A Polynomial-Cost Non-determinism Analysis*

Ricardo Peña and Clara Segura

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
e-mail: {ricardo,csegura}@sip.ucm.es

Abstract. This paper is an extension of a previous work where two non-determinism analyses were presented. One of them was efficient but not very powerful and the other one was more powerful but very expensive. Here, we develop an intermediate analysis in both aspects, efficiency and power. The improvement in efficiency is obtained by speeding up the fixpoint calculation by means of a widening operator, and the representation of functions through easily comparable signatures. Also details about the implementation and its cost are given.

1 Introduction

The parallel-functional language Eden [2] extends the lazy functional language Haskell by constructs to explicitly define and communicate processes. It is implemented by modifying the *Glasgow Haskell Compiler* (GHC) [11]. The three main new concepts are *process abstractions*, *process instantiations* and the non-deterministic process abstraction `merge`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, and process instantiations can be compared to function applications. An instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, a new parallel process is created to evaluate `(e1 e2)`. Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]` which *fairly* interleaves a set of input lists, to produce a single non-deterministic list.

The presence of non-determinism creates some problems in Eden: It affects the referential transparency [8, 17] of programs and invalidates some optimizations done in the GHC [14]. Such problems were precisely described in [9]. In [9] a solution was proposed to solve this problem: To develop a static analysis to determine when an Eden expression is sure to be deterministic and when it may be non-deterministic. Two different abstract interpretation based analyses were presented and compared with respect to expresiveness and efficiency. The first one $[\![\cdot]\!]_1$ was efficient (linear) but not very powerful, and the second one $[\![\cdot]\!]_2$ was powerful but less efficient (exponential). This paper presents an intermediate analysis $[\![\cdot]\!]_3$ that tries to be a compromise between power and efficiency and describes its implementation. Its definition is based on the second analysis $[\![\cdot]\!]_2$. The improvement in efficiency is obtained by speeding up the fixpoint calculation by means of a widening operator *wop*, and by using an easily comparable representation of functions. By choosing different operators we obtain

* Work partially supported by the Spanish-British Acción Integrada HB 1999-0102 and Spanish project TIC 2000-0738.

different variants of the analysis $[\![\cdot]\!]_3^{wop}$. The paper describes the analysis and one particular variant $[\![\cdot]\!]_3^{\mathcal{W}}$ in detail. It also describes an algorithm, written in Haskell, that implements the analysis and annotates the program expressions with non-determinism information, so that it can be used to avoid the harmful transformations.

The plan of the paper is as follows: In Section 2 the language and the analysis $[\![\cdot]\!]_2$ are briefly summarised (full details in [9, 16]). In Section 3 the new analysis $[\![\cdot]\!]_3^{\mathcal{W}}$ is described. First, some theoretical results that help in the implementation of the analysis are presented, and then its relation with $[\![\cdot]\!]_2$ is studied. We mention other variants of the analysis and their mutual relations. In Section 4 we describe the annotation algorithm and its cost. In Section 5 some conclusions are drawn. The proofs of all the propositions and examples of the output produced by the algorithm can be found in [16].

2 A Non-determinism Analysis

2.1 The Language

The language being analysed is an extension of Core-Haskell [11], i.e. a simple functional language with second-order polymorphism, so it includes type abstraction and type application. A program is a list of possibly recursive bindings from variables to expressions. Such expressions include variables, lambda abstractions, applications of a functional expression to an atom, constructor applications $C_j\ \overline{x_j}$, primitive operators applications, and also **case** and **let** expressions. We will use v to denote a variable, k to denote a literal, and x to denote an atom (a variable or a literal). Constructor and primitive operators applications are saturated. In **case** expressions there may be a default alternative, denoted as $[v \to e]$ to indicate it is optional.

The variables contain type information, so we will not write it explicitly in the expressions. When necessary, we will write $e :: t$ to make explicit the type of an expression. A type may be a basic type K, a type variable β, a tuple type $(t_1, \ldots, t_m)$, an algebraic (sum) type $T\ t_1 \ldots t_m$, a functional type $t_1 \to t_2$ or a polymorphic type $\forall \beta.t$. The new Eden expressions are a process abstraction **process** $v \to e$, and a process instantiation $v\ \#\ x$. There is also a new type *Process* $t_1\ t_2$ representing the type of a process abstraction **process** $v \to e$ where v has type t_1 and e has type t_2. Frequently t_1 and t_2 are tuple types and each tuple element represents an input or an output channel of the process.

2.2 The analysis

In Figure 1 the abstract domains for $[\![\cdot]\!]_2$ are shown. There is a domain *Basic* with two values: d represents *determinism* and n *possible non-determinism*, with the ordering $d \sqsubseteq n$. This is the abstract domain corresponding to basic types and algebraic types. The abstract domains corresponding to a tuple type and a function/process type are respectively the cartesian product of the components' domains and the domain of continuous functions between the domains of the argument and the result. Polymorphism is studied below.

$$
\begin{array}{l}
Basic = \{d, n\} \text{ where } d \sqsubseteq n \\
D_{2K} = D_{2T\ t_1 \ldots t_m} = D_{2\beta} = Basic \\
D_{2(t_1,\ldots,t_m)} = D_{2t_1} \times \ldots \times D_{2t_m} \\
D_{2t_1 \to t_2} = D_{2Process\ t_1\ t_2} = [D_{2t_1} \to D_{2t_2}] \\
D_{2\forall\beta.t} = D_{2t}
\end{array}
$$

Fig. 1. Abstract domains for the analysis

$$
\begin{array}{ll}
\alpha_t : D_{2t} \to Basic & \gamma_t : Basic \to D_{2t} \\
\alpha_K = \alpha_{T\ t_1 \ldots t_m} = \alpha_\beta = id_{Basic} & \gamma_K = \gamma_{T\ t_1 \ldots t_m} = \gamma_\beta = id_{Basic} \\
\alpha_{(t_1,\ldots,t_m)}(e_1, \ldots, e_m) = \bigsqcup_i \alpha_{t_i}(e_i) & \gamma_{(t_1,\ldots,t_m)}(b) = (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) \\
\alpha_{Process\ t_1\ t_2}(f) = \alpha_{t_1 \to t_2}(f) & \gamma_{Process\ t_1\ t_2}(b) = \gamma_{t_1 \to t_2}(b) \\
\alpha_{t_1 \to t_2}(f) = \alpha_{t_2}(f(\gamma_{t_1}(d))) & \gamma_{t_1 \to t_2}(b) = \begin{cases} \lambda z \in D_{2t_1}.\gamma_{t_2}(n) \text{ if } b = n \\ \lambda z \in D_{2t_1}.\gamma_{t_2}(\alpha_{t_1}(z)) \text{ if } b = d \end{cases} \\
\alpha_{\forall\beta.t} = \alpha_t & \gamma_{\forall\beta.t} = \gamma_t
\end{array}
$$

Fig. 2. Abstraction and concretisation functions, α_t and γ_t

In Figure 3 the abstract interpretation for this analysis is shown. It is an abstract interpretation based analysis in the style of [3]. We outline here only some cases. The interpretation of a tuple is the tuple of the abstract values of the components. Functions and processes are interpreted as abstract functions. So, application and process instantiation are interpreted as abstract functions applications. The interpretation of a constructor belongs to *Basic*, obtained as the least upper bound (lub) of the component's abstract values. But each component $x_i :: t_i$ has an abstract value belonging to D_{2t_i}, that must be first *flattened* to a basic abstract value. This is done by a function called *abstraction function* $\alpha_t : D_{2t} \to Basic$, defined in Figure 2. The idea is to flatten the tuples (by applying the lub operator) and to apply the functions to deterministic arguments.

In a recursive **let** expression the fixpoint can be calculated by using Kleene's ascending chain. We have three different kinds of *case* expressions (for tuple, algebraic types and primitive types). The more complex one is the algebraic *case*. Its abstract value is non-deterministic if either the discriminant or any of the expressions in the alternatives is non-deterministic. Note that the abstract value of the discriminant e, let us call it b, belongs to *Basic*. That is, when it was interpreted, the information about the components was lost. We want now to interpret each alternative's right hand side in an extended environment with abstract values for the variables $v_{ij} :: t_{ij}$ in the left hand side of the alternative. We do not have such information, but we can safely approximate it by using the *concretisation function* $\gamma_t : Basic \to D_{2t}$ defined in Figure 2. Given a type t, it *unflattens* a basic abstract value and produces an abstract value in D_{2t}. The idea is to obtain the best safe approximation both to d and n in a given domain. The abstraction and concretisation functions are mutually recursive. In [9] they were explained in detail and an example was given to illustrate their definitions. They have some interesting properties (e.g. they are a Galois insertion pair [5]), studied in [16, 15]. In abstract interpretation, abstraction and concretisation functions usually relate the standard and abstract semantics. We use here the same names for a different purpose because they are similar in spirit.

The abstract interpretation of a polymorphic expression is the abstract interpretation of its 'smallest instance' [1], i.e. that instance where K (the basic

$$
\begin{array}{l}
[\![v]\!]_2\ \rho_2 = \rho_2(v) \\
[\![k]\!]_2\ \rho_2 = d \\
[\![(x_1,\ldots,x_m)]\!]_2\ \rho_2 = ([\![x_1]\!]_2\ \rho_2,\ldots,[\![x_m]\!]_2\ \rho_2) \\
[\![C\ x_1 \ldots x_m]\!]_2\ \rho_2 = \bigsqcup_i \alpha_{t_i}([\![x_i]\!]_2\ \rho_2) \text{ where } x_i :: t_i \\
[\![e\ x]\!]_2\ \rho_2 = ([\![e]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![op\ x_1 \ldots x_m]\!]_2\ \rho_2 = (\gamma_{t_{op}}(d))\ ([\![x_1]\!]_2\ \rho_2) \ldots ([\![x_m]\!]_2\ \rho_2) \text{ where } op :: t_{op} \\
[\![p\#x]\!]_2\ \rho_2 = ([\![p]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \text{ where } v :: t_v \\
[\![\textbf{process}\ v \to e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \\
[\![merge]\!]_2\ \rho_2 = \lambda z \in Basic.n \\
[\![\textbf{let}\ v = e\ \textbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \mapsto [\![e]\!]_2\ \rho_2] \\
[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ (fix\ (\lambda\rho_2'.\rho_2\ \overline{[v_i \mapsto [\![e_i]\!]_2\ \rho_2']})) \\
[\![\textbf{case}\ e\ \textbf{of}\ (v_1,\ldots,v_m) \to e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v_i \mapsto \pi_i([\![e]\!]_2\ \rho_2)] \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}; [v \to e']]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i}\ [\ \sqcup [\![e']\!]_2\ \rho_2'\ ] \text{ otherwise} \end{cases} \\
\qquad \text{where } \rho_{2i} = \rho_2\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t \\
\qquad\qquad \rho_2' = \rho_2\ [v \mapsto d] \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{k_i \to e_i};\ [v \to e']]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_2\ [\ \sqcup [\![e']\!]_2\ \rho_2'\ ] \text{ otherwise} \end{cases} \\
\qquad \text{where } e_i :: t, \rho_2' = \rho_2\ [v \mapsto d] \\
[\![\Lambda\beta.e]\!]_2\ \rho_2 = [\![e]\!]_2\ \rho_2 \\
[\![e\ t]\!]_2\ \rho_2 = \gamma_{t'tinst}([\![e]\!]_2\ \rho_2)\ where\ e :: \forall\beta.t', tinst = t'[\beta := t]
\end{array}
$$

Fig. 3. Abstract interpretation $[\![\cdot]\!]_2$

type) is substituted for the type variables. This is the reason why the abstract domain corresponding to a type variable β is *Basic*, and the abstract domain corresponding to a polymorphic type is that of the type without qualifier.

When an application to a type t is done, the abstract value of the appropriate instance must be obtained. Such abstract value is in fact obtained as an approximation constructed from the abstract value of the smallest instance. From now on, the instantiated type $t'[\beta := t]$ will be denoted as *tinst*. The approximation to the instance abstract value is obtained by using a *polymorphic concretisation function* $\gamma_{t'tinst} : D_{2t'} \to D_{2tinst}$, which is defined in Figure 4 (where $tinst_i$ represents $t_i[\beta := t]$). This function *adapts* an abstract value in $D_{2t'}$ to one in D_{2tinst}. Another function, $\alpha_{tinst\,t'} : D_{2tinst} \to D_{2t'}$, which we will call the *polymorphic abstraction function*, is also defined in Figure 4. They are a generalisation of α_t and γ_t. These operated with values in *Basic* and D_{2t}. Now we operate with values in the domains $D_{2t'}$ and D_{2tinst}, that is, between the domains corresponding to the polymorphic type and each of its concrete instances. So in case $t' = \beta$ they will coincide with α_t and γ_t. These functions and their properties are described in detail in [16, 15].

3 The Intermediate Analysis

3.1 Introduction

The high cost of $[\![\cdot]\!]_2$ is due to the fixpoint calculation. At each iteration a comparison between abstract values is done. Such comparison is exponential in case

$$
\begin{array}{ll}
t' = K, T\ t_1 \ldots t_m & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = (t_1, \ldots, t_m) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \times^m((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), \ldots, (\gamma_{t_m\,tinst_m}, \alpha_{tinst_m\,t_m})) \\
t' = t_1 \to t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \ \to ((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = Process\ t_1\ t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \ \to ((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = \beta & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_t, \alpha_t) \\
t' = \beta'\ (\neq \beta) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = \forall \beta'.t_1 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1})
\end{array}
$$

$$
\begin{array}{c}
\times((f^e, f^c), (g^e, g^c)) = (f^e \times g^e, f^c \times g^c) \\
\to ((f^e, f^c), (g^e, g^c)) = (\lambda h.g^e \cdot h \cdot f^c, \lambda h'.g^c \cdot h' \cdot f^e)
\end{array}
$$

Fig. 4. Polymorphic abstraction and concretisation functions

functional domains are involved. So, a good way of speeding up the calculation of the fixpoint is finding a quickly comparable representation of functions. Some different techniques have been developed in this direction, such as frontiers algorithms [10] and widening/narrowing operators [4, 6, 12]. Here, we will represent functions by *signatures* in a way similar to [12]. A signature for a function is obtained by *probing* such function with some explicitly chosen combinations of arguments. For example, in the strictness analysis of [12], a function f with m arguments was probed with m combination of arguments, those where $\bot$ occupies each argument position and the rest of arguments are given a $\top$ value: $\bot, \top, \ldots, \top$; $\top, \bot, \top, \ldots, \top$; $\ldots$; $\top, \top, \ldots, \bot$. So, for example, the function $f = \lambda x :: Int.\lambda y :: Int.y$ has a signature $\top\ \bot$.

If we probe only with some arguments, different functions may have the same signature and consequently some information is lost. Then the fixpoint calculation is not exact, but just approximate. A compromise must be found between the amount of information the signature keeps and the cost of signatures comparison. Several probings can be proposed. Here we concentrate on the one we have implemented, and just mention other possibilities. We probe a function of m arguments with $m+1$ combinations of arguments. In the first m combinations, a non-deterministic abstract value (of the corresponding type) $\gamma_{t_i}(n)$ occupies each argument position while a deterministic abstract value $\gamma_{t_i}(d)$ is given to the rest of the arguments: $\gamma_{t_1}(n), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)$; $\gamma_{t_1}(d), \gamma_{t_2}(n), \ldots, \gamma_{t_m}(d)$; $\ldots$; $\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(n)$. In the $m+1$-th combination, all the arguments are given a deterministic value: $\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)$. This additional combination is very important for us, as we want the analysis to be more powerful than $[\![\cdot]\!]_1$, where the functions were probed with only this combination.

3.2 The Domain of Signatures

In Figure 5 the domains S_t of signatures are formally defined. The domain corresponding to a basic or an algebraic type is a two-point domain, very similar to the $Basic$ domain. However we will use uppercase letters D and N when talking about signatures. The domain corresponding to a tuple type is a tuple of signatures of the corresponding types, for example we could have (D, N) for the type (Int, Int). The ordering between tuples is the usual componentwise one.

$S_K = S_{T\ t_1 \ldots t_m} = S_\beta = \{D, N\}$where $D \preceq N$	$\mathcal{H}_K = \mathcal{H}_{T\ t_1 \ldots t_m} = \mathcal{H}_\beta = 1$
$S_{(t_1,\ldots,t_m)} = S_{t_1} \times \ldots \times S_{t_m}$	$\mathcal{H}_{(t_1,\ldots,t_m)} = \sum_{i=1}^{m} \mathcal{H}_{t_i}$
$S_{\forall\beta.t} = S_t$	$\mathcal{H}_{\forall\beta.t} = \mathcal{H}_t$
$S_t = \{s_1\ s_2\ \ldots\ s_m\ s_{m+1} \mid$	$\mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r}$
$\forall\ i \in \{1..(m+1)\}.s_i \in S_{t_r} \wedge s_{m+1} \preceq s_i\}$	where
where $t = t_1 \to t_2, Process\ t_1\ t_2$	$t = t_1 \to t_2, Process\ t_1\ t_2$
$m = nArgs(t), t_r = rType(t)$	$m = nArgs(t), t_r = rType(t)$

Fig. 5. The domain of signatures and its height $\mathcal{H}_t$

The domain corresponding to a polymorphic type is the domain corresponding to the smallest instance. With respect to the functions some intuition must be given. First of all, we will say that a type t is functional if $t = t_1 \to t_2$, $t = Process\ t_1\ t_2$ or $t = \forall\beta.t'$ where t' is functional. If a type t is functional we will write $fun(t)$; otherwise we will write $nonfun(t)$. If a function has m arguments then its signature is composed by $m+1$ signatures, each one corresponding to the (non-functional) type of the result. By m arguments, we mean that taking out all the polymorphism qualifiers, and transforming the process types into functional types, the type is $t_1 \to \ldots \to t_m \to t_r$, where t_r is not functional. We will call this type the *unrolled* version of the functional type. As an example, the unrolled version of $Int \to (\forall\beta.Process\ \beta\ (\beta, Int))$ is $Int \to \beta \to (\beta, Int)$. Three useful functions, *nArgs*, *rType* and *aTypes*, can be easily defined (for lack of space they do not appear here). Given a type t, the first one returns the number of arguments of t; the second one returns the (non-functional) type of its result (it is the identity in the rest of cases); and the third one returns the list (of length $nArgs(t)$) of the types of the arguments. Then the unrolled version of a type t has $nArgs(t)$ arguments of types $aTypes(t)$, and $rType(t)$ as result type. In order to make the signatures for a function type readable, in the examples the last component is separated with a + symbol. So, an example of signature for the type $Int \to (Int, Int)$ could be $(N, D) + (D, D)$. But not every sequence of signatures is a valid signature. As we have previously said, the last component is obtained by probing the function with all the arguments set to a deterministic value, while the rest of them are obtained by probing the function with one non-deterministic value. As the functions are monotone, this means that the last component must always be less than or equal to all the other components. The ordering between the signatures ($\preceq$) is componentwise, so least upper bound and greatest lower bound can also be obtained in the same way. It is easy to see that with this ordering, the domain of signatures S_t for a given type t is a complete lattice of height $\mathcal{H}_t$, see Figure 5.

3.3 The Probing

In this section we define the probing function $\wp_t :: D_{2t} \to S_t$, that given an abstract value in D_{2t}, obtains the corresponding signature in S_t. In Figure 6 the formal definition is shown. The signature of a basic value b is the corresponding basic signature B, that is, if $b = d$ then $B = D$ and if $b = n$ then $B = N$. The signature of a tuple is the tuple of signatures of the components. And finally, the signature for a function or a process $f :: t$ is a sequence of $m + 1$ signatures,

$$
\begin{array}{l}
\wp_t :: D_{2t} \to S_t \\
\wp_K(b) = \wp_{T\ t_1 \ldots t_m}(b) = \wp_\beta(b) = B \\
\wp_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = (\wp_{t_1}(e_1),\ldots,\wp_{t_m}(e_m)) \\
\wp_{\forall\beta.t} = \wp_t \\
\wp_t(f) = \wp_{t_r}(f\ \gamma_{t_1}(n)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(d))\ \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(n)\ldots\gamma_{t_m}(d))\ldots \\
\qquad \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(n))\ \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(d)) \\
\quad \text{where}\ \ t = t'_1 \to t'_2, Process\ t'_1\ t'_2,\ \ t_r = rType(t),\ \ [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 6. The probing function

$$
\begin{array}{l}
\Re_t :: S_t \to D_{2t} \\
\Re_K(B) = \Re_{T\ t_1 \ldots t_m}(B) = \Re_\beta(B) = b \\
\Re_{(t_1,\ldots,t_m)}(s1,\ldots,s_m) = (\Re_{t_1}(s_1),\ldots,\Re_{t_m}(s_m)) \\
\Re_{\forall\beta.t} = \Re_t \\
\Re_t(\overline{s_j}) = \lambda\overline{z_j \in D_{2t_j}}.
\begin{cases}
\Re_{t_r}(s_{m+1}) \text{ if } \bigwedge_{j=1}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \\
\Re_{t_r}(s_i) \text{ if } \bigwedge_{j=1,j\neq i}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \wedge z_i \not\sqsubseteq \gamma_{t_i}(d) \quad \forall i \in \{1..m\} \\
\gamma_{t_r}(n) \text{ otherwise } (m > 1)
\end{cases} \\
\quad \text{where}\ \ \overline{s_j} = s_1 \ldots s_m\ s_{m+1},\ \ t = t'_1 \to t'_2, Process\ t'_1\ t'_2 \\
\qquad\quad m = nArgs(t),\ \ t_r = rType(t),\ \ [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 7. The concretisation function corresponding to the probing

where $m = nArgs(t)$, that are obtained by probing f with the combinations of arguments we have previously mentioned.

We have already said that in the probing process some information is lost. This means that a signature represents several abstract values. When we want to recover the original value, we can only return an approximation. This is what the *signatures concretisation function* $\Re_t :: S_t \to D_{2t}$ does. This function is defined in Figure 7. All the cases but the functional one are simple. Given a signature $s = s_1 \ldots s_m\ s_{m+1}$, where $s \in S_t$, $\Re_t(s)$ is a function of m arguments $z_i \in D_{2t_i}$. We know that the last element s_{m+1} was obtained by probing the original function with $\gamma_{t_i}(d)$, $i \in \{1..m\}$. So, if all the arguments are less than or equal to the corresponding $\gamma_{t_i}(d)$, then the concretisation of s_{m+1} can be safely returned. The original function might have more precise information for some of the arguments combinations below $\gamma_{t_i}(d)$, but now it is lost. We already know that s_i was obtained by probing the original function with $\gamma_{t_i}(n)$ value for the ith argument and $\gamma_{t_j}(d)$ for the rest of them ($j \in \{1..m\}$, $j \neq i$). So, if all the arguments but the ith one are less than or equal to the corresponding $\gamma_{t_j}(d)$, then we can safely return the concretisation of s_i. Again we are losing information. If there is more than one value that is not less than or equal to the corresponding $\gamma_{t_j}(d)$, we can only return the pessimistic value $\gamma_{t_r}(n)$, as we do not have information for these combinations of arguments in the signature.

We have said that we will use a widening operator to speed up the fixpoint calculation. This is defined as $\mathcal{W}_t = \Re_t \cdot \wp_t$. In fact we will prove that $\mathcal{W}_t$ is an upper closure operator ($\mathcal{W}_t \sqsupseteq id_{D_{2t}}$). The definition of a widening operator is more general [4], but given an upper closure operator $\mathcal{W}_t$, we can define a corresponding widening operator $\nabla_t = \lambda(x,y).x \sqcup \mathcal{W}_t(y)$, as done in [6]. So we will use the word widening operator instead, as in [12].

The analysis The analysis is very similar to the second analysis, presented in Section 2. We will use a 3 underscript to identify it. The only expression where there are differences is the recursive **let** expression where a fixpoint must be calculated: $[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_3\ \rho_3 = [\![e']\!]_3\ (\mathit{fix}\ (\lambda\rho'_3.\rho_3\ \overline{[v_i \mapsto \mathcal{W}_{t_i}([\![e_i]\!]_3\ \rho'_3)]}))$, where $e_i :: t_i$. Notice that by modifying the widening operator we can have several different variants of the analysis. We can express them parameterised by the (collection of) widening operator wop_t, $[\![\cdot]\!]_3^{wop}$.

3.4 Some Theoretical Results

In this section we will prove some properties of the functions defined in the previous section and some others that will help in the implementation of the analysis. Proposition 1 tells us that $\wp_t$ and $\Re_t$ are a Galois insertion pair, which means that $\Re_t$ recovers as much information as possible, considering how the signature was built. As a consequence, $\mathcal{W}_t$ is a widening operator. Proposition 2 tells us that $\gamma_t(d)$ and $\gamma_t(n)$ can be represented by their corresponding signatures without losing any information, which will be very useful in the implementation of the analysis. Finally, Proposition 3 tells us that the comparison between an abstract value and $\gamma_t(d)$ can be done by comparing their corresponding signatures, which is much less expensive. This will be very useful in the implementation, as such comparison is done very often. In the worst case it is made in $\mathcal{H}_t$ steps.

Proposition 1 *For each type t,*

(a) *The functions $\wp_t$, $\Re_t$, and $\mathcal{W}_t$ are monotone and continuous.*
(b) $\mathcal{W}_t \sqsupseteq id_{D_{2t}}$.
(c) $\wp_t \cdot \Re_t = id_{S_t}$.

Proposition 2 *For each type t, $\mathcal{W}_t \cdot \gamma_t = \gamma_t$.*

Proposition 3 *For each type t, $\forall z \in D_{2t}.z \sqsubseteq \gamma_t(d) \Leftrightarrow \wp_t(z) \preceq \wp_t(\gamma_t(d))$.*

3.5 Polymorphism

We would like to have a property similar to Proposition 2, $\mathcal{W}_{tinst} \cdot \gamma_{t'tinst} = \gamma_{t'tinst}$ so that we could represent the instantiations' abstract value by its signature, but this is not true. We show a counterexample. Let the polymorphic type $\forall\beta.t'$, where $t' = (\beta, \beta) \to \beta$, be instantiated with the type $t = Int \to Int$. Then $tinst = t'[t/\beta] = (Int \to Int, Int \to Int) \to Int \to Int$. To abbreviate, we will call E_p to $Basic \times Basic$ and F_p to $[Basic \to Basic] \times [Basic \to Basic]$. Let $f \in D_{2t'}$ be $f = \lambda p \in E_p.\pi_1(p)$. By definition we have that $\gamma_{t'tinst}(f) = \lambda p \in F_p.\gamma_{t_2 tinst_2}(f\ (\alpha_{tinst_1 t_1}(p)))$. Also by definition we have that

$$\mathcal{W}_{tinst}(\gamma_{t'tinst}(f)) = \lambda p \in F_p. \begin{cases} \lambda u \in Basic.u \text{ if } p \sqsubseteq (\lambda z \in Basic.z, \lambda z \in Basic.z) \\ \lambda u \in Basic.n \text{ otherwise} \end{cases}$$

Let $q = (\lambda z \in Basic.z, \lambda z \in Basic.n)$. Then

$$\gamma_{t'tinst}(f)\ q = \lambda u \in Basic.u \sqsubset \lambda u \in Basic.n = (\mathcal{W}_{tinst}(\gamma_{t'tinst}(f)))\ q$$

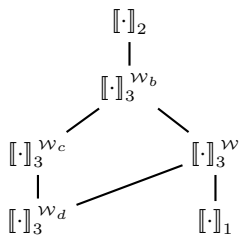

Fig. 8. A hierarchy of analyses

$e =$ **let rec**
$f = \lambda p.\lambda x.$**case** p **of**
$(p_1, p_2) \rightarrow$ **case** p_2 **of**
$0 \rightarrow (p_1, x)$
$z \rightarrow (f\ (p_1 * p_1, p_2 - 1))\ (x * p_2)$
in let
$f_1 = (f\ (q, 3))\ 4$
$f_2 = (f\ (1, 2))\ q$
$x_1 =$ **case** f_1 **of** $(f_{11}, f_{12}) \rightarrow f_{12}$
$x_2 =$ **case** f_2 **of** $(f_{21}, f_{22}) \rightarrow f_{21}$
in (x_1, x_2)

Fig. 9. An example expression e

3.6 A Hierarchy of Analyses

This analysis was intended to be an intermediate one between the two analyses presented in [9]. In this section we study its relation with $[\![\cdot]\!]_2$. It can also be proved that some of its variants are better than $[\![\cdot]\!]_1$, see [16, 15]. First, Proposition 4 tells us that the third analysis is less precise than the second one. This is true for any variant of the third analysis, and in particular for the one we have described. Also, Proposition 5 tells us that given two comparable widening operators, the corresponding variants of the third analysis are also comparable. In [16] other variants ($\mathcal{W}_b$, $\mathcal{W}_c$ and $\mathcal{W}_d$) of the third analysis were presented. The relations between them and with $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$ are shown in Figure 8. The main difference between them lies in their treatment of tuples, in the argument and/or in the result of the functions, either as indivisible entities or as componentwise ones.

Proposition 4 *Let $\mathcal{W}'_t : D_{2t} \rightarrow D_{2t}$ be a widening operator for each type t. Let ρ_2 and ρ_3 be such that for each variable $v :: t_v$ $\rho_2(v) \sqsubseteq \rho_3(v)$. Then for each expression $e :: t_e$, $[\![e]\!]_2\ \rho_2 \sqsubseteq [\![e]\!]_3^{\mathcal{W}'}\ \rho_3$*

Proposition 5 *Let $\mathcal{W}'_t$, $\mathcal{W}''_t$ be two widening operators for each type t. Let ρ_3, ρ'_3 such that for each variable $v :: t_v$, $\rho_3(v) \sqsubseteq \rho'_3(v)$. If for each type t, $\mathcal{W}'_t \sqsubseteq \mathcal{W}''_t$, then for each expression $e :: t_e$, $[\![e]\!]_3^{\mathcal{W}'}\ \rho_3 \sqsubseteq [\![e]\!]_3^{\mathcal{W}''}\ \rho'_3$*

An example In Figure 9 an example expression $e :: (Int, Int)$ shows the difference in power between $[\![\cdot]\!]_1$, $[\![\cdot]\!]_3^{\mathcal{W}}$ and $[\![\cdot]\!]_2$. In order to save some space, the syntax is sugared. Given a pair of integers (p_1, p_2) and another integer x, the function $f :: t$, where $t = (Int, Int) \rightarrow Int \rightarrow (Int, Int)$, calculates the pair $({p_1}^{2*p_2}, x * p_2!)$. q is a free variable in e. Let us assume that in our abstract environment it has an abstract value n, that is, $\rho = [q \mapsto n]$. Then, by applying the analyses definitions (see [9] for the details of $[\![\cdot]\!]_1$) we obtain that $[\![e]\!]_1\ \rho = (n, n) \sqsupset [\![e]\!]_3^{\mathcal{W}}\ \rho = (n, d) \sqsupset [\![e]\!]_2\ \rho = (d, d)$.

$$
\begin{array}{lll}
av \rightarrow & b & b \quad \rightarrow \mid d \\
& \mid (av_1, \ldots, av_m) & \quad\quad \mid n \\
& \mid \lambda v.(e, \rho) & aw \rightarrow \mid b \\
& \mid G\ t'\ t'[t/\beta]\ av & \quad\quad \mid (aw_1, \ldots, aw_m) \\
& \mid A\ t'[t/\beta]\ t'\ av & \quad\quad \mid <t,\ aw_1 \ldots aw_m + aw> \\
& \mid \lfloor \mathrm{F} \rfloor [av_1, \ldots, av_m] & \quad\quad \mid <t,\ + aw> \\
& \mid aw &
\end{array}
$$

Fig. 10. Abstract values definition

$$
\begin{array}{l}
\gamma'_t :: Basic \rightarrow S_t \\
\gamma'_t(b) = \left\{ \begin{array}{l}
B \text{ if } t = K, t = T\ t_1 \ldots t_m, t = \beta \\
(\gamma'_{t_1}(b), \ldots, \gamma'_{t_m}(b)) \text{ if } t = (t_1, \ldots, t_m) \\
\gamma'_{t_1}(b) \text{ if } t = \forall t.t_1 \\
<t,\ \gamma'_{t_r}(n)\ \overset{(m)}{\ldots}\ \gamma'_{t_r}(n)\ +\ \gamma'_{t_r}(b)> \quad \text{if } t = t_1 \rightarrow t_2, Process\ t_1\ t_2 \\
\quad \text{where}\ \ m = nArgs(t),\ \ t_r = rType(t)
\end{array} \right.
\end{array}
$$

Fig. 11. The signatures corresponding to $\gamma_t(n)$ and $\gamma_t(d)$.

4 Implementation of the analysis

4.1 Introduction

In this section we describe the main aspects of the analysis implementation. The algorithm we describe here not only obtains the abstract values of the expressions, but it also annotates each expression (and its subexpressions) with its corresponding signature. A full version of this algorithm has been implemented in Haskell. The implementation of the analysis includes also a little parser and a pretty printing [7]. It is important to annotate the subexpressions, even inside the body of a lambda-abstraction. In [9] we explained that the full laziness transformation [13] may change the semantics of a program when non-deterministic expressions are involved. Given an expression $f = \lambda y.\mathbf{let}\ x = e_1\ \mathbf{in}\ x + y$, where e_1 does not depend on y, the full laziness transformation would produce $f' = \mathbf{let}\ x = e_1\ \mathbf{in}\ \lambda y.x + y$. It was shown that the problem appears when e_1 is non-deterministic. So the annotation of expressions inside functions is necessary.

In the algorithm we make use of the fact that it is implemented in a lazy functional language. The interpretation of a lambda $\lambda v.e$ in an environment ρ is an abstract function. We will use a suspension $\lambda v.(e, \rho)$ to represent the abstract value of $\lambda v.e$. Only when the function is applied to an argument, the body e of the function will be interpreted in the proper environment, emulating in this way the behaviour of the abstract function. So, we use the lazy evaluation of Haskell as our interpretation machinery. Otherwise, we should build a whole interpreter which would be less efficient. But this decision introduces some problems. Sometimes we need to build an abstract function that does not come from the interpretation of a lambda in the program. There are several situations where this happens. One of these is the application of $\gamma_t(b)$ when t is a function or a process type. By Proposition 2 we can use the corresponding signature to represent $\gamma_t(b)$ without losing information. So, in this case we do not need to build a function. Given a basic value b, function $\gamma'_t = \wp_t \cdot \gamma_t$, see Figure 11, returns the signature of $\gamma_t(b)$.

	$\gamma'_{t'\,tinst} : D_{2\,t'} \to D_{2\,tinst}$	$\alpha'_{tinst\,t'} : D_{2\,tinst} \to D_{2\,t'}$
$t' = K, T\ t_1 \ldots t_m$, $\beta'(\neq \beta)$	$\gamma'_{t'\,tinst} = id_{Basic}$	$\alpha'_{tinst\,t'} = id_{Basic}$
$t' = (t_1, \ldots, t_m)$	$\gamma'_{t'\,tinst}(av_1, \ldots, av_m) = (\gamma'_{t_1\,tinst_1}(av_1), \ldots, \gamma'_{t_m\,tinst_m}(av_m))$	$\alpha'_{tinst\,t'}(av_1, \ldots, av_m) = (\alpha'_{tinst_1\,t_1}(av_1), \ldots, \alpha'_{tinst_m\,t_m}(av_m))$
$t' = t_1 \to t_2$, $Process\ t_1\ t_2$	$\gamma'_{t'\,tinst}(av) = G\ t'\ t'[t/\beta]\ av$	$\alpha'_{tinst\,t'}(av) = A\ t'[t/\beta]\ t'\ av$
$t' = \beta$	$\gamma'_{t'\,tinst} = \gamma'_t$	$\alpha'_{tinst\,t'} = \alpha_t$
$t' = \forall\beta'.t_1$	$\gamma'_{t'\,tinst} = \gamma'_{t_1\,tinst_1}$	$\alpha'_{tinst\,t'} = \alpha'_{tinst_1\,t_1}$

Fig. 12. Implementation of functions $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$

This also happens when computing $\alpha_{tinst\,t'}(av)$ and $\gamma_{t'\,tinst}(av)$ where t' is a function or a process type. But in this case, as we saw in Section 3.5, we cannot represent these values by their signatures without losing information. So we build two new suspensions $A\ t'[t/\beta]\ t'\ av$ and $G\ t'\ t'[t/\beta]\ av$ to represent them, see Figure 10. In Figure 12 the implementation of functions $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$, respectively called $\gamma'_{t'\,tinst}$ and $\alpha'_{tinst\,t'}$ are shown. In the functional case they just return the suspension. Only when they are applied to an argument, their definitions are applied, see Figure 14.

We also need to build a function when computing a lub of functions. In this case, we also use a new suspension $\lfloor_{\mathrm{F}}\rfloor[av_1, \ldots, av_m]$, see Figure 10. When the function is applied, the lub will be computed, see Figure 14.

4.2 Abstract values definition

In the implementation of the analysis, signatures are considered also as abstract values, where a signature $s \in S_t$ is just a representation of the abstract value $\Re_t(s)$. In Figure 10 the abstract values are defined. They can be basic abstract values d or n, that represent both a true basic abstract value or a basic signature. Tuples of abstract values are also abstract values. A functional abstract value may have several different representations: It may be represented by a signature or as a *suspension*. In Figure 10 a functional signature is either $<t, aw_1 \ldots aw_m + aw>$ or $<t, + aw>$. The first one is a normal signature. The signature $<t, + aw>$ represents a function returning aw when all the arguments are deterministic (that is, less than or equal to $\gamma_{t_i}(d)$) and $\gamma_{t_r}(n)$ otherwise. So, it is just a particular case of $<t, aw_1 \ldots aw_m + aw>$ where $aw_i = \gamma_{t_r}(n)$ $(i \in \{1..m\})$. A function may also be represented as a suspension. As we have previously said, it can be a suspended lambda abstraction $\lambda v.(e, \rho)$, a suspended lub $\lfloor_{\mathrm{F}}\rfloor[av_1, \ldots, av_m]$ or a polymorphism suspension.

In Figure 13 the lub operator between abstract values is defined. For basic values/signatures and tuples it is simple. In the functional case, if both functions are represented by a signature then we just apply the lub operator componentwise. If one of the functions is a suspension, then the result is a lub suspension.

4.3 Abstract application of a function

In Figure 14 the definition of the application of an abstract function to an abstract argument is shown. In case the abstract function is a signature of the

$$
\begin{array}{l}
n \sqcup b = n \\
d \sqcup b = b \\
(av_1, \ldots, av_m) \sqcup (av'_1, \ldots, av'_m) = (av_1 \sqcup av'_1, \ldots, av_m \sqcup av'_m) \\
< t,\ aw_1 \ldots aw_m + aw > \sqcup < t,\ aw'_1, \ldots, aw'_m + aw' > = \\
\qquad\qquad < t,\ (aw_1 \sqcup aw'_1) \ldots (aw_m \sqcup aw'_m) + (aw \sqcup aw') > \\
< t,\ + aw > \sqcup < t,\ + aw' > = < t,\ + (aw \sqcup aw') > \\
< t,\ aw_1 \ldots aw_m + aw > \sqcup < t,\ + aw' > = < t,\ + aw' > \sqcup < t,\ aw_1 \ldots aw_m + aw > \\
\qquad\qquad = < t,\ + (aw \sqcup aw') > \\
(\lfloor_{\mathrm{F}}\, avs) \sqcup av = av \sqcup (\lfloor_{\mathrm{F}}\, avs) = \lfloor_{\mathrm{F}}\, av : avs \\
av \sqcup \lambda v.(e, \rho) = \lambda v.(e, \rho) \sqcup av = \lfloor_{\mathrm{F}}\, [av, \lambda v.(e, \rho)] \\
(G\ t'\ t'[t/\beta]\ av) \sqcup av' = av' \sqcup (G\ t'\ t'[t/\beta]\ av) = \lfloor_{\mathrm{F}}\, [av', G\ t'\ t'[t/\beta]\ av] \\
(A\ t'[t/\beta]\ t'\ av) \sqcup av' = av' \sqcup (A\ t'[t/\beta]\ t'\ av) = \lfloor_{\mathrm{F}}\, [av', A\ t'[t/\beta]\ t'\ av] \\
\sqcup\, [av_1, \ldots, av_m] = av_1 \sqcup \ldots \sqcup av_m
\end{array}
$$

Fig. 13. Lub operator definition

form $< t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw >$ we check if the argument av' is less than or equal to $\gamma_{t_1}(d)$. This is done by comparing their signatures, $\wp_{t_1}(av')$ and $\gamma'_{t_1}(d)$ (this can be done by Proposition 3). Should this happen, then we discard the first element aw_1 of the signature and return $< t_2 \to \ldots \to t_m \to t_r, aw_2 \ldots aw_m\ +\ aw >$, as these elements have been obtained by giving the first argument a value $\gamma_{t_1}(d)$. Otherwise, we can return aw_1 as result of the function, only if the rest of the arguments are deterministic, so a signature $< t_2 \to \ldots \to t_m \to t_r, + aw_1 >$ is returned. If the abstract function is a signature $< t_1 \to \ldots \to t_m \to t_r,\ + aw >$, only if all the arguments are deterministic (that is, less than or equal to $\gamma_{t_i}(d)$) the value aw is returned. If any of them is not, a non-deterministic result is returned.

The suspensions are just a way of delaying the evaluation until the arguments are known. The application of a suspended function to an argument evaluates the function as far as it is possible, until the result of the function or a new suspension is obtained. If it is a suspension $\lambda v.(e, \rho)$, we continue by evaluating the body e with the interpretation function $[\![\cdot]\!]'$, studied in the following section. The environment ρ keeps the abstract values of all the free variables but v in e. So we just have to add a mapping from v to the abstract value of the argument.

If it is a suspended lub, we apply each function to the argument and then try to calculate the lub of the results. If it is a polymorphism suspension then we continue by applying the (temporally suspended) definition of $\gamma_{t' tinst}$ and $\alpha_{tinst\, t'}$. The algorithm proceeds by suspending and evaluating once and again.

4.4 The algorithm

In the algorithm there are two different interpretation functions $[\![\cdot]\!]'$ and $[\![\cdot]\!]$. Given a non-annotated expression e and an environment ρ, $[\![e]\!]\ \rho$ returns a pair $(av, e'@aw)$ where av is the abstract value of e, e' is e where all its subexpressions have been annotated, and aw is the external annotation of e. While annotations in the expressions are always signatures, the first component of the pair is intented to keep as much information as possible, except in the fixpoint calculation where it will be replaced by its corresponding signature. In Figure 15 the algorithm for $[\![\cdot]\!]$ is shown in pseudocode. The one for $[\![\cdot]\!]'$ is very similar:

$(\lambda v.(e,\rho))\ av = [\![e]\!]'\ \rho[v \mapsto (av, aw, b)]$ where $v :: t_v,\quad aw = \wp_{t_v}(av),\quad b = \alpha_{t_v}(av)$
$(\lfloor_{\mathrm{F}}\rfloor\,[av_1,\ldots,av_m])\ av' = \bigsqcup\ [av_1\ av',\ldots,av_m\ av']$
$(G\ t'\ t'[t/\beta]\ av)\ av' = \gamma'_{t_2 tinst_2}(av\ (\alpha'_{tinst_1\,t_1}(av')))$
$(A\ t'[t/\beta]\ t'\ av)\ av' = \alpha'_{tinst_2\,t_2}(av\ (\gamma'_{t_1\,tinst_1}(av')))$
$< t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw > av'\ (m > 1)$
  $\mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = < t_2 \to \ldots \to t_m \to t_r,\ aw_2 \ldots aw_m + aw >$
  $\mid$ otherwise $= < t_2 \to \ldots \to t_m \to t_r,\ + aw_1 >$
$< t_1 \to t_r,\ aw_1 + aw > av'$
  $\mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw$
  $\mid$ otherwise $= aw_1$
$< t_1 \to \ldots \to t_m \to t_r,\ + aw > av'\ (m > 1)$
  $\mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = < t_2 \to \ldots \to t_m \to t_r,\ + aw >$
  $\mid$ otherwise $= \gamma'_{t_2 \to \ldots \to t_m \to t_r}(n)$
$< t_1 \to t_r,\ + aw > av'$
  $\mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw$
  $\mid$ otherwise $= \gamma'_{t_r}(n)$

Fig. 14. Application of abstract functions

$[\![e]\!]'\ \rho$ returns just the abstract value of the expression. The rest of computations of $[\![e]\!]\ \rho$ are not done. In an environment ρ, there is a triple (av, aw, b) of abstract values associated to each program variable v. The first component av is the abstract value of the expression, aw is the corresponding signature $\wp_t(av)$, and b is the basic abstract value corresponding to $\alpha_t(av)$. As these three values may be used several times along the interpretation, they are calculated just once, when the variable is bound, and used wherever needed.

The computation of the first component of the result av follows the definition of $[\![\cdot]\!]_3^{\mathcal{W}}$, so we just explain the annotation part. In general, to annotate the expression we first recursively annotate its subexpressions and then calculate the annotation for the whole expression by probing the resulting abstract value (the first component) of the expression. But, in many cases the annotations of the subexpressions are used to build the annotation of the whole expression, which is more efficient.

4.5 Cost of the Analysis

Analysing the cost of the interpretation algorithm has proved to be a hard task. This is due to the fact that many of the functions involved —in particular $[\![\cdot]\!]$, $[\![\cdot]\!]'$, abstract application, $\gamma'_{t\,tinst}$, $\wp_t$, $\alpha'_{tinst\,t'}$, and α'_t— are heavily mutually recursive. Fortunately, there are small functions whose cost can be directly computed. For instance, a comparison between two signatures in S_t, or computing their lub, can be done in $O(\mathcal{H}_t)$. So, the lub of m abstract values of type t is in $O((m-1)\ \mathcal{H}_t)$. The cost of $\gamma'_t(b)$ is in $O(m + \mathcal{H}_{t_r})$, being $m = nArgs(t)$ and $t_r = rType(t)$. To analyse the cost of the main interpretation functions we define in [16] two functions $s, s' : Expr \to Int$ respectively giving the 'size' of an expression e when interpreted by $[\![\cdot]\!]$ and by $[\![\cdot]\!]'$. Then $[\![e]\!]'\ \rho \in O(s'(e))$ and $[\![e]\!]\ \rho \in O(s(e))$. Most of the time, $s(e)$ and $s'(e)$ are linear with e using any intuitive notion of size of an expression and including in this notion the size of the types involved. There are three exceptions to this linearity: (1) **Applications**: Interpreting a lambda binding with $[\![\cdot]\!]'$ costs $O(1)$ because a suspension is immediately created. But

the body of this lambda is interpreted as many times as the lambda appears applied in the text, each time with possibly different arguments. Being e_λ the body of a lambda, the algorithm costs $O(s'(e_\lambda))$ each time the lambda is applied. (2) **Probing a function**: It is heavily used by $[\![\cdot]\!]$ to annotate expressions with signatures and also by both $[\![\cdot]\!]$ and $[\![\cdot]\!]'$ in fixpoints. The cost of $\wp_t(e)$ involves $m+1$ abstract applications, each one to m parameters, being $m = nArgs(t)$. Calling e_λ to e's body, the cost will be in $O((m+1)\ s'(e_\lambda))$. (3) **Fixpoints** Assuming a recursive binding $v = e$ of functional type t, being $m = nArgs(t)$, $t_r = rType(t)$, and e_λ the body of e, algorithm $[\![\cdot]\!]'$ will compute a fixpoint in a maximum of $\mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r}$ iterations. At each iteration, the signature of e is obtained, so the cost of fixpoints is in $O(m^2\ \mathcal{H}_{t_r}\ s'(e_\lambda))$. The annotation algorithm $[\![\cdot]\!]$ will add to this cost that of completely annotating e, which involves m probings more, each one with one parameter less, i.e. in total $O(m^2\ s'(e_\lambda))$.

Summarizing, the complete interpretation/annotation algorithm is linear with e except in applications —where the interpretation of the body must be multiplied by the number of applications—, in the annotation of functions —where it is quadratic because of probing—, and in fixpoints where it can reach a cubic cost. We have tried the algorithm with actual definitions of typical Eden skeletons. For files of 3.000 net lines and 80 seconds of compilation time in a SUN 4 250 MHz Ultra Sparc-II, the analysis adds an overhead in the range of 0.5 to 1 second, i.e. less than 1 % overhead.

5 Conclusions

This paper has presented a non-determinism analysis, both from a theoretical and from a practical point of view. Although the main motivation for this work has been the correct compilation of our language Eden, most of the work presented here can be applied to any other non-deterministic polymorphic functional language. A possible application of the analysis could be to annotate a source text written in such a language with deterministic annotations, showing the programmer where equational reasoning would still be possible.

In [9] we presented two other analyses for this problem and related them to other abstract interpretation based analyses, such as strictness analysis. The first one had linear cost but it lost most of the information collected for a function when the function was applied. The second one has been summarized in Section 2. It is very powerful but has exponential cost. The one explained here represents a balance between power and cost: It needs polynomial time, and compared to the second analysis, it only loses information in the fixpoints. We have tried it with many example programs keeping a trace of the number of iterations at every fixpoint, and the results show that the upper bound $\mathcal{H}_t$ is almost never reached. So, for our purposes, this work closes the initial problem: We have achieved a powerful enough analysis with an acceptable cost. We have not found any previous analyses for this problem in the literature.

The implementation itself deserves a closing comment: Implementing the analysis in a lazy functional language such as Haskell has offered some advantages. The first one is that an abstract function can be represented by a suspended interpretation. Related to it, abstract interpretation can make use

of abstract application and the other way around without any danger of non-termination, thanks to lazy evaluation. Also, $[\![\cdot]\!]$ and $[\![\cdot]\!]'$ have in fact been implemented by a single Haskell function. $[\![\cdot]\!]'$ is just a call to $[\![\cdot]\!]$ after which the second component is simply ignored. Lazy evaluation actually does not compute this component in these calls. All this would have not been so easy in an eager and/or imperative language. Other features such as higher-order, polymorphism and overloading has contributed to a compact algorithm: The whole interpretation fits in a dozen of pages including comments.

References

1. G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions*. PhD thesis, University of Glasgow, February 1993.
2. S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany, 1998.
3. G. L. Burn, C. L. Hankin, and S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, October 1986.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles on Programming Languages*, pages 269–282. ACM, 1979.
6. C. Hankin and S. Hunt. Approximate fixed points in abstract interpretation. In B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582 of *LNCS*, pages 219–232. Springer, Berlin, 1992.
7. J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer Verlag, 1995.
8. R. J. M. Hughes and J. O'Donnell. Expressing and Reasoning About Non-Deterministic Functional Programs. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, pages 308–328. Springer-Verlag, 1990.
9. R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *12th International Workshop on Implementation of Functional Languages, IFL00*, volume 2011 of *LNCS*, pages 1–18. Springer-Verlag, 2001.
10. S. L. Peyton Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 11, pages 246–265. Ellis-Horwood, 1987.
11. S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele, DTI/SERC*, pages 249–257, 1993.
12. S. L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Workshop on Functional Programming 1993*, Workshops in Computing, pages 201–220. Springer-Verlag, 1993.
13. S. L. Peyton Jones, W. Partain, and A. L. M. Santos. Let-floating: Moving Bindings to give Faster Programs. *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP'96*, pages 1–12, 24-26 May 1996.

14. S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, September 1998.
15. R. Peña and C. Segura. A Comparison between three Non-determinism Analyses in a Parallel-Functional Language. In *Selected papers in Primeras Jornadas sobre Programación y Lenguajes, PROLE'01*, 2001.
16. R. Peña and C. Segura. Three Non-determinism Analyses in a Parallel-Functional Language. Technical Report 117-01 (SIP). Universidad Complutense de Madrid, Spain, 2001. (http://dalila.sip.ucm.es/miembros/clara/publications.html).
17. H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, May 1990.

$[\![\cdot]\!]' :: Expr \to Env \to AbsVal$
$[\![e]\!]'\ \rho = \pi_1([\![e]\!]\ \rho)$
$[\![\cdot]\!] :: Expr(\ ) \to Env \to (AbsVal, Expr\ AbsVal)$
$[\![v]\!]\ \rho = (av, v@aw)$
where $(av, aw, \_) = \rho(v)$;
$[\![k]\!]\ \rho = (d, k@d)$
$[\![(x_1, \ldots, x_m)]\!]\ \rho = ((av_1, \ldots, av_m), (x'_1, \ldots, x'_m)@(aw_1, \ldots, aw_m))$
where $(av_i, x'_i) = [\![x_i]\!]\ \rho$; $x_i@aw_i = x'_i$
$[\![C\ x_1 \ldots x_m]\!]\ \rho = (aw, C\ x'_1 \ldots x'_m@aw) \qquad \{x_i :: t_i\}$
where $(av_i, x'_i) = [\![x_i]\!]\ \rho$; $aw = \sqcup_{i=1}^m\ b_i$; $b_i = \text{if } isvar(x_i) \text{ then } (\pi_3(\rho(x_i))) \text{ else } d$
$[\![op\ x_1, \ldots, x_m]\!]\ \rho = (aw, op\ x'_1 \ldots x'_m@aw) \qquad \{op :: top\}$
where $(av_i, x'_i) = [\![x_i]\!]\ \rho$; $aw = \gamma'_{top}(d)\ aw_1\ \ldots\ aw_m$; $aw_i = \text{if } isvar(x_i) \text{ then } (\pi_2(\rho(x_i))) \text{ else } d$
$[\![\lambda v.e]\!]\ \rho = (a, (\lambda v@awv.e')@aw) \qquad \{v :: t_v, (\lambda v.e) :: t\}$
where $a = \lambda v.(e, \rho)$; $aw = \wp_t(a)$; $awv = \gamma'_{t_v}(n)$; $(\_, e') = [\![e]\!]\ \rho[v \mapsto (awv, awv, n)]$
$[\![\textbf{process}\ v \to e]\!]\ \rho = (a, (\textbf{process}\ v@awv \to e')@aw) \qquad \{v :: t_v, (\textbf{process}\ v \to e) :: t\}$
where $a = \lambda v.(e, \rho)$; $aw = \wp_t(a)$; $awv = \gamma'_{t_v}(n)$; $(\_, e') = [\![e]\!]\ \rho[v \mapsto (awv, awv, n)]$
$[\![e\ x]\!]\ \rho = (a, (e'\ x')@aw) \qquad \{(e\ x) :: t\}$
where $(ae, e') = [\![e]\!]\ \rho$; $(ax, x') = [\![x]\!]\ \rho$; $a = ae\ ax$; $aw = \wp_t(a)$
$[\![v\#x]\!]\ \rho = (a, (v'\#x')@aw) \qquad \{(v\#x) :: t\}$
where $(av, v') = [\![v]\!]\ \rho$; $(ax, x') = [\![x]\!]\ \rho$; $a = av\ ax$; $aw = \wp_t(a)$
$[\![merge]\!]\ \rho = (aw, merge@aw) \qquad \{merge :: t_{merge}\}$
where $aw = \gamma'_{t_{merge}}(n)$
$[\![\textbf{let}\ bind\ \textbf{in}\ e]\!]\ \rho = (a, (\textbf{let}\ bind'\ \textbf{in}\ e')@aw) \qquad \{e :: t\}$
where $(\rho', bind') = [\![bind]\!]_B\ \rho$; $(a, e') = [\![e]\!]\ \rho' \quad e''@aw = e'$
$[\![\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e']\!]\ \rho = (a, (\textbf{case}\ (e_1@awe)\ \textbf{of}\ (v'_1, \ldots, v'_m) \to (e'_1@aw))@aw) \qquad \{v_i :: t_i\}$
where $(ae, e1@awe) = [\![e]\!]\ \rho$; $aw_i = \pi_i(awe)$; $v'_i = v_i@aw_i$; $av_i = \pi_i(ae)$;
$b_i = \alpha_{t_i}(av_i)$; $\overline{(a, e'_1@aw) = [\![e']\!]\ \rho[v_i \mapsto (av_i, aw_i, b_i)]}$;
$[\![\textbf{case}\ e\ \textbf{of}\ \overline{alt_i}]\!]\ \rho = (av, (\textbf{case}\ e'\ \textbf{of}\ \overline{alt'_i})@aw) \qquad \{\textbf{case}\ e\ \textbf{of}\ \overline{alt_i} :: t\}$
where $(ae, e') = [\![e]\!]\ \rho$; $\overline{(av_i, alt'_i) = [\![alt_i]\!]_A\ ae\ \rho}$; $C_i\ v_{i1} \ldots v_{im_i} \to (e_i@wa_i) = alt'_i$;
$aw = \text{if } ae = n \text{ then } \gamma'_t(n) \text{ else } \sqcup_{i=1}^m\ aw_i$; $av = \text{if } ae = n \text{ then } \gamma'_t(n) \text{ else } \sqcup_{i=1}^m av_i$
$[\![\Lambda\beta.e]\!]\ \rho = (av, (\Lambda\beta.e')@awv)$
where $(av, e') = [\![e]\!]\ \rho$; $e''@awv = e'$
$[\![(e\ t)]\!]\ \rho = (av, e'\ t@aw) \qquad \{e :: (\forall\beta.t'), tinst = t'[t/\beta]\}$
where $(av', e') = [\![e]\!]\ \rho$; $av = \gamma'_{t'tinst}(av')$; $aw = \wp_{tinst}(av)$
$[\![v = e]\!]_B\ \rho = (\rho[v \mapsto (av, aw, b)], v@aw = e'@aw)$
where $(av, e'@aw) = \overline{[\![e]\!]\ \rho}$; $b = \alpha_{t_v}(av)$
$[\![\textbf{rec}\ \overline{v_i = e_i}]\!]_B\ \rho = (\rho_{fix}, \textbf{rec}\ \overline{v'_i = e''_i}) \qquad \{\overline{v_i :: t_i}\}$
where $\rho_{fix} = fix\ f\ init$; $init = \overline{\rho[v_i \mapsto (aw_i, aw_i, d)]}$; $aw_i = \gamma'_{t_i}(d)$
$f\rho' = \overline{\rho'[v_i \mapsto (aw'_i, aw'_i, b_i)]}$ where $av'_i = [\![e_i]\!]'\ \rho'$; $aw'_i = \wp_{t_i}(av'_i)$; $b_i = \alpha_{t_i}(av'_i)$
$(\_, e''_i) = [\![e_i]\!]\ \rho_{fix}$; $(\_, aw_i, \_) = \rho_{fix}(v_i)$; $v'_i = v_i@aw_i$
$[\![C\ v_1 \ldots v_m \to e]\!]_A\ avd\ \rho = (av, C\ v'_1 \ldots v'_m \to e') \qquad \{\overline{v_i :: t_i}\}$
where $aw_i = \gamma'_{t_i}(avd)$; $\overline{(av, e') = [\![e]\!]\ \rho[v_i \mapsto (aw_i, aw_i, avd)]}$; $v'_i = v_i@aw_i$
$[\![v \to e]\!]_A\ avd\ \rho = (av, v' \to e')$
where $(av, e') = [\![e]\!]\ \rho[v \mapsto (avd, avd, avd)]$; $v' = v@avd$

Fig. 15. The expressions annotation algorithm