple functional programs. World Wide Web page, 1998. `http://www.informatik.uni-muenchen.de/~abel/publications/foetus/`.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] J. Brauburger and J. Giesl. Termination analysis by inductive evaluation. In C. Kirchner and H. Kirchner, editors, *CADE 15*, volume 1421 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1998.

[4] 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.

[5] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

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

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings Sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas.* ACM, 1979.

[8] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92: Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992. Proceedings of the Fourth International Symposium, Leuven, Belgium, 13–17 August 1992.

[9] M. C. F. Ferreira and H. Zantema. Syntactical analysis of total termination. In G. Levi and M. Rodrigues-Artalejo, editors, *ALP '94*, volume 850 of *Lecture Notes in Computer Science*, pages 204–222. Springer-Verlag, 1994.

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

[11] 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.

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

[13] 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.

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

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

[16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised*. MIT Press, 1997.

[17] Eric G.J.M.H. Nöcker. Strictness analysis using abstract reduction. In *Proceedings of Conference on Functional Programming Languages and Computer Architectures*. ACM Press, 1993.

[18] S. Panitz and M. Schmidt-Shauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In P. Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[19] S.L. Peyton Jones, R.J.M. Hughes, et al. Haskell 98: A non-strict, purely functional language. WWW page, February 1999. `http://haskell.org/definition`.

[20] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[21] K. Slind. TFL: An environment for terminating functional programs. WWW page, 1998. `http://www.cl.cam.ac.uk/users/kxs/tfl.html`.

[22] A.J. Telford and D.A. Turner. Ensuring Streams Flow. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th Int. Conference, AMAST '97, Sydney Australia, December 1997*, pages 509–523. AMAST, December 1997.

[23] A.J. Telford and D.A. Turner. Ensuring the productivity of infinite structures. Technical Report 14-97, University of Kent at Canterbury, 1997.

[24] A.J. Telford and D.A. Turner. A Hierarchy of Languages with Strong Termination Properties. Technical Report TR 2-00, University of Kent at Canterbury, February 2000.

[25] 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.

[26] 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.

[27] H. Zantema. Termination of context-sensitive rewriting. In H. Comon, editor, *RTA '97*, volume 1232 of *Lecture Notes in Computer Science*, pages 172 – 186. Springer-Verlag, 1997.