UKC Computing Laboratory TR 2-00

A Hierarchy of Languages with Strong Termination Properties[1]

Alastair Telford
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/`

February 2000

---

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

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.

Contents

List of Tables

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 [20]. We would like our language to have a type system straightforwardly based on that of Hindley-Milner [22] and to be similar in usage to a language such as Miranda[1] [35]. The full case for such a language is set out in [36] 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 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 [28]). 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 [31] and have extended syntactic checks devised by Coquand [6] in Type Theory, and Giménez [17], in the Calculus of (Inductive) Constructions, to check whether corecursive definitions are well-formed. Our analysis, used the idea of *guardedness* (i.e. that corecursive occurrences only occur beneath constructors), first proposed by Milner in the area of process algebras [23].

In this paper we apply the dual ideas to the dual structures, data. This extends the Giménez work [17] 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

---

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

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.

This subtyping mechanism is itself extended using *projection sequences* so that we can show that the standard definition of *mergeSort* terminates. The projection sequence mechanism also enables us to accept programs that are defined recursively on nested inductive types. 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. We also suggest that our work may be suitable as an enhancement to the algorithm for recognising strongly normalising recursive forms in the Coq system [5].

Overview of this Paper. In § 2 we define our EFP language which may be seen as the rudimentary heart of any functional programming system. We then present a semantic property, constructed from standard termination theory, that guarantees termination of an EFP program in § 3. This termination condition then serves as the basis for the abstract interpretation-based analysis that we develop in § 4. This analysis is strong enough to show that both Ackerman's function and the standard, naive definition of quicksort both terminate. We then, in § 5, seek to broaden the class of algorithms permitted within the language by introducing a simple subtyping mechanism that allows certain partial functions provided that they are applied to terms of the correct subtypes. To cope with nested inductive types and the associated schemes of recursion we develop the analysis given in § 6. This method, using projection sequences, is then developed further in § 7 to produce a more sophisticated subtyping mechanism which allows the subtypes of substructures to be captured, thus widening further the class of ESFP programs. In § 8 we discuss how the analysis developed can be placed within a general analysis framework which may be parameterised by the reduction semantics of the language and hence the idea of normal form. This means that our analysis can be extended to show not only the termination of programs to weak normal form (the standard for strict functional languages) but also can show termination to (strong) normal form, including reductions under lambdas. Finally, in § 9 we discuss related work and in § 10 we conclude.

2 An ESFP Language

We now present the characteristics of types and terms in an ESFP language. We shall refer to the language that we describe below, which consists of the core of languages such as Miranda or Haskell together with some basic syntactic restrictions, as our *basic* elementary functional programming language which we shall call EFP. Our full ESFP language will consist of this basic language together with an augmentation to the type-checking system which ensures that a program will terminate.

2.1 Data and Codata

Firstly, in our basic EFP language, 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 [32].

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:

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

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:

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

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

Definition 2.1 *The* ***syntactic domains*** *of our* EFP *language are as follows:*

$\mathbb{D}$	*Definitions*
$\mathbb{F}$	*Function names*
$\mathbb{H}$	*Function parameter names*
$\mathbb{C}$	*Constructors*
$\mathbb{M}$	*Pattern variables*
$\mathbb{G}$	*Patterns*
$\mathbb{E}$	*Expressions*

Syntax

$d \in \mathbb{D}$
$x_{i,j} \in \mathbb{H}$
$C_i \in \mathbb{C}$

$f_i \in \mathbb{F}$
$e_i \in \mathbb{E}$
$p_i \in \mathbb{G}$
$v_{i,r} \in \mathbb{M}$

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

Operational Semantics

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

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

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

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

Table 1: The Syntax and Semantics of Data in EFP

Definition 2.2 *The* ***abstract syntax and applicative order operational semantics*** *of data within our language is given in Table 1.*

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* $\mathrm{nf}(c)$*.*

The set of normal forms of expressions of the language (i.e. the ***values*** *of the system) is denoted* $\mathbb{V}$*. This set includes,* $\perp$*, the undefined value.*

The ***set of algebraic values*** *of the basic* EFP *language is denoted* $\mathbb{V}_{\mathbb{A}}$ *and consists of the subset of* $\mathbb{V}$ *that are of algebraic type. This includes* $\perp$*, the undefined value.*

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.

In order to help ensure termination, we stipulate that ***case*** expressions must be *exhaustive* over the patterns of the type:

Definition 2.3 *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.*

Definition 2.4 *The typing system for basic* EFP *expressions is that of Hindley-Milner [22]. 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.*

Definition 2.5 *A* ***script****,* $\mathbf{S}$*, consists of a set of function definitions,* f_i *(where* i *is an integer) from the syntactic domain of function names,* $\mathbb{F}$*. The indices of* $\mathbb{F}$ *form a set,* $\mathbb{I}^{\mathbf{S}}_{f}$*. Each function* f_i *has formal parameters labelled* $x_{i,1}, x_{i,2} \ldots$*.*

We use $\mathrm{Ar}(f_i)$ *to denote the arity of function* f_i*. That is, the* ***variable index set****,* $\mathbb{I}^{\mathbf{S}}_{f_i}$*, of a function* f_i *consists of* (i, j) *pairs where* $0 \leq j \leq \mathrm{Ar}(f_i)$*.* $\mathrm{FT}(e)$ *is used to indicate that an expression is of non-ground type.*

Note in the above that 0 is always included in this set, even though $(i, 0)$ does not label any variable in the script. This, as we shall see in § 4, is because we need to find the contribution made by constant i.e. non-variable factors to the semantic size of an expression.

Additional assumptions. Pattern matching over an input to a function will be taken to mean the application of a ***case*** expression to an input. We shall use Haskell-style syntax for formal parameters and patterns. Furthermore, nested patterns will be unsugared as nested ***case*** expressions. We also assume that super-combinator abstraction (including lambda lifting) 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. This means that we can cope with ***where*** definitions in our programs. Finally, we assume that, due to the standard isomorphism, $A \times B \longrightarrow T \approx A \longrightarrow B \longrightarrow T$, uncurried programs are translated into their curried equivalents.

Termination and reduction sequence. Note that we have specified an applicative order reduction sequence in which expressions are reduced to weak normal form [29], which is similar to the reduction strategy and notion of normal form used in strict functional languages such as SML [24]. 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. Thus the system we shall present will, in fact, ensure termination in a suitable subset of all current functional programming systems such as Haskell and SML. We shall show in § 8 how this can be generalised further so that strong normalisation will be ensured i.e. programs will terminate even if reductions under lambda abstractions are allowed.

2.4 EFP

In the light of the above description, we are now in a position to give the definition of our basic language.

Definition 2.6 *The* ***elementary functional programming language****, written* EFP*, consists of a functional programming language where*

1. *Data and codata and consequently recursive and co-recursive functions are syntactically separate, as in § 2.1.*
2. *The syntax of types obeys that of § 2.2.*
3. *The syntax and semantics of the expressions and types of expressions obeys that given in § 2.3, including Defns 2.1 – 2.5.*

We write $\text{Accept}(\mathbf{S}, \text{EFP})$ *to denote the fact that a script,* $\mathbf{S}$*, meets the above conditions for* EFP*.*

3 A Semantic Termination Condition

We now exhibit a termination condition based upon abstracting the sizes of terms in the EFP 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* lexicographic ordering of the arguments for any recursive call of the function. The fact that well-founded descent upon one argument will ensure termination will mean that termination will be guaranteed in the lexicographic case for several arguments, as is discussed in [2]. 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.

3.1 The Monotonic Descent Property

Definition 3.1 *The* ***recursive sub-components*** *of a closed algebraic expression e, is defined as*

$$\mathrm{Rec}(e) \triangleq \begin{cases} \bigcup_{i=1}^{i=r} \mathrm{UnifySub}(e_i, e) & \textit{if}\, e \twoheadrightarrow C_j\, e_1 \ldots e_r \\ \{\} & \textit{otherwise} \end{cases}$$

Here, $\mathrm{UnifySub}(e_1, e_2)$ *denotes the recursive sub-components of* e_1 *that unify with* e_2*:-*

$$\mathrm{UnifySub}(e_i, e_2) \triangleq \begin{cases} \{e_1\} & \textit{if}\, \mathrm{Unify}(e_1.e_2) \\ \bigcup_{i=1}^{i} = s\mathrm{UnifySub}(e_i, e_2) & \textit{if}\, e \twoheadrightarrow C_j\, e_1 \ldots e_r \\ \{\} & \textit{otherwise} \end{cases}$$

Definition 3.2 *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,*

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

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 3.3 *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 3.4 *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 3.5 *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*

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

---

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

Definition 3.6 *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 $F_j^{desc} \triangleq \{f^k \mid f^k \in F \wedge \forall i.|e_j^{k,i}| < |t_j|\}$*

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.

3.2 Termination Theorem for MDP

In this section we state and prove that the monotonic descent property, coupled with exhaustive ***case*** expressions, ensures termination under the operational semantics of EFP.

We first show that there cannot be infinitely many calls that descend on an argument if that argument does not ascend.

Lemma 3.1 *Suppose that a function f has a descending argument, j, on $F \equiv \mathrm{Calls}(f\, t_1 \ldots t_n)$ for some $t_1 \ldots t_n$. Let $S = \max_{f^r \in F_j^{desc}} I(f^r, t_1 \ldots t_n)$ where F_j^{desc} is as given in the definition of the monotonic descent property (Defn. 3.6) and $I(f^r, t_1 \ldots t_n)$ is the ordinal number of times that f^r occurs within the evaluation of $f\, t_1 \ldots t_n$.*

Then $S \leq |t_j|$

Proof. By induction on $|t_j|$.

Base case (where $|t_j| = 0$)

In this case there cannot be any calls of any $f^r \in F_j^{desc}$ since then by definition then $|e_j^{r,i} < |t_j| = 0$, which contradicts our definition of size.

Inductive case (where $|t_j| > 0$)

If $f\, t_1 \ldots t_n \twoheadrightarrow E(f_{r,1}\, e_1^{r,1} \ldots e_n^{r,1})$ where $f^r \in F_j^{desc}$ then, due to the descending argument property, $|e_j^{r,1}| < |t_j|$. Thus, by the induction hypothesis (for $|e_j^{r,1}|$), there are at most $|e_j^{r,1}|$ calls of f^r in $f\, e_1^{r,1} \ldots e_n^{r,1}$ Consequently, there are at most $|t_j|$ calls of any f^r.

We thus obtain our termination theorem.

Theorem 3.1 *Suppose the following about the definition of a function f of arity n:*

- *f is defined according to the rules of* EFP.
- *Apart from recursive calls of f (which may indirectly occur in functions called by f), the definition of f comprises only constants and functions which terminate under the operational semantics of* EFP.

- *f has the monotonic descent property.*

Then f terminates on all inputs, $t_1 \ldots t_n$, following the operational semantics of EFP *given in Table 1.*

Proof. By induction on the number of elements in $F \equiv \text{Calls}(f\, t_1 \ldots t_n)$.

Base case (where $\text{Calls}(f\, t_1 \ldots t_n) = \{\}$)
In this case there are no recursive calls. It follows that since all other expressions are SN and ***case*** expressions are exhaustive, f must also be SN.

Inductive case
$\text{MDP}(f, F)$ implies that there exists a descending argument of f, j, say. By Lemma 3.1 there are at most $|t_j|$ calls of any $f^r \in F_j^{desc}$. Consequently, in the reduction sequence of $f\, t_1 \ldots t_n$, there must be an ith call in of some $f^r \in F_j^{desc}$ such that $\text{Calls}(f\, e_1^{r,i} \ldots e_n^{r,i}) \cap F_j^{desc} = \{\}$. Since f has the monotonic descent property on any inputs, it must have the monotonic descent property on $\text{Calls}(f\, e_1^{r,i} \ldots e_n^{r,i})$. Thus as the number of elements in $\text{Calls}(f\, e_1^{r,i} \ldots e_n^{r,i})$ is less than the number of elements in $\text{Calls}(f\, t_1 \ldots t_n)$, it follows by induction that $f\, e_1^{r,i} \ldots e_n^{r,i}$ is terminating and consequently $f\, t_1 \ldots t_n$ is terminating.

3.3 Example of a function with the MDP

We now show that Ackerman's function has the MDP.

Example 3.1 Ackerman's function, *ack* is defined as follows:

```
ack m n ≝
    case m of
    0 → n + 1
    (Succ m′) →
        case n of
        0 → ack m′ 1
        (Succ n′) → ack m′ (ack m n′)
```

We can argue that Ackerman's function has the MDP as follows: Note that if the first input, m, is 0 then the MDP holds trivially. Otherwise, there are three recursive calls of *ack*, $ack\, m'\, 1$, $ack\, m'\, (ack\, m\, n')$ and $ack\, m\, n'$ which are labelled as ack_1, ack_2 and ack_3, respectively.

Then, for arbitrary inputs m and n, $\text{MonDesc}(ack, 1, \text{Calls}(ack\, m\, n))$ (where, if $m > 0$, $\text{Calls}(ack\, m\, n) = \{ack_1, ack_2, ack_3\}$) since in ack_1 and ack_2, $|m'| < |m|$ whilst in ack_3, $|m| = |m|$.

It also follows that $\text{MonDesc}(ack, 2, \{ack_3\})$ since in ack_3, $|n'| < |n|$.

Hence, it follows that *ack* has the monotonic descent property.

◇

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.

We assume to start with that we do not have any nested or mutually inductive types. This means that where $E \equiv C_i e_1 \dots e_r$,

$$\mathrm{Rec}(E) = \{e_i \mid \mathrm{Unify}(e_i, E)\}$$

Obviously, this set is decidable. In Sect. 6 we shall see how the analysis may be extended to encompass nested inductive types.

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 [11], 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 4.1 *The set of* ***semantic properties*** *of our basic* ESFP *language, denoted* $\mathbb{P}$ *is defined as* $\mathbb{P} \overset{\triangle}{=} \wp(\mathbb{V})$*,*

The set of ***algebraic semantic properties*** *of our basic* ESFP *language, denoted* $\mathbb{P}_{\mathbb{A}}$ *is defined as* $\mathbb{P}_{\mathbb{A}} \overset{\triangle}{=} \wp(\mathbb{V}_{\mathbb{A}})$*.*

Definition 4.2 *The* ***static semantics*** *of basic* ESFP *expressions,* $\mathcal{O}\,[\![\, \bullet \,]\!] \in \mathbb{E} \times \mathsf{Env}(\mathbb{E}) \mapsto \mathbb{P}$ *is defined as follows:* $\mathcal{O}\,[\![\, e \,]\!]\, \mathsf{Env}(E) \overset{\triangle}{=} \{\begin{cases} c & \mathit{if}(e \twoheadrightarrow_{\mathsf{Env}(E)} c) \wedge \mathrm{nf}(c) \\ \bot & \mathit{otherwise} \end{cases}\}$

4.2 Relative size semantics

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

Definition 4.3 *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:*

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

Definition 4.4 *The* ***relative size semantics*** *of an expression, e, with respect to a parameter x, is defined as:* $\mathcal{R}\,[\![\, e \,]\!]_x \overset{\triangle}{=} \max\{\lambda \mathsf{Env}(E).|e|_{\mathsf{Env}(E)} - |\mathsf{Env}(E)(x)|\}$

---

[3] See [8] for an overview of abstract interpretation.

4.3 Abstract Expression Domain

Definition 4.5 *The set of all* ***type-correct substitution instances*** *of expressions, denoted* $\mathbb{E}^s$ *is defined as:*

$$\mathbb{E}^s \triangleq \{e[\boldsymbol{a}/\boldsymbol{x}] \mid e \in \mathbb{E}, \forall i.a_i \in \mathbb{E}^s, \mathrm{Unify}(x_i, a_i)\}$$

Definition 4.6 *The* ***abstract expression domain****, denoted* E*, consists of the powerset of all type-correct substitution instances of expressions i.e.* $\mathsf{E} \triangleq \wp(\mathbb{E}^s)$ *We denote the top of this complete lattice by* $\top_{\mathsf{E}}$*.*

Definition 4.7 *The domain of* ***pattern variable expression environments****,* M*, consists of functions binding pattern matching variables to elements of* E *i.e.* $\mathsf{M} \triangleq \mathbb{M} \mapsto \mathsf{E}$

Note that in the above, E is an infinite complete lattice. In order to ensure that the closure analysis calculation terminates we need to introduce approximations to the standard notion of expression substitution. These approximations are an example of a *widening*, a technique introduced and shown to be sound by the Cousots [12].

Definition 4.8 *The* ***abstract expression substitution*** *of an abstract expression,* b *for a variable* x *within an abstract expression* a*, denoted* $a[b \,/_{\mathsf{E}}\, x]$*, is defined via standard expression substitution thus:*

$$\begin{aligned} \top_{\mathsf{E}}[b \,/_{\mathsf{E}}\, x] &\triangleq \top_{\mathsf{E}} \\ \{e\}[\top_{\mathsf{E}} \,/_{\mathsf{E}}\, x] &\triangleq \begin{cases} \top_{\mathsf{E}} & \textit{if } x \in \mathrm{FV}(e) \\ \{e\} & \textit{otherwise} \end{cases} \\ \{e_1\}[\{e_2\} \,/_{\mathsf{E}}\, x] &\triangleq \{e_1[e_2 \,/_{\mathsf{E}}\, x]\} \end{aligned}$$

We can define a series of such substitutions, $a[b_1 \,/_{\mathsf{E}}\, x_1 \ldots b_r \,/_{\mathsf{E}}\, x_r]$*, also in an analogous way to that for standard substitution.*

In particular, we need to approximate in the case where we may be substituting expressions involving the parameters of a function for those parameters. The definition of approximation that we introduce below prevents an infinite growth in the size of substituted expressions.

Definition 4.9 *The* ***approximation of $\boldsymbol{b}$ with respect to $\boldsymbol{x}$****, where* $\boldsymbol{b}$ *is a vector of abstract expressions, is defined as,* $\boldsymbol{Apx}(\boldsymbol{b}, \boldsymbol{x}) \triangleq \boldsymbol{b'}$ *where*

$$b'_i \triangleq \begin{cases} \top_{\mathsf{E}} & \textit{if } b_i = \top_{\mathsf{E}} \\ \top_{\mathsf{E}} & \textit{if } b_i = \{e_i\} \wedge \exists j.j \neq i \wedge x_j \in \mathrm{FV}(e_i) \\ \{e_i\} & b_i = \{e_i\} \end{cases}$$

We can use the above definition in order to construct simultaneous substitutions over abstract expressions.

Definition 4.10 *The* ***simultaneous substitution*** *of a vector* $\boldsymbol{b}$ *of abstract expressions for a vector* $\boldsymbol{x}$ *of formal parameters within a vector of abstract expressions,* $\boldsymbol{a}$*, defined as,* $\boldsymbol{a}[\boldsymbol{b} /_{\mathsf{E}}\, \boldsymbol{x}] \triangleq \boldsymbol{a'}$ *where* $a'_i = a_i[b'_1 /_{\mathsf{E}}\, x_1 \ldots b'_{|\boldsymbol{x}|} /_{\mathsf{E}}\, x_{|\boldsymbol{x}|}]$ *and* $\boldsymbol{b'} = \boldsymbol{Apx}(\boldsymbol{b}, \boldsymbol{x})$*.*

Similarly we define substitutions of abstract expression environments within an abstract expression as follows: $b_j[\rho]$ *is the simultaneous* E *substitution,* $b_j[\rho(x)/x]_{x \in \mathrm{FV}(b_j) \wedge x \in \mathrm{Dom}(\rho)}$*.*

Likewise, we may substitute within an abstract expression environment, which we denote as $\sigma[\boldsymbol{b}/\boldsymbol{x_i}]$

4.4 Closure Analysis

We now describe an auxiliary analysis that allows us to abstract higher-order applications. This *closure analysis*, which is based on that given devised by Palsberg, Bondorf and Sestoff [18, 26], takes an application, $F\,a$ and produces a set of triples of the form $(f_i, \boldsymbol{a}, \sigma)$, where f_i is a function label, $\boldsymbol{a}$ is an actual parameter sequence and σ is an environment binding expressions to pattern matching variables. We shall see that the latter is necessary in order to determine whether a reduction in the size of an argument to a recursive call has occurred. Furthermore, $\mathrm{Ar}(f_i) \geq |\boldsymbol{a}|$, where $|\boldsymbol{a}|$ is the length of $\boldsymbol{a}$ i.e. we ensure that each actual parameter can be bound to some formal parameter of a function.

We first need to define the abstract domains that comprise our closure analysis.

Definition 4.11 *The* ***abstract function label*** *domain, denoted* F*, consists of all possible singleton sets of function labels together with the empty set and the set of all function labels (which is equivalent to* $\mathbb{F}$ *and which we denote here* $\top_{\mathsf{F}}$*) i.e.*

$$\mathsf{F} \triangleq \{\{\}\} \cup \{\mathbb{F}\} \cup \bigcup_{f_i \in \mathbb{F}} \{\{f_i\}\}$$

The powerset of the product of the above definitions then defines our abstract space of closures.

Definition 4.12 *The* ***abstract closure domain****,* C*, is defined as follows:*

$$\mathsf{C} \triangleq \wp(\mathsf{F} \times \mathsf{E}^* \times \mathsf{M})$$

where E^* *denotes finite sequences of elements of* E*. The top of* C *is denoted* $\top_{\mathsf{C}}$*.*

Definition 4.13 *The* ***closure analysis*** *semantic operator,* $\mathcal{C} \in \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \mathsf{E}^* \mapsto \mathsf{C}$*, is defined in Table 2.*

Definition 4.14 *The* ***abstract closure function*** *of a function,* $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$*, is defined for a given environment of non-ground expressions* ρ*, and a sequence of actual parameter expressions,* $\boldsymbol{a}$*, as* $f_i^m\, \rho\, \boldsymbol{a} \triangleq \mathcal{C}\, [\![\, e_i \,]\!]_{\rho,\{\}}\, \boldsymbol{a}$

An abstract closure function, f_i^m, produces a set of function, actual parameter sequence pairs where the actual parameter expressions are in terms of the formal parameters of f_i. However, these actual parameter expressions need to be transformed into expressions involving the parameters of the calling context.

$$\mathcal{C}\llbracket x \rrbracket_{\rho,\sigma}\,\boldsymbol{a} \triangleq \begin{cases} \top_{\lambda} & \text{if } \rho(x) = \top_{\mathsf{E}} \vee \sigma(x) = \top_{\mathsf{E}} \\ \{(\{\},\boldsymbol{a},\sigma)\} & \text{if } \rho(x) = \{\} \\ \mathcal{C}\llbracket e \rrbracket_{\rho,\sigma}\,\boldsymbol{a} & \text{if } \rho(x) = \{e\} \vee \sigma(x) = \{e\} \end{cases} \tag{1}$$

$$\mathcal{C}\llbracket f_i \rrbracket_{\rho,\sigma}\,\boldsymbol{a} \triangleq \begin{cases} \{(\{f_i\},\boldsymbol{a},\sigma)\} & \text{if } \mathrm{Ar}(f_i) \geq |\boldsymbol{a}| \\ \{(f,\boldsymbol{e},\sigma') \mid (f,\boldsymbol{d},\sigma) \in f_i^m\,\rho'\,\boldsymbol{c}\} & \text{otherwise} \end{cases} \tag{2}$$

$$\mathcal{C}\llbracket C_t\,a_1 \ldots a_r \rrbracket_{\rho,\sigma}\,\boldsymbol{a} \triangleq \bigcup_{i=1}^{i=r} \{(f,\boldsymbol{b},\sigma) \mid (f,\boldsymbol{b},\sigma) \in e_i \wedge \mathrm{TC}(f,\boldsymbol{b})\} \tag{3}$$

$$\mathcal{C}\llbracket \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r \rangle \rrbracket_{\rho,\sigma}\,\boldsymbol{a} \triangleq \bigcup_{i=1}^{i=r} \mathcal{C}\llbracket e_i \rrbracket_{\rho,\sigma_i}\,\boldsymbol{a} \tag{4}$$

$$\mathcal{C}\llbracket G\,d \rrbracket_{\rho,\sigma}\,\boldsymbol{a} \triangleq \mathcal{C}\llbracket G \rrbracket_{\rho,\sigma}\,(\langle\{d\}\rangle \mathbin{+\!\!+} \boldsymbol{a}) \tag{5}$$

In (2), if $\boldsymbol{x}_i$ are the formal parameters of f_i,
$\rho' \triangleq \{(x_{i,j} \mapsto b_j[\rho]) \mid j \in \{1 \ldots \mathrm{Ar}(f_i)\}, \mathrm{FT}(b_j)\}$
where $\boldsymbol{b} \triangleq \langle a_1 \ldots a_{\mathrm{Ar}(f_i)} \rangle$, $\boldsymbol{c} \triangleq \langle a_{\mathrm{Ar}(f_i)+1} \ldots a_{|\boldsymbol{a}|} \rangle$, $\boldsymbol{e} \triangleq \boldsymbol{d}[\boldsymbol{b} /_{\mathsf{E}}\, \boldsymbol{x_i}]$ and $\sigma' \triangleq \sigma[\boldsymbol{b} /_{\mathsf{E}}\, \boldsymbol{x_i}]$.
In (4), $\sigma_i \triangleq (\bigcup_{j=1}^{j=|p_i|} B^{\mathcal{C}}(p_{i,j}, s, \sigma))$ and $B^{\mathcal{C}}(p_{i,j}, s, \sigma) \triangleq \sigma\{p_{i,j} := s\}$
In (3), $\mathrm{TC}(f,\boldsymbol{b})$ indicates that $f\,\boldsymbol{b}$ is a type-correct application; $\forall \boldsymbol{b}.\mathrm{TC}(\top_{\mathsf{C}}, \boldsymbol{b},)$

Table 2: Definition of $\mathcal{C}\llbracket E \rrbracket_{\rho,\sigma}\,\boldsymbol{a}$

4.4.1 Correctness.

Finally, we show that our closure analysis is correct in the sense that it is a superset of the closures evaluated during computation of an application.

Theorem 4.1 *The closure analysis is safe in the sense that any application that would be evaluated in the standard semantics is captured by the closure analysis.*

Proof. By induction on the structure of expressions. □

4.5 Abstract interpretation of relative size

We now construct an abstract interpretation over the expression syntax to approximate the idea of relative size. 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.

$$\mathcal{A}_{i,j}\llbracket\, x \,\rrbracket_{\rho,\sigma} \triangleq \begin{cases} 0 & \text{if } x \equiv x_{i,j} \\ -\omega & \text{if } x \equiv x_{i,k} \\ \mathcal{A}_{i,j}\llbracket\, t \,\rrbracket_{\rho,\sigma} - 1 & \text{if } \sigma(x) = \{t\} \wedge \text{Unify}(x,t) \\ \omega & \text{otherwise} \end{cases} \tag{6}$$

$$\mathcal{A}_{i,j}\llbracket\, f_k \,\rrbracket_{\rho,\sigma} \triangleq \begin{cases} f^{a}_{k,0}\,\{\} & \text{if } \text{Ar}(f_k) = 0 \wedge j = 0 \\ -\omega & \text{otherwise} \end{cases} \tag{7}$$

$$\mathcal{A}_{i,j}\llbracket\, C_t\, a_1 \ldots a_r \,\rrbracket_{\rho,\sigma} \triangleq \text{cs}(\text{Rec}(E), i, j, \rho, \sigma) \tag{8}$$

$$\mathcal{A}_{i,j}\llbracket\, \textbf{case}\ s\ \textbf{of}\ \langle p_r, e_r\rangle \,\rrbracket_{\rho,\sigma} \triangleq \max_{k=1}^{k=r} \mathcal{A}_{i,j}\llbracket\, e_k \,\rrbracket_{\rho,\sigma_k} \tag{9}$$

$$\mathcal{A}_{i,j}\llbracket\, F\, a \,\rrbracket_{\rho,\sigma} \triangleq \max\{\text{ap}^{a}(f, i, j, \boldsymbol{a}, \rho, \sigma) \mid (f, \boldsymbol{a}, \sigma_k) \in \mathcal{C}\llbracket\, F \,\rrbracket_{\sigma,\rho}\langle\{a\}\rangle\} \tag{10}$$

Table 3: Definition of $\mathcal{A}_{i,j}\llbracket\, E \,\rrbracket_{\rho,\sigma}$

Before giving our full abstract interpretation, we must first abstract the idea of recursive sub-components, as given in Defn 3.1. To start with we shall make a simplifying assumption, that *no nested type definitions are allowed.* We shall relax this stipulation in Sect. 6. This means that in this case, we have the following definition.

Definition 4.15 *The* ***set of recursive subcomponents*** *of an algebraic expression is defined for our abstract interpretation as follows:*

$$\mathrm{RecAbs}(C_i e_1 \ldots e_r) \triangleq \{e_i \mid \mathrm{Unify}(e_i, (C_i e_1 \ldots e_r))\}$$

We can now give a definition for our semantic operator that is going to abstract the idea of relative size.

Definition 4.16 *The* ***relative size analysis operator***, $\mathcal{A} \in \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathbb{E} \times \mathsf{Env}(\mathsf{E}) \times \mathsf{M} \mapsto \mathsf{R}$, *is defined over the structure of expressions in Table 3 with auxiliary definitions given in Defns 4.17–4.19. 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 $0 \leq j \leq \mathrm{Ar}(f_i)$.*

Definition 4.17 *We define the* ***constructor abstract size*** *function,* $\mathrm{cs} \in \wp(\mathbb{E}) \times \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathsf{Env}(\mathsf{E}) \times \mathsf{M} \mapsto \mathsf{R}$, *which appears in (8) in Table 3, as follows:*

$$\mathrm{cs}(\{\}, i, 0, \rho, \sigma) \triangleq 0 \tag{11}$$

$$\mathrm{cs}(\{\}, i, j, \rho, \sigma) \triangleq -\omega \tag{12}$$

$$\mathrm{cs}(R, i, 0, \rho, \sigma) \triangleq 1 + \mathrm{sv} \tag{13}$$

$$\mathrm{cs}(R, i, j, \rho, \sigma) \triangleq \begin{cases} \omega & \textit{if}\, \exists s_{k_1}, s_{k_2} \in S.(k_1 \neq k_2) \wedge (s_{k_1} > -\omega) \wedge (s_{k_2} > -\omega) \\ -\omega & \textit{if}\, \mathrm{sv} = -\omega \\ 1 + \mathrm{sv} & \textit{otherwise} \end{cases} \tag{14}$$

In the above,

$$S \triangleq \mathrm{Map}\,(\mathcal{A}_{i,j}\, [\![\bullet]\!]_{\rho,\sigma})\, R \tag{15}$$

$$\mathrm{sv} \triangleq \sum_{s_k \in S} s_k \tag{16}$$

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

Definition 4.18 *The $\mathcal{A}$ operator is lifted to the E domain as follows:*

$$\mathcal{A}_{i,j}\, [\![\top_{\mathsf{E}}]\!]_{\rho,\sigma} \triangleq \omega \tag{17}$$

$$\mathcal{A}_{i,j}\, [\![\{e\}]\!]_{\rho,\sigma} \triangleq \mathcal{A}_{i,j}\, [\![e]\!]_{\rho,\sigma} \tag{18}$$

Definition 4.19 *We define the* ***abstract applicator for size analysis***, ap^a, *which is used in (10) in Table 3, as follows.*

$$\mathrm{ap}^a(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \sigma, \rho) \triangleq \omega \tag{19}$$

$$\mathrm{ap}^a(\{\}, i, j, \boldsymbol{a}, \sigma, \rho) \triangleq \omega \tag{20}$$

$$\mathrm{ap}^a(\{f_k\}, i, j, \boldsymbol{a}, \sigma, \rho) \triangleq (\boldsymbol{f^a_k} * \boldsymbol{a^a}) + v_j \tag{21}$$

In the above, $\boldsymbol{f^a_k} \triangleq [f^a_{k,1}\,\rho' \ldots f^a_{k,\mathrm{Ar}(f_k)}\,\rho']$ *and* $\boldsymbol{a^a} \triangleq [\mathcal{A}_{i,j}\, [\![a_1]\!]_{\rho,\sigma} \ldots \mathcal{A}_{i,j}\, [\![a_{|\boldsymbol{a}|}]\!]_{\rho,\sigma}]$.

$$v_j \triangleq \begin{cases} f^a_{k,0}\,\rho' & \textit{if}\, j = 0 \\ -\omega & \textit{otherwise} \end{cases}$$

Definition 4.20 *The* ***relative size abstraction*** *of a function,* $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$, ***relative to parameter*** j, *is defined for a given environment of non-ground expressions* ρ, *as:*

$$f^a_{i,j}\,\rho \overset{\triangle}{=} \mathrm{lfp}(F_{i,j,\rho})$$

Here, $F_{i,j,\rho}$ *is the* functional *defined as,* $F_{i,j,\rho}(f^a_{i,j}\,\rho) \overset{\triangle}{=} \mathcal{A}_{i,j}\,[\![\, e_i \,]\!]_{\rho,\{\}}$. *§ 4.5.1 details how the least fixpoints are calculated. Where there is no ambiguity, we shall write a functional simply as* F.

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

Discussion of the $\mathcal{A}$ operator. The key clauses in the definition given in Table 3 are (6) and constructor expressions (8) (and Defn 4.17). In the case of variables, 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, 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. This reflects the fact that we cannot determine, in general, the sizes of data elements of structures.

In the case of constructors, we have to determine which are the recursive subcomponents of the expression and take the abstract relative sizes of those (see (15) in Defn 4.17). However, if $j \neq 0$ and the variable $x_{i,j}$ contributes to the abstract size of the constructor expression more than once (through separate subtrees of the constructor expression) then ω results (see (14)). This is because a multiplicative factor of relative size has been detected (i.e. the size of the expression is $kx_{i,j}$ where $k \geq 2$) which cannot be accepted by our analysis.

As would be expected, if we are finding the size of an expression, e relative to a variable, $x_{i,j}$ then the result is $-\omega$ if $x_{i,j}$ does not occur in e.

Lemma 4.1 *Let* $x_{i,j}$ *be a formal parameter of a function* f_i. *Then if* $x_{i,j}$ *does not occur within an expression* e, $\mathcal{A}_{i,j}\,[\![\, e \,]\!]_{\rho,\sigma} = -\omega$.

Proof. By a simple structural induction over e. □

4.5.1 Determining least fixpoints for size analysis.

We form abstract size functions which may be recursive. We now discuss how their fixpoints are calculated, in view of the fact that we have an infinite chain as our abstract domain R.

Lemma 4.2 *The functionals defined for the abstract size functions are monotonic and continuous. That is,*

$$\forall a_1, a_2 \in \mathsf{R}. a_1 \leq a_2 \Rightarrow F(a_1) \leq F(a_2)$$

and

$$F(\max(a_1, a_2)) = \max(F(a_1), F(a_2))$$

Proof. By structural induction over functionals which are constructed from the max, +, − and $*$ operators, together with constants from the R domain. □

The least fixpoint of each functional instance corresponding to an abstract size function thus exists and can be found by computing the ascending Kleene chain, $F^r(-\omega)$, for $0 \leq r$, where $F^0(-\omega) \triangleq - - \omega$, and $F^{r+1}(-\omega) \triangleq F(F^r(-\omega))$. 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:

Lemma 4.3 *Let F be a functional corresponding to an abstract recursion equation formed from our abstract interpretation of relative sizes. Then either* $\mathrm{lfp}(F) = F^2(-\omega)$ *or* $\mathrm{lfp}(F) = \omega$.

Proof. By induction on the structure of functionals. □

Widening of the fixpoint iteration process. 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 [12]. Here, the widening consists of a family of operations that depend upon the iteration, similar to that in [7].

Definition 4.21 *Let L be a complete lattice*[4]*. Then a* ***widening*** *is a family of operators (indexed over $\mathbb{N}$), $\bigtriangledown_n \in L \times L \mapsto L$, which meets the following conditions:*

1. $\forall x, y \in L, r \in \mathbb{N}.(x \sqsubseteq (x \bigtriangledown_r y)) \wedge (y \sqsubseteq (x \bigtriangledown_r y))$
2. *For all increasing chains, $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$, the increasing chain defined by, $y^0 \triangleq x^0; y^{r+1} \triangleq y^r \bigtriangledown_{r+1} x^{r+1}$ is not strictly increasing.*

In the above, $\sqsubseteq$ is the ordering on L.

Definition 4.22 *The* ***upward iteration sequence with widening*** *is defined as follows:*

$$\begin{array}{lll} \mathrm{U}^0 & \triangleq \bot & \\ \mathrm{U}^{r+1} & \triangleq \mathrm{U}^r & \text{if } F(\mathrm{U}^r) \sqsubseteq \mathrm{U}^r \\ & \triangleq \mathrm{U}^r \bigtriangledown_r F(\mathrm{U}^r) & \text{otherwise} \end{array}$$

As shown in [12], the upward iteration sequence with widening reaches a fixpoint within finitely many steps and, furthermore, is a sound upper approximation of the least fixpoint of the functional.

We thus define the widening operator for our size analysis.

---

[4] Actually, the domain need only be a CPO.

Definition 4.23 *The **widening operator for relative size analysis**, $\overset{a}{\bigtriangledown}_r \in \mathsf{R}\times\mathsf{R} \mapsto \mathsf{R}$, is defined as follows:*

$$\begin{array}{rcll} x_r \overset{a}{\bigtriangledown}_r y_r & \triangleq & \omega & \text{if } (r > 3) \wedge x_r \neq y_r \\ & \triangleq & \max(x_r, y_r) & \text{otherwise} \end{array}$$

Lemma 4.4 *The $\overset{a}{\bigtriangledown}_r$ operator is a widening operator in the sense of Defn 4.21.*

Proof. The proof is simply by an examination of the definitions. □

Definition 4.24 *The **fixpoint computation** of the functional associated with each relative size abstraction of a function is defined by the upward iteration sequence given in Defn 4.22 where the widening operator used is $\overset{a}{\bigtriangledown}_r$, which was presented in Defn 4.23.*

Note that in the light of Defn 4.24 and Defn 4.22 we could have shortened the last clause of Defn 4.23 so that simply y_r results rather than $\max(x_r, y_r)$.

Lemma 4.5 *The fixpoint computation given in Defn 4.24 finds the least fixpoint of the relevant functional and computes it in finitely many steps.*

Proof. By Lemma 4.3, Lemma 4.4 and [12]. □

4.5.2 Combining abstract size components.

The above has given a method of calculating the size component of an expression due to a given parameter or, in the case where $j = 0$ in the size analysis, due to constant factors other than variables. We now show how we combine the relative size information with respect to all the parameters of a function and with respect to constant factors, to given an abstraction of the total size of an expression relative to a given input.

Definition 4.25 *The **abstract size vector** of an expression e, with respect to the environments of function expressions, ρ, and pattern matching expressions, σ, is defined as follows:*

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

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 $\mathcal{A}_{i,j} [\![e]\!]_{\rho,\sigma}$ is not $-\omega$ for $j \neq k$ then $\mathcal{R} [\![e]\!]_{x_{i,k}}$ is unknown in general. In such a situation we must safely approximate with the ω value, which leads to the following definitions.

Definition 4.26 *The* j***th weighting vector*** *is a vector with a* 0 *in the* j *position if* $x_{i,j}$ *is of algebraic type. Otherwise, where* $x_{i,j}$ *is not algebraic,* ω *is in the* j*th position.* ω *is in all other positions, regardless of their types.*

Definition 4.27 *The* abstract interpretation *of relative sizes over expressions is defined by the* ***component size semantics*** *of an expression,* e*, with respect to a parameter,* $x_{i,j}$*:* $\mathcal{R}^{\#}\,[\![\, e \,]\!]_{i,j} \triangleq \lambda \mathsf{Env}(E).(\boldsymbol{w_j s}(e, i, \{\}, \{\}))$ *where juxtaposition indicates vector product.*

Theorem 4.2 *The component size semantics is a safe approximation of the relative size semantics.*

Proof. By structural induction for some expression E and the definition of the abstract size vector.

4.6 Detecting Recursive Calls

$$\mathcal{G}_{i[j]}\,[\![\, x \,]\!]_{\rho,\sigma} \triangleq \langle\rangle \tag{22}$$

$$\mathcal{G}_{i[j]}\,[\![\, f_k \,]\!]_{\rho,\sigma} \triangleq \begin{cases} f^{g}_{k[j]}\,\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k \neq j \\ \langle \boldsymbol{\Omega} \rangle & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k = j \\ \langle\rangle & \text{otherwise} \end{cases} \tag{23}$$

$$\mathcal{G}_{i[j]}\,[\![\, C_t\, a_1 \ldots a_r \,]\!]_{\rho,\sigma} \triangleq \biguplus_{k=1}^{k=r} a_k \tag{24}$$

$$\mathcal{G}_{i[j]}\,[\![\, \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r \rangle \,]\!]_{\rho,\sigma} \triangleq \biguplus(\mathcal{G}_{i[j]}\,[\![\, s \,]\!]_{\rho,\sigma}, (\biguplus_{k=1}^{k=r}(\mathcal{G}_{i[j]}\,[\![\, e_k \,]\!]_{\rho,\sigma_k}))) \tag{25}$$

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

Table 4: Definition of $\mathcal{G}_{i[j]}\,[\![\, E \,]\!]_{\rho,\sigma}$

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 a *component size transformation.*

Definition 4.28 *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* $\boldsymbol{c}(i, \boldsymbol{e}, \sigma, \rho)$ *and* $\mathbf{v}(i, \boldsymbol{e}, \sigma, \rho)$*, respectively, and defined as* *follows:*

$$\boldsymbol{c}(i,\boldsymbol{e},\sigma,\rho) \triangleq \begin{bmatrix} \mathcal{A}_{i,0}\,[\![\, e_1 \,]\!]_{\rho,\sigma} \\ \vdots \\ \mathcal{A}_{i,0}\,[\![\, e_{|\boldsymbol{e}|} \,]\!]_{\rho,\sigma} \end{bmatrix} \mathbf{v}(i,\boldsymbol{e},\sigma,\rho) \triangleq \begin{bmatrix} \mathcal{A}_{i,1}\,[\![\, e_1 \,]\!]_{\rho,\sigma} & \cdots & \mathcal{A}_{i,\mathrm{Ar}(f_i)}\,[\![\, e_1 \,]\!]_{\rho,\sigma} \\ \vdots & & \vdots \\ \mathcal{A}_{i,1}\,[\![\, e_n \,]\!]_{\rho,\sigma} & \cdots & \mathcal{A}_{i,\mathrm{Ar}(f_i)}\,[\![\, e_n \,]\!]_{\rho,\sigma} \end{bmatrix}$$

Definition 4.29 *The* ***component size transformation*** *(CST) for a sequence of expressions, $\boldsymbol{e}$, and with respect to the parameters of function f_i and environments, ρ and σ, is defined as a pair of a variable factors matrix and a constant factors vector thus:*

$$\mathbf{T}(i,\boldsymbol{e},\rho,\sigma) \triangleq (\mathbf{v}(i,\boldsymbol{e},\rho,\sigma), \boldsymbol{c}(i,\boldsymbol{e},\rho,\sigma))$$

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.

The set of CSTs is denoted T and $\top_{\mathsf{T}}$ is the CST with all ω components.

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 that may be reached by reduction from a call of function f_i. A sequence of CSTs, corresponding to the recursive calls will be computed by the following operator.

Definition 4.30 *The* ***abstract calls operator****, $\mathcal{G} \in \mathbb{I}_f^{\mathsf{S}} \times \mathbb{I}_f^{\mathsf{S}} \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \mapsto \mathsf{T}^*$, and is defined over the structure of expressions in Table 4. In the definition, $\uplus$, denotes the concatenation of sequences of CSTs and other auxiliary definitions follow below.*

Definition 4.31 *We define the* ***abstract applicator for calls analysis****, $\mathrm{ap}^g \in \mathsf{F} \times \mathbb{I}_f^{\mathsf{S}} \times \mathbb{I}_f^{\mathsf{S}} \times \mathsf{E}^* \times \mathsf{M} \times \mathit{Env}(\mathsf{E}) \mapsto \mathsf{T}^*$ which is used in (26) in Table 4, as follows*

$$\mathrm{ap}^g(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \rho, \sigma) \triangleq \langle \top_{\mathsf{T}} \rangle \tag{26}$$

$$\mathrm{ap}^g(\{\}, i, j, \boldsymbol{a}, \rho, \sigma) \triangleq \langle\rangle \tag{27}$$

$$\mathrm{ap}^g(\{f_k\}, i, j, \boldsymbol{a}, \rho, \sigma) \triangleq \begin{cases} \langle\rangle & \text{if } (|\boldsymbol{a}| < \mathrm{Ar}(f_k)) \\ \langle \mathbf{T}(i,\boldsymbol{a},\rho,\sigma) \rangle & \text{if } f_k \equiv f_j \\ \biguplus_{\mathbf{T'} \in f^g_{k[j]}\rho_k} (\mathrm{Map}\,(\star\mathbf{T}(i,\boldsymbol{a},\rho,\sigma))\;\mathbf{T'} & \text{if } f_k \not\equiv f_j \end{cases} \tag{28}$$

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.*

Definition 4.32 *For each function, there is 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:*

$$f^g_{i[j]}\rho \triangleq \mathrm{lfp}(F_{i[j],\rho})$$

Here, $F_{i[j],\rho}$ is the functional *defined as, $F_{i[j],\rho}(f^g_{i[j]}\,\rho) \triangleq \mathcal{G}_{i[j]}\,[\![\, e_i \,]\!]_{\rho,\{\}}$. As before, we write F for $F_{i[j],\rho}$ and details of the computation are given in § 4.6.1.*

Discussion of the $\mathcal{G}$ operator. In the definition of $\mathcal{G}$, the significant clause is (26). 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.

As with size analysis, the following holds.

Lemma 4.6 *Consider functions f_i and f_j. Then if f_j does not occur within the definition, E, of f_i or, transitively, any function called by f_i then, $\mathcal{G}_{i[j]} [\![E]\!]_{\rho,\sigma} = \langle\rangle$.*

Proof. By a simple structural induction over E. □

4.6.1 Calculating fixpoints for calls analysis.

As with size analysis, there is a potential for the calls analysis to spawn an infinite ascending Kleene chain during the calculation of fixpoints. Indeed, since each CST is composed of elements of R it is a consequence of Lemma 4.3 that the calls analysis must converge to a fixpoint by the third iteration in the Kleene ascending chain computation or else the fixpoint contains an element of a CST that is $TopAR$. We consequently define a widening operation (see § 4.5.1) to make the computation finite.

Definition 4.33 *The **widening operator for calls analysis**, $\overset{g}{\bigtriangledown}_r \in \mathsf{T}^* \times \mathsf{T}^* \mapsto \mathsf{T}^*$, is defined as the pointwise application of $\overset{a}{\bigtriangledown} r$ across corresponding elements in the two sequences of CSTs, x_r and y_r. Where one sequence is longer than the other, those CSTs are included in the same positions in the resulting sequence.*

Lemma 4.7 *The $\overset{g}{\bigtriangledown}_r$ operator is a widening operator in the sense of Defn 4.21.*

Proof. The proof is again simply by an examination of the definitions. □

Definition 4.34 *The **fixpoint computation** of the functional associated with each calls abstraction of a function is defined by the upward iteration sequence given in Defn 4.22 where the widening operator used is $\overset{g}{\bigtriangledown}_r$, which was presented in Defn 4.33.*

Lemma 4.8 *The fixpoint computation given in Defn 4.34 finds the least fixpoint of the relevant functional and computes it in finitely many steps.*

Proof. Again, by Lemma 4.3, Lemma 4.7 and [12]. □

4.7 Abstract Descent Property

We are now in a position to present an abstract property that will guarantee the termination of programs with EFP. The main concept is that, 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 a matrix that represents the sizes of arguments to the recursive calls of a function.

Firstly, we define a matrix that gives the relative abstract sizes of the arguments to all potential recursive calls of a given function.

Definition 4.35 *The* ***abstract calls matrix*** *of recursive calls of function* f_i *is defined thus:*

$$\mathbf{ACM}(i) \overset{\triangle}{=} \{\boldsymbol{r} \mid (\mathbf{v}, \boldsymbol{c}) \in f^{g}_{i[i]}\, \{\}\}$$

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

Lemma 4.9 *Let* $t_1 \ldots t_{\mathrm{Ar}(f_i)}$ *be arbitrary inputs to a function* f_i. *Then there exists a bijection between* $\mathbf{ACM}(i)$ *and* $\mathrm{Calls}(f_i\, t_1 \ldots t_{\mathrm{Ar}(f_i)})$ *where each row of a structure is mapped to the row with the same index in the other. Furthermore, each row of* $\mathbf{ACM}(i)$ *corresponds to the same program point as the corresponding row in* $\mathrm{Calls}(f_i\, t_1 \ldots t_{\mathrm{Ar}(f_i)})$.

Proof. This can be shown by consdiering program points with respect to the definitions of $\mathrm{Calls}(f_i\, t_1 \ldots t_{\mathrm{Ar}(f_i)})$ and $\mathbf{ACM}(i)$. □

Definition 4.36 *The* j*th argument to* f_i *(i.e.* $x_{i,j}$*) is said to be an* ***abstractly monotonic descending*** *argument, written* $\mathrm{AMD}(x_{i,j})$ *(or simply* $\mathrm{AMD}(j)$ *where the context is clear), if*

$$\forall \boldsymbol{r_k} \in \mathbf{ACM}(i).(r_{k,j} \leq 0) \wedge (\exists d. r_{d,j} < 0)$$

Definition 4.37 *A function* f_i *has the* ***abstract descent property****, denoted* $\mathrm{ADP}(A)$, *where* $A \equiv \mathbf{ACM}(i)$, *if and only if*

$$\exists j. \mathrm{AMD}(j) \wedge \mathrm{ADP}(A')$$

where $A' = \{\boldsymbol{r_e} \mid (\boldsymbol{r_e} \in A) \wedge (r_{e,j} = 0)\}$

Lemma 4.10 *Let* A *be the abstract calls matrix of a function* f_i *and suppose that* f_i *has the abstract descent property. Then if* A' *is an matrix formed by eliminating any number of rows from* A, $\mathrm{ADP}(A')$.

Proof. Follows directly from the definition. □

The above result means that if the abstract descent property holds for all recursive calls of a function then it holds for a subset of those calls.

Theorem 4.3 *A function* f_i *that has the abstract descent property has the monotonic descent property.*

Proof. The proof follows from the safety of the previous components of the analysis. □

Corollary 4.1 *Suppose the following of a function* f_i:

- f_i *is defined according to the rules of* EFP.
- *Apart from recursive calls of* f_i *(which may indirectly occur in functions called by* f_i*), the definition of* f_i *comprises only terminating constants and functions.*
- f_i *has the abstract descent property.*

Then f_i *terminates under the* EFP *reduction relation.*

Proof. By Theorems 3.1 and 4.3. □

4.8 ESFP^0

Our analysis, which can ensure termination, means that we can define an ESFP language thus:

Definition 4.38 *The **language*** ESFP^0 *consists of* EFP *together with a check that all definitions within a script have the abstract descent property. That is, for a script,* **S**.

$$\text{Accept}(\mathbf{S}, \text{ESFP}^0) \overset{\triangle}{\iff} \text{Accept}(\mathbf{S}, \text{EFP}) \wedge \forall i \in \mathbb{I}^{\mathbf{S}}_{f}.\text{ADP}(\mathbf{ACM}(f_i))$$

4.9 Examples

We now show that the above analysis is powerful enough to accept Ackerman's function (which we showed in Ex 3.1 had the monotonic descent property) and also the standard (naive) definition of the *qsort* function as being terminating on all type-correct and terminating arguments.

Example 4.1 [Ackerman's Function] The analysis of Ackerman's function (defined in Ex 3.1), which shows that ADP(**ACM**(*ack*)), proceeds as follows:

We refer to the clauses of the outer ***case*** expression as E' and of the inner ***case*** expression as E''.

We make the following definitions for environments of abstract expressions:

$$\begin{array}{lcl} \sigma & = & \{m' := \{m\}\} \\ \sigma' & = & \{m' := \{m\}, n' := \{n\}\} \end{array}$$

We also need to perform closure analysis for the three (recursive) applications that occur within the function definition.

$$\begin{array}{lcllr} \mathcal{C}[\![\, ack\, m' \,]\!]_{\{\},\sigma}\, \langle\{1\}\rangle & = & \mathcal{C}[\![\, ack \,]\!]_{\{\},\sigma}\, \langle\{m'\}, \{1\}\rangle & [\text{By } (5)] & \\ & = & \{(\{ack\}, \langle\{m'\}, \{1\}\rangle, \sigma)\} & [(2)] & (29) \\ \mathcal{C}[\![\, ack\, m' \,]\!]_{\{\},\sigma'}\, \langle\{ack\, m\, n'\}\rangle & = & \{(\{ack\}, \langle\{m'\}, \{ack\, m\, n'\}\rangle, \sigma')\} & [\text{Sim. to } (29)] & (30) \\ \mathcal{C}[\![\, ack\, m \,]\!]_{\{\},\sigma'}\, \langle\{n'\}\rangle & = & \{(\{ack\}, \langle\{m\}, \{n'\}\rangle, \sigma')\} & [\text{Sim. to } (29)] & (31) \end{array}$$

We assume the following abstractions of the + operator which has its standard recursive definition.

$$+^a_0\,\{\} = 0 \qquad [\text{From base case of when 2nd arg is 0}] \qquad (32)$$

$$+^a_1\,\{\} = \omega \qquad [\text{As 1st arg occurs in result and recursion is by 2nd arg}] \qquad (33)$$

$$+^a_2\,\{\} = -\omega \qquad [\text{As 2nd arg does not occur in the result}] \qquad (34)$$

The relevant applications of the abstract size operator are as follows:

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, m' \,]\!]_{\{\},\sigma} &= \mathcal{A}_{ack,1}[\![\, m \,]\!]_{\{\},\sigma} - 1 = 0 - 1 = -1 && \text{[(6)](35)}\\
\mathcal{A}_{ack,2}[\![\, m' \,]\!]_{\{\},\sigma} &= \mathcal{A}_{ack,2}[\![\, m \,]\!]_{\{\},\sigma} = -\omega && \text{[(6)](36)}\\
\mathcal{A}_{ack,0}[\![\, m' \,]\!]_{\{\},\sigma} &= \mathcal{A}_{ack,0}[\![\, m \,]\!]_{\{\},\sigma} = -\omega && \text{[(6)](37)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, 1 \,]\!]_{\{\},\sigma} &= -\omega && \text{[Lemma 4.1](38)}\\
\mathcal{A}_{ack,2}[\![\, 1 \,]\!]_{\{\},\sigma} &= -\omega && \text{[Lemma 4.1](39)}\\
\mathcal{A}_{ack,0}[\![\, 1 \,]\!]_{\{\},\sigma} &= 1 && \text{[(8)](40)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, m' \,]\!]_{\{\},\sigma'} &= -1 && \text{[As (35)](41)}\\
\mathcal{A}_{ack,2}[\![\, m' \,]\!]_{\{\},\sigma'} &= -\omega && \text{[As (36)](42)}\\
\mathcal{A}_{ack,0}[\![\, m' \,]\!]_{\{\},\sigma'} &= -\omega && \text{[As (37)](43)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, ack\, m\, n' \,]\!]_{\{\},\sigma'} &= \mathrm{ap}^a(\{ack\}, ack, 1, \langle\{m\},\{n'\}\rangle, \{\}, \sigma') && \text{[(10) and (31)]}\\
&= \omega && \text{[(62) below](44)}\\
\mathcal{A}_{ack,2}[\![\, ack\, m\, n' \,]\!]_{\{\},\sigma'} &= \omega && \text{[(63) below](45)}\\
\mathcal{A}_{ack,0}[\![\, ack\, m\, n' \,]\!]_{\{\},\sigma'} &= -\omega && \text{[(64) below](46)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, m \,]\!]_{\{\},\sigma'} &= 0 && \text{[(6)](47)}\\
\mathcal{A}_{ack,2}[\![\, m \,]\!]_{\{\},\sigma'} &= -\omega && \text{[(6)](48)}\\
\mathcal{A}_{ack,0}[\![\, m \,]\!]_{\{\},\sigma'} &= -\omega && \text{[(6)](49)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, n' \,]\!]_{\{\},\sigma'} &= -\omega && \text{[(6)](50)}\\
\mathcal{A}_{ack,2}[\![\, n' \,]\!]_{\{\},\sigma'} &= -1 && \text{[Sim. to (35)](51)}\\
\mathcal{A}_{ack,0}[\![\, n' \,]\!]_{\{\},\sigma'} &= -\omega && \text{[(6)](52)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, n+1 \,]\!]_{\{\},\sigma} &= -\omega && \text{[Lemma 4.1](53)}\\
\mathcal{A}_{ack,2}[\![\, n+1 \,]\!]_{\{\},\sigma} &= \omega && \text{[(33)](54)}\\
\mathcal{A}_{ack,0}[\![\, n+1 \,]\!]_{\{\},\sigma'} &= 0 && \text{[(32) and (34)](55)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,1}[\![\, ack\, m'\, 1 \,]\!]_{\{\},\sigma} &= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [-1, -\omega] && \text{[(10), (29), (35) and (38)]}\\
&= -1 * ack_1^a\,\{\} && \text{[Mult](56)}\\
\mathcal{A}_{ack,0}[\![\, ack\, m'\, 1 \,]\!]_{\{\},\sigma} &= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [-\omega, 1] + ack_0^a\,\{\} && \text{[(10), (29), (37) and (40)]}\\
&= ack_2^a * 1\,\{\} + ack_0^a\,\{\} && \text{[Mult](57)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_{ack,0}[\![\, ack\, m\, n' \,]\!]_{\{\},\sigma'} &= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [-\omega, -\omega] + ack_0^a\,\{\} && \text{[(10), (49) \& (52)]}
\end{aligned}$$

$$\begin{aligned}
&= ack_0^a\,\{\} && [\text{Mult}](58)\\
\mathcal{A}_{ack,1}\llbracket\, ack\; m\; n' \,\rrbracket_{\{\},\sigma'} &= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [0,-\omega] && [(10),\ (47)\ \&\ (50)]\\
&= ack_1^a\,\{\} * 0 && [\text{Mult}](59)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{A}_{ack,0}\llbracket\, ack\; m'\; (ack\; m\; n') \,\rrbracket_{\{\},\sigma'}\\
&\quad= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [-\omega, ack_0^a\,\{\}] \quad [(10),\ (30),\ (31),\ (43),\ (46)\ \&\ (58)\ ]\\
&\quad= ack_2^a\,\{\} * ack_0^a\,\{\} + ack_0^a\,\{\} \qquad [\text{Mult}] \qquad (60)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{A}_{ack,1}\llbracket\, ack\; m'\; (ack\; m\; n') \,\rrbracket_{\{\},\sigma'}\\
&\quad= [ack_1^a\,\{\}, ack_2^a\,\{\}] * [-1, ack_1^a\,\{\} * 0] \quad [(10),\ (30),\ (31),\ (41),\ (44)\ \&\ (59)\ ]\\
&\quad= (ack_1^a\,\{\} * -1) + (ack_2^a\,\{\} * (ack_1^a\,\{\} * 0)) \qquad [\text{Mult}] \qquad (61)
\end{aligned}$$

We need to compute the following instances of the abstract size operator:

$$\begin{aligned}
&\mathrm{ap}^a(\{ack\}, ack, 1, \langle\{m\},\{n'\}\rangle, \{\}, \sigma')\\
&\quad= [f_{ack,1}^a\,\{\}, f_{ack,2}^a\,\{\}] * [0,-\omega] \quad [\text{Defn 4.19, (47) \& (50)}]\\
&\quad= [f_{ack,1}^a\,\{\} * 0, -\omega] \qquad [\text{Mult}] \qquad (62)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{ap}^a(\{ack\}, ack, 2, \langle\{m\},\{n'\}\rangle, \{\}, \sigma')\\
&\quad= [f_{ack,1}^a\,\{\}, f_{ack,2}^a\,\{\}] * [-\omega,-1] \quad [\text{Defn 4.19, (48) \& (51)}]\\
&\quad= [-\omega, f_{ack,2}^a\,\{\} * -1] \qquad [\text{Mult}] \qquad (63)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{ap}^a(\{ack\}, ack, 0, \langle\{m\},\{n'\}\rangle, \{\}, \sigma')\\
&\quad= [f_{ack,1}^a\,\{\}, f_{ack,2}^a\,\{\}] * [-\omega,-\omega] + f_{ack,0}^a\,\{\} \quad [\text{Defn 4.19, (49) \& (52)}]\\
&\quad= f_{ack,0}^a\,\{\} \qquad [\text{Mult}] \qquad (64)
\end{aligned}$$

We need to compute the following relative size abstractions of the *ack* function:

$$\begin{aligned}
f_{ack,2}^a\,\{\} &= \mathrm{lfp}(F_{ack,2,\{\}}) && [\text{Defn 4.20}]\\
F_{ack,2,\{\}}(f_{ack,2}^a\,\{\}) &= \mathcal{A}_{ack,2}\llbracket\, \textbf{\textsf{case}}\; m\; \textbf{\textsf{of}}\, E' \,\rrbracket_{\{\},\{\}} && [\text{Defn 4.20}]\\
&= \max(\omega, \mathcal{A}_{ack,2}\llbracket\, E'' \,\rrbracket_{\{\},\sigma}) && [(9)\text{ and }(54)]\\
&= \omega && [\max] \qquad (65)
\end{aligned}$$

$$\begin{aligned}
f_{ack,0}^a\,\{\} &= \mathrm{lfp}(F_{ack,0,\{\}}) && [\text{Defn 4.20}]\\
F_{ack,0,\{\}}(f_{ack,0}^a\,\{\}) &= \mathcal{A}_{ack,0}\llbracket\, \textbf{\textsf{case}}\; m\; \textbf{\textsf{of}}\, E' \,\rrbracket_{\{\},\{\}} && [\text{Defn 4.20}]\\
&= \max(0, \mathcal{A}_{ack,0}\llbracket\, E'' \,\rrbracket_{\{\},\sigma}) && [(9)\text{ and }(55)]\\
\mathcal{A}_{ack,0}\llbracket\, E'' \,\rrbracket_{\{\},\sigma} &= \max(\mathcal{A}_{ack,0}\llbracket\, ack\; m'\; 1 \,\rrbracket_{\{\},\sigma}, && [(9)]
\end{aligned}$$

$$
\begin{array}{lcll}
 & & \mathcal{A}_{ack,0}\,\llbracket\, ack\, m'\,(ack\, m\, n')\,\rrbracket_{\{\},\sigma'}) & \\
 & = & \max((1 * ack^a_2\,\{\} + ack^a_0\,\{\}), & [(57) \text{ and } (60)] \\
 & & \quad (ack^a_1\,\{\} * 0 + ack^a_0\,\{\})) & \\
\mathrm{lfp}(F_{ack,0,\{\}}) & = & \omega & [\text{From } (65)] \qquad (66) \\
f^a_{ack,1}\,\{\} & = & \mathrm{lfp}(F_{ack,1,\{\}}) & [\text{Defn } 4.20] \\
F_{ack,1,\{\}}(f^a_{ack,1}\,\{\}) & = & \mathcal{A}_{ack,1}\,\llbracket\, \textbf{case}\; m\; \textbf{of}\, E'\,\rrbracket_{\{\},\{\}} & [\text{Defn } 4.20] \\
 & = & \max(-\omega, \mathcal{A}_{ack,1}\,\llbracket\, E''\,\rrbracket_{\{\},\sigma}) & [(9) \text{ and } (53)] \\
\mathcal{A}_{ack,1}\,\llbracket\, E''\,\rrbracket_{\{\},\sigma} & = & \max(\mathcal{A}_{ack,1}\,\llbracket\, ack\, m'\, 1\,\rrbracket_{\{\},\sigma}, & [(9)] \\
 & & \quad \mathcal{A}_{ack,1}\,\llbracket\, ack\, m'\,(ack\, m\, n')\,\rrbracket_{\{\},\sigma'}) & \\
 & = & \max((-1 * ack^a_1\,\{\}), & [(56) \text{ and } (61)] \\
 & & \quad ((ack^a_1\,\{\} * -1)+ & \\
 & & \quad (ack^a_2\,\{\} * (ack^a_1\,\{\} * 0)))) & \\
F^1_{ack,0,\{\}}(-\omega) & = & -\omega = F^0_{ack,0,\{\}}(-\omega) & [\text{Mult}]
\end{array}
$$

We consequently generate the following CSTs:

$$
\begin{array}{lcll}
\mathbf{T}(ack, ack\, m'\, 1, \{\}, \sigma) & = & \left(\begin{bmatrix} -1 & -\omega \\ -\omega & -\omega \end{bmatrix}, \begin{bmatrix} -\omega \\ 1 \end{bmatrix}\right) & (67) \\[2ex]
\mathbf{T}(ack, ack\, m'\,(ack\, m\, n'), \{\}, \sigma) & = & \left(\begin{bmatrix} -1 & -\omega \\ -\omega & \omega \end{bmatrix}, \begin{bmatrix} -\omega \\ \omega \end{bmatrix}\right) & (68) \\[2ex]
\mathbf{T}(ack, ack\, m\, n', \{\}, \sigma) & = & \left(\begin{bmatrix} 0 & -\omega \\ -\omega & -1 \end{bmatrix}, \begin{bmatrix} -\omega \\ -\omega \end{bmatrix}\right) & (69)
\end{array}
$$

We calculate the calls analysis of *ack* as follows:

$$
\begin{array}{lcll}
ack^g_{[ack]}\,\{\} & = & \mathcal{G}_{ack[ack]}\,\llbracket\, \textbf{case}\; m\; \textbf{of}\; E'\,\rrbracket_{\{\},\{\}} & [\text{Defn } 4.32] \\
 & = & \mathcal{G}_{ack[ack]}\,\llbracket\, E'\,\rrbracket_{\{\},\{\}} & [\text{Lemma } 4.6] \\
 & = & \biguplus \mathcal{G}_{ack[ack]}\,\llbracket\, ack\, m'\, 1\,\rrbracket_{\{\},\sigma} \mathcal{G}_{ack[ack]}\,\llbracket\, ack\, m'\,(ack\, m\, n')\,\rrbracket_{\{\},\sigma'} & [(25)] \\
 & = & \langle \mathbf{T}(ack, ack\, m'\, 1, \{\}, \sigma), \rangle & [(26),\ (29)\text{–}(31)\ \&\ \text{Defn } 4.31] \\
 & & \quad \mathbf{T}(ack, ack\, m'\,(ack\, m\, n'), \{\}, \sigma), & \\
 & & \quad \mathbf{T}(ack, ack\, m\, n', \{\}, \sigma) & \\
 & & & \qquad (70) \\
 & = & \langle (\begin{bmatrix} -1 & -\omega \\ -\omega & -\omega \end{bmatrix}, \begin{bmatrix} -\omega \\ 1 \end{bmatrix}) & \\
 & & \quad (\begin{bmatrix} -1 & -\omega \\ -\omega & \omega \end{bmatrix}, \begin{bmatrix} -\omega \\ \omega \end{bmatrix}) & \\
 & & \quad (\begin{bmatrix} 0 & -\omega \\ -\omega & -1 \end{bmatrix}, \begin{bmatrix} -\omega \\ -\omega \end{bmatrix}) \rangle &
\end{array}
$$

We finally have the following result for $\mathbf{ACM}(ack)$ (an instance of Defn 4.35):

$$
\mathbf{ACM}(ack) = \begin{bmatrix} -1 & 1 \\ -1 & \omega \\ 0 & -1 \end{bmatrix}
$$

◇

Example 4.2 [Quicksort] The quicksort ($qsort$) function is defined as follows:

$$
\begin{array}{lll}
y@[\,] +\!\!+\, z & \overset{def}{=} & z \\
y@(f : r) +\!\!+\, z & \overset{def}{=} & f : (r +\!\!+\, z) \\[1ex]
\mathit{filter}\; p\, m@[\,] & \overset{def}{=} & [\,] \\
\mathit{filter}\; p\, m@(h : t) & & \\
\quad \mid p\, h & \overset{def}{=} & h : \mathit{filter}\, p\, t \\
\quad \mid \textbf{\textit{otherwise}} & \overset{def}{=} & \mathit{filter}\, p\, t \\[1ex]
\mathit{qsort}\ l@[\,] & \overset{def}{=} & [\,] \\
\mathit{qsort}\ l@(a : x) & \overset{def}{=} & s +\!\!+\, [a] +\!\!+\, b \\
\quad \textbf{\textit{where}} & & \\
\qquad s \overset{def}{=} \mathit{qsort}\,(\mathit{filter}\,(\leq a)\, x) & & \\
\qquad b \overset{def}{=} \mathit{qsort}\,(\mathit{filter}\,(> a)\, x) & &
\end{array}
$$

The analysis of $qsort$ proceeds as follows: The fact that $qsort$ has the abstract descent property follows from the following:

$$
\begin{array}{lll}
\mathit{filter}^{a}_{2}\,\{p := \{(\leq a)\}\} & = & 0 \\
\mathit{filter}^{a}_{2}\,\{p := \{(\leq a)\}\} & = & 0
\end{array}
$$

We get also, for the analysis for the list input and constant factors, respectively,

$$
\begin{array}{lll}
[\omega, 0] * [-\omega, -1] & = & -1 \\
[\omega, 0] * [-\omega, -1] + 0 & = & 0
\end{array}
$$

Consequently, $\mathbf{ACM}(\mathit{qsort}) = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$.

◇

5 Adding Subtyping

As we have seen, the above analysis is powerful enough to show that 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 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 now proceed to give an overview of how the analysis is modified.

- 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.

- As well as a set of CSTs, our new analyses, modified for subtyping, need to indicate whether or not function applications have been at the correct subtypes. This can be done by pairing the result with a Boolean flag to indicate the subtype-correctness of each application or, as we have chosen, to return the top of the CST domain ($\top_{\mathrm{T}}$ as the result if a function does not have the abstract descent property for the subtypes of the arguments to which it is applied.

- 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,

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

  where $G_k = \begin{cases} \mathcal{G}^1_{i[j,\phi_j]}\, [\![\, e_k \,]\!]^{\phi_k}_{\sigma_k,\rho} & \text{if } H(p_k) \in \mathcal{S}\, [\![\, s \,]\!]^{\phi_k}_{\rho,\sigma} \\ \{\} & \text{otherwise} \end{cases}$

  Here, $H(p_k)$ is the head constructor of the pattern p_k and $\mathcal{S}\, [\![\, s \,]\!]^{\phi_k}_{\rho,\sigma}$, 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 whilst for an inductive case constructor, such as ***Succ*** it must be at least 1. Thus, if a base constructor results for a function then it represents size descent from an input $x_{i,j}$ that is assumed to reduce to an expression with an inductive case constructor, such as ***Succ***.

5.1 An Elementary Functional Language with Explicitly Undefined Values

We extend the algorithms permitted in the system that we are developing by introducing an explicit undefined value ***error*** into our language. This is the counterpart of the the error expressions of Miranda or uncaught exceptions in SML which produce a runtime error together with a diagnostic. However, the point of the ***error*** construct is that it indicates a clause that should never be reached and it is up to the analyser to check that it is impossible for the program to evaluate to that program point. In that sense, when the termination analysis described below has been performed to ensure that a function will terminate, the ***error*** expressions correspond to the *abort* construct that appears in Martin-Löf's type theory — *abort* expressions only appear so as to adhere to the *principle of complete presentation* [33].

Definition 5.1 *For each type, A, there is an* **error**$_A$ ***expression*** *that does not have any associated reduction rules. The semantics of expressions involving* **error**$_A$ *(which we do not give here explicitly) corresponds to the semantics of exceptions in a strict language such as ML [24].*

Generally, we write **error** *when the context is clear or irrelevant.*

Consequently, we define a new variant of our EFP language.

Definition 5.2 *The* EFP^e *language consists of the EFP language together with the addition of* **error** *expressions. If a script,* $\mathbf{S}$*, meets the criteria of* EFP^e *then we write* $\mathrm{Accept}(\mathbf{S}, \mathrm{EFP}^e)$.

5.1.1 The abstract semantics of *error*.

In the abstract analyses which follow below § 5.3–5.4 we do not give the abstract semantics for the ***error*** construct since in each case it is the $\top$ of the relevant domain. For size analysis (see Defn 5.14), for example, $\mathcal{A}^1_{i,j}\,[\![\, \textit{\textbf{error}} \,]\!]^{\phi}_{\rho,\sigma} \overset{\triangle}{=} \omega$.

5.2 The Abstract Subtyping Domain

Definition 5.3 *Let T be an algebraic type in our basic ESFP language. Then the* ***abstract subtyping domain*** *for T, denoted S_T, is defined as, $\mathsf{S}_T \overset{\triangle}{=} \wp(Cs_T)$. For non-algebraic types, S_T is $\{\{\}\}$.*

We normally write S instead of S_T where the type is either clear from, or irrelevant to, the context.

The concretisation of such abstract values is straightforward.

Definition 5.4 *The concretisation of elements of the abstract subtyping domain is defined via the mapping $\gamma_{\mathsf{S}_T} \in \mathsf{S}_T \mapsto \mathbb{P}_{\mathbb{A}}$, so that for $s \in \mathsf{S}_T$*

$$\gamma_{\mathsf{S}_T} s \overset{\triangle}{=} \{v \mid (v \equiv C_i v_1 \ldots v_{\mathrm{Ar}(C_i)}); \forall j.\mathrm{nf}(v_j); C_i \in s\} \cup \{\bot\}$$

We also write $\mathsf{Env}(\mathsf{S})$ to mean the environment where each $x_{i,j}$ is bound to elements of the appropriate subtyping domain. Since such environments are used to constrain the domain over which a function may terminate, we now define them more fully.

Definition 5.5 *A **subtyping environment** for a function f_i is an environment, ϕ, in which each $x_{i,j}$ is bound to an element of $\mathsf{S}_{\mathrm{T}(x_{i,j})}$.*

*A **valid subtyping environment** is a subtyping environment, ϕ, in which each $x_{i,j}$ of* algebraic type *is bound to a non-empty value. We write* $\mathrm{ValidSub}(\phi)$.

Definition 5.6 *Let ϕ_1 and ϕ_2 be two subtyping environments of function f_i. Then the **join** of ϕ_1 and ϕ_2, denoted $\phi_1 \sqcup \phi_2$, is defined thus:*

$$\phi_1 \sqcup \phi_2 \stackrel{\triangle}{=} \{x_{i,j} \mapsto \phi_1(x_{i,j}) \cup \phi_2(x_{i,j}) \mid x_{i,j} \in \mathrm{Dom}(\phi_1).\}$$

*Similarly the **meet**, denoted $\phi_1 \sqcap \phi_2$, is defined,*

$$\phi_1 \sqcap \phi_2 \stackrel{\triangle}{=} \{x_{i,j} \mapsto \phi_1(x_{i,j}) \cap \phi_2(x_{i,j}) \mid x_{i,j} \in \mathrm{Dom}(\phi_1).\}$$

Since we need to determine whether a recursive call of a function has been matched with the correct subtypes, we need to acertain whether a subtyping environment includes the one we are trying to match.

Definition 5.7 *A **sub-subtyping environment** (often written simply as sub-environment where the meaning is clear) of a subtyping environment of a function f_i, ϕ, is a subtyping environment, ϕ', for which, $\forall j.\phi'(x_{i,j}) \subseteq \phi(x_{i,j})$. We denote the fact that ϕ is a sub-subtyping environment by $\phi' \sqsubseteq \phi$.*

*Conversely, we also use the term, **super-subtyping environment**.*

If we one environment does include another we still need to perform an analysis on the subtyping environment that lies outside the intersection.

Definition 5.8 *The **sub-environment difference** between two environments, ϕ and ϕ', where $\phi' \sqsubseteq \phi$, and denoted $\phi - \phi'$ is defined as the set difference upon corresponding bindings in the two environments i.e.*

$$\phi - \phi' \stackrel{\triangle}{=} \{x_{i,j} \mapsto \phi(x_{i,j}) - \phi'(x_{i,j})\}$$

In our actual analyses we only take one constructor per algebraic argument in our subtyping environments and then join the results on each of these sub-environments to determine the subtyping environment over which the function is defined.

Definition 5.9 *Let ϕ be a subtyping environment. Then the **singleton partition** of ϕ, denoted* $\mathrm{SP}(()\phi)$ *consists of all the sub-subtyping environments containing only singleton sets as algebraic subtypes.*

$$
\begin{aligned}
&\mathcal{S}\,[\![\, x \,]\!]^{\phi}_{\rho,\sigma} && \triangleq \begin{cases} \phi(x) & \text{if } x \in \mathrm{Dom}(\phi) \\ \top_{\mathsf{S}} & \text{otherwise} \end{cases} && (71)\\
&\mathcal{S}\,[\![\, f_i \,]\!]^{\phi}_{\rho,\sigma} && \triangleq \begin{cases} f_i^s\,\{\}\,\{\} & \text{if } \mathrm{Ar}(f_i) = 0 \\ \{\} & \text{otherwise} \end{cases} && (72)\\
&\mathcal{S}\,[\![\, C_t\, a_1 \ldots a_r \,]\!]^{\phi}_{\rho,\sigma} && \triangleq \{C_t\} && (73)\\
&\mathcal{S}\,[\![\, \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r\rangle \,]\!]^{\phi}_{\rho,\sigma} && \triangleq \bigcup_{i=1}^{i=n} \bigcup \{\mathcal{S}\,[\![\, e_i \,]\!]^{\phi}_{\rho,\sigma_i} \mid H(p_i) \in \mathcal{S}\,[\![\, s \,]\!]^{\phi}_{\rho,\sigma}\} && (74)\\
&\mathcal{S}\,[\![\, F\, a \,]\!]^{\phi}_{\rho,\sigma} && \triangleq \bigcup \{f_k^s\, \rho'\, \phi' \mid (f_k, \boldsymbol{a}, \sigma') \in \mathcal{C}^1\,[\![\, F \,]\!]^{\phi}_{\rho,\sigma}\, \sigma\rho\phi\langle a\rangle\} && (75)
\end{aligned}
$$

Table 5: Definition of $\mathcal{S}\,[\![\, E \,]\!]^{\phi}_{\rho,\sigma}$

5.3 The Analysis of Subtypes

We now describe an analysis which safely approximates the subtype of any algebraic expression within the language. Firstly, we need to define how subtypes match a pattern in a ***case*** expression.

Definition 5.10 *$H(p_i)$ is the* ***head constructor*** *of the pattern p_i and is defined as $H(C_t\, a_1 \ldots a_r) \triangleq C_t$.*

Definition 5.11 *The* ***analysis of subtypes operator****, $\mathcal{S} \in \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{S}$, is presented in Table 5.*

5.4 Modified Termination Analyses

We now give definitions that are analagous to those in § 4.

5.4.1 Closure analysis with subtyping.

Definition 5.12 *The* ***closure analysis with subtyping*** *semantic operator, $\mathcal{C}^1 \in \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \mathit{Env}(\mathsf{S}) \times \mathsf{E}^* \mapsto \mathsf{C}$, is defined in Table 6.*

Definition 5.13 *The* ***abstract closure function with subtyping environment****, ϕ of a function, $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$, is defined for a given environment of non-ground expressions ρ, and a sequence of actual parameter expressions, $\boldsymbol{a}$, as $f_i^{m_1}\, \rho\, \phi\, \boldsymbol{a} \triangleq \bigcup_{\phi' \in \mathrm{SP}(()\phi)} \mathcal{C}^1\,[\![\, e_i \,]\!]^{\phi'}_{\rho,\{\}}\, \boldsymbol{a}$*

As would be expected, subtyping produces more precise results than for the basic analysis without subtyping.

$$\mathcal{C}^1\,[\![\,x\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \triangleq \begin{cases} \top_{\mathsf{C}} & \text{if } \rho(x) = \top_{\mathsf{E}} \vee \sigma(x) = \top_{\mathsf{E}} \\ \{(\{\},\boldsymbol{a},\sigma)\} & \text{if } \rho(x) = \{\} \\ \mathcal{C}^1\,[\![\,e\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} & \text{if } \rho(x) = \{e\} \vee \sigma(x) = \{e\} \end{cases} \tag{76}$$

$$\mathcal{C}^1\,[\![\,f_i\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \triangleq \begin{cases} \{(\{f_i\},\boldsymbol{a},\sigma)\} & \text{if } \mathrm{Ar}(f_i) \geq |\boldsymbol{a}| \\ \{(f,\boldsymbol{e},\sigma'') \mid (f,\boldsymbol{d},\sigma') \in f_i^{m_1}\,\rho'\,\phi'\,\boldsymbol{c}\} & \text{otherwise} \end{cases} \tag{77}$$

$$\mathcal{C}^1\,[\![\,C_t\,a_1\ldots a_r\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \triangleq \bigcup_{i=1}^{i=r}\{(f,\boldsymbol{b},\sigma') \mid (f,\boldsymbol{b},\sigma') \in \mathcal{C}^1\,[\![\,e_i\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \wedge \mathrm{TC}(f,\boldsymbol{b})\} \tag{78}$$

$$\mathcal{C}^1\,[\![\,\textbf{\textit{case}}\; s\; \textbf{\textit{of}}\;\langle p_r, e_r\rangle\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \triangleq \bigcup_{k=1}^{k=r}\bigcup\{\mathcal{C}^1\,[\![\,e_k\,]\!]^{\phi}_{\sigma_k,\rho}\,\boldsymbol{a} \mid H(p_k) \in \mathcal{S}\,[\![\,s\,]\!]^{\phi}_{\rho,\sigma}\} \tag{79}$$

$$\mathcal{C}^1\,[\![\,G\,d\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \triangleq \mathcal{C}^1\,[\![\,G\,]\!]^{\phi}_{\rho,\sigma}\,(\langle\{d\}\rangle +\!\!+\, \boldsymbol{a}) \tag{80}$$

Auxiliary definitions are as in Table 2.

Table 6: Definition of $\mathcal{C}^1\,[\![\,E\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a}$

Lemma 5.1 *For any well-formed basic ESFP expression, e, with well-formed function environment ρ, well-formed pattern-matching variable expression environment, σ, well-formed subtyping environments, ϕ and well-formed abstract expression vector, $\boldsymbol{a}$,*

$$\mathcal{C}^1\,[\![\,e\,]\!]^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \subseteq \mathcal{C}\,[\![\,e\,]\!]_{\rho,\sigma}\,\boldsymbol{a}$$

Proof. By inspection of the definitions, in particular the clauses for ***case*** expressions. □

Corollary 5.1 *The abstract closure function with subtyping, ϕ, is more precise than the abstract closure function without subtyping.*

Proof. Follows from Defn 5.13. □

5.4.2 Size analysis with subtyping.

Definition 5.14 *The* ***relative size analysis operator with subtyping****, $\mathcal{A}^1 \in \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{R}$, is the $\mathcal{A}$ operator extended with subtyping and defined over the structure of expressions in Table 7 with auxiliary definitions given below. In the definition, ρ is an environment binding function type expressions to variables, σ is an environment binding pattern-matching variables of algebraic types to expressions, and ϕ is an environment binding subtypes to the formal parameters. i is a function index whilst $0 \leq j \leq \mathrm{Ar}(f_i)$.*

Definition 5.15 *The* ***constructor abstract size function with subtyping****, $\mathrm{cs}^1 \in \wp(\mathbb{E}) \times \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathit{Env}(\mathsf{E}) \times \mathit{Env}(\mathsf{S})\mathsf{M} \mapsto \mathsf{R}$, is defined analagously to Defn 4.17, with $\mathcal{A}^1$ replacing $\mathcal{A}$.*

$$\mathcal{A}^1_{i,j}\llbracket\, x \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \begin{cases} 0 & \text{if } x \equiv x_{i,j} \\ -\omega & \text{if } x \equiv x_{i,k} \\ \mathcal{A}^1_{i,j}\llbracket\, t \,\rrbracket^{\phi}_{\rho,\sigma} - 1 & \text{if } \sigma(x) = \{t\} \wedge \mathrm{Unify}(x,t) \\ \omega & \text{otherwise} \end{cases} \tag{81}$$

$$\mathcal{A}^1_{i,j}\llbracket\, f_k \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \begin{cases} f^{a}_{k,0}\,\{\}\,\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge j = 0 \\ -\omega & \text{otherwise} \end{cases} \tag{82}$$

$$\mathcal{A}^1_{i,j}\llbracket\, C_t\, a_1 \ldots a_r \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \mathrm{cs}^1(\mathrm{Rec}(E), i, j, \rho, \sigma, \phi) \tag{83}$$

$$\mathcal{A}^1_{i,j}\llbracket\, \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r\rangle \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \max(\bigcup_{k=1}^{k=r} \{\mathcal{A}^1_{i,j}\llbracket\, e_k \,\rrbracket^{\phi}_{\rho,\sigma_k} \mid H(p_k) \in \mathcal{S}\llbracket\, s \,\rrbracket^{\phi}_{\rho,\sigma}\}) \tag{84}$$

$$\mathcal{A}^1_{i,j}\llbracket\, F\, a \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \max\{\mathrm{ap}^{a_1}(f, i, j, \boldsymbol{a}, \rho, \sigma, \phi) \mid (f, \boldsymbol{a}, \sigma_k) \in \mathcal{C}^1\llbracket\, F \,\rrbracket^{\phi}_{\sigma,\rho}\,\langle\{a\}\rangle\} \tag{85}$$

In addition, if $\neg\mathrm{ADP}(k, \phi')$, then $\boldsymbol{f^{r}_{k,\phi'} a^{s}} \triangleq \omega$.

Table 7: Definition of $\mathcal{A}^1_{i,j}\llbracket\, E \,\rrbracket^{\phi}_{\rho,\sigma}$

Definition 5.16 *The $\mathcal{A}^1$ operator is lifted to the E domain as follows:*

$$\mathcal{A}^1_{i,j}\llbracket\, \top_{\mathsf{E}} \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \omega \tag{86}$$

$$\mathcal{A}^1_{i,j}\llbracket\, \{e\} \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \mathcal{A}^1_{i,j}\llbracket\, e \,\rrbracket^{\phi}_{\rho,\sigma} \tag{87}$$

Definition 5.17 *The* ***abstract applicator for size analysis with subtyping****,* ap^{a_1}*, is defined as follows.*

$$\mathrm{ap}^{a_1}(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \sigma, \rho, \phi) \triangleq \omega \tag{88}$$

$$\mathrm{ap}^{a_1}(\{\}, i, j, \boldsymbol{a}, \sigma, \rho, \phi) \triangleq \omega \tag{89}$$

$$\mathrm{ap}^{a_1}(\{f_k\}, i, j, \boldsymbol{a}, \sigma, \rho, \phi) \triangleq (\boldsymbol{f_k}^{a_1} * \boldsymbol{a}^{a_1}) + vj \tag{90}$$

In the above, $\phi' \triangleq \{x_{k,1} := \mathcal{S}\llbracket\, a_1 \,\rrbracket^{\phi}_{\rho,\sigma} \ldots x_{k,\mathrm{Ar}(f_k)} := \mathcal{S}\llbracket\, a_{\mathrm{Ar}(f_k)} \,\rrbracket^{\phi}_{\rho,\sigma}\}$*. In addition,* $\phi'\{x_{k,l} := \top_{\mathsf{S}}\}$*, if* $l > |\boldsymbol{a^s}|$.

$\boldsymbol{f_k^{a_1}} \triangleq [f^{a_1}_{k,1}\,\rho'\,\phi' \ldots f^{a_1}_{k,\mathrm{Ar}(f_k)}\,\rho'\,\phi']$ *and* $\boldsymbol{a^{a_1}} \triangleq [\mathcal{A}^1_{i,j}\llbracket\, a_1 \,\rrbracket^{\phi}_{\rho,\sigma} \ldots \mathcal{A}^1_{i,j}\llbracket\, a_{|\boldsymbol{a}|} \,\rrbracket^{\phi}_{\rho,\sigma}]$.

$$v_j \triangleq \begin{cases} f^{a_1}_{k,0}\,\rho'\,\phi' & \textit{if } j = 0 \\ -\omega & \textit{otherwise} \end{cases}$$

Definition 5.18 *The* ***abstract size function with subtyping*** *of a function,* $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$*,* ***relative to parameter*** j *is defined for a given subtyping environment,* ϕ_i *and a given environment of function-type parameters,* ρ *as,* $f^{a_1}_{i,j}\,\phi_i\,\rho \triangleq \max_{\phi' \in \mathrm{SP}(()\phi_i)} \mathcal{A}^1_{i,j}\llbracket\, e_i \,\rrbracket^{\phi'}_{\rho,\{\}}$

Again, subtyping produces more precise results for size analysis than for the basic analysis without subtyping.

Lemma 5.2 *For any well-formed basic ESFP expression, e, with well-formed function environment ρ, well-formed pattern-matching variable expression environment, σ, and well-formed subtyping environment, ϕ,*

$$\mathcal{A}^1_{i,j} [\![e]\!]^{\phi}_{\rho,\sigma} \leq_{\mathsf{R}} \mathcal{A}_{i,j} [\![e]\!]_{\rho,\sigma}$$

Proof. Again, by inspection of the definitions. □

Corollary 5.2 *The abstract size function with subtyping, ϕ, is more precise than the abstract size function without subtyping.*

Proof. Follows from Defn 5.18. □

Definition 5.19 *The **abstract size vector** of an expression e, with respect to the environments of function expressions, ρ, pattern matching expressions, σ, and subtypes, ϕ, is defined as follows:*

$$\boldsymbol{s}(e, i, \sigma, \rho, \phi) \triangleq \begin{bmatrix} \mathcal{A}^1_{i,1} [\![e]\!]^{\phi}_{\rho,\sigma} \\ \vdots \\ \mathcal{A}^1_{i,\mathrm{Ar}(f_i)} [\![e]\!]^{\phi}_{\rho,\sigma} \end{bmatrix}$$

Definition 5.20 *The* abstract interpretation *of relative sizes over expressions is defined by the **component size semantics** of an expression, e, with respect to a parameter, $x_{i,j}$ and a subtyping environment, ϕ: $\mathcal{R}^{\#} [\![e]\!]_{i,j} \triangleq \lambda \mathsf{Env}(E).(\boldsymbol{w_j} \boldsymbol{s}(e, i, \{\}, \{\}, \phi))$ where juxtaposition indicates vector product.*

Definition 5.21 *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, σ and ϕ (an environment of subtypes) are denoted $\boldsymbol{c}(i, \boldsymbol{e}, \sigma, \rho, \phi)$ and $\mathbf{v}(i, \boldsymbol{e}, \sigma, \rho, \phi)$, respectively, and defined as follows:*

$$\boldsymbol{c}(i, \boldsymbol{e}, \sigma, \rho, \phi) \triangleq \begin{bmatrix} \mathcal{A}^1_{i,0} [\![e_1]\!]^{\phi}_{\rho,\sigma} \\ \vdots \\ \mathcal{A}^1_{i,0} [\![e_{|\boldsymbol{e}|}]\!]^{\phi}_{\rho,\sigma} \end{bmatrix} \mathbf{v}(i, \boldsymbol{e}, \sigma, \rho, \phi) \triangleq \begin{bmatrix} \mathcal{A}^1_{i,1} [\![e_1]\!]^{\phi}_{\rho,\sigma} & \cdots & \mathcal{A}^1_{i,\mathrm{Ar}(f_i)} [\![e_1]\!]^{\phi}_{\rho,\sigma} \\ \vdots & & \vdots \\ \mathcal{A}^1_{i,1} [\![e_n]\!]^{\phi}_{\rho,\sigma} & \cdots & \mathcal{A}^1_{i,\mathrm{Ar}(f_i)} [\![e_n]\!]^{\phi}_{\rho,\sigma} \end{bmatrix}$$

Definition 5.22 *The **component size transformation** (CST) for a sequence of expressions, $\boldsymbol{e}$, and with respect to the parameters of function f_i and environments, ρ, σ and ϕ (an environment of subtypes) is defined: $\mathbf{T}(i, \boldsymbol{e}, \sigma, \rho, \phi) \triangleq (\mathbf{v}(i, \boldsymbol{e}, \sigma, \rho, \phi), \boldsymbol{c}(i, \boldsymbol{e}, \sigma, \rho, \phi))$ If $(\mathbf{V_1}, \boldsymbol{k_1}), (\mathbf{V_2}, \boldsymbol{k_2})$ are CSTs then, if the relevant matrix multiplications are defined,*

$$(\mathbf{V_1}, \boldsymbol{k_1}) \star (\mathbf{V_2}, \boldsymbol{k_2}) \triangleq (\mathbf{V_1}\mathbf{V_2}, (\mathbf{V_1}\boldsymbol{k_2} + \boldsymbol{k_1}))$$

$$\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, x \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \langle\rangle \tag{91}$$

$$\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, f_k \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \begin{cases} f^{g}_{k[j]}\,\{\}\,\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k \neq j \\ \langle \boldsymbol{\Omega} \rangle & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k = j \\ \{\} & \text{otherwise} \end{cases} \tag{92}$$

$$\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, C_t\, a_1 \ldots a_r \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, a_k \,\rrbracket^{\phi}_{\rho,\sigma} \tag{93}$$

$$\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, \textbf{case}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r\rangle \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \biguplus(\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, s \,\rrbracket^{\phi}_{\rho,\sigma}, (\biguplus_{k=1}^{k=r} G_k)) \tag{94}$$

$$\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, F\, a \,\rrbracket^{\phi}_{\rho,\sigma} \triangleq \biguplus_{(f,\boldsymbol{a},\sigma')\in\mathcal{C}^1\,\llbracket\, F \,\rrbracket^{\phi}_{\rho,\sigma}\,\langle\{a\}\rangle} (\mathrm{ap}^{g_1}(f, i, j, \boldsymbol{a}, \sigma', \rho, \phi) \biguplus(\biguplus_{i=1}^{i=|\boldsymbol{a}|} \mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, a_i \,\rrbracket^{\phi}_{\rho,\sigma})) \tag{95}$$

In (94), $G_k = \begin{cases} \mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, e_k \,\rrbracket^{\phi}_{\sigma_k,\rho} & \text{if } H(p_k) \in \mathcal{S}\,\llbracket\, s \,\rrbracket^{\phi}_{\rho,\sigma} \\ \{\} & \text{otherwise} \end{cases}$

Table 8: Definition of $\mathcal{G}^1_{i[j,\phi_j]}\,\llbracket\, E \,\rrbracket^{\phi}_{\rho,\sigma}$

5.4.3 Calls analysis with subtyping.

Definition 5.23 *The* ***abstract calls operator****,* $\mathcal{G}^1 \in \mathbb{I}^{\mathsf{S}}_f \times \mathbb{I}^{\mathsf{S}}_f \times \mathit{Env}(\mathsf{S}) \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{T}^*$*, is the* $\mathcal{G}$ *operator extended with subtyping to locate calls of function* f_j *with subtype environment* ϕ_j *within function* f_i *which has input subtype environment* ϕ_i*. It is defined over the structure of expressions in Table 8.*

Definition 5.24 *The* ***abstract applicator for calls analysis with subtyping****,* $\mathrm{ap}^{g_1} \in \mathsf{F} \times \mathbb{I}^{\mathsf{S}}_f \times \mathbb{I}^{\mathsf{S}}_f \times \mathsf{E}^* \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \mathit{Env}(\mathsf{S}) \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{T}^*$*, is defined as follows*

$$\mathrm{ap}^{g_1}(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \rho, \sigma, \phi, \phi_j) \triangleq \langle \top_{\mathsf{T}} \rangle \tag{96}$$

$$\mathrm{ap}^{g_1}(\{\}, i, j, \boldsymbol{a}, \rho, \sigma, \phi, \phi_j) \triangleq \langle\rangle \tag{97}$$

$$\mathrm{ap}^{g_1}(\{f_k\}, i, j, \boldsymbol{a}, \rho, \sigma, \phi, \phi_j) \triangleq \begin{cases} \langle\rangle & \text{if } (|\boldsymbol{a}| < \mathrm{Ar}(f_k)) \\ \{\mathbf{T}(i, \boldsymbol{a}, \rho, \sigma, \phi)\} \cup R & \text{if } (f_k \equiv f_j) \wedge \phi_j \sqsubseteq \phi \\ \langle \top_{\mathsf{T}} \rangle & \textit{if}\,(f_k \not\equiv f_j) \vee (\phi_j \not\sqsubseteq \phi) \\ & \quad \wedge \neg \mathrm{ADP}(f_k, \phi') \\ \biguplus_{\mathbf{T}' \in f^{g}_{k[j]}\,\rho_k\,\phi_k} (\mathrm{Map}\,(\star\mathbf{T}(i, \boldsymbol{a}, \rho, \sigma, \phi))\,\mathbf{T}') & \text{if } f_k \not\equiv f_j \end{cases} \tag{98}$$

In the above, if $\phi'' = \phi - \phi_j$ *is a valid subtyping environment then* $R \equiv \mathrm{ap}^{g_1}(\{f_k\}, i, \boldsymbol{a}, \sigma, \rho, \phi'', \phi_j)$*. Otherwise,* $R \equiv \langle\rangle$

Definition 5.25 *For each function, there is a family of* ***abstract calls functions*** *which give the CSTs for the recursive calls of function* f_j *with subtyping environment* ϕ_j *within* *the definition of function f_i for subtyping environment ϕ_i and environment of function-type arguments, ρ:*

$$f^{g_1}_{i[j,\phi_j]}\,\rho\,\phi_i \triangleq \bigcup_{\phi'\in \mathrm{SP}(()\phi_i)} \mathcal{G}^1_{i[j,\phi_j]}\,[\![\, e_i \,]\!]^{\phi'}_{\rho,\{\}}$$

Note that now it is not only necessary to scan for occurrences of f_j within the definition of f_i but that the occurrences of f_j must occur at the stipulated subtyping environment. Furthermore, the search for occurrences of f_j is directed by the subtyping environment, ϕi i.e. the given subtypes of the parameters of f_i.

Again, subtyping produces more precise results for the calls operator than for the basic analysis without subtyping.

Lemma 5.3 *For any well-formed basic ESFP expression, e, with well-formed function environment ρ, well-formed pattern-matching variable expression environment, σ, and well-formed subtyping environment, ϕ,*

$$\mathcal{G}^1_{i[j,\phi_j]}\,[\![\, e \,]\!]^{\phi}_{\rho,\sigma} \leq_{\mathsf{R}} \mathcal{G}_{i[j]}\,[\![\, e \,]\!]_{\rho,\sigma}$$

Proof. Again, by inspection of the definitions. □

Corollary 5.3 *The abstract calls function with subtyping, ϕ, is more precise than the abstract calls function without subtyping.*

Proof. Follows from Defn 5.25. □

5.5 Termination Criteria Using Subtyping

As mentioned at the beginning of this section, we first need to refine our idea of a weighting vector to take account of the fact that subtypes give information as to the size of each input.

Definition 5.26 *Let C_t be some constructor. Then the* ***minimal size*** *of an expression that has C_t at its head is denoted* $\mathrm{mcs} \in \mathrm{C} \mapsto \mathsf{R}$*, is defined thus:*

$$\mathrm{mcs}(C_t) \triangleq \begin{cases} 0 & \textit{if } C_t \textit{ is a base case constructor} \\ 1 & \textit{if } C_t \textit{ is an inductive case constructor} \end{cases}$$

Definition 5.27 *Assume we have $s \in \mathsf{S}$. Then the* ***minimal subtype size*** *of s, denoted* $\mathrm{mss} \in \mathsf{S} \mapsto \mathsf{R}$*, is defined thus for non-empty s:*

$$\mathrm{mss}(s) \triangleq \min\{\mathrm{mcs}(C_t) \mid C_t \in s\}$$

For empty s (i.e. non-algebraic arguments), $\mathrm{mss}(\{\}) \triangleq -\omega$.

Definition 5.28 *The* ***jth weighting vector with respect to a subtyping environment*** *ϕ is a vector with, in the jth position,* $\mathrm{mss}(\phi(x_{i,j})$*. ω is in all other positions.*

Definition 5.29 *The* ***abstract call matrix*** *of recursive calls of function f_i is defined with respect to a subtyping environment, ϕ, thus:*

$$\mathbf{ACM}(i,\phi) \overset{\triangle}{=} \{\boldsymbol{r} \mid (\mathbf{v}, \boldsymbol{c}) \in f^{g_1}_{i[i,\phi]}\,\{\}\,\phi\}$$

where, if $x_{i,j}$ is an algebraic argument, $r_j \overset{\triangle}{=} \boldsymbol{w_j}\mathbf{v}_j + c_j - \mathrm{mss}(\phi(x_{i,j}))$, $\boldsymbol{w_j}$ is the jth weighting vector with respect to the subtyping environment ϕ and $\mathbf{v}_j$ is the jth column of $\mathbf{v}$.

If $x_{i,j}$ is non-algebraic then $r_j \overset{\triangle}{=} \omega$.

Definition 5.30 *The jth argument to f_i (i.e. $x_{i,j}$) with subtyping environment ϕ is said to be an* ***abstractly monotonic descending*** *argument, written $\mathrm{AMD}(x_{i,j}, \phi)$ (or simply $\mathrm{AMD}(j, \phi)$ where the context is clear), if*

$$\forall \boldsymbol{r_k} \in \mathbf{ACM}(i,\phi).(r_{k,j} \leq 0) \wedge (\exists d. r_{d,j} < 0)$$

The jth argument is said to be ***abstractly strictly descending****, written $\mathrm{ASD}(x_{i,j}, \phi)$ if*

$$\forall \boldsymbol{r_k} \in \mathbf{ACM}(i,\phi).(r_{k,j} < 0)$$

Definition 5.31 *A function f_i has the* ***abstract descent property for the subtyping environment*** *ϕ, denoted $\mathrm{ADP}(A)$, where $A \equiv \mathbf{ACM}(i,\phi)$, if and only if*

$$\exists j. \mathrm{AMD}(j, \phi) \wedge \mathrm{ADP}(A')$$

where $A' = \{\boldsymbol{r_e} \mid (\boldsymbol{r_e} \in A) \wedge (r_{e,j} = 0)\}$

Lemma 5.4 *If a function f_i has the abstract descent property for the subtyping environment ϕ then it has the abstract descent property for any ϕ' where ϕ' is a proper sub-environment of ϕ.*

Proof. This is a consequence of Lemma 4.10 and Defns 5.13, 5.18 and 5.25 where we use the singleton partition of a subtyping environment to define the respective abstract functions. □

However, if we take two subtyping environments on both of which f_i has the abstract descent property then it is not necessarily the case that f_i has the ADP on the join of the two environments if f_i has more than one argument. (There may be different lexicographical orderings used to fulfill the ADP in each case.) However, the following does hold.

Lemma 5.5 *Suppose that a function f_i has the abstract descent property on subtype environments ϕ_1 and ϕ_2 and that there exists a j such that $\mathrm{ASD}(x_{i,j}, \phi_1)$ and $\mathrm{ASD}(x_{i,j}, \phi_2)$. Then f_i has the abstract descent property on $\phi_1 \sqcup \phi_2$.*

Proof. The definitions of the closure, size and subtyping analyses mean that in each case their results are the joins of the results on the two sub-environments. This means that all entries in the abstract calls matrix for the joined subtype environment must be less than 0 as the entries for ϕ_1 and ϕ_2 are less than 0. Furthermore, the number of rows in the abstract calls matrix is the sum of the rows for the matrices pertaining to ϕ_1 and ϕ_2. □

The abstract descent property for a particular subtyping environment means that it has the monotonic descent property for those subtypes.

Theorem 5.1 *Suppose that a function f_i has the abstract descent property for the subtyping environment ϕ. Then f_i restricted to the subtypes of ϕ has the monotonic descent property.*

Proof. Similar arguments apply as for Theorem 4.3 □

Corollary 5.4 *Suppose that a function f_i is defined according to the rules of the basic ESFP language and that f_i has the abstract descent property for the subtyping environment ϕ. Then f_i terminates on all arguments restricted to the subtyping environment ϕ.*

5.6 ESFP with Subtyping — ESFP^1

Our new analysis, which is enhanced by subtyping, means that we can define a more expressive ESFP language.

Definition 5.32 *The **language** ESFP^1 consists of EFP^e together with a check that all definitions within a script have the abstract descent property for some valid subtyping environment. That is,*

$$\text{Accept}(\mathbf{S}, \text{ESFP}^1) \overset{\triangle}{\Longleftrightarrow} \text{Accept}(\mathbf{S}, \text{EFP}^e) \wedge \forall i \in \mathbb{I}^{\mathsf{S}}_f . \exists \phi i \in \mathsf{Env}_i(\mathsf{S}).\text{ValidSub}(\phi i) \wedge \text{ADP}(\mathbf{ACM}(f_i, \phi_i)$$

where ADP$(\mathbf{ACM}(f_i), \phi_i))$ *follows Defns 5.29–5.31.*

5.7 Example of the Analysis Using Subtyping

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

$$\begin{array}{l} gcd\; m\; n \overset{def}{=} \\ \quad \textbf{case}\; m\; \textbf{of}\;\; 0 \rightarrow \textbf{case}\; n\; \textbf{of}\; 0 \rightarrow \textbf{error}; (\textbf{Succ}\; n') \rightarrow n \\ \quad (\textbf{Succ}\; m') \rightarrow \;\; \textbf{case}\; compare\; m\; n\; \textbf{of}\;\; \textbf{EQ} \rightarrow m;\; \textbf{LT} \rightarrow gcd\; m\; (n-m);\; \textbf{GT} \rightarrow gcd\; (m-n)\; n \end{array}$$

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

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

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

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

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

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

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

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

The least fixed point 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 $\mathcal{G}^1_{gcd[gcd,\phi]} \llbracket \, gcd\,(m-n)\,n \, \rrbracket^{\phi'}_{\{\},\sigma} = \left(\begin{bmatrix} 0 \\ -\omega \end{bmatrix}, \begin{bmatrix} -1 & -\omega \\ -\omega & 0 \end{bmatrix} \right)$

Similarly, we get: $\mathcal{G}^1_{gcd[gcd,\phi]} \llbracket \, gcd\,m\,(m-n) \, \rrbracket^{\phi'}_{\{\},\sigma} = \left(\begin{bmatrix} -\omega \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & -\omega \\ -\omega & -1 \end{bmatrix} \right)$

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

This satisfies the abstract descent property.

6 Nested Inductive Types

Our abstract interpretation operates by recognising syntactic sub-components of other expressions. These sub-components occur as pattern-matching variables within ***case*** expressions. We may consequently have ***case*** expressions applied to expressions involving pattern-matching variables. Some pattern-matching variables will indicate a size descent within a recursive structure whilst others will indicate arbitrary data extracted from the structure. For example, in the case of lists where we may match a list l against a pattern of the form $(h : t)$ for non-empty lists, $|t| < |l|$ for all lists. However, the head, h, may be of arbitrary size. In the case of rosetrees, though, where a list of rosetrees is a sub-component of an internal node, an element of such a list will be a subtree of the original tree and consequently represents size descent.

We thus make our basic EFP language less restrictive by removing two of the constraints upon the definition of algebraic types.

Definition 6.1 *The language* EFP^+ *consists of* EFP^e *with restrictions 2 and 3 of* § *2.2 removed. If a script,* $\mathbf{S}$*, meets the criteria of* EFP^+ *then we write* $\mathrm{Accept}(\mathbf{S}, \mathrm{EFP}^+)$.

6.1 Projection Sequences

We consequently need an extended space of expressions which relates pattern-matching variables to the expressions that they match. We firstly define *projection sequences* which will represent the sequence of operations required to extract an element from a structure. These will be constrained so as to enable the calculation of least fixed points within the abstract interpretation framework.

Definition 6.2 *The set of* ***projections****,* Π*, is defined as follows:*

$$\Pi \triangleq \{\pi_{i,j} \mid \exists C_i \in \mathbb{C}.\mathrm{Ar}(C_i) \geq j\}$$

Definition 6.3 *The set of* ***projections from type S to type T****, denoted* $\Pi_{S \to T}$*, is defined as the restriction of* Π *to projections with domain* S *and range* T*.*

The projections above have the following interpretation. $\pi_{i,j}\, e \mapsto e_{i,j}$ if and only if $e \twoheadrightarrow C_i e_{i,1} \ldots e_{i,r}$. We shall only use such a projection in a context where it is defined i.e. where e does reduce to the appropriate pattern.

We form length-constrained sequences of projections as follows:

Definition 6.4 *The set of projection sequences,* P^d*, consists of singleton subsets of the set of all sequences of projections of length* $\leq d$*,* Π^d*, together with* $\top_{\mathsf{P}}$*, the set of all possible projections and* $\{\}$*, the bottom of the lattice induced by the subset ordering on* P^d*.*

$\top_{\mathsf{P}}$ indicates any possible composition of projections from a given data structure. Where there is no ambiguity, we represent singleton sets of sequences of projections simply by the projection sequence itself e.g. $\boldsymbol{\pi}$. In addition, we shall assume in the rest of this section that d is 2 and thus we shall write P^d simply as P. In Sect. 8, we shall discuss the effect of other possible values of d on the analysis. We shall write $|\boldsymbol{\pi}|$ to denote the length of the sequence $\boldsymbol{\pi}$.

Definition 6.5 *The set of* ***projection sequences****,* $\mathsf{P}^d_{S \to T}$*, consists of singleton subsets of the set of all sequences of projections of length* $\leq d$ *and of type* $S \to T$*,* $\Pi^d_{S \to T}$*, together with* $\top$*, the set of all possible projections of the required type, and* $\{\}$*.*

Definition 6.6 *The set of* ***types projectable*** *by an d length projection sequence from a type* T*, denoted* P^d_T*, is defined as,* $\mathsf{P}^d_T \triangleq \{V \mid \mathsf{P}^d_{T \to V} \neq \{\}\}$*.*

The projection sequences are used to determine whether a composition of projections upon a structure is *endomorphic* i.e. the types of the domain and range are the same.

Definition 6.7 *The* ***composition*** *of a non-*$\top_{\mathsf{P}}$ *sequence of projections is denoted* $\bigodot(\boldsymbol{\pi})$ *and is defined as* $\bigodot(\{\}) \triangleq \mathrm{Fold} \circ id\,\boldsymbol{\pi}$ *where* Fold *is the standard fold catamorphism over sequences. However,* $\bigodot(\top_{\mathsf{P}})$ *is undefined (reflecting the fact that this represents any possible combination of projections).*

Definition 6.8 *A projection sequence, $\boldsymbol{\pi}$, is termed **endomorphic**, and denoted* $\mathrm{Endo}(\boldsymbol{\pi})$ *if and only if* $\exists A. \bigodot(\boldsymbol{\pi}) :: A \to A$

The empty projection sequence is endomorphic (as it represents the identity) whilst $\top_{\mathsf{P}}$ *is not endomorphic.*

Definition 6.9 *A projection sequence, $\boldsymbol{\pi}$, is termed **reducing** and denoted* $\mathrm{Red}(\boldsymbol{\pi})$ *if it is both endomorphic and not the empty projection sequence.*

6.2 Projection Expressions

We can now combine the projection sequences defined above with our basic expression syntax to form new, abstract expressions as follows.

Definition 6.10 *A **projection expression**, denoted* π_e^d *(where d indicates that projection expressions are constructed from P^d sequences), is a pair of a sequence of projections and a basic expression (as defined in Sect. 2.3) or a substitution instance of a basic expression. The set of projection expressions,* P_{E}*, is thus defined,* $\mathsf{P}_{\mathsf{E}} \overset{\triangle}{=} \mathsf{P} \times \mathsf{E}$

*An **endomorphic projection expression** is a projection expression that includes an endomorphic projection sequence. A **reducing projection expression** is a projection expression that includes a reducing projection sequence.*

The informal concrete semantics of a projection expression is that it is the application of the composition of the sequence of projections to the (basic) expression e.

6.3 Binding Sets of Projection Expressions

We bind sets of projection expressions to pattern-matching variables within an environment, σ. Since, in our language, we assume that we only have single-level patterns, we shall only bind pattern-matching variables to elements of $\mathsf{P}^1_{\mathsf{E}}$.

We need to define a new domain of environments binding projection expressions to pattern matching variables since that given in Defn 4.7 only makes bindings to abstract expressions.

Definition 6.11 *The domain of **pattern variable projection expression environments**,* M_{P}*, consists of functions binding pattern matching variables to elements of* $\mathsf{P}^1_{\mathsf{E}}$ *i.e.* $\mathsf{M}_{\mathsf{P}} \overset{\triangle}{=} \mathbb{M} \mapsto \mathsf{P}^1_{\mathsf{E}}$

The actual binding process, for a ***case*** expression, is defined as follows.

Definition 6.12 *Let σ bind pattern-matching variables to sets of projection expressions. Then, for a **case** expression of the form,* $\textbf{\textit{case}}\, s\, \textbf{\textit{of}} \langle p_1, e_1\rangle \ldots \langle p_r, e_r\rangle$ *the environment of pattern-matching variables pertaining to each e_k is defined as,* $\sigma_k \overset{\triangle}{=} \bigcup_{l=1}^{l=|p_k|} B(p_{k,l}, s, \sigma)$*, where* $B(p_{k,l}, s, \sigma) \overset{\triangle}{=} \sigma\{p_{k,l} := (\langle \pi_{k,l}\rangle, \{s\})\}$*.*

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, \top_{\mathsf{E}})\,]\!]^{\phi}_{\rho,\sigma} \triangleq \mathsf{P}^{d}_{\mathsf{E}} \tag{99}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, (\{\}, e))\,]\!]^{\phi}_{\rho,\sigma} \triangleq \{(\{\}, e)\} \tag{100}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, x)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \{(\boldsymbol{\pi}, x)\} \tag{101}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, f_k)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \{\} \tag{102}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, C_t\, a_1 \ldots a_r)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \begin{cases} \{(\{\}, a_j)\} & \text{if } \boldsymbol{\pi} \equiv \langle \pi_{i,j} \rangle \\ \mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi'}, a_j)\,]\!]^{\phi}_{\rho,\sigma} & \text{if } \boldsymbol{\pi} \equiv \boldsymbol{\pi'} +\!\!+ \langle \pi_{i,j} \rangle \\ \{\} & \text{otherwise} \end{cases} \tag{103}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, \textbf{\textit{case}}\; s\; \textbf{\textit{of}}\; \langle p_r, e_r \rangle)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \bigcup_{k=1}^{k=r} \bigcup \{\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, \{e_k\})\,]\!]^{\phi}_{\rho,\sigma_k} \mid H(p_k) \in \mathcal{S}\,[\![\,s\,]\!]^{\phi}_{\rho,\sigma}\} \tag{104}$$

$$\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, F\, a)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \bigcup \left\{ b[\boldsymbol{a} \;/_{\mathsf{E}}\; \boldsymbol{x_k}] \;\middle|\; \begin{array}{l} b = f_f^{p_2,d}\, \rho'\, \phi''\, \boldsymbol{\pi}, \\ (\{f_k\}, \boldsymbol{a}, \sigma') \in \mathcal{C}^1\,[\![\,F\,]\!]^{\phi}_{\rho,\sigma}\, \langle a \rangle \end{array} \right\} \tag{105}$$

In (105), if, for some $\boldsymbol{a}, \sigma'$, $\mathcal{C}^1\,[\![\,F\,]\!]^{\phi}_{\rho,\sigma}\, \langle a \rangle = (\{\}, \boldsymbol{a}, \sigma')$ then $\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, F\, a)\,]\!]^{\phi}_{\rho,\sigma} \triangleq \top_{\mathsf{E}}$.

Table 9: Definition of $\mathcal{P}^{2,d}\,[\![\,(\boldsymbol{\pi}, E)\,]\!]^{\phi}_{\rho,\sigma}$

6.4 Approximating Projection Applications

Syntactic descent can only occur via a composition of projections that is of endomorphic type. Intermediate enclosing structures (such as the list of subtrees in the rosetree example) cannot be added to if the recursion is to be well-founded. In the case of rosetrees, if an arbitrary tree was added to the list of subtrees then descent could not be guaranteed. We thus require a method of approximating the expressions that may result from applying a non-endomorphic projection. We need to be able to approximate the set of endomorphic projection expressions that correspond to a (non-endomorphic) projection expression. To form our approximation, we map an element of $\mathsf{P}^1_{\mathsf{E}}$ into $\wp(\mathsf{P}^l_{\mathsf{E}})$. This mapping will reduce a projection expression to a set of projection expressions in which the projection sequence is either empty ($\langle\rangle$) or the projection expression is of the form, $(\boldsymbol{\pi}, v)$, where v is either $\{x\}$, where x is a variable, or v is $\top_{\mathsf{E}}$.

We shall call this mapping, *projection analysis* which approximates the set of expressions that may result from applying a projection other than $\top_{\mathsf{P}}$ to an expression.

Firstly, we need a method of adding a projection to a projection expression to deal with the situation where we have ***case*** constructs applied to pattern matching variables — we may take the head of the tail of a list, for example. This leads to the following definition.

Definition 6.13 *The addition of a projection, $\pi_{i,j}$, to a projection expression, $p \in \mathsf{P}^l_{\mathsf{E}}$ is denoted $\pi_{i,j} \oplus p$ and is defined as follows:*

$$\begin{array}{rcl} \pi_{i,j} \oplus (\top_{\mathsf{P}}, e) & \triangleq & (\top_{\mathsf{P}}, e) \\ \pi_{i,j} \oplus (\boldsymbol{\pi}, e) & \triangleq & \begin{cases} (\pi_{i,j} : \boldsymbol{\pi}, e) & \textit{if } |\boldsymbol{\pi}| < l \\ (\top_{\mathsf{P}}, e) & \textit{otherwise} \end{cases} \end{array}$$

Having broadened the class of types that may be permitted in the language we need to redefine the operator that gives the recursive sub-components of an expression. This operator will then be used in our projection analysis below when calculating the set of terms that may be projected from a data structure by a non-endomorphic projection. We wish, for example, for this to correspond to all the elements of a list constant.

Definition 6.14 *The* ***recursive sub-components abstraction*** *operator,* RecAbs($\in$)$\mathbb{E} \mapsto$ E*, is defined as follows:*

$$\mathrm{RecAbs}(e) \triangleq \{ \begin{array}{l} s \in \Pi^{!} \mid \\ \mathrm{TC}(\bigodot(s), e, \wedge)\mathrm{Unify}(\bigodot(s)e, e) \\ \wedge\neg\exists s' \subset s.\mathrm{Unify}(\bigodot(s')e, e) \end{array} \}$$

Here, $\Pi^{!}$ *is the set of ordered* sets *of projections and* $\mathrm{TC}(\bigodot(s), e,)$ *denotes a type-correct application of the composition of the elements of s to e. The final restriction on the elements of S ensures that there is not any proper subsequence that represents a type-recursive projection.*

Note that $\mathbb{C}$ is finite and thus $\Pi^{!}$ is finite since it is the ordered counterpart of $\wp(\mathbb{C})$.

We now define the abstract interpretation used to approximate the set of projection expressions corresponding to the application of a non-endomorphic projection. In this and the subsequent analyses given (see § 6.6) we do not give the result for ***error*** explicitly: as in § 5.1.1 the result is in each case the $\top$ of the relevant abstract domain.

Definition 6.15 *The* ***projection analysis operator****,* $\mathcal{P}^{2,d} \in \mathsf{P}^1_{\mathsf{E}} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M}_{\mathsf{P}} \times \mathit{Env}(\mathsf{S}) \mapsto \wp(\mathsf{P}^l_{\mathsf{E}})$*, is defined in Table 9 for projection expressions where the projection sequence is neither endomorphic nor* $\top_{\mathsf{P}}$*.*

- *In the case of the endomorphic projection sequences,* $\mathcal{P}^{2,d}$ *corresponds to the injection,* $p \mapsto \{p\}$*.*
- *In the case where the projection sequence is* $\top_{\mathsf{P}}$*, the result is* $\mathsf{P}^l_{\mathsf{E}}$*, the top of* $\wp(\mathsf{P}^l_{\mathsf{E}})$*.*

This mapping will be particularly useful in the case of closure analysis, where we previously found all possible subcomponents that could be applied as functions, even though the subcomponents would not be projected from a structure and then applied.

In the second clause of the definition we have direct descent due to the projection implicit in the pattern match. In the third clause, the composition of projections when applied to an expression produces size descent — this thus takes care of the case of nested-type data structures. However, in the last clause, where the composition of two projections has not produced descent, we approximate by using the $\top_{\mathsf{P}}$ projection.

We now show that the projection analysis has the required behaviour in the following respects:

1. It reduces non-endomorphic projection expressions to sets of projection expressions of the form $(\boldsymbol{\pi}, \{e\})$ (in the non-pathological case where the result is not $\top_{\mathsf{P}_{\mathsf{E}}}$), which are either endomorphic or non-endomorphic and the expression, e is a parameter, $x_{i,j}$.
2. If the expression, e, corresponds to the projection expression $(\boldsymbol{\pi}, a)$, then if e reduces to e' then there exists a $p \in \mathcal{P}^{2,d}(\boldsymbol{\pi}, a)$, such that p is equivalent to e'' which is convertible to e'.

Lemma 6.1 *The projection analysis operator,* $\mathcal{P}^{2,d}$*, either reduces* $p \in \mathsf{P}^1_{\mathsf{E}}$ *to* $\top_{\mathsf{P}^1_{\mathsf{E}}}$ *or to a set* S *consisting only of endomorphic projection expressions or non-endomorphic projection expressions where the expression is a parameter,* $x_{i,j}$*.*

Proof. By structural induction over E. □

Theorem 6.1 *Our projection analysis operator is correct.*

Proof. By structural induction over E. □

6.5 The Projection-Size Abstract Domain

In correspondence to the space of projection expressions we now describe a new abstract domain of *projection sizes*. The point of this new domain is that it allows the detection of syntactic descent even when this occurs as the composition of separate projections applied to the actual and formal parameters of a called function.

Definition 6.16 *The* ***projection-size*** *abstract domain, denoted* $\mathsf{R_P}$ *is defined as the cardinal product of the projection domain and the relative size domain i.e.* $\mathsf{R_P} \triangleq \mathsf{P} \times \mathsf{R}$. *The top of this domain is denoted* $\top_{\mathsf{R_P}}$ *(*$\equiv (\top_{\mathsf{P}}, \omega)$*) and the bottom is denoted* $\bot_{\mathsf{R_P}}$ *(*$\equiv (\{\}, -\omega)$*). The least upper bound operator on this complete lattice is denoted* $\mathrm{max}_{\mathsf{R_P}}$*, although it will normally be written simply* max *as it will be clear upon which domain we shall be operating.* $\mathrm{max}_{\mathsf{R_P}}$ *is defined (for non-*$\top$ *elements) via* $\mathrm{max}_{\mathsf{R}}$ *as follows:*

$$\begin{array}{rcll} \mathrm{max}_{\mathsf{R_P}}(\boldsymbol{\pi_1}, s_1)(\boldsymbol{\pi_2}, s_2) & \triangleq & (\boldsymbol{\pi_1}, (\mathrm{max}_{\mathsf{R}}(s_1, s_2))) & \textit{if } \pi_1 \equiv \pi_2 \\ & \triangleq & \top_{\mathsf{R_P}} & \textit{otherwise} \end{array} \tag{106}$$

As for the relative size domain, we define addition and multiplication operators as follows:

$$\begin{array}{rcll} (\boldsymbol{\pi_1}, s_1) + (\boldsymbol{\pi_2}, s_2) & \triangleq & (\boldsymbol{\pi_1}, (s_1 +_{\mathsf{R}} s_2)) & \textit{if } \pi_1 \equiv \pi_2 \\ & \triangleq & \top_{\mathsf{R_P}} & \textit{otherwise} \end{array} \tag{107}$$

$$\begin{array}{rcll} (\boldsymbol{\pi_1}, s_1) * (\{\}, s_2) & \triangleq & (\boldsymbol{\pi_1}, (s_1 *_{\mathsf{R}} s_2)) & \\ (\{\}, s_1) * (\boldsymbol{\pi_2}, s_2) & \triangleq & (\boldsymbol{\pi_2}, (s_1 *_{\mathsf{R}} \omega)) & \\ (\boldsymbol{\pi_1}, s_1) * (\boldsymbol{\pi_2}, s_2) & \triangleq & (\{\}, (s_1 *_{\mathsf{R}} s_2) - 1) & |\boldsymbol{\pi_3}| \leq l \wedge \mathrm{Red}(\pi_3) \\ (\boldsymbol{\pi_1}, s_1) * (\boldsymbol{\pi_2}, s_2) & \triangleq & (\boldsymbol{\pi_3}, (s_1 *_{\mathsf{R}} s_2)) & |\boldsymbol{\pi_3}| \leq l \\ (\boldsymbol{\pi_1}, s_1) * (\boldsymbol{\pi_2}, s_2) & \triangleq & \top_{\mathsf{R_P}} & \textit{otherwise} \end{array} \tag{108}$$

In equation (108), $\boldsymbol{\pi_3} = \boldsymbol{\pi_1} \mathbin{+\!\!+} \boldsymbol{\pi_2}$.

6.5.1 Concrete Semantics of $\mathsf{R_P}$

We now discuss the meaning of our projection-size domain, $\mathsf{R_P}$, with respect to the concrete semantics of our basic ESFP language and to the domain of relative sizes discussed in Sect. 4.2.

Informal concretisation. For an expression e, if $(\{\}, s)$ is its abstract semantics in $\mathsf{R_P}$, relative to some parameter, $x_{i,j}$ then its concrete semantics corresponds to that of s in AR, again relative to $x_{i,j}$. If, however, the abstract semantics of e in $\mathsf{R_P}$ is $(\boldsymbol{\pi}, s)$ (again relative to some parameter $x_{i,j}$), then e is convertible to $\bigodot(\boldsymbol{\pi})e'$ for some e' and the size of e' is s (relative to $x_{i,j}$) and the relative size of e itself is unknown. Furthermore, $(\boldsymbol{\pi}, s)$ is only a valid projection-size representation of e in the case where $\boldsymbol{\pi}$ is an endomorphic projection sequence and where e' is a formal parameter. Our analysis which we describe below will enforce this latter requirement and we will thus also be able to show that s will thus be either 0 or $-\omega$.

Given the above informal description, we have the following operator that maps projection sizes to their counterparts in R.

Definition 6.17 *The **projection-size norm**, denoted* N*, is a mapping from* $\mathsf{R_P}$ *to* R *which is defined as follows:*

$$\begin{aligned} \mathrm{N}(\{\}, s) &\triangleq s \\ \mathrm{N}(\boldsymbol{\pi}, s) &\triangleq \omega \end{aligned}$$

The idea here is similar to that discussed in the original analysis in § 4.5 — despite knowing that an expression e has size of s relative to $x_{i,j}$, we cannot determine the size of e' where e' is equivalent to $\bigodot(\boldsymbol{\pi})e$ and $\boldsymbol{\pi}$ is not the identity. Thus we must safely approximate using ω.

6.6 Modifying the Analyses with Projection Expressions

We now show how the analyses are modified in the light of the foregoing discussion on projection analysis. We give definitions that are developed from those in § 5.

6.6.1 Closure analysis with projection sequences.

Definition 6.18 *The **closure analysis semantic operator with** d **length projection sequences** $\mathcal{C}^{2,d} \in \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M_P} \times \mathit{Env}(\mathsf{S}) \times \mathsf{E}^* \mapsto \mathsf{C}$, is defined in Table 10.*

Definition 6.19 *The **abstract closure function using** d **length projection sequences**, of a function f_i, denoted $f_i^{m_{2,d}}$, is defined as in Defn 5.13, except that $\mathcal{C}^{2,d}$ replaces $\mathcal{C}^1$.*

6.6.2 Size analysis with projection sequences.

Definition 6.20 *The **relative size analysis operator with** d **length projection sequences**, $\mathcal{A}^{2,d} \in \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M_P} \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{R}$, is the $\mathcal{A}$ operator extended with projection expressions and subtyping and defined over the structure of expressions in Table 11.*

As an auxiliary operation, we need to define the size of a projection expression that is produced by projection analysis.

Definition 6.21 *The **size of a projection expression**, e, denoted* $\mathrm{PES}(e)$*, relative to the environments of pattern matching variables (σ), functions (ρ) and subtypes (ϕ), is defined as follows:*

$$\begin{aligned} \mathrm{PES}((\top_{\mathsf{P}}, e')) &\triangleq \top_{\mathsf{R_P}} \\ \mathrm{PES}((\boldsymbol{\pi}, e')) &\triangleq \begin{cases} \mathcal{A}^{2,d}_{i,j}[\![\, e' \,]\!]^{\phi}_{\rho,\sigma} - 1 & \text{if } \mathrm{Red}(\boldsymbol{\pi}) \\ \mathcal{A}^{2,d}_{i,j}[\![\, e' \,]\!]^{\phi}_{\rho,\sigma} & \textbf{\textit{otherwise}} \end{cases} \end{aligned}$$

$$\mathcal{C}^{2,d}\,\llbracket\,E\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \{\}\ \text{ if } E \text{ is of ground type}$$

$$\mathcal{C}^{2,d}\,\llbracket\,x\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \begin{cases} \top_{\mathsf{C}} & \text{if } \rho(x) = \top_{\mathsf{E}} \\ \{(\{\},\boldsymbol{a},\sigma)\} & \text{if } \rho(x) = \{\} \\ \mathcal{C}^{2,d}\,\llbracket\,e\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} & \text{if } \rho(x) = \{e\} \\ \bigcup_{(\{\},\{e\})\in\mathcal{P}^{2,d}\,\llbracket\,(\boldsymbol{\pi},\sigma(x))\,\rrbracket^{\phi}_{\rho,\sigma}\}} \mathcal{C}^{2,d}\,\llbracket\,e\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} & \text{if } x \in \mathrm{Dom}(\sigma) \wedge \mathcal{P}^{2,d}\,\llbracket\,(\boldsymbol{\pi},\sigma(x))\,\rrbracket^{\phi}_{\rho,\sigma} \\ \top_{\mathsf{C}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}^{2,d}\,\llbracket\,f_i\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \begin{cases} \{(\{f_i\},\boldsymbol{a},\sigma)\} & \text{if } \mathrm{Ar}(f_i) \geq |\boldsymbol{a}| \\ \{(f,\boldsymbol{e},\sigma'') \mid (f,\boldsymbol{d},\sigma') \in f_i^m\,\rho'\,\boldsymbol{c}\} & \text{otherwise} \end{cases}$$

$$\mathcal{C}^{2,d}\,\llbracket\,C_t\,a_1\ldots a_r\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \bigcup_{i=1}^{i=r}(f,\boldsymbol{b},\sigma') \mid (f,\boldsymbol{b},\sigma') \in \mathcal{C}^{2,d}\,\llbracket\,e_i\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \wedge \mathrm{TC}(f,\boldsymbol{b})\}$$

$$\mathcal{C}^{2,d}\,\llbracket\,\textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r,e_r\rangle\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \bigcup_{k=1}^{k=r}\bigcup\{\mathcal{C}^{2,d}\,\llbracket\,e_k\,\rrbracket^{\phi}_{\rho_k,\sigma_k}\,\boldsymbol{a} \mid H(p_k) \in \mathcal{S}\,\llbracket\,s\,\rrbracket^{\phi}_{\rho,\sigma}\}$$

$$\mathcal{C}^{2,d}\,\llbracket\,G\,d\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a} \;\triangleq\; \mathcal{C}^{2,d}\,\llbracket\,G\,\rrbracket^{\phi}_{\rho,\sigma}\,(\langle\{d\}\rangle \mathbin{+\!\!\!+} \boldsymbol{a})$$

Table 10: Definition of $\mathcal{C}^{2,d}\,\llbracket\,E\,\rrbracket^{\phi}_{\rho,\sigma}\,\boldsymbol{a}$

$$\mathcal{A}^{2,d}_{i,j}\,\llbracket\,x\,\rrbracket^{\phi}_{\rho,\sigma} \;\triangleq\; \begin{cases} (\{\},0) & \text{if } x \equiv x_{i,j} \\ (\{\},-\omega) & \text{if } x \equiv x_{i,k} \\ \max_{p\in\mathcal{P}^{2,d}\,\llbracket\,(\boldsymbol{\pi},\sigma(x))\,\rrbracket^{\phi}_{\rho,\sigma}\}} \mathrm{PES}(p) & \text{if } x \in \mathrm{Dom}(\sigma) \wedge \mathcal{P}^{2,d}\,\llbracket\,(\boldsymbol{\pi},\sigma(x))\,\rrbracket^{\phi}_{\rho,\sigma} \neq \top_{\mathsf{P_E}} \\ \top_{\mathsf{R_P}} & \text{otherwise} \end{cases} \tag{115}$$

$$\mathcal{A}^{2,d}_{i,j}\,\llbracket\,f_k\,\rrbracket^{\phi}_{\rho,\sigma} \;\triangleq\; \begin{cases} f^a_{k,0}\,\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \\ (\{\},-\omega) & \text{if } \mathrm{Ar}(f_k) \neq 0 \end{cases} \tag{116}$$

$$\mathcal{A}^{2,d}_{i,j}\,\llbracket\,C_t\,a_1\ldots a_r\,\rrbracket^{\phi}_{\rho,\sigma} \;\triangleq\; \mathrm{cs}^{2,d}(\mathrm{RecAbs}(E),i,j,\rho,\sigma,\phi) \tag{117}$$

$$\mathcal{A}^{2,d}_{i,j}\,\llbracket\,\textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r,e_r\rangle\,\rrbracket^{\phi}_{\rho,\sigma} \;\triangleq\; \max(\bigcup_{k=1}^{k=r}\{\mathcal{A}^{2,d}_{a_k,\sigma_k}\,\llbracket\,\rho\,\rrbracket_{|,}\,H(p_k) \in \mathcal{S}\,\llbracket\,s\,\rrbracket^{\phi}_{\rho,\sigma}\}) \tag{118}$$

$$\mathcal{A}^{2,d}_{i,j}\,\llbracket\,F\,a\,\rrbracket^{\phi}_{\rho,\sigma} \;\triangleq\; \max\{\mathrm{ap}^{a_{2,d}}(f,i,j,\boldsymbol{a},\rho',\sigma',\phi') \mid (f,\boldsymbol{a},\sigma') \in \mathcal{C}^{2,d}\,\llbracket\,F\,\rrbracket^{\phi}_{\sigma,\rho}\,\langle\{a\}\rangle\} \tag{119}$$

Table 11: Definition of $\mathcal{A}^{2,d}_{i,j}\,\llbracket\,E\,\rrbracket^{\phi}_{\rho,\sigma}$

Definition 6.22 *The* ***abstract size function with*** d ***length projection sequences*** *of a function,* f_i*,* ***relative to parameter*** j *is denoted* $f^{a_2,d}_{i,j}$ *and defined as in Defn 5.18, except that the* $\mathcal{A}^{2,d}$ *operator replaces* $\mathcal{A}^1$*.*

6.6.3 Calls analysis with projection sequences.

$$\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket x \rrbracket^{\phi}_{\rho,\sigma} \triangleq \langle\rangle \tag{120}$$

$$\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket f_k \rrbracket^{\phi}_{\rho,\sigma} \triangleq \begin{cases} f^{g_{2,d}}_{k[j,\phi_j]}\,\{\}\,\{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k \neq j \\ \langle \boldsymbol{\Omega} \rangle & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k = j \\ \langle\rangle & \text{otherwise} \end{cases} \tag{121}$$

$$\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket C_t\, a_1 \ldots a_r \rrbracket^{\phi}_{\rho,\sigma} \triangleq \biguplus_{k=1}^{k=r} \mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket a_k \rrbracket^{\phi}_{\rho,\sigma} \tag{122}$$

$$\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r \rangle \rrbracket^{\phi}_{\rho,\sigma} \triangleq \biguplus(\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket s \rrbracket^{\phi}_{\rho,\sigma}, (\biguplus_{k=1}^{k=r} G_k)) \tag{123}$$

$$\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket F\, a \rrbracket^{\phi}_{\rho,\sigma} \triangleq \biguplus_{\substack{(f,\boldsymbol{a},\sigma') \in \\ \mathcal{C}^1 \llbracket F \rrbracket^{\phi}_{\rho,\sigma} \langle\{a\}\rangle}} (\mathrm{ap}^{g_{2,d}}(f, i, \boldsymbol{a}, \sigma', \rho, \phi, \phi_j) \biguplus (\biguplus_{i=1}^{i=|\boldsymbol{a}|} \mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket a_i \rrbracket^{\phi}_{\rho,\sigma})) \tag{124}$$

In (123), $G_k = \begin{cases} \mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket e_k \rrbracket^{\phi_k}_{\sigma_k,\rho} & \text{if } H(p_k) \in \mathcal{S} \llbracket s \rrbracket^{\phi_k}_{\rho,\sigma} \\ \{\} & \text{otherwise} \end{cases}.$

Table 12: Definition of $\mathcal{G}^{2,d}_{i[j,\phi_j]} \llbracket E \rrbracket^{\phi}_{\rho,\sigma}$

Definition 6.23 *The* ***abstract calls operator with*** d ***length projection sequences****,* $\mathcal{G}^{2,d} \in \mathbb{I}^{\mathsf{S}}_f \times \mathbb{I}^{\mathsf{S}}_f \times \mathit{Env}(\mathsf{S}) \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M_P} \times \mathit{Env}(\mathsf{S}) \mapsto \mathsf{T}^*$*, is the* $\mathcal{G}$ *operator extended with projection expressions and subtyping to locate calls of function* f_j *with subtype environment* ϕ_j *within function* f_i *which has input subtype environment* ϕ_i*. It is defined over the structure of expressions in Table 12.*

Definition 6.24 *For each function, there is a family of* ***abstract calls functions with*** d ***length projection sequences*** *which is denoted* $f^{g_{2,d}}_{i[j,\phi_j]}$ *and defined as in Defn 5.25 except that the* $\mathcal{G}^{2,d}$ *operator replaces* $\mathcal{G}^1$*.*

6.6.4 Other modified definitions.

The other definitions of the analysis and the abstract termination criteria follow analogously to those of Defns 5.19–5.22 and Defns 5.29–5.31.

6.7 Extending ESFP — ESFP^{2,d}

Once again, we can now define a more expressive ESFP language.

Definition 6.25 *For some given natural number d, the* ***language*** ESFP^{2,d} *consists of* EFP^{+} *together with a check that all definitions within a script have the abstract descent property for some valid subtyping environment and analysing with projection expressions of length d. Formally, the definition follows that given in Defn 5.32, with the appropriate modifications to the definitions of the abstract descent property and the abstract calls matrix.*

6.8 Example: *Maptree*

We now proceed to show how the termination of recursive functions over such nested inductive types can be shown in the case where the length of the projection sequences is 2.

Example 6.1 [Maptree] Suppose that we have the following definition of a rosetree type:

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

We then define a mapping function, *maptree*, over such structures as follows:

```
maptree f t ≝
    case t of
        (Leaf a)   →    (Leaf f a)
        (Node s)   →    (Node map (maptree f) s)
```

The definition of *map* is standard.

```
map g l ≝
    case l of
        []         →    []
        (h : t)    →    (g h) : map g t
```

The above can be shown to be an ESFP^{2,d} program for $d \geq 2$ since we get:

$$[(\langle \pi_{hd} \rangle, 0)] * [(\langle \pi_{Node} \rangle, 0)] \ = \ [(\langle\rangle, -1)] \quad [\text{As Endo}(\langle \pi_{hd}, \pi_{Node} \rangle)] \tag{125}$$

7 Arbitrary Precision Subtyping

The method of subtyping given in Sect. 5 may be seen to be unsatisfactory for the following reasons:

1. Consider the general form of the ***case*** expression:

   $$\begin{array}{lll} \textbf{\textit{case}}\ s\ \textbf{\textit{of}} & & \\ \quad C_1\, v_{1,1} \ldots v_{1,\mathrm{Ar}(C_1)} & \rightarrow & e_1 \\ \quad \quad \vdots & \vdots & \vdots \\ \quad C_n\, v_{n,1} \ldots v_{n,\mathrm{Ar}(C_n)} & \rightarrow & e_n \end{array}$$

   We know the subtype of the switch expression, s for the ith clause (i.e. $\{C_i\}$) but what we wish to infer is the subtype of each variable, $x_{k,j}$. Furthermore, it would be useful if we could discover precise subtyping information for pattern matching variables. For example, if another ***case*** expression was nested within e_i, then it would be desirable to find, the subtype pertaining to $v_{i,l}$. Consequently, we would be able to deduce the subtype of the head or tail of a list, for example. This would generally appear to be impossible, given the evidence from strictness analysis[5] [19], if we use the approach given previously.

2. We cannot use partial functions as arguments to functors such as *map*, even if we know, for example, that the function is defined on all elements of a given list. This is because the subtyping mechanism is not strong enough to convey the subtypes of elements of data structures.

3. Dependencies in the subtyping information are lost when using the subtype environments with the other analyses such as the size analysis. This is because the environments only contain subtype constants and the relationship between the subtypes of the various parameters is lost. However, consider an ESFP language expression such as:

   $$\mathit{take}\ (\mathit{length}\, x\ \mathit{div}\ 2)\, x$$

   In the above, subtype constants will be bound to each of the parameters of *take* but the information that each subtype depends on the subtype of x will be lost.

4. We need to analyse every function with respect to every possible permutation of subtypes of the algebraic arguments. This process is naturally akin to the satisfiability problem and thus is of exponential complexity. This is despite the fact that

---

[5] Strictness analysis, used to optimise lazy functional languages by eliminating closure formation, determines whether for a function f that $f\,\bot = \bot$, where $\bot$ is the undefined value. In such a case, f is said to be *strict* in its argument.

subtyping information is not normally required for every algebraic argument. This computational complexity cannot be improved without a consdierable weaking of the precision of the analysis, as shown in [13].

7.1 Arbitrary Precision Subtype Domains

We proceed to define a domain, the *arbitrary precision subtyping domain*, that allows us to assign subtypes to elements which may be projected from an algebraic structure.

Definition 7.1 *A* ***projection subtype****, denoted* π^i_S *(where i indicates that projection subtypes are constructed from* P^i *sequences — see Defn 6.4), is a mapping from sequences of projections (from some type* S *to a type* T*) to a basic subtype of type* T *as defined in § 5.2.*

The ***set of projection subtypes****,* $\mathsf{P}^i_{\mathsf{S},(S\to T)}$*, is thus defined,* $\mathsf{P}^i_{\mathsf{S},(S\to T)} \stackrel{\triangle}{=} ((\mathsf{P}^i_{S\to T}) \mapsto \mathsf{S}_T)$*.*

Definition 7.2 *The* ***arbitrary precision subtyping domain*** *for type* T *of order d, denoted* S^d_T*, is defined as,* $\mathsf{S}^d_T \stackrel{\triangle}{=} \bigcup_{V\in\mathsf{P}^i_T} \mathsf{P}^i_{\mathsf{S},(T\to V)}$ *The ordering on this set is given in Defn 7.7 and ensures that the set forms a complete lattice.*

We shall normally write this domain as S^d where T is either clear from the context or applies universally to all algebraic types and the top is denoted $\top_{\mathsf{S}^d}$. We shall also employ the convention of writing elements of S^d as a union of a mapping between the empty (representing the identity) projection sequence and S_T and a partial mapping from non-empty projection sequences to S_V for some type V. The projection sequences not included in the domain of the resulting map will thus implicitly be mapped to $\top_{\mathsf{S}_V}$ for the appropriate V.

We need to be able to extract the relevant components from an element of our arbitrary precision subtyping domain.

Definition 7.3 *Let* S *be an arbitrary precision subtype for the type* T *of order d. The* ***part of*** S ***prefixed by*** $\boldsymbol{\pi}$ *(where* $\boldsymbol{\pi}$ *is a valid projection sequence on* T*), denoted* $pp(S,\boldsymbol{\pi}) \in \bigcup_{W\in\mathsf{P}^l_V} \mathsf{P}^l_{\mathsf{S},(V\to W)}$ *(where* $\mathrm{T}(\bigodot(\boldsymbol{\pi})) = T \to V$ *and* $l = d - |\boldsymbol{\pi}|$*) and defined as follows:*

$$pp(S,\boldsymbol{\pi}) \stackrel{\triangle}{=} \{(\boldsymbol{\pi}', s) \mid (\boldsymbol{\pi}' +\!\!+ \boldsymbol{\pi}, s) \in S \wedge \boldsymbol{\pi}' \neq \{\}\}$$

Definition 7.4 *The* ***atomic part*** *of an arbitrary precision subtype,* S*, denoted* $at(S) \in \mathsf{S}$ *is defined as follows:*

$$at(S) \stackrel{\triangle}{=} \bigsqcup \{a \mid (\{\}, a) \in pp(S, \{\})\}$$

The ***subsidiary part*** *of an arbitrary precision subtype,* S*, denoted* $sp(S) \in \bigcup_{V\in\mathsf{P}^d_T} \mathsf{P}^d_{\mathsf{S},(T\to V)}$ *is defined as follows:*

$$sp(S) \stackrel{\triangle}{=} \{r \mid r \in S - pp(S, \{\})\}$$

As a consequence of the above definitions, we shall normally write our arbitrary precision subtypes as sets containing pairs of the atomic and subsidiary parts rather than as a set of pairs of projection sequences and basic subtypes. An example of this form of notation is given in the following paragraph.

Each element, $(\boldsymbol{\pi}, s)$ of the subsidiary part indicates that the subtype of the subcomponent, c, projected by $\boldsymbol{\pi}$ from the enclosing structure, e, is s. However, s is, of course, an element of S and not an arbitrary precision subtype. However, other elements of the subsidiary part may indicate the subtypes of c. These will be those elements that have $\boldsymbol{\pi}$ as a suffix in the projection sequence. For example, consider the following possible subtype, S, for a list of naturals:

$$\{(\{:\}, \{(\langle \mathit{tail} \rangle, \{:\}), (\langle \mathit{hd} \rangle, \{0, \boldsymbol{\mathit{Succ}}\}), (\langle \mathit{hd}, \mathit{tail} \rangle, \{\boldsymbol{\mathit{Succ}}\})\})\}$$

The above indicates that we have a non-empty list and, in fact, a list of at least two elements since the tail is non-empty. Elements of the list may be any natural number but elements of the tail must be non-zero. Consider now what the full, arbitrary precision subtype of the tail of this list should be, given the above subtype. The $(\langle \mathit{tail} \rangle, \{:\})$ element of the subsidiary part of S indicates that the atomic part of the subtype of the tail should be $\{:\}$. Now we examine the subsidiary part of the subtype of the tail of the list. In S we have, $(\langle \mathit{hd}, \mathit{tail} \rangle, \{\boldsymbol{\mathit{Succ}}\})$. This means that $(\langle \mathit{hd} \rangle, \{\boldsymbol{\mathit{Succ}}\})$ should be included in the subsidiary part of the subtype of the tail of the list. Thus, given S, the full subtype of the tail of the list should be

$$\{(\{:\}, \{(\langle \mathit{hd} \rangle, \{\boldsymbol{\mathit{Succ}}\}), \})\}$$

Consequently, we have the following definition.

Definition 7.5 *Let S be an arbitrary precision subtype for the type T of order d. Then the arbitrary precision subtype of type V and order d* ***indexed by the projection sequence*** $\boldsymbol{\pi}$ *(where $\boldsymbol{\pi} \in \mathsf{P}^{d}_{(T \to V)}$ for some V) is denoted $\mathit{ist}(S, \boldsymbol{\pi})$ and defined as follows:*

$$\mathit{ist}(S, \boldsymbol{\pi}) \stackrel{\triangle}{=} \{(a, r) \mid (\boldsymbol{\pi}, a) \in \mathit{sp}(S) \wedge r \in \mathit{pp}(S, \boldsymbol{\pi})\}$$

In the opposite direction, we wish to add a projection sequence to each component of an arbitrary precision subtype. This is required when we determine the subtype of a sub-structure and then wish to integrate that subtype within the subtype for the entire structure.

Definition 7.6 *Let S be an arbitrary precision subtype of order d and let l be a natural $\geq d$. Then S* ***lifted by*** $\boldsymbol{\pi}$ *(where $\boldsymbol{\pi}$ is a valid projection sequence) is an arbitrary precision subtype of order l, denoted by $\mathit{lst}(S, \boldsymbol{\pi})$ is defined as follows:*

$$\mathit{lst}(S, \boldsymbol{\pi}) \stackrel{\triangle}{=} \{(\boldsymbol{\pi'} +\!\!+_l \boldsymbol{\pi}, S') \mid (\boldsymbol{\pi'}, S') \in S\}$$

We now proceed to define the lattice operations over arbitrary precision subtypes.

Definition 7.7 *The **join** (denoted $\sqcup$) and **meet** (denoted $\sqcap$) over atomic parts of arbitrary precision subtypes is as for S i.e. $\cap$ and $\cup$, respectively. Similarly, the ordering, $\sqsubseteq$ is just subset inclusion.*

Over subsidiary parts the ordering $\sqsubseteq$ is defined as follows:

$$r_1 \sqsubseteq r_2 \stackrel{\triangle}{=} \forall(\boldsymbol{\pi}, s) \in r_1.(\exists(\boldsymbol{\pi}, s') \in r_2 \Rightarrow s \sqsubseteq s')$$

*The **join** and **meet** over subsidiary parts are defined as follows:*

$$\begin{aligned} r_1 \sqcup r_2 &\stackrel{\triangle}{=} \{(p, t_1 \cup t_2) \mid (p, t_1) \in r_1 \wedge (p, t_2) \in r_2\} \\ r_1 \sqcap r_2 &\stackrel{\triangle}{=} \{(p, t_1 \cap t_2) \mid (p, t_1) \in r_1 \wedge (p, t_2) \in r_2\} \\ &\quad \cup\{(p, t_1) \mid (p, t_1) \in r_1 \wedge (\,\nexists t_2.(p, t_2) \in r_2\} \\ &\quad \cup\{(p, t_2) \mid (p, t_2) \in r_2 \wedge (\,\nexists t_1.(p, t_1) \in r_1\} \end{aligned}$$

The definitions of $\sqcup$ and $\sqcap$ given above may be seen to be almost dual to that which might be expected. This is because if a projection sequence, $\boldsymbol{\pi}$ does not occur within a subsidiary part it is implicit that $(\boldsymbol{\pi}, \top)$ is included within the subsidiary part. Concomitant with this, note that the definition of $\sqsubseteq$ is such that $(\boldsymbol{\pi}, s)$ may be in r_1 and not in r_2 but that $r_1 \sqsubseteq r_2$. Indeed, for all S and d, the empty set is the top of $\mathsf{P}^d_{\mathsf{S},(S\rightarrow T)}$.

Definition 7.8 *The **join** operation on arbitrary precision subtypes, s_1 and s_2, denoted $s_1 \sqcup s_2$, is defined as follows (using the representation of subtypes as pairs of the atomic and subsidiary parts):*

$$\begin{aligned} s_1 \sqcup s_2 &\stackrel{\triangle}{=} \{(a, r \sqcup r') \mid (a, r_1) \in s_1, (a, r_2) \in s_2\} \\ &\quad \cup\{(a_1, r_1) \mid (a_1, r_1) \in s_1 \wedge (\,\nexists r_2.(a_1.r_2) \in s_2\} \\ &\quad \{\cup(a_2, r_2) \mid (a_2, r_2) \in s_2 \wedge (\,\nexists r_1.(a_2.r_1) \in s_1\} \end{aligned}$$

*The **meet** operation on arbitrary precision subtypes, s_1 and s_2, denoted $s_1 \sqcap s_2$ is defined as follows:*

$$s_1 \sqcap s_2 \stackrel{\triangle}{=} \{(a, r_1 \sqcap r_2) \mid (a, r_1) \in s_1, (a, r_2) \in s_2\}$$

7.2 Arbitrary Precision Subtype Environments

Subtyping environments need to capture a richer set of program properties than before and, furthermore, need to both assign subtypes to variables and to give subtypes to expressions. The latter is necessary since, for example, we still need to determine the subtypes with which each function is called.

7.2.1 Environments used to determine the subtypes of expressions.

Our subtyping environments thus come in two forms. The first, which is used to determine the subtypes of expressions, is the analogue of Defn 5.5, which assigns subtypes to the formal parameters. Thus we modify Defns 5.5–5.6.

Definition 7.9 *A* ***subtyping environment of order*** *d for a function f_i is an environment in which each $x_{i,j}$ is bound to an element of $\mathsf{S}^d_{\mathrm{T}(x_{i,j})}$ d is fixed for all elements of the environment.*

A ***valid subtyping environment of order*** *d for a function f_i is a subtyping environment of order d in which each $x_{i,j}$ is* not *bound to a subtype with atomic part $\{\}$. If ϕ is a valid subtyping environment we write* ValidSub(ϕ).

Definition 7.10 *Let ϕ_1 and ϕ_2 be two subtyping environments (of order d) of function f_i. Then the* ***join*** *of ϕ_1 and ϕ_2, denoted $\phi_1 \sqcup \phi_2$, is defined thus:*

$$\phi_1 \sqcup \phi_2 \stackrel{\triangle}{=} \{x_{i,j} \mapsto \phi_1(x_{i,j}) \sqcup \phi_2(x_{i,j}) \,|\, x_{i,j} \in \mathrm{Dom}(\phi_1).\}$$

Similarly the ***meet****, denoted $\phi_1 \sqcap \phi_2$,*

$$\phi_1 \sqcup \phi_2 \stackrel{\triangle}{=} \{x_{i,j} \mapsto \phi_1(x_{i,j}) \sqcap \phi_2(x_{i,j}) \,|\, x_{i,j} \in \mathrm{Dom}(\phi_1).\}$$

As before, in order to recognise when a subtyping environment we need to determine whether a subtyping environment is included within another, as for the simple subtyping environment given in § 5.

Definition 7.11 *A* ***sub-subtyping environment of order*** *d (often written simply as sub-environment where there is no ambiguity) of a subtyping environment of order d of a function f_i, ϕ, is a subtyping environment, ϕ', for which, $\forall j.\phi'(x_{i,j}) \subseteq \phi(x_{i,j})$. We denote the fact that ϕ is a sub-subtyping environment by $\phi' \sqsubseteq \phi$.*

Analagously, we speak of ***sub-subtyping environments relative to*** *$x_{i,j}$ and conversely, we also speak of* ***super-subtyping environments****.*

Definition 7.12 *The* ***difference between two (arbitrary precision) subtype environments*** *ϕ_1 and ϕ_2, denoted $\phi_1 - \phi_2$, is defined thus:*

$$\{x_{i,j} := \phi_1(x_{i,j}) - \phi_2(x_{i,j}) \,|\, x_{i,j} \in \mathrm{Dom}(\phi_1)\}$$

However, as stated in 3 at the beginning to this section, we also wish to include information about the dependencies of the subtypes of parameters to functions. To do this, we use the standard technique of *lazy evaluation*, using the formation of closures to encode subtyping information that is used to give the subtypes of expressions. We will thus use *environment closures* rather than simple environments as parameters to our analyses.

Definition 7.13 *An* ***arbitrary precision subtype environment transformer*** *(which we shorten to environment transformer) is a function from arbitrary precision subtype environments (for the variables of some function f_i) of order d to arbitrary precision subtypes of order d.*

We write such environment transformers in the form, $\lambda\phi.E(\phi)$ and denote the set of environment transformers for f_i as $\Phi_i^d \triangleq \mathsf{Env}_i(\mathsf{S}^d) \mapsto \mathsf{S}^d$.

We normally use the shorthand form, Φ^d where i is clear from the context.

Definition 7.14 *An* ***arbitrary precision subtype closure environment*** *(written simply as subtype closure environments) consists of a pair of an environment (binding to the parameters of a function f_i) of environment transformers (where the environments bind the parameters to some function f_j) and a subtyping environment (again binding to variables of the same f_j). That is, the set of subtype closure environments for a function f_i with respect to the variables of some f_j is denoted as Ψ_i^d and defined as $\Psi_i^d \triangleq \Sigma_{f_j \in \mathbb{F}}(\mathsf{Env}_i(\Phi^d j) \times \mathsf{Env}_j(\mathsf{S}^d))$*

Again, we normally use the shorthand form, Ψ^d where i is clear from the context.

We can assign identity environment transformers to each parameter to shadow a given subtyping environment.

Definition 7.15 *Let ϕ be a subtyping environment of order d for some function f_i. Then the* ***simple subtype closure environment*** *formed from ϕ is denoted ψ_ϕ and defined, $\psi_\phi \triangleq (T, \phi)$ where $T \triangleq \{x_{i,j} := \lambda\phi.\phi(x_{i,j}) \mid x_{i,j} \in \mathrm{FP}(f_i)\}$.*

We shall need to evaluate such subtype closure environments to produce a subtype environment.

Definition 7.16 *Let ψ be a subtype closure environment. Then the* ***subtype environment evaluated from*** *ψ, denoted $\mathrm{E}(\psi) \in \mathsf{Env}_i(\mathsf{S}^d)$, is defined thus:*

$$\mathrm{E}(\psi) \triangleq \{x_{i,j} := (\mathrm{Fst}\,\psi)(x_{i,j})\,(\mathrm{Snd}\,\psi) \mid x_{i,j} \in \mathrm{Dom}(\mathrm{Fst}\,\psi)\}$$

As with subtyping environments, we need to define the ordering on subtype closure environments and the difference between two such environments.

Definition 7.17 *Let ψ_1 and ψ_2 be subtype closure environments.*

Then $\psi_1 \sqsubseteq \psi_2$ if and only if, on the subtype environment ordering (Defn 7.11), $\mathrm{E}(\psi_1) \sqsubseteq \mathrm{E}(\psi_2)$.

Analagously to subtyping environments, we refer to ψ_1 as a ***sub-subtype environment closure*** *of ψ_2.*

The ordering induces an equality, =, over subtype environment closures.

Definition 7.18 *The* ***difference between subtype closure environments*** *ψ_1 and ψ_2, denoted $\psi_1 - \psi_2 \in \Psi^d$, is defined as $\psi_1 - \psi_2 \triangleq \psi_3$, where*

$$\mathrm{Fst}(\psi_3) = \{x_{i,j} := \lambda\phi.\phi(x_{i,j}) \mid x_{i,j} \in \mathrm{Dom}(\mathrm{Fst}\psi_1)\} \quad \textit{and} \quad \mathrm{Snd}(\psi_3) = \mathrm{E}(\psi_1) - \mathrm{E}(\psi_2)$$

We can use the equality predicate over subtype closure environments to determine when a recursive invocation of one of our abstract operators has been reached.

Definition 7.19 *Two subtype closure environments,* ψ_1, ψ_2, ***match****, denoted* $\mathrm{Match}(\psi_1, \psi_2)$ *if and only if* $\psi_1 = \psi_2$ *where the equality predicate is that given in Defn 7.17.*

7.2.2 Environments used to determine the subtypes of variables.

We now define the environments used to compute the subtype of a particular parameter, $x_{i,j}$. When analysing backwards to determine the subtype of a particular variable, we cannot, naturally, start with an environment of subtypes but rather with an environment containing values which will produce a new subtype given an input subtype.

Definition 7.20 *A* ***subtype transformer****,* t*, is a function of type* $\mathsf{S}^d_T \mapsto \mathsf{S}^d_T$ *(for some type* T *and order* d*) with the additional property that* $t\,\{\} = \{\}$*. The set of subtype transformers (for arbitrary* T *and* d *is denoted* $\mathsf{S}^d_{\mapsto}$*.*

Subtype transformer terms are written in the form, $\overline{\lambda} s.E(s)$*, where* $E(s)$ *is an expression involving* s*, elements of* S^d_T*, the* $\sqcup$ *and* $\sqcap$ *operators and applications of subtype transformers.*

Definition 7.21 *A* ***backwards subtyping environment of order*** d ***relative to*** $x_{i,j}$ *(where* $x_{i,j}$ *is a formal parameter of the function* f_i*) is an environment,* $\phi_{x_{i,j}}$*, in which each formal parameter* $x_{i,l}$ *and* $x_{i,j}$ *itself is bound to an element of* $\mathsf{S}^d_{\mathrm{T}(x)} \mapsto \mathsf{S}^d_{\mathrm{T}(x)}$*.* d *is fixed for all elements of the environment.*

An ***initial backwards subtyping environment relative to*** $x_{i,j}$ *is a subtyping environment,* $\phi^I_{x_{i,j}}$*, of order* d *relative to* $x_{i,j}$ *where* $x_{i,j}$ *is bound to* $\overline{\lambda} c.c$ *and all formal parameters apart from* $x_{i,j}$ *are bound to* $\overline{\lambda} c.\top_{\mathsf{S}^d_{\mathrm{T}(x_{i,j})}}$*.*

7.2.3 Analyses to determine subtypes.

We now present the abstract interpretations which give more precise subtypes as a result. In these and the subsequent analyses, the result for ***error*** is always the $\top$ of the relevant domain.

Definition 7.22 $\mathcal{S}^d_{\mathbf{f}} \in \mathbb{E} \times \mathit{Env}(\mathsf{P}_{\mathsf{E}}) \times \mathit{Env}(\mathsf{E}) \times \Psi^d \mapsto \mathsf{S}^d$*, the* ***forwards subtyping abstract semantic operator****, is defined in Table 13.*

Definition 7.23 $\mathcal{S}^d_{\mathbf{b}} \in \mathbb{E} \times \mathit{Env}(\mathsf{P}_{\mathsf{E}}) \times \mathit{Env}(\mathsf{E}) \times \mathit{Env}(\mathsf{S}^d_{\mapsto}) \times \mathsf{S}^d \mapsto \mathsf{S}^d$*, the* ***backwards subtyping abstract semantic operator****, is defined in Table 14.*

7.2.4 Subtyping environments induced by *case* clauses.

The above subtyping regime has been introduced purely so that we can infer a more precise subtyping environment once we encounter a ***case*** expression. In order to obtain a more precise environment we need to:

$$\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, x \,\rrbracket_{\rho,\sigma}^{\psi} \triangleq \begin{cases} (\mathrm{E}(\psi))(x) & \text{if } x \in \mathrm{Dom}(\mathrm{E}(\psi)) \\ \mathrm{pp}(\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, e \,\rrbracket_{\rho,\sigma}^{\psi}, \boldsymbol{\pi}) & \text{if } \sigma(x) \equiv (\boldsymbol{\pi}, \{e\}) \\ \top_{\mathsf{S}^{d}} & \text{otherwise} \end{cases} \tag{126}$$

$$\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, f_i \,\rrbracket_{\rho,\sigma}^{\psi} \triangleq \begin{cases} f_i^{s_{\mathbf{f}},3,d}\,\{\}\,(\{\},\{\}) & \text{if } \mathrm{Ar}(f_i) = 0 \\ \{\} & \text{otherwise} \end{cases} \tag{127}$$

$$\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, C_t\, a_1 \ldots a_r \,\rrbracket_{\rho,\sigma}^{\psi} \triangleq \{(\langle\rangle, C_t)\} \cup \bigcup_{j=1}^{j=r} \mathrm{lst}(\mathcal{S}_{\mathbf{f}}^{d-1}\llbracket\, a_j \,\rrbracket_{\rho,\sigma}^{\psi}, \langle \pi_{t,j} \rangle) \tag{128}$$

$$\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, \textbf{\textit{case}}\; s\; \textbf{\textit{of}}\; \langle p_r, e_r \rangle \,\rrbracket_{\rho,\sigma}^{\psi} \triangleq \bigsqcup_{H(p_i) \in \mathcal{S}_{\mathbf{f}}^{d}\llbracket\, s \,\rrbracket_{\rho,\sigma}^{\psi}} (\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, e_i \,\rrbracket_{\rho,\sigma_i}^{\psi}) \tag{129}$$

$$\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, F\, a \,\rrbracket_{\rho,\sigma}^{\psi} \triangleq \bigcup \{ f_k^{s_{\mathbf{f}},3,d}\, \rho'\, \psi'' \mid (f_k, \boldsymbol{a}, \sigma', \psi') \in \mathcal{C}^{3,d}\llbracket\, F \,\rrbracket_{\rho,\sigma}^{\psi} \langle a \rangle \} \tag{130}$$

Table 13: Definition of $\mathcal{S}_{\mathbf{f}}^{d}\llbracket\, E \,\rrbracket_{\rho,\sigma}^{\psi}$

$$\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, x \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \triangleq \begin{cases} \chi(x)\, \mathbf{s} & \text{if } x \in \mathrm{Dom}(\chi) \\ \mathrm{pp}(\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, e \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s}, \boldsymbol{\pi}) & \text{if } \sigma(x) \equiv (\boldsymbol{\pi}, \{e\}) \\ \top_{\mathsf{S}^{d}} & \text{otherwise} \end{cases} \tag{131}$$

$$\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, f_i \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \triangleq \begin{cases} f_i^{s_{\mathbf{b}},3,d}\,\{\}\,\{\} & \text{if } \mathrm{Ar}(f_i) = 0 \\ \{\} & \text{otherwise} \end{cases} \tag{132}$$

$$\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, C_t\, a_1 \ldots a_r \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \triangleq \bigsqcup_{i=1}^{i=r} \mathcal{S}_{\mathbf{b}}^{d}\llbracket\, a_i \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \tag{133}$$

$$\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, \textbf{\textit{case}}\; s\; \textbf{\textit{of}}\; \langle p_r, e_r \rangle \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \triangleq (\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, s \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s}) \sqcap \bigsqcup_{i=1}^{i=n} (\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, e_i \,\rrbracket_{\rho,\sigma_i}^{\chi}\, \mathbf{s}) \tag{134}$$

$$\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, F\, a \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s} \triangleq \bigcup \{ f_k^{s_{\mathbf{b}},3,d}\, \rho'\, \chi'\, \mathbf{s} \mid (f_k, \boldsymbol{a}, \sigma', \psi) \in \mathcal{C}^{3,d}\llbracket\, F \,\rrbracket_{\rho,\sigma}^{\psi} \langle a \rangle \} \tag{135}$$

Table 14: Definition of $\mathcal{S}_{\mathbf{b}}^{d}\llbracket\, E \,\rrbracket_{\rho,\sigma}^{\chi}\, \mathbf{s}$

1. Obtain the subtype inferred for each formal parameter, $x_{i,j}$ of the enclosing function. We thus get a new subtype environment, ϕ_i for the ith clause of the ***case*** expression.

2. Note that we shall already have a subtyping environment, ϕ_0 (represented, in fact, by a subtype closure environment, ψ_0) and that the subtyping environment, ϕ_i, inferred from the ith clause of the ***case*** expression, should be a refinement (in the sense of being a sub-environment) of the original environment, ϕ_0. Consequently, we shall take the meet of the two environments, $\phi_0 \sqcap \phi_i$ to be the environment ϕ'_i inferred for the ith clause of the ***case*** expression.

Consequently, we have the following series of definitions.

Definition 7.24 *Let C_i be a constructor of some algebraic type T. Then the* ***subtype of order*** d ***induced by the constructor,*** C_i*, denoted* $\mathrm{I}(C_i, d)$*, is the subtype of order d defined thus:*

$$\mathrm{I}(C_i, d) \triangleq \{(\{\}, \{C_i\}\} \cup \{(p, \top_{\mathsf{S}}) \mid p \in \mathsf{P}^d, p \neq \{\}\}$$

Definition 7.25 *Let f_i be a function with formal parameters $x_{i,1} \ldots x_{i,\mathrm{Ar}(f_i)}$. Consider the* **case** *expression,* **case** s **of** $\langle p_n, e_n \rangle$*. Then the* ***subtyping environment of order*** d ***induced by the*** n***th clause of the case expression****, denoted ϕ_n, is the following subtyping environment of order d:*

$$\phi_n \triangleq \{x_{i,j} := \mathcal{S}^d_{\mathsf{b}} [\![\, s \,]\!]^{\chi}_{\rho,\sigma} (\mathrm{I}(C_n, d)) \mid x_{i,j} \in \mathrm{FP}(f_i)\}$$

Definition 7.26 *Suppose we have a function f_i and a* **case** *expression as in Defn 7.25, above. In addition, assume that we have a subtype closure environment, ψ. Then the* ***subtype closure environment refinement of*** ψ ***with respect to the*** n***th clause of the case expression****, denoted ψ_n, is defined as a simple subtyping environment, thus:*

$$\psi_n \triangleq \psi_{\phi'} \qquad \textit{where} \qquad \phi' \triangleq \phi_n \sqcap \mathrm{E}(\psi)$$

7.3 Modifying the Analyses

We consequently produce new versions of our analyses. The changes are relatively minimal since the subtyping mechanism is in general separated from our analyses. The changes to be made to the analyses are as follows:

- We require a new subtyping environment to be calculated for each clause of a case expression in the calls analysis. The flow of information is from the head constructor of the pattern to a parameter of the function f_i that forms the current context. This idea is encapsulated in Defn 7.26 above.

- However, the environment must not be refined during size analysis. The reason for this is that otherwise each argument to a recursive call could then potentially be given a different subtyping environment, which would be unsound. Nevertheless, we need to attempt to maintain dependency information between the abstract subtypes of various parameters. This is why we use subtype closure environments (see Defn 7.15) rather than subtyping environments.

- Calls analysis must now produce a sequence of *pairs*, each consisting of a CST and a subtyping environment.

7.3.1 Closure analysis with arbitrary precision subtyping.

$$
\begin{aligned}
\mathcal{C}^{3,d} \llbracket E \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \{\} \text{ if } E \text{ is of ground type} \\
\mathcal{C}^{3,d} \llbracket x \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \begin{cases} \top_{\mathsf{C}} & \text{if } \rho(x) = \top_{\mathsf{E}} \\ \{(\{\}, \boldsymbol{a}, \sigma, \psi)\} & \text{if } \rho(x) = \{\} \\ \mathcal{C}^{3,d} \llbracket e \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} & \text{if } \rho(x) = \{e\} \\ \bigcup_{(\{\},\{e\}) \in \mathcal{P}^{3,d} \llbracket (\boldsymbol{\pi}, \sigma(x)) \rrbracket^{\psi}_{\rho,\sigma}\}} \mathcal{C}^{3,d} \llbracket e \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} & \text{if } x \in \mathrm{Dom}(\sigma) \wedge \mathcal{P}^{3,d} \llbracket (\boldsymbol{\pi}, \sigma(x)) \rrbracket^{\psi}_{\rho,\sigma} \\ \top_{\mathsf{C}} & \text{otherwise} \end{cases} \\
\mathcal{C}^{3,d} \llbracket f_i \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \begin{cases} \{(\{f_i\}, \boldsymbol{a}, \sigma, \psi)\} & \text{if } \mathrm{Ar}(f_i) \geq |\boldsymbol{a}| \\ \{(f, \boldsymbol{e}, \sigma'', \psi') \\ \quad \mid (f, \boldsymbol{d}, \sigma', \psi') \in f_i^m \, \rho' \, \boldsymbol{c}\} & \text{otherwise} \end{cases} \\
\mathcal{C}^{3,d} \llbracket C_t \, a_1 \ldots a_r \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \bigcup_{i=1}^{i=r} \{(f, \boldsymbol{b}, \sigma', \psi') \mid (f, \boldsymbol{b}, \sigma', \psi') \in \mathcal{C}^{3,d} \llbracket e_i \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} \wedge \mathrm{TC}(f, \boldsymbol{b})\} \\
\mathcal{C}^{3,d} \llbracket \textbf{\textit{case}} \; s \; \textbf{\textit{of}} \; \langle p_r, e_r \rangle \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \bigcup_{k=1}^{k=r} \bigcup \{\mathcal{C}^{3,d} \llbracket e_k \rrbracket^{\psi_k}_{\rho_k,\sigma_k} \boldsymbol{a} \mid H(p_k) \in \mathcal{S} \llbracket s \rrbracket^{\phi}_{\rho,\sigma}\} \\
\mathcal{C}^{3,d} \llbracket G \, d \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a} &\triangleq \mathcal{C}^{3,d} \llbracket G \rrbracket^{\psi}_{\rho,\sigma} (\langle \{d\} \rangle + \!\!\! + \, \boldsymbol{a})
\end{aligned}
$$

Table 15: Definition of $\mathcal{C}^{3,d} \llbracket E \rrbracket^{\psi}_{\rho,\sigma} \boldsymbol{a}$

Definition 7.27 *The* ***closure analysis operator with*** d***th order subtyping****,* $\mathcal{C}^{3,d} \in \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M}_{\mathsf{P}} \times \Psi^d \times \mathsf{E}^* \mapsto \mathsf{C}$*, is defined in Table 15.*

Definition 7.28 *The* ***abstract closure function with subtype closure environment****,* ψ *of a function,* $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$*, is defined for a given environment of non-ground expressions* ρ*, and a sequence of actual parameter expressions,* $\boldsymbol{a}$*, as* $f_{f_i}^{m_{3,d}}\, \psi\, \rho\, \boldsymbol{a} \triangleq \bigcup_{\phi' \in \mathrm{SP}(()\phi)} \mathcal{C}^{3,d}\, [\![\, e_i \,]\!]_{\rho,\{\}}^{\phi'}\, \boldsymbol{a}$

7.3.2 Projection analysis with arbitrary precision subtyping.

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, \top_{\mathsf{E}}) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \pi_e^l \tag{142}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, (\{\}, e)) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \{(\{\}, e)\} \tag{143}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, x) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \{(\boldsymbol{\pi}, x)\} \tag{144}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, f_k) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \{\} \tag{145}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, C_t\, a_1 \ldots a_r) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \begin{cases} \{a_j\} & \text{if } \boldsymbol{\pi} \equiv \langle \pi_{i,j} \rangle \\ \mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi'}, a_j) \,]\!]_{\rho,\sigma}^{\psi} & \text{if } \boldsymbol{\pi} \equiv \boldsymbol{\pi'} +\!\!+ \langle \pi_{i,j} \rangle \\ \{\} & \text{otherwise} \end{cases} \tag{146}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, \textbf{case}\, s\, \textbf{\textit{of}}\, \langle p_r, e_r \rangle) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \bigcup_{k=1}^{k=r} \bigcup \{\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, \{e_k\}) \,]\!]_{\rho,\sigma_k}^{\psi} \mid H(p_k) \in \mathcal{S}\, [\![\, s \,]\!]_{\rho,\sigma}^{\phi}\} \tag{147}$$

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, F\, a) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \bigcup \begin{array}{l} \{b[\boldsymbol{a} /_{\mathsf{E}}\, \boldsymbol{x_k}] \\ \quad \mid \begin{array}{l} b = f_f^{p_2,d}\, \rho'\, \psi''\, \boldsymbol{\pi}, \\ (\{f_k\}, \boldsymbol{a}, \sigma', \psi') \in \mathcal{C}^1\, [\![\, F \,]\!]_{\rho,\sigma}^{\psi}\, \langle a \rangle \end{array} \} \end{array} \tag{148}$$

In (148), if, for some $\boldsymbol{a}, \sigma', \psi'$, $\mathcal{C}^1\, [\![\, F \,]\!]_{\rho,\sigma}^{\psi}\, \langle a \rangle = (\{\}, \boldsymbol{a}, \sigma', \psi')$ then

$$\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, F\, a) \,]\!]_{\rho,\sigma}^{\psi} \triangleq \top_{\mathsf{E}}$$

Table 16: Definition of $\mathcal{P}^{3,d}\, [\![\, (\boldsymbol{\pi}, E) \,]\!]_{\rho,\sigma}^{\psi}$

Definition 7.29 *The* ***projection analysis operator with*** d***th order subtyping****,* $\mathcal{P}^{3,d} \in \mathsf{P}_{\mathsf{E}}^1 \times \mathit{Env}(\mathsf{E}) \times \mathsf{M} \times \Psi^d \mapsto \wp(\mathsf{P}_{\mathsf{E}}^l)$*, is defined in Table 16 for projection expressions where the projection sequence is neither endomorphic nor* $\top_{\mathsf{P}}$*.*

In the case of the endomorphic projection sequences, $\mathcal{P}$ *corresponds to the injection,* $p \mapsto \{p\}$*.*

In the case where the projection sequence is $\top_{\mathsf{P}}$*, the result is* $\mathsf{P}_{\mathsf{E}}^l$*, the top of* $\wp(\mathsf{P}_{\mathsf{E}}^l)$*.*

$$\mathcal{A}^{3,d}_{i,j} \llbracket x \rrbracket^{\psi}_{\rho,\sigma} \triangleq \begin{cases} (\{\}, 0) & \text{if } x \equiv x_{i,j} \\ (\{\}, -\omega) & \text{if } x \equiv x_{i,k} \\ \max_{p \in \mathcal{P}^{3,d} \llbracket (\boldsymbol{\pi}, \sigma(x)) \rrbracket^{\psi}_{\rho,\sigma}\}} \mathrm{PES}(p) & \text{if } x \in \mathrm{Dom}(\sigma) \wedge \mathcal{P}^{3,d} \llbracket (\boldsymbol{\pi}, \sigma(x)) \rrbracket^{\psi}_{\rho,\sigma} \neq \top_{\mathsf{P_E}} \\ \top_{\mathsf{R_P}} & \text{otherwise} \end{cases} \tag{149}$$

$$\mathcal{A}^{3,d}_{i,j} \llbracket f_k \rrbracket^{\psi}_{\rho,\sigma} \triangleq \begin{cases} f^{a}_{k,0}\, \{\} & \text{if } \mathrm{Ar}(f_k) = 0 \\ (\{\}, -\omega) & \text{if } \mathrm{Ar}(f_k) \neq 0 \end{cases} \tag{150}$$

$$\mathcal{A}^{3,d}_{i,j} \llbracket C_t\, a_1 \ldots a_r \rrbracket^{\psi}_{\rho,\sigma} \triangleq \mathrm{cs}^{3,d}(\mathrm{Rec}(E), i, j, \rho, \sigma, \psi) \tag{151}$$

$$\mathcal{A}^{3,d}_{i,j} \llbracket \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r \rangle \rrbracket^{\psi}_{\rho,\sigma} \triangleq \max(\bigcup_{k=1}^{k=r} \{\mathcal{A}^{3,d}_{i,j} \llbracket e_k \rrbracket^{\psi}_{\rho,\sigma_k} \mid H(p_k) \in \mathcal{S} \llbracket s \rrbracket^{\phi}_{\rho,\sigma}\}) \tag{152}$$

$$\mathcal{A}^{3,d}_{i,j} \llbracket F\, a \rrbracket^{\psi}_{\rho,\sigma} \triangleq \max\{\mathrm{ap}^{a_{3,d}}(f, i, j, \boldsymbol{a}, \rho', \sigma, \psi) \mid (f, \boldsymbol{a}, \sigma', \psi') \in \mathcal{C}^1 \llbracket F \rrbracket^{\psi}_{\sigma,\rho} \langle \{a\} \rangle\} \tag{153}$$

Table 17: Definition of $\mathcal{A}^{3,d}_{i,j} \llbracket E \rrbracket^{\psi}_{\rho,\sigma}$

7.3.3 Size analysis with arbitrary precision subtyping.

Definition 7.30 *The* ***relative size analysis operator with*** d***th order subtyping****,* $\mathcal{A}^{3,d} \in \mathbb{I}^{\mathsf{S}}_{f_i} \times \mathbb{E} \times \textsf{\textit{Env}}(\mathsf{E}) \times \mathsf{M_P} \times \Psi^d \mapsto \mathsf{R_P}$*, is the extension of* $\mathcal{A}$ *with arbitrary precision subtyping of order* d*, and defined over the structure of expressions in Table 17. In the definition,* ρ *is an environment binding function type expressions to variables,* σ *is an environment binding pattern-matching variables of algebraic types to expressions, and* ψ *is a subtype closure environment binding subtypes and environment transformers to the formal parameters.* i *is a function index whilst* $0 \leq j \leq \mathrm{Ar}(f_i)$*.*

Definition 7.31 *The* ***constructor abstract size function with arbitrary precision subtyping****,* $\mathrm{cs}^{3,d} \in \wp(\mathbb{E}) \times \mathbb{I}^{\mathsf{S}}_{f_i} \times \textsf{\textit{Env}}(\mathsf{E}) \times \Psi^d \times \mathsf{M_P} \mapsto \mathsf{R_P}$*, is defined analagously to Defn 5.15, with* $\mathcal{A}^{3,d}$ *replacing* $\mathcal{A}^1$*.*

Definition 7.32 *The* $\mathcal{A}^1$ *operator is lifted to the* E *domain as follows:*

$$\mathcal{A}^{3,d}_{i,j} \llbracket \top_{\mathsf{E}} \rrbracket^{\psi}_{\rho,\sigma} \triangleq \top_{\mathsf{R_P}} \tag{154}$$

$$\mathcal{A}^{3,d}_{i,j} \llbracket \{e\} \rrbracket^{\psi}_{\rho,\sigma} \triangleq \mathcal{A}^{3,d}_{i,j} \llbracket e \rrbracket^{\psi}_{\rho,\sigma} \tag{155}$$

Definition 7.33 *The* ***abstract applicator for size analysis with arbitrary precision subtyping of order*** d*,* $\mathrm{ap}^{a_{3,d}}$*, is defined as follows.*

$$\mathrm{ap}^{a_{3,d}}(\top_{\mathsf{F}}, i, j, \boldsymbol{a}, \sigma, \rho, \psi) \triangleq \omega \tag{156}$$

$$\mathrm{ap}^{a_{3,d}}(\{\}, i, j, \boldsymbol{a}, \sigma, \rho, \psi) \triangleq \omega \tag{157}$$

$$\mathrm{ap}^{a_{3,d}}(\{f_k\}, i, j, \boldsymbol{a}, \sigma, \rho, \psi) \triangleq (\boldsymbol{f_k}^{a_{3,d}} * \boldsymbol{a}^{a_{3,d}}) + vj \tag{158}$$

In the above, $\phi' \triangleq \{x_{k,1} := \mathcal{S} \llbracket a_1 \rrbracket^{\phi}_{\rho,\sigma} \ldots x_{k,\mathrm{Ar}(f_k)} := \mathcal{S} \llbracket a_{\mathrm{Ar}(f_k)} \rrbracket^{\phi}_{\rho,\sigma}\}$. *In addition,* $\phi'\{x_{k,l} := \top_{\mathsf{S}^d}\}$, *if* $l > |\boldsymbol{a^s}|$.

$\boldsymbol{f_k^{a_{3,d}}} \triangleq [f^{a_{3,d}}_{k,1}\, \rho'\, \phi' \ldots f^{a_{3,d}}_{k,\mathrm{Ar}(f_k)}\, \rho'\, \phi']$ *and* $\boldsymbol{a^{a_{3,d}}} \triangleq [\mathcal{A}^{3,d}_{i,j} \llbracket a_1 \rrbracket^{\psi}_{\rho,\sigma} \ldots \mathcal{A}^{3,d}_{i,j} \llbracket a_{|\boldsymbol{a}|} \rrbracket^{\psi}_{\rho,\sigma}]$.

$$v_j \triangleq \begin{cases} f^{a_{3,d}}_{k,0}\, \rho'\, \phi' & \textit{if } j = 0 \\ (\{\}, -\omega) & \textit{otherwise} \end{cases}$$

Definition 7.34 *The* ***abstract size function with arbitrary precision subtyping of order*** d *of a function,* $f_i \overset{def}{=} \lambda x_{i,1} \ldots x_{i,n}.e_i$, ***relative to parameter*** j *is defined for a given subtype closure environment,* ψ *and a given environment of function-type parameters,* ρ *as,* $f^{a_{3,d}}_{i,j}\, \psi\, \rho \triangleq \max_{\phi' \in \mathrm{SP}(()\phi_i)} \mathcal{A}^{3,d}_{i,j} \llbracket e_i \rrbracket^{\phi'}_{\rho,\{\}}$

7.3.4 Calls analysis with arbitrary precision subtyping.

$$\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket x \rrbracket^{\psi}_{\rho,\sigma} \triangleq \langle\rangle \tag{159}$$

$$\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket f_k \rrbracket^{\psi}_{\rho,\sigma} \triangleq \begin{cases} f^{g_{3,d}}_{k[j]}\, \{\}\, \{\} & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k \neq j \\ \langle(\boldsymbol{\Omega}, \mathrm{E}\psi)\rangle & \text{if } \mathrm{Ar}(f_k) = 0 \wedge k = j \\ \langle\rangle & \text{otherwise} \end{cases} \tag{160}$$

$$\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket C_t\, a_1 \ldots a_r \rrbracket^{\psi}_{\rho,\sigma} \triangleq \biguplus_{k=1}^{k=r} \mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket a_k \rrbracket^{\psi}_{\rho,\sigma} \tag{161}$$

$$\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket \textbf{\textit{case}}\ s\ \textbf{\textit{of}}\ \langle p_r, e_r\rangle \rrbracket^{\psi}_{\rho,\sigma} \triangleq \biguplus(\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket s \rrbracket^{\psi}_{\rho,\sigma}, (\biguplus_{k=1}^{k=r} G_k)) \tag{162}$$

$$\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket F\, a \rrbracket^{\psi}_{\rho,\sigma} \triangleq \biguplus_{\substack{(f, \boldsymbol{a}, \sigma', \psi') \in \\ \mathcal{C}^1 \llbracket F \rrbracket^{\psi}_{\rho,\sigma} \langle\{a\}\rangle}} (\mathrm{ap}^{g_{3,d}}(f, i, j, \boldsymbol{a}, \rho', \sigma', \psi') \biguplus (\biguplus_{i=1}^{i=|\boldsymbol{a}|} \mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket a_i \rrbracket^{\psi}_{\rho,\sigma})) \tag{163}$$

In (162),

$$G_k = \begin{cases} \mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket e_k \rrbracket^{\psi_k}_{\sigma_k,\rho} & \text{if } H(p_k) \in \mathcal{S} \llbracket s \rrbracket^{\psi_k}_{\rho,\sigma} \\ \{\} & \text{otherwise} \end{cases}$$

Table 18: Definition of $\mathcal{G}^{3,d}_{i[j,\phi_j]} \llbracket E \rrbracket^{\psi}_{\rho,\sigma}$

Definition 7.35 *The* ***abstract calls operator with*** d***th order subtyping***, $\mathcal{G}^{3,d} \in \mathbb{I}^{\mathsf{S}}_{f} \times \mathbb{I}^{\mathsf{S}}_{f} \times \mathbb{E} \times \mathit{Env}(\mathsf{E}) \times \mathsf{M}_{\mathsf{P}} \times \Psi^d \mapsto (\mathsf{T} \times \mathit{Env}(\mathsf{S}))^*$, *(the extension of* $\mathcal{G}$ *with arbitrary precision* *subtyping of order d), locates calls of function f_j within function f_i (which has input subtype closure environment, ψ), and is defined over the structure of expressions in Table 18.*

Note that, in contrast to previous members of the hierachy of abstract calls operators which each produced a sequence of CSTs, $\mathcal{G}^{3,d}$ produces a sequence of CST, subtyping environment pairs.

Definition 7.36 *The* ***abstract applicator for calls analysis with arbitrary precision subtyping****,* $\mathrm{ap}^{g_{3,d}} \in \mathsf{F} \times \mathbb{I}^{\mathbf{S}}_f \times \mathbb{I}^{\mathbf{S}}_f \times \mathsf{E}^* \times \mathsf{M}_\mathsf{P} \times \mathit{Env}(\mathsf{E}) \times \mathit{Env}(\mathsf{S}) \mapsto (\mathsf{T} \times \mathit{Env}(\mathsf{S}))^*$*, is defined as follows*

$$\mathrm{ap}^{g_{3,d}}(\top_\mathsf{F}, i, j, \boldsymbol{a}, \rho, \sigma, \phi) \stackrel{\triangle}{=} \langle \top_\mathsf{T} \rangle \tag{164}$$

$$\mathrm{ap}^{g_{3,d}}(\{\}, i, j, \boldsymbol{a}, \rho, \sigma, \phi) \stackrel{\triangle}{=} \langle\rangle \tag{165}$$

$$\mathrm{ap}^{g_{3,d}}(\{f_k\}, i, j, \boldsymbol{a}, \rho, \sigma, \phi) \stackrel{\triangle}{=} \begin{cases} \langle\rangle & \text{if } (|\boldsymbol{a}| < \mathrm{Ar}(f_k)) \\ \{(\mathbf{T}(i, \boldsymbol{a}, \rho, \sigma, \phi), \phi)\} \cup R & \text{if } (f_k \equiv f_j) \\ \langle(\top_\mathsf{T}, \phi)\rangle & \mathit{if} (f_k \not\equiv f_j) \\ & \quad \wedge \neg \mathrm{ADP}(f_k, \phi') \\ \biguplus_{(\mathbf{T}', \phi') \in f^{g_{3,d}}_{k[j]} \rho' \psi'} (\mathrm{Map}\,(\star\mathbf{T}(i, \boldsymbol{a}, \rho, \sigma, \phi), \phi')\,\mathbf{T}') & \text{if } f_k \not\equiv f_j \end{cases} \tag{166}$$

In the above, if $\phi'' = \phi - \phi_j$ *is a valid subtyping environment then* $R \equiv \mathrm{ap}^{g_{3,d}}(\{f_k\}, i, \boldsymbol{a}, \sigma, \rho, \phi'')$*. Otherwise,* $R \equiv \langle\rangle$

Definition 7.37 *For each function, there is 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 with subtype closure environment, ψ and environment of function-type arguments, ρ.*

$$f^{g_{3,d}}_{i[j]}\, \rho\, \psi \stackrel{\triangle}{=} \bigcup_{\phi' \in \mathrm{SP}(()\phi_i)} \mathcal{G}^{3,d}_{i[j,\phi_j]} [\![\, e_i \,]\!]^{\phi'}_{\rho,\{\}}$$

7.4 Termination Criteria Using Arbitrary Precision Subtyping

We now proceed, in the light of our arbitrary precision subtyping constructions, to redefine our abstract semantic criteria that assure termination.

Firstly, as the subtypes have become more sophisticated, so their minimal size can be other than a binary value.

Definition 7.38 *Assume we have $s \in \mathsf{S}$. Then the* ***minimal subtype size*** *of s, denoted* $\mathrm{mss} \in \mathsf{S} \mapsto \mathsf{R}_\mathsf{P}$*, is defined thus:*

$$\mathrm{mss}(s) \stackrel{\triangle}{=} \min\{\mathrm{mcs}(C_t) \mid C_t \in s\}$$

Definition 7.39 *The j***th weighting vector with respect to an arbitrary precision environment** *ϕ is a vector with, in the jth position,* $\mathrm{mss}^{3,d}(\phi(x_{i,j})$ $\top_{\mathsf{R}_\mathsf{P}}$ *is in all other positions, regardless of their types.*

Definition 7.40 *The **abstract subtyped calls set** of a function f_i, is denoted, for the order d, $\mathbf{ASC}(i,d) \in \mathsf{T} \times \mathit{\mathsf{Env}}(\mathsf{S}^d)$ and is defined as, $\mathbf{ASC}(i,d) \triangleq f_{i[i]}^{g_{3,d}}\,\{\}$.*

Definition 7.41 *The **abstract call matrix** of recursive calls of function f_i is defined with respect to a dth-order subtyping environment, ϕ, thus:*

$$\mathbf{ACM}(i,d,\phi) \triangleq \{\boldsymbol{r} \mid ((\mathbf{v},\boldsymbol{c}),\phi') \in \mathbf{ASC}(i,d); \phi' \sqsubseteq \phi\}$$

where, if $x_{i,j}$ is an algebraic argument, $r_j \triangleq \mathrm{N}(\boldsymbol{w_j}\mathbf{v}_j + c_j - \mathrm{mss}^{3,d}(\phi(x_{i,j})))$, $\boldsymbol{w_j}$ is the jth weighting vector with respect to the subtyping environment ϕ and $\mathbf{v}_j$ is the jth column of $\mathbf{v}$.

If $x_{i,j}$ is non-algebraic then $r_j \triangleq \omega$.

Definition 7.42 *The jth argument to f_i (i.e. $x_{i,j}$) with subtyping environment ϕ is said to be an **abstractly monotonic descending argument**, written $\mathrm{AMD}(x_{i,j},d,\phi)$ (or simply $\mathrm{AMD}(j,\phi)$ where the function and subtyping contexts are clear), if*

$$\forall \boldsymbol{r_l} \in \mathbf{ACM}(i,d,\phi).(r_{l,j} \leq 0) \wedge (\exists d.r_{d,j} < 0)$$

*The jth argument is said to be **abstractly strictly descending**, written $\mathrm{ASD}(x_{i,j},d,\phi)$ if*

$$\forall \boldsymbol{r_l} \in \mathbf{ACM}(i,d,\phi).(r_{l,j} < 0)$$

Definition 7.43 *A function f_i has the **abstract descent property for the subtyping environment** ϕ, denoted $\mathrm{ADP}(A)$, where $A \equiv \mathbf{ACM}(i,\phi)$, if and only if*

$$\exists j.\mathrm{AMD}(j,\phi) \wedge \mathrm{ADP}(A')$$

where $A' = \{\boldsymbol{r_e} \mid (\boldsymbol{r_e} \in A) \wedge (r_{e,j} = 0)\}$

Theorem 7.1 *If a function has the abstract descent property for a subtyping environment, ϕ, then it has the monotonic descent property on the same set of subtyping assumptions, where the subtypes are of order d for some fixed d.*

Proof. The proof follows the same structure as previously. □

7.5 ESFP^{3,d}

The modified analysis above produces the final version of our ESFP language.

Definition 7.44 *For some given natural number d, the **language** $\mathrm{ESFP}^{3,d}$ consists of EFP^{+} together with a check that all definitions within a script have the abstract descent property for some valid subtyping environment with arbitrary precision subtypes of order d and analysing with projection expressions of length d.*

7.6 Example Using Arbitrary Precision Subtyping

We now give, as an example of the backwards subtyping termination analysis, an account of the analysis of *mergeSort*.

Example 7.1 [Mergesort] The definition of the function, which is that used in [34], is as follows:

$$
\begin{array}{l}
\mathit{mergeSort}\ \mathit{merge}\ x\\
\quad \mid \mathit{length}\, x < 2 \overset{def}{=} x\\
\quad \mid \textbf{otherwise} \overset{def}{=} \mathit{merge}\ (\mathit{mergeSort}\ \mathit{merge}\ \mathit{first})\\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{mergeSort}\ \mathit{merge}\ \mathit{second})\\
\quad \textbf{where}\\
\quad\ \mathit{first} \quad\ \overset{def}{=} \mathit{take}\ \mathit{half}\ x\\
\quad\ \mathit{second} \overset{def}{=} \mathit{drop}\ \mathit{half}\ x\\
\quad\ \mathit{half} \quad\ \overset{def}{=} (\mathit{length}\, x)\ \mathit{div}\ 2
\end{array}
$$

The analysis can show that *mergeSort* is in $\mathrm{ESFP}^{3,d}$ for $d \geq 2$ since the information that the size of the list x is at least 2 is propagated throughout the part of the analysis corresponding to the second clause. Thus the analysis is capable of detecting that both *take half x* and *drop half x* produce a reduction in the length of their arguments.

8 Strong Normalisation and Analysis Frameworks

We can generalise our analysis further to allow different notions of reduction and to develop a generalised framework for our analysis. As discussed in § 2, our operational semantics only reduces to *weak normal form*. Consequently, our analysis only assures termination under the given reduction order. This is sufficient with respect to languages such as Haskell or ML, since both do not have a stronger notion of normal form. Conversely, Miranda, Haskell and other so-called lazy languages only reduce to weak head normal form (WHNF). Both for pedagogical reasons and the desire to have sound program transformations, we believe that strong normalisation is worth pursuing. With regard to the former, the assurance that a program is strongly normalising will, we believe, help students to construct better programs. With regard to the latter, program transformations may fail in the case where we expand the expressions bound by lambda abstractions.

8.1 Analyses Parameterised By The Operational Semantics

It has been proposed by Cousot that strictness analyses can be parameterised by their semantics [9]. We take a similar approach here in sketching out how our analyses can be generalised to take account of weaker (*à la* Haskell) or stronger notions of normal form.

For each operational semantics (and consequent definition of normal form) the main point of departure is for function applications. In that case we can use the operational semantics to determine whether parameters or, indeed, function bodies should be reached by the analysis. This can be achieved by adding a reachability predicate (which would depend on the operational semantics) to the ap^a and $\mathcal{A}^{2,d}$ operators. With WHNF, for instance, we would not scan an actual parameter of a function for recursive calls if the function did not use that argument.

9 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. [39]). Most of this work does not address the issue of fully automated termination checks for programs, with [14] being an exception. In more specific programming areas, Giesl has worked on automated termination proofs for nested, mutually recursive and partial functional programs [16, 4]. Closely related to this, Brauburger has produced an automated termination analysis for partial functions [3] using Giesl's synthesising techniques for polynomial orderings [15]. Closely related is the work of Slind on TFL [30], and like the previous work is based on automatically generating term orderings and termination predicates within a theorem-proving environment. A decidable test for a broader class of definitions than primitive recursion has also been established for Walther recursion [38, 21]. 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 [27] has used Nöcker's abstract reduction technique (whereby the standard evaluation of a program is replicated with abstract values; [25]) 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.

10 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, for the basic case uses the same domain of values employed to analyse the dual, corecursive case. We have then developed the abstract interpretation so that partial functions may be admitted due to a subtyping mechanism. Furthermore, by using representations of projections we have been able to add functions that recurse over nested data structures. This methodology was then used to develop a more sophisticated subtyping mechanism which meant that more complex descent mechanisms could be recognised.

The methods that we have developed could be incorporated within a compiler for an elementary strong functional programming language. Indeed, we are currently working on the implementation of our basic methodology for the EFP language. We suggest also that this method could be used to extend the current algorithm within systems such as Coq [5].

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 [10]. 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 [18], 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.

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. Automatic termination analysis for partial functions using polynomial orderings. In Van Hentenryck [37].

[4] 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.

[5] The Coq project. WWW page by INRIA and CNRS, France, 1996. `http://pauillac.inria.fr/~coq/coq-eng.html`.

[6] 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.

[7] P. Cousot. Semantic foundations of program analysis. In S.S. Mucknick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

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

[9] P. Cousot. Abstract interpretation based static analysis parameterized by semantics (invited paper). In Van Hentenryck [37].

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. Giesl. Termination analysis for functional programs using term orderings. In A. Mycroft, editor, *SAS '95*, volume 983 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17(1):47–62, January 1995.

[27] S. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In Van Hentenryck [37].

[28] 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`.

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

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

[31] 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.

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

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

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

[35] 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.

[36] 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.

[37] P. Van Hentenryck, editor. *Static Analysis, 4th International Symposium, SAS '97. Paris, France, September 1997*, volume 1302 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[38] C. Walther. On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[39] 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.