Three Non-determinism Analyses in a Parallel-Functional Language

Ricardo Peña and Clara Segura

21 June 2001

Technical Report n° 117-01
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Abstract. This paper is an extension of a previous work where two non-determinism analyses were presented. The first of them was efficient (linear) but not very powerful and the second one was more powerful but very expensive (exponential). 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. Additionally: (1) the second and third analyses are completed with polymorphism, (2) we prove that the domains in the second and third analyses form a category in which the morphisms are embedding-closure pairs of functions; respectively called *abstraction* and *concretisation* functions; and (3) we formally relate the analyses and prove that the first analysis is a safe approximation to the third one and that the third one is a safe approximation to the second one. In this way the three analyses become totally ordered by increasing cost and precision.

1 Introduction

The parallel-functional language Eden [BLOP98] extends the lazy functional language Haskell by constructs to explicitly define and communicate processes. It is implemented by modifying the *Glasgow Haskell Compiler* (GHC) [PHH+93]. 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 [HO90,SS90,SS92] of programs and invalidates some optimizations done in the GHC [San95] [PS98]. Such problems were precisely described in [PS01,PPRS00]. In [PS01] 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 first 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 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 that not only implements the analysis but also annotates the program expressions with non-determinism information, so that it can be used to avoid the harmful transformations. The algorithm has been implemented in Haskell and it is detailed in Section 8.

In [PS01], the first and second analyses were presented and compared with respect to expressiveness and efficiency. In that paper some open questions were left: (1) to complete the second analysis with polymorphism; (2) to prove that the domains in the second analysis form a category in which the morphisms are embedding-closure pairs of functions, respectively called *abstraction* and *concretisation*

$$
\begin{array}{lll}
prog \rightarrow bind_1; \ldots; bind_m & & \\
bind \rightarrow \quad v = expr & & \{\text{non-recursive binding}\} \\
\quad\quad | \ \mathbf{rec}\ v_1 = expr_1; \ldots; v_m = expr_m & & \{\text{recursive binding}\} \\
expr \rightarrow \quad expr\ x & & \{\text{application to an atom}\} \\
\quad\quad | \ \lambda v.expr & & \{\text{lambda abstraction}\} \\
\quad\quad | \ \mathbf{case}\ expr\ \mathbf{of}\ alts & & \{case \text{ expression}\} \\
\quad\quad | \ \mathbf{let}\ bind\ \mathbf{in}\ expr & & \{let \text{ expression}\} \\
\quad\quad | \ C\ x_1 \ldots x_m & & \{\text{saturated constructor application}\} \\
\quad\quad | \ op\ x_1 \ldots x_m & & \{\text{saturated primitive operator application}\} \\
\quad\quad | \ x & & \{\text{atom}\} \\
\quad\quad | \ \Lambda\alpha.expr & & \{\text{type abstraction}\} \\
\quad\quad | \ expr\ type & & \{\text{type application}\} \\
\quad\quad | \ v \# x & & \{\text{process instantiation}\} \\
\quad\quad | \ \mathbf{process}\ v \rightarrow expr & & \{\text{process abstraction}\} \\
alts \rightarrow \quad Calt_1; \ldots; Calt_m; [Deft] & m \geq 0 & \\
\quad\quad | \ Lalt_1; \ldots; Lalt_m; [Deft] & m \geq 0 & \\
Calt \rightarrow C\ v_1 \ldots v_m \rightarrow expr & m \geq 0 & \{\text{algebraic alternative}\} \\
Lalt \rightarrow k \rightarrow expr & & \{\text{primitive alternative}\} \\
Deft \rightarrow v \rightarrow expr & & \{\text{default alternative}\} \\
type \rightarrow \quad K & & \{\text{basic types: Integers, characters}\} \\
\quad\quad | \ \alpha & & \{\text{type variables}\} \\
\quad\quad | \ T\ type_1 \ldots type_m & & \{\text{type constructor application}\} \\
\quad\quad | \ type_1 \rightarrow type_2 & & \{\text{function type}\} \\
\quad\quad | \ Process\ type_1\ type_2 & & \{\text{process type}\} \\
\quad\quad | \ \forall\alpha.type & & \{\text{polymorphic type}\}
\end{array}
$$

Fig. 1. Language definition and type expressions

functions; and (3) to prove that the first analysis is a safe approximation to the second one, i.e. whenever it characterizes an expression as being deterministic, so will do the second one.

In this paper, such questions are extensively treated: (1) and (2) are developed both for the second and the third analyses, and (3) we formally relate the three analyses by proving that the first analysis is a safe approximation to the third one, and also that the third one is a safe approximation to the second one. In this way the three analyses become totally ordered by increasing cost and precision.

This paper is a continuation of [PS01] and it subsumes the report [PS00], where the first and the second analyses were presented. This paper is intended to be self-contained, so it includes some results already presented there.

The plan of the paper is as follows: In Section 2 the language and the analyses $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$ are summarised. In Section 3 polymorphism is included in the second analysis. In Section 4 the new third analysis $[\![\cdot]\!]_3^{\mathcal{W}}$ is described. In Section 5 the properties of the abstraction-concretisation function pairs are proved. In Section 6 the properties of polymorphism in the second and third analyses are treated in detail. Section 7 defines functions that relate the three analyses and shows the correctness relations between them. In Section 8 we describe an annotation algorithm that implements the third analysis. Some previous theoretical results help in this implementation. In Section 9 the output produced by this algorithm for some example programs is shown. In Section 10 the cost of the algorithm is studied in detail. Finally, in Section 11 some conclusions are drawn.

2 Two Analyses for Non determinism in Eden

2.1 The Language

The language being analysed is an extension of Core-Haskell [PHH+93], i.e. a simple functional language with second-order polymorphism, so it includes type abstraction and type application.

In Figure 1 the syntax of the language and of the type expressions are shown. There, v denotes a variable, k denotes a literal, x denotes an atom (a variable or a literal), and T denotes a type constructor. 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, primitive operators applications, and also *case* and *let* expressions. Constructor and primitive operators applications are saturated. 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 tuple typle $(t_1, \ldots, t_m)$, an algebraic 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/output channel of the process.

2.2 The First Analysis

Figure 2 shows the abstract domains for the first analysis. 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 (except tuples). Tuples are specially treated as tuples of basic abstract values[1] (see [PS01] for justification). The ordering between basic values is naturally extended to tuples. Several least upper bound (lub) operators as well as an operator to flatten the internal tuples can be defined, shown in Figure 2. The domains corresponding to functions and processes *are identified with their range domains* (this is the most important difference with the second analysis). The abstract domain of a polymorphic type is that of its smallest instance [Bar93], i.e. that one in which K is substituted for the type variable. So the domain corresponding to a type variable is $Basic$.

In Figure 3 the abstract interpretation is shown. The interpretation of a tuple is a tuple of basic abstract values, so the abstract value of each component is previously flattened (by using $\widehat{\square}$). In the interpretation of a constructed value, we take a step further: Once the components are flattened, the lub is applied so that a final basic value is obtained. This means that the information about the components is lost. As `merge` process is the only source of non-determinism, we will say that an expression may be non-deterministic when it 'contains' any instantiation of this process. So we will consider that a function/process is deterministic if it does not generate non-deterministic results from deterministic arguments. Then, the interpretation of a function/process is the interpretation of its body when the argument is given a deterministic abstract value.

Such value is in fact an adaptation of the basic abstract value d to the type of the argument, see Figure 2. Given a type t, $-_t$ takes a basic abstract value b and produces an abstract value in D_{1t}. Its behaviour is the opposite to the flattening operator: If t is a m-tuple, it replicates b to obtain the m-tuple $(b, \ldots, b)$. Note that $\widehat{\square} b_t = b$ and $(\widehat{\square} a)_t \sqsupseteq a$.

So coming back to the function interpretation, if the type of the argument is a n-tuple, the argument should be an n-tuple $(d, \ldots, d)$. If the argument is non-deterministic we are always assuming that the result is non-deterministic. This means that we are not expressing how the output depends on the input, what leads to some limitations of the analysis. The lack of such information is reflected in the interpretation of the function application: The result of the application is non-deterministic either when the function is non-deterministic or when the argument is non-deterministic. As an example, let $h = \lambda v_1.\lambda v_2.\mathbf{let}\ v_3 = merge \# v_2\ \mathbf{in}\ (3, v_3)$. The abstract value of this function is (d, n). For this example, the analysis is able to tell us that the first component is deterministic, while the second one may be non-deterministic. But, when this function is applied to any non-deterministic argument, we lose this information: $[\![(h\ x)\ y]\!]_1\ \ [x \mapsto n, y \mapsto d] = (n, n)$. This is because we are not able to express

[1] We use b to denote a basic abstract value and a to denote either a basic value or a tuple of basic values.

$Basic = \{d, n\}$ where $d \sqsubseteq n$	$-_t : Basic \to D_{1\,t}$	$n \sqcup b = n \quad \widehat{\sqcup} b = b$
$D_{1\,K} = D_{1\,T\ t_1 \ldots t_m} = D_{1\,\alpha} = Basic$	$b_{(t_1,\ldots,t_m)} = (\overset{1}{b}, \ldots, \overset{m}{b})$	$d \sqcup b = b \quad \widehat{\sqcup}(b_1, \ldots, b_m) = \bigsqcup_i b_i$
$D_{1\,(t_1,\ldots,t_m)} = \{(b_1, \ldots, b_m) \mid b_i \in Basic\}$	$b_{t_1 \to t_2} = b_{Process\ t_1\ t_2} = b_{t_2}$	$b \sqcup (b_1, \ldots, b_m) = (b_1 \sqcup b, \ldots, b_m \sqcup b)$
$D_{1\,t_1 \to t_2} = D_{1\,Process\ t_1\ t_2} = D_{1\,t_2}$	$b_{\forall \alpha.t} = b_t$	
$D_{1\,\forall \alpha.t} = D_{1\,t}$	$b_t = b$ if $t = K, \alpha, T\ t_1 \ldots t_m$	

Fig. 2. Abstract domains for the first analysis, adaptation function definition, lub and flattening operators

$$
\begin{array}{l}
[\![v]\!]_1\ \rho_1 = \rho_1\ v \\
[\![k]\!]_1\ \rho_1 = d \\
[\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 = (\widehat{\sqcup}([\![x_1]\!]_1\ \rho_1), \ldots, \widehat{\sqcup}([\![x_m]\!]_1\ \rho_1)) \\
[\![C\ x_1 \ldots x_m]\!]_1\ \rho_1 = \bigsqcup_i \widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \\
[\![e\ x]\!]_1\ \rho_1 = (\widehat{\sqcup}([\![x]\!]_1\ \rho_1)) \sqcup [\![e]\!]_1\ \rho_1 \\
[\![op\ x_1 \ldots x_m]\!]_1\ \rho_1 = (\bigsqcup_i \widehat{\sqcup}([\![x_i]\!]_1\ \rho_1))_t \text{ where } op :: t_1 \to (t_2 \to \ldots (t_m \to t)) \\
[\![p\#x]\!]_1\ \rho_1 = \widehat{\sqcup}([\![x]\!]_1\ \rho_1) \sqcup [\![p]\!]_1\ \rho_1 \\
[\![\lambda v.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \mapsto d_t] \text{ where } v :: t \\
[\![\textbf{process}\ v \to e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \mapsto d_t] \text{ where } v :: t \\
[\![merge]\!]_1\ \rho_1 = n \\
[\![\textbf{let}\ v = e\ \textbf{in}\ e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1] \\
[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]_1\ \rho_1 = [\![e']\!]_1\ (fix\ (\lambda \rho_1'.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho_1']})) \\
[\![\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto \pi_i([\![e]\!]_1\ \rho_1)_{t_i}]} \text{ where } v_i :: t_i \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]_1\ \rho_1 = b \sqcup (\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i}) \\
\qquad\qquad \text{where } b = [\![e]\!]_1\ \rho_1 \text{ and } \rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto b_{t_{ij}}]}, v_{ij} :: t_{ij} \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{k_i \to e_i}]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1 \sqcup (\bigsqcup_i [\![e_i]\!]_1\ \rho_1) \\
[\![\Lambda \alpha.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1 \\
[\![e\ t]\!]_1\ \rho_1 = ([\![e]\!]_1\ \rho_1)_{tinst} \text{ where } (e\ t) :: tinst
\end{array}
$$

Fig. 3. Abstract interpretation of the first analysis

that the determinacy of the first component is independent of the arguments. This problem is solved in the second analysis.

In the recursive *let* expression, the fixpoint can be obtained by using the Kleene's ascending chain. The number of iterations is linear with the number of tuple components in the bindings.

We have three different kinds of *case* expressions (for tuple, algebraic and primitive types). The more complex one is the algebraic *case*. It is non-deterministic if either the discriminant expression 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 adaptation function, see Figure 2.

When a type application $e\ t$ is interpreted, it is necessary to adapt the abstract value of the type abstraction to the instantiated type, as new structure may arise from the instantiation. As an example, if $e :: t_e$, where $t_e = \forall \beta.\beta \to \beta$ and $t = (Int, Int)$, then $[\![e]\!]_1\ \rho \in Basic$, while $[\![e\ t]\!]_1\ \rho \in (Basic \times Basic)$. If no new structure arises (i.e. the domain corresponding to the smallest instance is already a cartesian product), then we do not need to make any adaptation. So, by overloading the adaptation function notation, we could write $(b_1, \ldots, b_m)_{(t_1,\ldots,t_m)} = (b_1, \ldots, b_m)$. As an example, if $t_e = \forall \beta.\beta \to (\beta, \beta)$ and $t = (Int, Int)$, we have that $[\![e]\!]_1\ \rho \in (Basic \times Basic)$ and also $[\![e\ t]\!]_1\ \rho \in (Basic \times Basic)$.

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

Fig. 4. Abstract domains for the second analysis

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

Fig. 5. Abstraction and concretisation function definitions

2.3 The Second Analysis

To overcome the limitations of the first analysis (see [PS01] for a thorough discussion and an example) we define a new analysis where interpreting functions and processes as abstract functions allows us to express several levels of determinism and non-determinism, for instance $\lambda z.d \sqsubseteq \lambda z.z \sqsubseteq \lambda z.n$. It is also an abstract interpretation based analysis, in the style of Burn, Hankin and Abramsky [BHA86].

In Figure 4 the abstract domains for the second analysis are shown. Now the abstract domain of a tuple type is the cartesian product of the domains corresponding to the types of the components. The domains corresponding to functions and processes are the domains of continuous functions between the domains of the argument and the result.

In Figure 6 the abstract interpretation is shown. The interpretation of a tuple is now a tuple of the abstract values of the components. The interpretation of a constructor belongs to *Basic* but, as each component $x_i :: t_i$ has an abstract value belonging to D_{2t_i}, we cannot directly apply the lub. First the information of each component must be flattened. The function responsible for this is called the *abstraction function* $\alpha_t : D_{2t} \to Basic$, defined in Figure 5.

Given a type t, the abstraction function takes an abstract value in D_{2t} and *flattens* it to a value in *Basic*. The idea is to flatten the tuples (by applying the lub operator) and to apply the functions to deterministic arguments.

Application and process instantiation are interpreted as functions applications. In a recursive *let* expression the fixpoint can be calculated using again Kleene's ascending chain. As there are functional domains, the number of iterations is, in the worst case, exponential in the number of bindings.

In the algebraic *case* expression we again find a safe representative of the discriminant's abstract value in the domain of the variables v_{ij}, now $D_{2t_{ij}}$. This is obtained with a function $\gamma_t : Basic \to D_{2t}$, called the *concretisation function*, defined in Figure 5.

Given a type t, the concretisation function γ_t *unflattens* a basic abstract value and produces an abstract value in D_{2t}. The idea of this function is to obtain the best safe approximation both to d and n in a given domain. In particular, n is mapped to the top of the domain D_{2t}, and d to a value in D_{2t} that reflects our original idea of determinism: All the values below it will have different levels of determinism.

The function type needs explanation, the rest of them being inmediate. We consider that a function is deterministic if it produces deterministic results from deterministic arguments; but if the argument is non-deterministic, the only safe result we can produce is a non-deterministic one. So, the unflattening of d for a function type is a function that takes an argument, flattens it to see whether it is deterministic or not and again applies the concretisation function corresponding to the type of the result. The unflattening of n for a function type is the function that returns a non-deterministic result independently of the argument.

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

Fig. 6. Abstract interpretation for the second analysis

$$
\begin{array}{l}
D_{2\beta} = Basic \qquad D_{2\forall\beta.t} = D_{2t} \\
\\
[\![\Lambda\beta.e]\!]_2\ \rho_2 = [\![e]\!]_2\ \rho_2 \\
[\![e\ t]\!]_2\ \rho_2 = \gamma_{t'tinst}([\![e]\!]_2\ \rho_2)\ where\ e :: \forall\beta.t', \\
\qquad\qquad\qquad\qquad\qquad\qquad tinst = t'[\beta := t]
\end{array}
$$

Fig. 7. Abstract domains and abstract interpretation for polymorphism

In Section 5 we will show some important properties of the abstraction and concretisation functions. Particularly, we will see that they are a Galois insertion and that $\gamma_t(d)$ is the biggest value in D_{2t} that has the property of preserving the determinism. As an example, for $t = Int \to Int$, $\gamma_t(d) = \lambda z.z$, while $\lambda z.d$ is below it and it also has the property.

To conclude, if the discriminant has as abstract value n, then the whole *case* expression is non-deterministic and consequently it has as abstract value $\gamma_t(n)$. If it has d as abstract value, the right hand sides of the alternatives are interpreted in extended environments where each v_{ij} has $\gamma_{t_{ij}}(d)$ as abstract value. Then, the lub of all the alternatives is computed.

3 Polymorphism

Polymorphism is now incorporated to the second analysis. Figure 7 shows the abstract domains corresponding to type variables and polymorphic types, and also the abstract interpretation of a type abstraction and a type application.

The abstract interpretation of a polymorphic expression is the abstract interpretation of its 'smallest instance', i.e. that instance where K (the basic 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 funcion* $\gamma_{t'tinst} : D_{2t'} \to D_{2tinst}$. This function *adapts* an abstract value in D_{2t} to one in D_{2tinst}. Another func-

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

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

Fig. 8. Polymorphic abstraction and concretisation functions definitions

$p \in F_p$	$\alpha_{tinst_1\,t_1}(p)$	$f\ (\alpha_{tinst_1\,t_1}(p))$	$\gamma_{t_2\,tinst_2}(f\ (\alpha_{tinst_1\,t_1}(p)))$
$(\lambda z \in Basic.d, \lambda z \in Basic.d)$	(d, d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.d, \lambda z \in Basic.z)$	(d, d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.z, \lambda z \in Basic.d)$	(d, d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.z, \lambda z \in Basic.z)$	(d, d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.n, \lambda z \in Basic.z)$	(n, d)	n	$\lambda u \in Basic.n$
$(\lambda z \in Basic.z, \lambda z \in Basic.n)$	(d, n)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.n, \lambda z \in Basic.n)$	(n, n)	n	$\lambda u \in Basic.n$

Fig. 9. An example of polymorphism

tion, $\alpha_{tinst\,t'} : D_{2\,tinst} \to D_{2\,t'}$, which we will call the *polymorphic abstraction function*, is also defined. They are mutually recursive and will be shown to be a Galois insertion.

Given two types t, t' and a type variable β functions $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ are formally defined in Figure 8 (where $tinst_i$ represents $t_i[\beta := t]$). In [Bar93] the category of domains and embedding-closure pairs is presented. Two functors, $\times$ (and $\times^m$) and $\to$, can be defined in this category (see Figure 8 bottom). They build a new embedding-closure pair from two (or more in the case of the cartesian product) embedding-closure pairs. We define $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ by means of these functors.

These functions are a generalisation of α_t and γ_t. The functions α_t and γ_t operated with values in *Basic* and $D_{2\,t}$. Now we operate with values in the domains $D_{2\,t'}$ 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. This is why we use the same names. From now on, we will call abstraction function indistinctly to α_t and $\alpha_{tinst\,t'}$, and concretisation function to γ_t and $\gamma_{t'\,tinst}$. If we need to specifically refer to $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$, we will use the 'polymorphic' qualification.

Properties of $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ are shown in Section 6.

As an example, let the polymorphic type $\forall\beta.t'$, where $t' = (\beta, \beta) \to \beta$, be applied to 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 $f \in D_{2\,t'}$ be $f = \lambda p \in E_p.\pi_1(p)$. By definition we have that $\gamma_{t'\,tinst}(f) = \lambda p \in F_p.\gamma_{t_2\,tinst_2}(f\ (\alpha_{tinst_1\,t_1}(p)))$. Figure 9 shows a table for $\gamma_{t'\,tinst}(f)$ and the intermediate steps that have been followed to compute it.

4 The Intermediate Analysis

4.1 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 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

$nArgs :: Type \to \mathbb{N}$	$rType :: Type \to Type$	$aTypes :: Type \to [Type]$
$nArgs(t) = 0$ if $nonfun(t)$	$rType(t) = t$ if $nonfun(t)$	$aTypes(t) = [\,]$ if $nonfun(t)$
$nArgs(t_1 \to t_2) = 1 + nArgs(t_2)$	$rType(t_1 \to t_2) = rType(t_2)$	$aTypes(t_1 \to t_2) = t_1 : aTypes(t_2)$
$nArgs(Process\ t_1\ t_2) = 1 + nArgs(t_2)$	$rType(Process\ t_1\ t_2) = rType(t_2)$	$aTypes(Process\ t_1\ t_2) = t_1 : aTypes(t_2)$
$nArgs(\forall\beta.t) = nArgs(t)$ if $fun(t)$	$rType(\forall\beta.t) = rType(t)$ if $fun(t)$	$aTypes(\forall\beta.t) = aTypes(t)$ if $fun(t)$

Fig. 10. Functions *nArgs*, *rType* and *aTypes*

[CP85,JC87,MH87] and widening/narrowing operators [CC77,CC79,HH92,PP93]. Here, we will represent functions by *signatures* in a way similar to [PP93]. 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 [PP93], a function f with m arguments was probed with m combination of arguments, those where $-$ occupies each argument position and the rest of arguments are given a $\top$ value: $-, \top, \ldots, \top;\ \top, -, \top, \ldots, \top;\ \ldots;\ \top, \top, \ldots, -$. So, for example, the function $f = \lambda x :: Int.\lambda y :: Int.y$ has a signature $\top\ -$.

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.

4.2 The Domain of Signatures

In Figure 11 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. 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 $Int \to \beta \to (\beta, Int)$.

In Figure 10, three useful functions, *nArgs*, *rType* and *aTypes*, are defined. 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. With these definitions, we can see that 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

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

Fig. 11. The domain of signatures and its height $\mathcal{H}_t$

$(N, D) + (D, D)$. 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 11. In Figure 12, the domain S_t, where $t = Int \rightarrow Int \rightarrow (Int, Int)$ is shown.

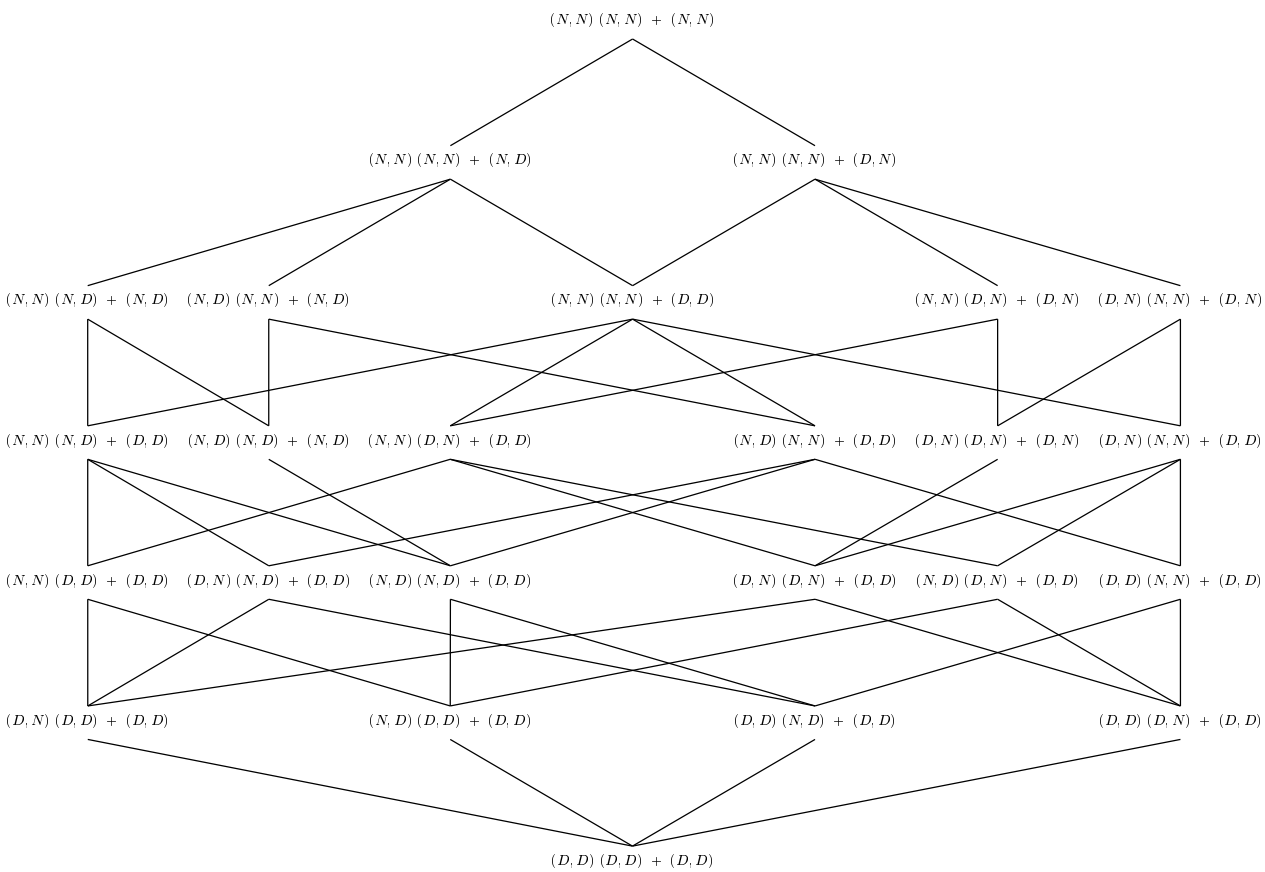

Fig. 12. Signatures for $Int \rightarrow Int \rightarrow (Int, Int)$

4.3 The Probing

In this section we define the probing function $\wp_t :: D_{2t} \rightarrow S_t$, that given an abstract value in D_{2t}, obtains the corresponding signature in S_t. In Figure 13 the formal definition is shown. The signature

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

Fig. 13. The probing function

$$
\begin{array}{l}
\Re_t :: S_t \to D_{2t} \\
\Re_K(B) = \Re_{T\ t_1 \ldots t_m}(B) = \Re_\beta(B) = b \\
\Re_{(t_1,\ldots,t_m)}(s1,\ldots,s_m) = (\Re_{t_1}(s_1),\ldots,\Re_{t_m}(s_m)) \\
\Re_{\forall\beta.t} = \Re_t \\
\Re_t(s_1\ \ldots\ s_m\ s_{m+1}) = \lambda z_1 \in D_{2t_1}.\ldots.\lambda z_m \in D_{2t_m}.\begin{cases} \Re_{t_r}(s_{m+1}) \text{ if } \bigwedge_{j=1}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \\ \Re_{t_r}(s_i) \text{ if } \bigwedge_{j=1, j\neq i}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \wedge z_i \not\sqsubseteq \gamma_{t_i}(d)\ \ i \in \{1..m\} \\ \gamma_{t_r}(n) \text{ otherwise } (m > 1) \end{cases} \\
\quad \text{where } t = t'_1 \to t'_2, Process\ t'_1\ t'_2,\ \ m = nArgs(t),\ \ t_r = rType(t),\ \ [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 14. The concretisation function corresponding to the probing

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, 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 14. 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 operator[2] is defined as $\mathcal{W}_t = \Re_t \cdot \wp_t$. By applying the definitions we obtain the definition in Figure 15.

In the following proposition we prove that $\wp_t$ and $\Re_t$ are a Galois insertion pair [CC79,NNH99], 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 1 *For each type t,*

[2] In Proposition 1 we will prove that $\mathcal{W}_t$ is an upper closure operator. The definition of a widening operator is more general [CC77,NNH99], but given an upper closure operator $\mathcal{W}_t \sqsupseteq id_{D_{2t}}$, we can define a corresponding widening operator $\nabla_t = \lambda(x,y).x \sqcup \mathcal{W}_t(y)$, as done in [HH92]. So we will use the word widening operator instead, as in [PP93].

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

Proof. (Proposition 1) We prove first 1(a). We will prove that $\wp_t$ and $\Re_t$ are monotone by structural induction on t. Then, as D_{2t} is finite, they are continuous. Finally, as $\mathcal{W}_t = \Re_t \cdot \wp_t$, $\mathcal{W}_t$ is monotone and continuous as well.

- $t = K, T\ t_1 \ldots t_m, \beta$. This is a trivial case as $\wp_t(b) = B$, $\Re_t(B) = b$, and $d \sqsubset n$, $D \preceq N$.
- $t = (t_1, \ldots, t_m)$. Let $z, z' \in D_{2t}$, where $z = (z_1, \ldots, z_m) \sqsubseteq (z'_1, \ldots, z'_m) = z'$. Then

$$\begin{array}{rcll} \wp_t(z) & = & (\wp_{t_1}(z_1), \ldots, \wp_{t_m}(z_m)) & \{\text{by } \wp_t \text{ definition}\} \\ & \preceq & (\wp_{t_1}(z'_1), \ldots, \wp_{t_m}(z'_m)) & \{\text{by induction hypothesis (ih)}\} \\ & = & \wp_t(z') & \{\text{by } \wp_t \text{ definition}\} \end{array}$$

  Let $s, s' \in S_t$, where $s = (s_1, \ldots, s_m) \preceq (s'_1, \ldots, s'_m) = s'$. Then

$$\begin{array}{rll} \Re_t(s) & = (\Re_{t_1}(s_1), \ldots, \Re_{t_m}(s_m)) & \{\text{by } \Re_t \text{ definition }\} \\ & \sqsubseteq (\Re_{t_1}(s'_1), \ldots, \Re_{t_m}(s'_m)) & \{\text{by ih}\} \\ & = \Re_t(s') & \{\text{by } \Re_t \text{ definition }\} \end{array}$$

- $t = t_1 \to t_2, Process\ t_1\ t_2$. Let $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$ and $t_r = rType(t)$. Let $f, f' \in D_{2t}$ where $f \sqsubseteq f'$. Then

$$\begin{array}{rcll} \wp_t(f) & = & \overline{\wp_{t_r}(f\ (\gamma_{t_1}(d) \ldots \gamma_{t_i}(n) \ldots \gamma_{t_m}(d)))}\ \wp_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \{\text{by } \wp_t \text{ definition}\} \\ & \preceq & \overline{\wp_{t_r}(f'\ (\gamma_{t_1}(d) \ldots \gamma_{t_i}(n) \ldots \gamma_{t_m}(d)))}\ \wp_{t_r}(f'\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \{\text{by ih and } f \sqsubseteq f'\} \\ & = & \wp_t(f') & \end{array}$$

  Let $s = s_1 \ldots s_m\ s_{m+1} \in S_t$ and $s' = s'_1 \ldots s'_m\ s'_{m+1}$, such that $s \preceq s'$. Let $z_i \in D_{2t_i}$ $(i \in \{1..m\})$. We have to prove that $\Re_t(s)\ \overline{z_i} \sqsubseteq \Re_t(s')\ \overline{z_i}$. We have to distinguish several cases. If $\bigwedge_{i=1}^m z_i \sqsubseteq \gamma_{t_i}(d)$ then by definition of $\Re_t$ and induction hypothesis

$$\Re_t(s)\ \overline{z_i} = \Re_{t_r}(s_{m+1}) \sqsubseteq \Re_{t_r}(s'_{m+1}) = \Re_t(s')\ \overline{z_i}$$

  Otherwise, there is a $j \in \{1..m\}$ such that $z_j \not\sqsubseteq \gamma_{t_j}(d)$. If $\bigwedge_{i=1, i\neq j}^m z_i \sqsubseteq \gamma_{t_i}(d)$, then by definition of $\Re_t$ and induction hypothesis

$$\Re_t(s)\ \overline{z_i} = \Re_{t_r}(s_j) \sqsubseteq \Re_{t_r}(s'_j) = \Re_t(s')\ \overline{z_i}$$

  Otherwise by definition of $\Re_t$

$$\Re_t(s)\ \overline{z_i} = \gamma_{t_r}(n) = \Re_t(s')\ \overline{z_i}$$

- $t = \forall \beta.t_1$. This is trivial by induction hypothesis as $\wp_t = \wp_{t_1}$ and $\Re_t = \Re_{t_1}$.

  Now we are going to prove 1(b) and 1(c) simultaneously, also by structural induction.

- $t = K, T\ t_1 \ldots t_m, \beta$. This is a trivial case as $\mathcal{W}_t = id_{Basic}$ and $\wp_t(\Re_t(B)) = \wp_t(b) = B$.
- $t = (t_1, \ldots, t_m)$. Let $z_i \in D_{2t_i}$ $(i \in \{1..m\})$. In this case, by induction hypothesis

$$\mathcal{W}_t(z_1, \ldots, z_m) = (\mathcal{W}_{t_1}(z_1), \ldots \mathcal{W}_{t_m}(z_m)) \sqsupseteq (z_1, \ldots, z_m)$$

  Additionally

$$\begin{array}{rll} \wp_t(\Re_t(s_1, \ldots, s_m)) & = (\wp_{t_1}(\Re_{t_1}(s_1)), \ldots, \wp_{t_m}(\Re_{t_m}(s_m))) & \{\text{by } \wp_t \text{ and } \Re_t \text{ definition}\} \\ & = (s_1, \ldots, s_m) & \{\text{by induction hypothesis}\} \end{array}$$

$$
\begin{array}{l}
\mathcal{W}_t :: D_{2t} \to D_{2t} \\
\mathcal{W}_K = \mathcal{W}_{T\ t_1 \ldots t_m} = \mathcal{W}_\beta = id_{Basic} \\
\mathcal{W}_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = (\mathcal{W}_{t_1}(e_1),\ldots,\mathcal{W}_{t_m}(e_m)) \\
\mathcal{W}_{\forall\beta.t} = \mathcal{W}_t \\
\mathcal{W}_t(f) = \\
\quad \lambda z_1 \in D_{2t_1}.\ldots.\lambda z_m \in D_{2t_m}.\begin{cases} \mathcal{W}_{t_r}(f\ \gamma_{t_1}(d)\ldots\gamma_{t_m}(d)) \text{ if } \bigwedge_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d) \\ \mathcal{W}_{t_r}(f\ \gamma_{t_1}(d)\ldots\gamma_{t_i}(n)\ldots\gamma_{t_m}(d)) \text{ if } \bigwedge_{j=1,j\neq i} z_j \sqsubseteq \gamma_{t_j}(d) \wedge z_i \not\sqsubseteq \gamma_{t_i}(d) \quad i \in \{1..m\} \\ \gamma_{t_r}(n) \text{ otherwise } (m > 1) \end{cases} \\
\quad \text{where } \ t = t'_1 \to t'_2, Process\ t'_1\ t'_2, \quad m = nArgs(t), \quad t_r = rType(t), \quad [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 15. The widening operator

- $t = t_1 \to t_2, Process\ t_1\ t_2$. Let $m = nArgs(t)$, $[t_1,\ldots,t_m] = aTypes(t)$ and $t_r = rType(t)$.
  We prove first that $\mathcal{W}_t(d) \sqsupseteq id_{D_{2t}}$. Let $f \in D_{2t}$. Let $z_i \in D_{2t_i}$ $(i \in \{1..m\})$. We have to prove that $\mathcal{W}_t(f)\ \overline{z_i} \sqsupseteq f\ \overline{z_i}$. We can have different cases. If (1) $\bigwedge_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$, then

$$
\begin{array}{rll}
\mathcal{W}_t(f)\ \overline{z_i} = \mathcal{W}_{t_r}(f\ \overline{\gamma_{t_i}(d)}) & \{\text{by } \mathcal{W}_t \text{ definition }\} \\
\sqsupseteq f\ \overline{\gamma_{t_i}} & \{\text{by induction hypothesis}\} \\
\sqsupseteq f\ \overline{z_i} & \{\text{by (1) and } f \text{ monotone}\}
\end{array}
$$

  Otherwise there is a $j \in \{1..m\}$ such that $z_j \not\sqsubseteq \gamma_{t_j}(d)$. It has been proved that for each type t (2) $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(n)$.
  If (3) $\bigwedge_{i=1,i\neq j}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$ then

$$
\begin{array}{rll}
\mathcal{W}_t(f)\ \overline{z_i} = \mathcal{W}_{t_r}(f\ \gamma_{t_1}(d)\ldots\gamma_{t_j}(n)\ldots\gamma_{t_m}(d)) & \{\text{by } \mathcal{W}_t \text{ definition}\} \\
\sqsupseteq f\ \gamma_{t_1}(d)\ldots\gamma_{t_j}(n)\ldots\gamma_{t_m}(d) & \{\text{by induction hypothesis}\} \\
\sqsupseteq f\ \overline{z_i} & \{\text{by (2), (3) and f monotone}\}
\end{array}
$$

  Otherwise, by $\mathcal{W}_t$ definition and (2)

$$
\mathcal{W}_t(f)\ \overline{z_i} = \gamma_{t_r}(n) \sqsupseteq f\ \overline{z_i}
$$

  Now we prove that $\wp_t \cdot \Re_t = id_{S_t}$. Let $s_1 \ldots s_m\ s_{m+1} \in S_t$.

$$
\begin{array}{rll}
\wp_t(\Re_t(s_1\ldots s_m\ s_{m+1})) = \wp_{t_r}(\Re_{t_r}(s_1))\ldots\wp_{t_r}(\Re_{t_r}(s_m))\ \wp_{t_r}(\Re_{t_r}(s_{m+1})) & \{\text{by } \wp_t \text{ and } \Re_t \text{ definition}\} \\
= s_1\ldots s_m\ s_{m+1} & \{\text{by induction hypothesis}\}
\end{array}
$$

- $t = \forall\beta.t_1$. This case is trivial by induction hypothesis as $\mathcal{W}_t = \mathcal{W}_{t_1}$, $\wp_t = \wp_{t_1}$ and $\Re_t = \Re_{t_1}$.

□

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. In this third analysis its interpretation is:

$$
[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_3\ \rho_3 = [\![e']\!]_3\ (\mathit{fix}\ (\lambda\rho'_3.\rho_3\ \overline{[v_i \mapsto \mathcal{W}_{t_i}([\![e_i]\!]_3\ \rho'_3)]})) \quad \text{where } \ e_i :: t_i
$$

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, $[\![\cdot]\!]_3^{wop}$.

5 Properties of the Abstraction and Concretisation Functions

In the second and third analyses, and for each type t, we need two functions called the abstraction function, α_t, and the concretisation function, γ_t, both defined in Figure 5. They are mutually recursive and are respectively used in constructor applications and in *case* expressions.

We prove now that α_t and γ_t are a Galois insertion [NNH99], or equivalently a Galois surjection [CC92], or an embedding-closure pair[3] [Bar93] (see Proposition 2), and also that $\gamma_{t_1 \to t_2}(d)$ faithfully reflects the notion of deterministic function (see Proposition 4).

In the following proposition we prove that α_t and γ_t are monotone and continuous, and that they are a Galois insertion. This means that α_t loses information but just all at once; and that γ_t recovers as much as possible of it, so that another application of α_t does not lose more information. This fact guarantees that in the interpretation of *case* expressions, the information recovered from the discriminant is not only safe but also the best we can obtain considering how its abstraction was done (giving a non-deterministic abstract value to each variable in the alternative is also safe, but less accurate). Item 2(d) says that the unflattening of n is the top of the corresponding domain.

Proposition 2 *For each type t:*

(a) *The functions α_t and γ_t are monotone and continuous.*
(b) $\alpha_t \cdot \gamma_t = id_{Basic}$
(c) $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$
(d) $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(n)$

Proof. (Proposition 2) Items 2(a) and 2(d) can be proved by structural induction (s.i.) on t, and they are needed to prove, also by s.i., 2(b) and 2(c).

First, the item 2(a) is proved. If we prove that for each t, α_t and γ_t are monotone, then we will also have proved they are continuous, as their domains are finite (they satisfy the Ascending Chain Condition) [NNH99]. Monotonicity can be proved by structural induction on t.

We first prove that α_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\alpha_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. By i.h., for each $i = 1, \ldots, m$, α_{t_i} is monotone. Let $e_i, e'_i \in D_{2t_i}$. Let us assume that $(e_1, \ldots, e_m) \sqsubseteq (e'_1, \ldots, e'_m)$, that is, for each i, $e_i \sqsubseteq e'_i$. We have to prove that $\alpha_t(e_1, \ldots, e_m) \sqsubseteq \alpha_t(e'_1, \ldots, e'_m)$:

$$\begin{aligned} \alpha_t(e_1, \ldots, e_m) &= \bigsqcup_i \alpha_{t_i}(e_i) && \text{by definition of } \alpha_t \\ &\sqsubseteq \bigsqcup_i \alpha_{t_i}(e'_i) && \text{by i.h. and } e_i \sqsubseteq e'_i \\ &= \alpha_t(e'_1, \ldots, e'_m) && \text{by definition of } \alpha_t \end{aligned}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. By i.h. each α_{t_i} is monotone, for $i = 1, 2$. We have to prove that α_t is monotone. Let $f, f' \in [D_{2t_1} \to D_{2t_2}]$. Let us assume that $f \sqsubseteq f'$, that is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. We have to prove that $\alpha_t(f) \sqsubseteq \alpha_t(f')$:

$$\begin{aligned} \alpha_t(f) &= \alpha_{t_2}(f(\gamma_{t_1}(d))) && \text{by definition of } \alpha_t \\ &\sqsubseteq \alpha_{t_2}(f'(\gamma_{t_1}(d))) && \text{by i.h. and } f \sqsubseteq f' \\ &= \alpha_t(f') && \text{by definition of } \alpha_t \end{aligned}$$

- $t = \forall \beta.t'$. As $\alpha_t = \alpha_{t'}$, it trivially holds by i.h.

Let us prove now that γ_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These case are trivial as $\gamma_t = id_{Basic}$.

[3] Where γ_t is the embedding and α_t is the closure.

- $t = (t_1, \ldots, t_m)$. By i.h. for each $i = 1, \ldots, m$, γ_{t_i} is monotone. Let $b, b' \in \mathit{Basic}$. Let us assume $b \sqsubseteq b'$ and let us see that $\gamma_t(b) \sqsubseteq \gamma_t(b')$:

$$\begin{array}{rll} \gamma_t(b) & = (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & \sqsubseteq (\gamma_{t_1}(b'), \ldots, \gamma_{t_m}(b')) & \text{by i.h. and } b \sqsubseteq b' \\ & = \gamma_t(b') & \text{by definition of } \gamma_t \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. By i.h. each γ_{t_i} is monotone, for $i = 1, 2$. Let $b, b' \in \mathit{Basic}$. Let us assume that $b \sqsubseteq b'$ and let us prove that $\gamma_t(b) \sqsubseteq \gamma_t(b')$.
  When $b = b'$ this is trivial to prove. The non-trivial case is $b = d$ and $b' = n$. In such a case $\gamma_t(b) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ and $\gamma_t(b') = \lambda z.\gamma_{t_2}(n)$. As n is the top in Basic, we have that $\alpha_{t_1}(e) \sqsubseteq n$ for each $e \in D_{2t_1}$. So, by i.h.: $\forall e \in D_{2t_1}.\ \gamma_{t_2}(\alpha_{t_1}(e)) \sqsubseteq \gamma_{t_2}(n)$, from which we can obtain that $\gamma_t(d) \sqsubseteq \gamma_t(n)$.
- $t = \forall \beta.t'$. As $\gamma_t = \gamma_{t'}$, it trivially holds by i.h.

Now we prove item 2(d) also by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\gamma_t = id_{\mathit{Basic}}$, and n is the top in Basic.
- $t = (t_1, \ldots, t_m)$. If $e \in D_{2t}$, then $e = (e_1, \ldots, e_m)$, where $e_i \in D_{2t_i}$ for $i = 1, \ldots, m$. By i.h., for each $i = 1, \ldots, m$ we have that $e_i \sqsubseteq \gamma_{t_i}(n)$, so $e = (e_1, \ldots, e_m) \sqsubseteq (\gamma_{t_1}(n), \ldots, \gamma_{t_m}(n)) = \gamma_t(n)$.
- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in D_{2t}$. And let $e \in D_{2t_1}$. Then $f(e) \in D_{2t_2}$, and by i.h. $f(e) \sqsubseteq \gamma_{t_2}(n)$. On the other hand $\gamma_t(n) = \lambda z.\gamma_{t_2}(n)$. That is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq (\gamma_t(n))(e)$, so $f \sqsubseteq \gamma_t(n)$.
- $t = \forall \beta.t'$. As $\gamma_t = \gamma'_t$, it holds by i.h..

Let us prove now item 2(b).

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial, as $\alpha_t = \gamma_t = id_{\mathit{Basic}}$.
- $t = (t_1, \ldots, t_m)$. Let $b \in \mathit{Basic}$. Then:

$$\begin{array}{rll} \alpha_t(\gamma_t(b)) & = \alpha_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & = \bigsqcup_i \alpha_{t_i}(\gamma_{t_i}(b)) & \text{by definition of } \alpha_t \\ & = \bigsqcup_i b & \text{by i.h.} \\ & = b & \end{array}$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Let $b \in \mathit{Basic}$. We distinguish two cases: $b = d$ and $b = n$.
  If $b = d$, then:

$$\begin{array}{rll} \alpha_t(\gamma_t(d)) & = \alpha_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by definition of } \gamma_t \\ & = \alpha_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \text{by definition of } \alpha_t \\ & = d & \text{by i.h.} \end{array}$$

  If $b = n$, then:

$$\begin{array}{rll} \alpha_t(\gamma_t(n)) & = \alpha_t(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ & = \alpha_{t_2}(\gamma_{t_2}(n)) & \text{by definition of } \alpha_t \\ & = n & \text{by i.h.} \end{array}$$

- $t = \forall \beta.t'$. It directly holds by i.h. as $\alpha_t = \alpha_{t'}$ and $\gamma_t = \gamma_{t'}$.
  It remains to prove item 2(c).
- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial, as $\alpha_t = \gamma_t = id_{\mathit{Basic}}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ for each $i = 1, \ldots, m$. Then:

$$\begin{array}{ll} \gamma_t(\alpha_t(e_1, \ldots, e_m)) & \\ \quad = \gamma_t(\bigsqcup_i \alpha_{t_i}(e_i)) & \text{by def. of } \alpha_t \\ \quad = (\gamma_{t_1}(\bigsqcup_i \alpha_{t_i}(e_i)), \ldots, \gamma_{t_m}(\bigsqcup_i \alpha_{t_i}(e_i))) & \text{by def. of } \gamma_t \\ \quad \sqsupseteq (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \gamma_t \text{ monotone} \\ \quad \sqsupseteq (e_1, \ldots, e_m) & \text{by i.h.} \end{array}$$

– $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$. Then:

$$\gamma_t(\alpha_t(f)) = \gamma_t(\alpha_{t_2}(f(\gamma_{t_1}(d)))) \text{ by definition of } \alpha_t$$

We distinguish two cases:

- If $\alpha_{t_2}(f(\gamma_{t_1}(d)) = d$ then by definition of γ_t we have that $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Let $e \in D_{2t_1}$. We want to prove that $f(e) \sqsubseteq (\lambda z.\gamma_{t_2}(\alpha_{t_1}(z)))(e)$. That is, $f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$. We distinguish again two cases:
  * $e \sqsubseteq \gamma_{t_1}(d)$. By monotonicity of f and α_t, $\alpha_{t_2}(f(e)) \sqsubseteq \alpha_{t_2}(f(\gamma_{t_1}(d)))$ (which is equal to d), that is,

    $$\alpha_{t_2}(f(e)) \sqsubseteq d \sqsubseteq \alpha_{t_1}(e)$$

    The last inequality holds because d is the infimum in *Basic*. So, by monotonicity of γ_t, $\gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$, and by i.h. we have

    $$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$$

  * $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case $\alpha_{t_1}(e) = n$. Let us prove this by contradiction. Let us assume that $\alpha_{t_1}(e) \neq n$, i.e. that $\alpha_{t_1}(e) = d$. Then $\gamma_{t_1}(\alpha_{t_1}(e)) = \gamma_{t_1}(d)$. By i.h. we know that $e \sqsubseteq \gamma_{t_1}(\alpha_{t_1}(e))$, so $e \sqsubseteq \gamma_{t_1}(d)$, which contradicts the case in which we are.
    So, we have to prove that $f(e) \sqsubseteq \gamma_{t_2}(n)$, which holds by Proposition 2(d), as $f(e) \in D_{2t_2}$.
- If $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$, then $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$, so we have to prove that given $e \in D_{2t_1}$, $f(e) \sqsubseteq \gamma_{t_2}(n)$, which in fact holds by Proposition 2(d), as $f(e) \in D_{2t_2}$.

□

As a direct consequence of this proposition, we have the following (quite standard) one, that describes the relation between *Basic* and D_{2t} for each t. It asserts that all the values below $\gamma_t(d)$ are flattened to d. That is, $\gamma_t(d)$ represents d. But it also says that any other value, bigger that $\gamma_t(d)$ or incomparable with it, is flattened to n. This confirms that $\gamma_t(d)$ is the biggest value representing d.

Proposition 3 *For each type t: $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(d) \Leftrightarrow \alpha_t(e) = d$*

Proof. (Proposition 3) This can be proved by using propositions 2(a), 2(b) and 2(c).

(⇒) If $e \sqsubseteq \gamma_t(d)$ then, as α_t is monotone $\alpha_t(e) \sqsubseteq \alpha_t(\gamma_t(d))$. By Proposition 2(b), $\alpha_t(\gamma_t(d)) = d$, and as d is the infimum in *Basic*, we have that $\alpha_t(e) = d$.
(⇐) If $\alpha_t(e) = d$, then by Proposition 2(c), $e \sqsubseteq \gamma_t(\alpha_t(e)) = \gamma_t(d)$.

□

A function/process is considered as deterministic if when applied to a deterministic argument its result is also deterministic. This means that, when a function (resp. process) of type $t_1 \to t_2$ (resp. *Process* $t_1\ t_2$) is applied to a value less or equal than $\gamma_{t_1}(d)$, the result is less or equal than $\gamma_{t_2}(d)$. The following proposition says that a function of type $t_1 \to t_2$ is deterministic if and only if it is less or equal than $\gamma_{t_1 \to t_2}(d)$, i.e. $\gamma_{t_1 \to t_2}(d)$ is the biggest representative of the deterministic functions of that type. The same happens for processes.

Proposition 4 *Given a function $f \in D_{2t}$, where $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$:*

$$f \sqsubseteq \gamma_t(d) \Leftrightarrow \forall e \sqsubseteq \gamma_{t_1}(d).f(e) \sqsubseteq \gamma_{t_2}(d)$$

Proof. (Proposition 4) This can be proved by using propositions 2(d) and 3.

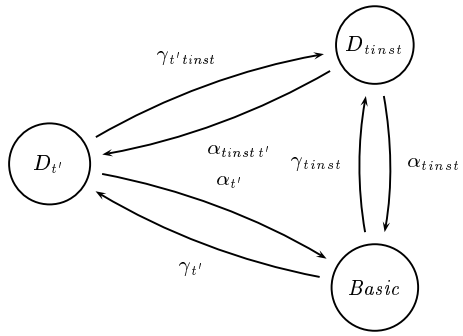

Fig. 16. Conmutativity properties of the abstraction and concretisation functions

(⇒) Let $f \sqsubseteq \gamma_t(d)$ and $e \sqsubseteq \gamma_{t_1}(d)$. By Proposition 3 we have that $\alpha_{t_1}(e) = d$. On the other hand $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ by definition of γ_t.
As $f \sqsubseteq \gamma_t(d)$ then:

$$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(d)$$

(⇐) Let us assume that for all $e \sqsubseteq \gamma_{t_1}(d)$ we have that $f(e) \sqsubseteq \gamma_{t_2}(d)$. We have to prove that $f \sqsubseteq \gamma_t(d)$. Let $e \in D_{2_{t_1}}$. We distinguish two cases:

- $e \sqsubseteq \gamma_{t_1}(d)$. In this case, by Proposition 3 we have that $\alpha_{t_1}(e) = d$ and:

$$\begin{aligned} f(e) &\sqsubseteq \gamma_{t_2}(d) && \text{by hypothesis} \\ &= \gamma_{t_2}(\alpha_{t_1}(e)) && \text{by Proposition 3} \\ &= (\gamma_t(d))(e) && \text{by definition of } \gamma_t \end{aligned}$$

- Otherwise, by Proposition 3 we have that $\alpha_{t_1}(e) = n$. So:

$$\begin{aligned} f(e) &\sqsubseteq \gamma_{t_2}(n) && \text{by Proposition 2(d)} \\ &= \gamma_{t_2}(\alpha_{t_1}(e)) && \text{by Proposition 3} \\ &= (\gamma_t(d))(e) && \text{by definition of } \gamma_t \end{aligned}$$

□

6 Polymorphism properties

We prove now some properties of the polymorphic versions of the abstraction and concretisation functions. Such versions were defined in Figure 8 by using functors $\times$ and $\rightarrow$ in the category of embedding-closure pairs. So the following proposition is trivially true by Proposition 2.

Proposition 5 *Given two types t, t' and a type variable β:*

(a) $\gamma_{t'tinst}$ *and* $\alpha_{tinst\,t'}$ *are monotone and continuous.*
(b) $\alpha_{tinst\,t'} \cdot \gamma_{t'tinst} = id_{D_{2_{t'}}}$
(c) $\gamma_{t'tinst} \cdot \alpha_{tinst\,t'} \sqsupseteq id_{D_{2_{tinst}}}$

In the following proposition we study the relationship between the abstraction/concretisation functions and its polymorphic counterparts. They have some conmutativity properties, shown in Figure 16.

Proposition 6 *Given two types t, t' and a type variable β:*

(a) $\gamma_{t'tinst} \cdot \gamma_{t'} = \gamma_{tinst}$
(b) $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$
(c) $\alpha_{tinst} \cdot \gamma_{t'tinst} = \alpha_{t'}$

(d) $\alpha_{tinst\,t'} \cdot \gamma_{tinst} = \gamma_{t'}$

Proof. (Proposition 6) Items 6(a) and 6(b) can be proved by s.i. on t'. Items 6(c) and 6(d) can be proved from the previous items and Proposition 5(b).

We prove simultaneously items 6(a) and 6(b) by structural induction on t':

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'\ (\neq \beta)$. In such cases $\gamma_{t'\,tinst} = \gamma_{t'} = \gamma_{tinst} = id_{Basic} = \alpha_{tinst\,t'} = \alpha_{t'} = \alpha_{tinst}$.
- $t' = (t_1, \ldots, t_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$ where $tinst_i = t_i[\beta := t]$, for each $i = 1, \ldots, m$. By i.h. we have for each $i = 1, \ldots, m$ that

$$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i}$$

and

$$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}$$

Let us prove first that $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Let $b \in Basic$.

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(b)) & \\
\quad = \gamma_{t'\,tinst}(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by def. of } \gamma_t \\
\quad = (\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b)), \ldots, \gamma_{t_m\,tinst_m}(\gamma_{t_m}(b))) & \text{by } \gamma_{t'\,tinst} \text{ def.} \\
\quad = (\gamma_{tinst_1}(b), \ldots, \gamma_{tinst_m}(b)) & \text{by i.h.} \\
\quad = \gamma_{tinst}(b) & \text{by def. of } \gamma_t
\end{array}$$

Let us prove now that $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Let $e_i \in D_{2\,tinst_i}$, for each $i = 1, \ldots, m$.

$$\begin{array}{ll}
\alpha_{t'}(\alpha_{tinst\,t'}(e_1, \ldots, e_m)) & \\
\quad = \alpha_{t'}(\alpha_{tinst_1\,t_1}(e_1), \ldots, \alpha_{tinst_m\,t_m}(e_m)) & \text{by def. of } \alpha_{tinst\,t'} \\
\quad = \bigsqcup_i \alpha_{t_i}(\alpha_{tinst_i\,t_i}(e_i)) & \text{by def. of } \alpha_t \\
\quad = \bigsqcup_i \alpha_{tinst_i}(e_i) & \text{by i.h.} \\
\quad = \alpha_{tinst}(e_1, \ldots, e_m) & \text{by def. of } \alpha_t
\end{array}$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. In this case $tinst = tinst_1 \to tinst_2$ (respectively $tinst = Process\ tinst_1\ tinst_2$) where $tinst_i = t_i[\beta := t]$. By i.h. we have that for each $i = 1, 2$,

$$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i}$$

and

$$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}$$

Let us prove first that $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Let $b \in Basic$. We distinguish two cases: $b = d$ and $b = n$.

If $b = d$:

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(d)) & \\
\quad = \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by def. of } \gamma_t \\
\quad = \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(\alpha_{t_1}(\alpha_{tinst_1\,t_1}(z)))) & \text{by def. of } \gamma_{t'\,tinst} \\
\quad = \lambda z.\gamma_{tinst_2}(\alpha_{tinst_1}(z)) & \text{by i.h.} \\
\quad = \gamma_{tinst}(d) & \text{by def. of } \gamma_t
\end{array}$$

If $b = n$ then:

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(n)) & \\
\quad = \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(n)) & \text{by def. of } \gamma_t \\
\quad = \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) & \text{by def. of } \gamma_{t'\,tinst} \\
\quad = \lambda z.\gamma_{tinst_2}(n) & \text{by i.h.} \\
\quad = \gamma_{tinst}(n) & \text{by def. of } \gamma_t
\end{array}$$

Let us prove now that $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Let $f \in D_{2\,tinst}$:

$$\begin{array}{ll}
\alpha_{t'}(\alpha_{tinst\,t'}(f)) & \\
\quad = \alpha_{t'}(\lambda z.\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(z)))) & \text{by def. of } \alpha_{tinst\,t'} \\
\quad = \alpha_{t_2}(\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(d))))) & \text{by def. of } \alpha_t \\
\quad = \alpha_{tinst_2}(f(\gamma_{tinst_1}(d))) & \text{by i.h.} \\
\quad = \alpha_{tinst}(f) & \text{by def. of } \alpha_t
\end{array}$$

– $t' = \beta$. In this case $tinst = t$ and by definition we have that $\gamma_{t'\,tinst} = \gamma_t$, $\alpha_{tinst\,t'} = \alpha_t$, $\gamma_{t'} = \alpha_{t'} = id_{Basic}$, so

$$\begin{aligned}
&\gamma_{t'\,tinst} \cdot \gamma_{t'} \\
&\quad = \gamma_t \cdot id_{Basic} && \text{by definition of } \gamma_{t'\,tinst} \text{ and } \gamma_t \\
&\quad = \gamma_{tinst} && \text{as } tinst = t
\end{aligned}$$

and

$$\begin{aligned}
&\alpha_{t'} \cdot \alpha_{tinst\,t'} \\
&\quad = id_{Basic} \cdot \alpha_t && \text{by definition of } \alpha_{tinst\,t'} \text{ and } \alpha_t \\
&\quad = \alpha_{tinst} && \text{as } tinst = t
\end{aligned}$$

– $t' = \forall\beta'.t_1$. In this case $tinst = tinst_1 = t_1[\beta := t]$ and then

$$\begin{aligned}
&\gamma_{t'\,tinst} \cdot \gamma_{t'} \\
&\quad = \gamma_{t_1\,tinst_1} \cdot \gamma_{t_1} && \text{by definition of } \gamma_{t'\,tinst} \text{ and } \gamma_t \\
&\quad = \gamma_{tinst_1} && \text{by i.h.} \\
&\quad = \gamma_{tinst} && \text{by definition of } \gamma_t
\end{aligned}$$

and

$$\begin{aligned}
&\alpha_{t'} \cdot \alpha_{tinst\,t'} \\
&\quad = \alpha_{t_1} \cdot \alpha_{tinst_1\,t_1} && \text{by definition } \alpha_{tinst\,t'} \text{ and } \alpha_t \\
&\quad = \alpha_{tinst_1} && \text{by i.h.} \\
&\quad = \alpha_{tinst} && \text{by definition of } \alpha_t
\end{aligned}$$

Items 6(c) and 6(d) can be proved easily from the previous ones:

$$\begin{aligned}
&\alpha_{tinst} \cdot \gamma_{t'\,tinst} \\
&\quad = (\alpha_{t'} \cdot \alpha_{tinst\,t'}) \cdot \gamma_{t'\,tinst} && \text{by Proposition 6(b)} \\
&\quad = \alpha_{t'} && \text{by associativity and Prop. 5(b)}
\end{aligned}$$

$$\begin{aligned}
&\alpha_{tinst\,t'} \cdot \gamma_{tinst} \\
&\quad = \alpha_{tinst\,t'} \cdot (\gamma_{t'\,tinst} \cdot \gamma_{t'}) && \text{by Proposition 6(a)} \\
&\quad = \gamma_{t'} && \text{by associativity and Prop. 5(b)}
\end{aligned}$$

□

We have already seen that the semantics of a type application is obtained by using the function $\gamma_{t'\,tinst}$. But we could also have defined the semantics of the instance in another way, first abstracting to *Basic* the abstract value of the smallest instance and then applying the concretisation function to *tinst*, that is:

$$[\![e\ t]\!]_i\ \rho_i = \gamma_{tinst}(\alpha_{t'}([\![e]\!]_i\ \rho_i)) \qquad i \in \{2, 3\}$$

The first item of the following lemma tells us that this would be a worse choice, that is, we would lose more information. The second one completes the conmutativity properties of interest shown in Figure 16.

Proposition 7 *Given two types t, t' and a type variable β:*

(a) $\gamma_{tinst} \cdot \alpha_{t'} \sqsupseteq \gamma_{t'\,tinst}$
(b) $\gamma_{t'} \cdot \alpha_{tinst} \sqsupseteq \alpha_{tinst\,t'}$

Proof. (Proposition 7) The first item can be proved from propositions 6(a) and 2(c), and the second one from propositions 6(b) and 2(c):

$$\begin{aligned}
&\gamma_{tinst} \cdot \alpha_{t'} \\
&\quad = (\gamma_{t'\,tinst} \cdot \gamma_{t'}) \cdot \alpha_{t'} && \text{by Proposition 6(a)} \\
&\quad \sqsupseteq \gamma_{t'\,tinst} && \text{by associativity and Prop. 2(c)}
\end{aligned}$$

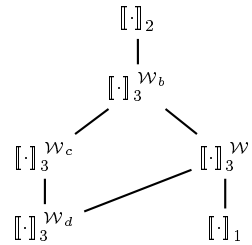

Fig. 17. A hierarchy of analyses

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

Fig. 18. An example expression e

$$
\begin{array}{ll}
\gamma_{t'} \cdot \alpha_{tinst} & \\
\quad = \gamma_{t'} \cdot (\alpha_{t'} \cdot \alpha_{tinst\,t'}) & \text{by Proposition 6(b)} \\
\quad \sqsupseteq \alpha_{tinst\,t'} & \text{by associativity and Prop. 2(c)}
\end{array}
$$

□

A generalization of the pair $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'})$ can be defined, where several type variables are instantiated in sequence. This corresponds to a type $\forall\beta_1.\ldots.\forall\beta_m.t'$ and an instantiation $tinst = t'[\beta_1 := t_1, \ldots, \beta_m := t_m]$. The analysis domains together with these pairs as morphisms form a category (by propositions 6(a) and 6(b)).

7 Relation between the Analyses

The third analysis was intended to be an intermediate analysis between the two analyses presented in [PS01]. In this section we will mentione other possible widening operators that could be defined ($\mathcal{W}_b$, $\mathcal{W}_c$ and $\mathcal{W}_d$) and their relation with $\mathcal{W}$. The main difference between them lies in their treatment of the tuples, in the arguments and/or in the result of the functions, either as indivisible entities or componentwise. In this section we show first that the third analysis is a safe approximation to the second one, and afterwards we show that the first analysis is a safe approximation to only some variants of the third one. This will allow us to establish a hierarchy of analyses, shown in Figure 17.

In Figure 18 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 $({p_1}^{2*p_2}, x * p_2!)$. The variable q is free 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 we obtain that $[\![e]\!]_1\ \rho = (n, n) \sqsupset [\![e]\!]_3^{\mathcal{W}} \rho = (n, d) \sqsupset [\![e]\!]_2\ \rho = (d, d)$.

7.1 Relation between the second and third analyses

The following proposition 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.

Proposition 8 *Let $\mathcal{W}'_t : D_{2t} \rightarrow D_{2t}$ be a widening operator for each type t. Given ρ_2 and ρ_3 such that for each variable $v :: t_v$ $\rho_2(v) \sqsubseteq \rho_3(v)$, then for each expression $e :: t_e$, $[\![e]\!]_2\ \rho_2 \sqsubseteq [\![e]\!]_3^{\mathcal{W}'} \rho_3$.*

Proof. (Proposition 8) Let $\mathcal{W}'_t : D_{2t} \rightarrow D_{2t}$ be a widening operator for each type t. This proposition can be proved by structural induction on e. All the cases but the recursive *let* expression can be easily proved by using the hypothesis over the environments, the induction hypothesis and the monotonicity properties of α_t, γ_t, $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$. So, here we only look at the recursive *let* expression. This is a standard proof.

Let $e = \textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e' :: t$, where $e' :: t$, and each v_i and e_i have type t_i. The fixpoint can be caculated by using the Kleene ascending chain:

$$[\![e]\!]_2\ \rho_2 = [\![e']\!]_2\ (\bigsqcup_{n\in\mathbb{N}} (\lambda\rho_2'.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho_2']})^n(\rho_{02}))$$

where ρ_{02} is the initial environment where each variable $y :: t_y$ has $-_{t_y}$ as abstract value (that is, the infimum of the corresponding domain). Let us call F to the function between environments $\lambda\rho_2'.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho_2']}$. Let us call ρ_2^{fix} to $\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{02})$.

On the other side:

$$[\![e]\!]_3^{\mathcal{W}'}\ \rho_t = [\![e']\!]_3^{\mathcal{W}'}(\bigsqcup_{n\in\mathbb{N}} (\lambda\rho_3'.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_3')]})^n(\rho_{03}))$$

where ρ_{03} is the initial environment where each variable $y :: t_y$ has $-_{t_y}$ as abstract value (that is, the infimum of the corresponding domain). Let us call G to the function between environments $\lambda\rho_3'.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_3')]}$. Let us call ρ_3^{fix} to $\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{03})$.

If we proved that for each variable $y :: t_y$, $\rho_2^{fix}(y) \sqsubseteq \rho_3^{fix}(y)$, then by i.h. we would have that

$$[\![e']\!]_2\ \rho_2^{fix} \sqsubseteq [\![e']\!]_3^{\mathcal{W}'}\ \rho_3^{fix}$$

which is what we wanted to prove. Let us see that for each $n \geq 0$, the following holds

$$\forall y :: t_y.(F^n(\rho_{02}))(y) \sqsubseteq (G^n(\rho_{03}))(y)$$

If this were true then

$$\forall y :: t_y.(\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{02}))(y) \sqsubseteq (\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{03}))(y)$$

and this is what we want to prove. It can be proved by induction on n:

- $n = 0$. This is a trivial case as $F^0(\rho_{02}) = \rho_{02}$, $G^0(\rho_{03}) = \rho_{03}$, which are equal.
- $n = m + 1$. Then

$$\begin{aligned} F^{m+1}(\rho_{02}) &= F(F^m(\rho_{02})) \\ &= \rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ (F^m(\rho_{02}))]} \text{ by definition of } F \end{aligned}$$

and

$$\begin{aligned} G^{m+1}(\rho_{03}) &= G(G^m(\rho_{03})) \\ &= \rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_{03})))]} \text{ by definition of } G \end{aligned}$$

Let $y :: t_y$. We want to prove that $(F^{m+1}(\rho_{02}))(y) \sqsubseteq (G^{m+1}(\rho_{03}))(y)$. We distinguish two cases. If y is not any of the v_i, then it holds by the hypothesis over the environments ρ_2 and ρ_3. If it is one of the v_i, then we have to prove that

$$[\![e_i]\!]_2\ (F^m(\rho_{02})) \sqsubseteq \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_{03})))$$

This holds by induction hypothesis and hypothesis on $\mathcal{W}'_t$:

$$[\![e_i]\!]_2\ (F^m(\rho_{02})) \sqsubseteq [\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_{03})) \sqsubseteq \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_{03})))$$

□

7.2 Other variants of the third analysis

In this subsection other variants of the third analysis are presented. The relations between them and with $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$ analyses are shown in Figure 17.

First of all, we are going to define a variant $[\![\cdot]\!]_3^{\mathcal{W}_b}$ of the third analysis that is more powerful than $[\![\cdot]\!]_3^{\mathcal{W}}$.The idea is to curry the tuple arguments to obtain more information. In the current version, a function of type $(t_1, \ldots, t_m) \to t$ is probed with $\gamma_{(t_1,\ldots,t_m)}(n)$ and $\gamma_{(t_1,\ldots,t_m)}(d)$. In this new variant, it is probed with $(\gamma_{t_1}(n), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)), (\gamma_{t_1}(d), \gamma_{t_2}(n), \ldots, \gamma_{t_m}(d)) \ldots (\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(n))$, that is, the tuple components are considered as separate arguments. The results of these probings are grouped using a special notation $\mathcal{U}(A_1, \ldots, A_m)$ to distinguish a tuple probing from a result tuple.

As an example, let $f \in D_{2t}$, where $t = (Int, Int) \to Int \to (Int, Int)$, be $\lambda x \in Basic \times Basic.\lambda x \in Basic.(\pi_1(p), x)$. In this case, our current version would return $s = (N, D)(D, N) + (D, D)$, corresponding to the probings $f\ (n, n)\ d$, $f\ (d, d)\ n$ and $f\ (d, d)\ d$. In this new probing $s' = \wp_t^B(f) = \mathcal{U}((N, D), (D, D))(D, N) + (D, D)$. The first component $\mathcal{U}((N, D), (D, D))$ corresponds to the probings $f\ (n, d)\ d$ and $f\ (d, n)\ d$, and the other two, correspond to the probings $f\ (d, d)\ n$ and $f\ (d, d)\ d$. Notice that the number of probings is higher.

We can also define variants that are less powerful than the variant we have chosen or uncomparable with it. One possibility is to flatten the tuples in the result of the function. Consequently we are losing information. We will call this variant $[\![\cdot]\!]_3^{\mathcal{W}_d}$. Another possibility is to curry the tuple arguments as in $[\![\cdot]\!]_3^{\mathcal{W}_b}$, but flatten the tuples in the result (so it is less accurate than this one). We will call this variant $[\![\cdot]\!]_3^{\mathcal{W}_c}$. As it is shown in Figure 17, this variant is uncomparable with $[\![\cdot]\!]_3^{\mathcal{W}}$. In those functions with a tuple type as result but no tuple arguments $[\![\cdot]\!]_3^{\mathcal{W}}$ produces more accurate results; while in those functions with tuple arguments but no tuple result, $[\![\cdot]\!]_3^{\mathcal{W}_c}$ is better. Finally $[\![\cdot]\!]_3^{\mathcal{W}_c}$ is clearly better than $[\![\cdot]\!]_3^{\mathcal{W}_b}$. The relation of these variants with $[\![\cdot]\!]_1$ are shown in Figure 17, although this should be proved in more detail.

The following proposition tells us that given two comparable widening operators, the corresponding variants of the third analysis are also comparable.

Proposition 9 *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 $e :: t_e$, $[\![e]\!]_3^{\mathcal{W}'} \rho_3 \sqsubseteq [\![e]\!]_3^{\mathcal{W}''} \rho'_3$*

Proof. (Proposition 9) This proof is very similar to the previous one. The same steps can be followed in the recursive *let* expression, being now

$$F = \lambda\rho'_3.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'} \rho'_3)]})^n(\rho_{03}))$$

and

$$G = \lambda\rho''_3.\rho_3\ \overline{[v_i \to \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''} \rho''_3)]})^n(\rho_{03}))$$

The same induction on n is done, and at the end we have to prove that

$$\mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_{02}))) \sqsubseteq \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''}(G^m(\rho_{03})))$$

which is true by induction hypothesis, $\mathcal{W}'_t \sqsubseteq \mathcal{W}''_t$ and monotonicity of $\mathcal{W}''_t$:

$$\begin{aligned}\mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_{02}))) &\sqsubseteq \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_{02}))) \\ &\sqsubseteq \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''}(G^m(\rho_{03})))\end{aligned}$$

□

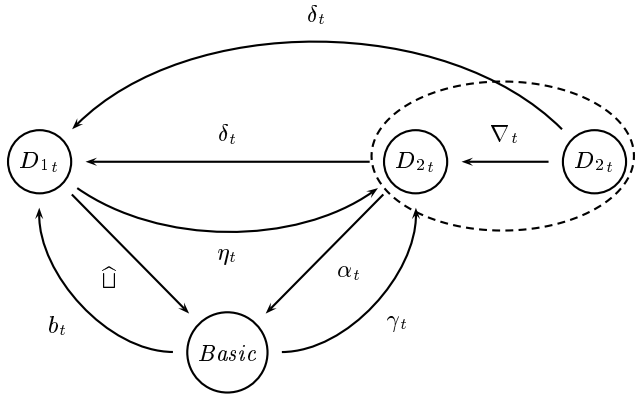

Fig. 19. Representative scheme of the functions in Section 7

7.3 Relation between the first and third analyses

Grosso modo we will see that the first analysis is worse than some variants of the third analysis, but in a safe way, that is, it is a safe approximation to them. In particular we will prove that the first analysis is an upper aproximation to those variants of the third analysis that fulfill a property (see Theorem 19). We will prove that the particular widening operator $\mathcal{W}_t$ defined in Figure 15 fulfills the property. As all the variants of the third analysis are upper approximations to the second one, it will also happen that the first analysis is an upper approximation to the second analysis.

In Figure 19 an scheme of the functions defined in this section and their definition domains are shown. Many propositions in this section can be easily visualized using this figure. Recall that the abstract domains for the second and the third analyses are the same, so all the propositions involving only the domains are true for both analyses.

In the first analysis there are neither nested tuples nor abstract functions. However, the abstract value a of a function in such analysis represents an abstract function. We already know that a represents the behaviour of the function when given a deterministic argument. If the argument is non-deterministic, or it has any 'dose' of non-determinism, the result of the application is always non-deterministic. What we are really doing is to simulate the behaviour of an abstract function. Let us assume that the abstract value of a function in the first analysis is a. Which is the abstract function a represents? It is a function that takes an abstract value; if that value is non-deterministic the result is also non-deterministic, and if it is deterministic it gives as result the abstract value a. This leads us to define a function η_t, see Figure 20 (left), called the *expansion function* that *expands* the abstract values of the first analysis domains into abstract values belonging to the second and third analyses abstract domains. Recall also that in the first analysis there are no nested tuples so it will be necessary to adapt each of the basic values to the types of the components. For basic types, algebraic types and type variables η_t is trivially the identity function, as in both analyses the corresponding abstract domains are *Basic*. The polymorphic type case is reduced to the type without qualifier.

We can also look at the second and third analyses from the point of view of the first one. We just have to flatten the tuples up to the first level and apply the functions to $\gamma_t(d)$, to obtain the behaviour of the function when it is applied to a deterministic argument. This is represented by the *compression function* δ_t defined in Figure 20 (right). The cases of basic type, algebraic type, type variable and polymorphic type are similar to those of the expansion function.

We prove now that δ_t and η_t are a Galois insertion.

Proposition 10 *For each type t:*

(a) *The functions δ_t and η_t are monotone and continuous.*
(b) $\delta_t \cdot \eta_t = id_{D_{1_t}}$

(c) $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$

Proof. (Proposition 10) Item 10(a) can be easily proved by structural induction on t and Proposition 2(a). Items 10(b) and 10(c) can be proved by structural induction on t and by propositions 2(d), 2(b) and 2(c).

We first prove item 10(a). If we prove that they are monotone, as their domains of definition are finite, then they will also be continuous. The monotonicity can be proved by structural induction on t. Let us prove that δ_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\delta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i, e'_i \in D_{2t_i}$. Let us assume that $(e_1, \ldots, e_m) \sqsubseteq (e'_1, \ldots, e'_m)$, that is, for each $i = 1, \ldots, m$, $e_i \sqsubseteq e'_i$. We have to prove that $\delta_t(e_1, \ldots, e_m) \sqsubseteq \delta_t(e'_1, \ldots, e'_m)$:

$$\begin{aligned}
&\delta_t(e_1, \ldots, e_m) \\
&\quad = (\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) && \text{by definition of } \delta_t \\
&\quad \sqsubseteq (\alpha_{t_1}(e'_1), \ldots, \alpha_{t_m}(e'_m)) && \text{by monotonicity of } \alpha_t \\
&\quad = \delta_t(e'_1, \ldots, e'_m) && \text{by definition of } \delta_t
\end{aligned}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f, f' \in [D_{2t_1} \to D_{2t_2}]$. Let us assume that $f \sqsubseteq f'$, that is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. We have to prove that $\delta_t(f) \sqsubseteq \delta_t(f')$.

$$\begin{aligned}
&\delta_t(f) \\
&\quad = \delta_{t_2}(f(\gamma_{t_1}(d))) && \text{by definition of } \delta_t \\
&\quad \sqsubseteq \delta_{t_2}(f'(\gamma_{t_1}(d))) && \text{by i.h. and } f \sqsubseteq f' \\
&\quad = \delta_t(f') && \text{by definition of } \delta_t
\end{aligned}$$

- $t = \forall\beta.t'$. As $\delta_t = \delta_{t'}$, it trivially holds by i.h.

And now let us prove the monotonicity of η_t.

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\eta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $b_i, b'_i \in \mathit{Basic}$. Let us assume that $(b_1, \ldots, b_m) \sqsubseteq (b'_1, \ldots, b'_m)$, that is, for each $i = 1, \ldots, m$, $b_i \sqsubseteq b'_i$. We have to prove that $\eta_t(b_1, \ldots, b_m) \sqsubseteq \eta_t(b'_1, \ldots, b'_m)$:

$$\begin{aligned}
&\eta_t(b_1, \ldots, b_m) \\
&\quad = (\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) \\
&\quad \sqsubseteq (\gamma_{t_1}(b'_1), \ldots, \gamma_{t_m}(b'_m)) && \text{by monotonicity of } \gamma_t \\
&\quad = \eta_t(b'_1, \ldots, b'_m)
\end{aligned}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $a, a' \in D_{1t} = D_{1t_2}$. Let us assume that $a \sqsubseteq a'$. We have to prove that for each $e \in D_{2t_1}$, $(\eta_t(a))(e) \sqsubseteq (\eta_t(a'))(e)$. Let $e \in D_{2t_1}$. We distinguish two cases: $e \sqsubseteq \gamma_{t_1}(d)$ or $e \not\sqsubseteq \gamma_{t_1}(d)$.
  If $e \sqsubseteq \gamma_{t_1}(d)$, then

$$\begin{aligned}
&(\eta_t(a))(e) \\
&\quad = \eta_{t_2}(a) && \text{by def. of } \eta_t \\
&\quad \sqsubseteq \eta_{t_2}(a') && \text{by i.h. and } a \sqsubseteq a' \\
&\quad = (\eta_t(a'))(e) && \text{by def. of } \eta_t
\end{aligned}$$

  Otherwise, $e \not\sqsubseteq \gamma_{t_1}(d)$, and by definition of η_t: $(\eta_t(a))(e) = \gamma_{t_2}(n) = (\eta_t(a'))(e)$.
- $t = \forall\beta.t'$. As $\eta_t = \eta_{t'}$, it trivially holds by i.h.

We now prove items 10(b) and 10(c). Both things can be proved by structural induction on t. Let us prove first that $\delta_t \cdot \eta_t = id_{D_{1t}}$:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $D_{1t} = \mathit{Basic}$ and $\eta_t = \delta_t = id_{Basic}$.

– $t = (t_1, \ldots, t_m)$. Let $b_i \in \mathit{Basic}$, where $i = 1, \ldots, m$. Then:

$$\begin{array}{ll} \delta_t(\eta_t(b_1, \ldots, b_m)) & \\ \quad = \delta_t(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) & \text{by def. of } \eta_t \\ \quad = (\alpha_{t_1}(\gamma_{t_1}(b_1)), \ldots, \alpha_{t_m}(\gamma_{t_m}(b_m))) & \text{by def. of } \delta_t \\ \quad = (b_1, \ldots, b_m) & \text{by Prop. 2(b)} \end{array}$$

– $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $a \in D_{1t}$. Then

$$\begin{array}{ll} \delta_t(\eta_t(a)) & \\ \quad = \delta_t\left(\lambda z. \begin{cases} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}\right) & \text{by def. of } \eta_t \\ \quad = \delta_{t_2}(\eta_{t_2}(a)) & \text{by def. of } \delta_t \\ \quad = a & \text{by i.h.} \end{array}$$

– $t = \forall\beta.t'$. In this case $D_{1t} = D_{1t'}$, $\eta_t = \eta_{t'}$ and $\delta_t = \delta_{t'}$, so it trivially holds by i.h..

Let us prove now that $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$:

– $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $D_{2t} = \mathit{Basic}$ and $\eta_t = \delta_t = id_{\mathit{Basic}}$.
– $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$, where $i = 1, \ldots, m$. Entonces, tenemos:

$$\begin{array}{ll} \eta_t(\delta_t(e_1, \ldots, e_m)) & \\ \quad = \eta_t(\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) & \text{by def. of } \delta_t \\ \quad = (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by def. of } \eta_t \\ \quad \sqsupseteq (e_1, \ldots, e_m) & \text{by Prop. 2(c)} \end{array}$$

– $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in D_{2t}$. Then

$$\begin{array}{ll} \eta_t(\delta_t(f)) & \\ \quad = \eta_t(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \text{by def. of } \delta_t \\ \quad = \lambda z. \begin{cases} \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases} & \text{by def. of } \eta_t \end{array}$$

Let $e \in D_{2t_1}$. We have to prove that $f(e) \sqsubseteq (\eta_t(\delta_t(f)))(e)$. We distinguish two cases::

- $e \sqsubseteq \gamma_{t_1}(d)$. In this case

$$(\eta_t(\delta_t(f)))(e) = \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d))))$$

As f is continuous and consequently monotone, we have that $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. So

$$\begin{array}{ll} (\eta_t(\delta_t(f)))(e) & \\ \quad = \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \\ \quad \sqsupseteq f(\gamma_{t_1}(d)) & \text{by i.h.} \\ \quad \sqsupseteq f(e) & \text{as } f \text{ is monotone} \end{array}$$

- $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case $(\eta_t(\delta_t(f)))(e) = \gamma_{t_2}(n)$. By Lemma 2(d) $\gamma_{t_2}(n) \sqsupseteq f(e)$, as $f(e) \in D_{2t_2}$.

– $t = \forall\beta.t'$. In this case $D_{2t} = D_{2t'}$, $\eta_t = \eta_{t'}$ and $\delta_t = \delta_{t'}$, so it trivially holds by i.h..

□

The following proposition asserts that given an abstract value in D_{2t}, if we apply δ_t to it to obtain a value in D_{1t} and then we apply $\widehat{\square}$ to the result (it could be a tuple of basic abstract values) we obtain the same as applying α_t to it directly to obtain a value in *Basic*. This shows a similarity between the abstraction function α_t used in constructor applications and the compression function δ_t. In the end the ideas are the same.

$$\eta_t : D_{1t} \to D_{2t}$$
$$\eta_K = \eta_{T\ t_1 \ldots t_m} = \eta_\beta = id_{Basic}$$
$$\eta_{(t_1,\ldots,t_m)}(b_1,\ldots,b_m) = (\gamma_{t_1}(b_1),\ldots,\gamma_{t_m}(b_m))$$
$$\eta_{t_1 \to t_2}(a) = \eta_{Process\ t_1\ t_2}(a) = \lambda z \in D_{2t_1}. \begin{cases} \eta_{t_2}(a) \ if\ z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n)\ otherwise \end{cases}$$
$$\eta_{\forall\beta.t} = \eta_t$$

$$\delta_t : D_{2t} \to D_{1t}$$
$$\delta_{T\ t_1 \ldots t_m} = \delta_\beta = id_{Basic}$$
$$\delta_{(t_1,\ldots,t_m)}(b_1,\ldots,b_m) = (\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m))$$
$$\delta_{t_1 \to t_2}(f) = \delta_{Process\ t_1\ t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d)))$$
$$\delta_{\forall\beta.t} = \delta_t$$

Fig. 20. Expansion and compression functions

Proposition 11 *For each type t:* $\widehat{\sqcup} \cdot \delta_t = \alpha_t$

Proof. (Proposition 11) This proposition can be proved by s.i. on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\delta_t = \alpha_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$, where $i = 1, \ldots, m$. Then:

$$\begin{aligned} &\widehat{\sqcup}(\delta_t(e_1,\ldots,e_m)) \\ &\quad = \widehat{\sqcup}(\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m)) && \text{by definition of } \delta \\ &\quad = \bigsqcup_i \alpha_{t_i}(e_i) && \text{by definition of } \widehat{\sqcup} \\ &\quad = \alpha_t(e_1,\ldots,e_m) && \text{by definition of } \alpha_t \end{aligned}$$

- $t = t_1 \to t_2$ or $t = Process\ t_1\ t_2$. Let $f \in D_{2t}$. Then:

$$\begin{aligned} &\widehat{\sqcup}(\delta_t(f)) \\ &\quad = \widehat{\sqcup}(\delta_{t_2}(f(\gamma_{t_1}(d)))) && \text{by definition of } \delta_t \\ &\quad = \alpha_{t_2}(f(\gamma_{t_1}(d))) && \text{by i.h.} \\ &\quad = \alpha_t(f) && \text{by definition of } \alpha_t \end{aligned}$$

- $t = \forall\beta.t'$. In this case $\delta_t = \delta'_t$ and $\alpha_t = \alpha_{t'}$, so it trivially holds by i.h..

□

A direct consequence of Proposition 10 is that δ_t is strict, as in an embedding-closure pair the closure is always strict.

Proposition 12 *For each type t:* $\delta_t(-_{2t}) = d_t$

Proposition 10 tells us that $\eta_t \cdot \delta_t$ is a widening operator in D_{2t}. This composition will be widely used in the following, so we define $\nabla_t = \eta_t \cdot \delta_t$. We will call it the *rising function* (just to differentiate it from $\mathcal{W}_t$). Its unfolded definition is shown in Figure 21.

It also tells us that for each type t, the range of ∇_t is isomorphic to D_{1t}: $\nabla_t(D_{2t}) \simeq D_{1t}$. This means that the range of ∇_t is a subdomain of D_{2t} where we have lost the additional information provided by the second and third analyses. For example, we still have nested tuples, but they are maintained in a fictitious way, that is, they have been flattened up to the first level and then unflattened again. So, all the internal tuples will be formed by only n's or only d's. For example both $((n,d),d,(d,d))$ and $((d,n),d,(d,d))$ are transformed by ∇_t into $((n,n),d,(d,d))$. We also have abstract functions, but only some of them: Those that can be represented with an abstract value in the first analysis domains. These are the functions such that for all values below $\gamma_{t_1}(d)$, give the same result as the one obtained for $\gamma_{t_1}(d)$, and for the rest of the values give as result the top of the corresponding domain, $\gamma_{t_2}(n)$.

We have seen that $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$ and $\nabla_t \sqsupseteq id_{D_{2t}}$. But, how are they related? The following proposition tells us that ∇_t is smaller (i.e. better) than $\gamma_t \cdot \alpha_t$.

Proposition 13 *For each type t:* $\nabla_t \sqsubseteq \gamma_t \cdot \alpha_t$

Proof. (Proposition 13) This can be proved by structural induction on t and propositions 2(a), 2(d) and 3:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \alpha_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ for $i = 1, \ldots, m$. Then:

$$\begin{aligned}
&\nabla_t(e_1, \ldots, e_m) \\
&\quad = (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) \\
&\qquad \{\text{by definition of } \nabla_t\} \\
&\quad \sqsubseteq (\gamma_{t_1}(\bigsqcup_i \alpha_{t_i}(e_i)), \ldots, \gamma_{t_m}(\bigsqcup_i \alpha_{t_i}(e_i))) \\
&\qquad \{\text{as } \alpha_{t_j}(e_j) \sqsubseteq \bigsqcup_i \alpha_{t_i}(e_i) \text{ for each } j = 1, \ldots, m \\
&\qquad \text{and } \gamma_{t_i} \text{ is monotone by Proposition 2(a)}\} \\
&\quad = \gamma_t(\alpha_t(e_1, \ldots, e_m)) \\
&\qquad \{\text{by definition of } \alpha_t \text{ and } \gamma_t\}
\end{aligned}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$ By definition of ∇_t

$$\nabla_t(f) = \lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}$$

  By definition of γ_t and α_t

$$\gamma_t(\alpha_t(f)) = \lambda z. \begin{cases} \gamma_{t_2}(\alpha_{t_1}(z)) \text{ if } \alpha_{t_2}(f(\gamma_{t_1}(d))) = d \\ \gamma_{t_2}(n) \quad \text{otherwise} \end{cases}$$

  We distinguish two cases:
  - $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$. In this case $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$. Let $e \in D_{2t_1}$. We have that $(\gamma_t(\alpha_t(f)))(e) = \gamma_{t_2}(n)$. So, by Lemma 2(d) we have that $\gamma_{t_2}(n) \sqsupseteq (\nabla_t(f))(e)$ as $(\nabla_t(f))(e) \in D_{2t_2}$.
  - $\alpha_{t_2}(f(\gamma_{t_1}(d))) = d$. Now $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Let $e \in D_{2t_1}$. We distinguish again two cases: $e \sqsubseteq \gamma_{t_1}(d)$ or not.
    If $e \sqsubseteq \gamma_{t_1}(d)$ then $(\nabla_t(f))(e) = \nabla_{t_2}(f(\gamma_{t_1}(d)))$. By Proposition 3 $\alpha_{t_1}(e) = d$. So, what we really want to prove is that $\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(d)$. By i.h. we know that $\nabla_{t_2} \sqsubseteq \gamma_{t_2} \cdot \alpha_{t_2}$, so:

$$\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(\gamma_{t_1}(d)))) = \gamma_{t_2}(d)$$

    In case $e \not\sqsubseteq \gamma_{t_1}(d)$, then $(\nabla_t(f))(e) = \gamma_{t_2}(n)$. By Lemma 3 $\alpha_{t_1}(e) = n$, so $\gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(n)$, and we have the equality.
- $t = \forall \beta.t'$. As $\alpha_t = \alpha_{t'}$, $\gamma_t = \gamma_{t'}$ and $\nabla_t = \nabla_{t'}$ by definition, it trivially holds by i.h.

□

Theorem 19 below asserts the correctness of the first analysis with respect to the third one, but first we will see some more properties that are needed to prove it. The following proposition relates the adaptation function of the first analysis and the concretisation function γ_t. The are made equal through the application of δ_t.

Proposition 14 *For each type t:* $\forall b \in Basic.\ b_t = (\delta_t \cdot \gamma_t)(b)$

Proof. (Proposition 14) This can be proved by s.i. on t and by Proposition 2(b):

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $b_t = b$ and $\delta_t = \gamma_t = id_{Basic}$.

– $t = (t_1, \ldots, t_m)$. In this case we have:

$$\begin{array}{ll} \delta_t(\gamma_t(b)) & \\ \quad = \delta_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by def. of } \gamma_t \\ \quad = (\alpha_{t_1}(\gamma_{t_1}(b)), \ldots, \alpha_{t_m}(\gamma_{t_m}(b))) & \text{by def. of } \delta_t \\ \quad = (b, \ldots, b) & \text{by Prop. 2(b)} \\ \quad = b_t & \text{by def. of } b_t \end{array}$$

– $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. We distinguish two cases:: $b = d$ and $b = n$.

- $b = d$. In this case:

$$\begin{array}{ll} \delta_t(\gamma_t(d)) & \\ \quad = \delta_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by def. of } \gamma_t \\ \quad = \delta_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \text{by definition of } \delta_t \\ \quad = \delta_{t_2}(\gamma_{t_2}(d)) & \text{by Prop. 2(b)} \\ \quad = d_{t_2} & \text{by i.h.} \\ \quad = d_t & \text{by def. of } b_t \end{array}$$

- $b = n$. Now we have

$$\begin{array}{ll} \delta_t(\gamma_t(n)) = \delta_t(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ \qquad\qquad\quad = \delta_{t_2}(\gamma_{t_2}(n)) & \text{by definition of } \delta_t \\ \qquad\qquad\quad = n_{t_2} & \text{by i.h.} \\ \qquad\qquad\quad = n_t & \text{by definition of } b_t \end{array}$$

– $t = \forall\beta.t'$. In this case $b_t = b_{t'}$, $\delta_t = \delta'_t$ and $\gamma_t = \gamma_{t'}$, so it trivially holds by i.h..

□

The following proposition relates the rising function ∇_t with the abstraction and concretisation functions α_t and γ_t. The basic idea is that once we have gone up in the domain with ∇_t, the functions α_t and γ_t behave in the same way in the subdomain of D_{2t} that is the range of ∇_t, than in the whole domain. This means that in fact α_t and γ_t are moving inside this subdomain. In Figure 19 this fact is represented by two D_{2t} domains encircled by a dashed line, where α_t and γ_t appear between the range of ∇_t and *Basic*.

Proposition 15 *For each type t:*

(a) $\alpha_t \cdot \nabla_t = \alpha_t$
(b) $\nabla_t \cdot \gamma_t = \gamma_t$

Proof. (Proposition 15) Item 15(a) can be proved by s.i. on t and by Proposition 2(b). Item 15(b) can be proved also by s.i. on t and propositions 2(b) and 3.

We first prove item 15(a):

– $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \alpha_t = id_{Basic}$.
– $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ where $i = 1, \ldots, m$. Then:

$$\begin{array}{ll} \alpha_t(\nabla_t(e_1, \ldots, e_m)) & \\ \quad = \alpha_t(\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by def. of } \nabla_t \\ \quad = \bigsqcup_i \alpha_{t_i}(\gamma_{t_i}(\alpha_{t_i}(e_i))) & \text{by def. of } \alpha_t \\ \quad = \bigsqcup_i \alpha_{t_i}(e_i) & \text{by Prop. 2(b)} \\ \quad = \alpha_t(e_1, \ldots, e_m) & \text{by def. of } \alpha_t \end{array}$$

– $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in [D_{2\,t_1} \to D_{2\,t_2}]$. Then:

$$\begin{aligned}
&\alpha_t(\nabla_t(f)) \\
&\quad = \alpha_{t_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) && \text{by def. of } \alpha_t \text{ and } \nabla_t \\
&\quad = \alpha_{t_2}(f(\gamma_{t_1}(d))) && \text{by i.h.} \\
&\quad = \alpha_t(f) && \text{by definition of } \alpha_t
\end{aligned}$$

– $t = \forall\beta.t'$. It trivially holds by i.h. as $\nabla_t = \nabla_{t'}$ and $\alpha_t = \alpha_{t'}$.

Now, item 15(b) is proved:

– $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \gamma_t = id_{\mathit{Basic}}$.
– $t = (t_1, \ldots, t_m)$. Let $b \in \mathit{Basic}$. In this case:

$$\begin{aligned}
&\nabla_t(\gamma_t(b)) \\
&\quad = \nabla_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) \\
&\qquad\quad \{\text{by definition of } \gamma_t\} \\
&\quad = (\gamma_{t_1}(\alpha_{t_1}(\gamma_{t_1}(b))), \ldots, \gamma_{t_m}(\alpha_{t_m}(\gamma_{t_m}(b)))) \\
&\qquad\quad \{\text{by definition of } \nabla_t\} \\
&\quad = (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) \\
&\qquad\quad \{\text{by Proposition 2(b)}\} \\
&\quad = \gamma_t(b) \\
&\qquad\quad \{\text{by definition of } \gamma_t\}
\end{aligned}$$

– $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $b \in \mathit{Basic}$. We distinguish two cases: $b = d$ or $b = n$. Let us see first the case when $b = d$. On one side we have that:

$$\begin{aligned}
&\nabla_t(\gamma_t(d)) \\
&\quad = \nabla_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) \\
&\qquad\quad \{\text{by definition of } \gamma\} \\
&\quad = \lambda z. \begin{cases} \nabla_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \text{if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) & \text{otherwise} \end{cases} \\
&\qquad\quad \{\text{by definition of } \nabla_t\} \\
&\quad = \lambda z. \begin{cases} \gamma_{t_2}(d) & \text{if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) & \text{otherwise} \end{cases} \\
&\qquad\quad \{\text{by i.h. and Proposition 2(b)}\}
\end{aligned}$$

On the other side we have that $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. We have to prove that for each $e \in D_{2\,t_1}$, $(\nabla_t(\gamma_t(d)))(e) = (\gamma_t(d))(e)$. We distinguish two cases:

- $e \sqsubseteq \gamma_{t_1}(d)$. In this case

$$(\nabla_t(\gamma_t(d)))(e) = \gamma_{t_2}(d)$$

  By Lemma 3, if $e \sqsubseteq \gamma_{t_1}(d)$ then $\alpha_{t_1}(e) = d$, so

$$(\gamma_t(d))(e) = \gamma_{t_2}(d)$$

- Otherwise, by Lemma 3, $\alpha_{t_1}(e) = n$ and we have:

$$\begin{aligned}
(\nabla_t(\gamma_t(d)))(e) &= \gamma_{t_2}(n) \\
&= \gamma_{t_2}(\alpha_{t_1}(e)) \\
&= (\gamma_t(d))(e)
\end{aligned}$$

$$
\begin{array}{l}
\nabla_t : D_{2t} \to D_{2t} \\
\nabla_K = \nabla_{T\ t_1 \ldots t_m} = \nabla_\beta = id_{Basic} \\
\nabla_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \\
\qquad (\gamma_{t_1}(\alpha_{t_1}(e_1)),\ldots,\gamma_{t_m}(\alpha_{t_m}(e_m))) \\
\nabla_{t_1 \to t_2}(f) = \nabla_{Process\ t_1\ t_2}(f) = \\
\qquad \lambda z \in D_{2t_1}. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases} \\
\nabla_{\forall \beta.t} = \nabla_t
\end{array}
$$

Fig. 21. The unfolding of the rising function ∇_t

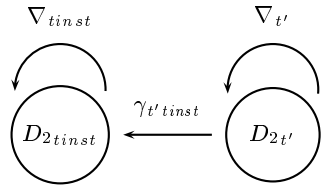

Fig. 22. Proposition 16(a)

If $b = n$:

$$
\begin{array}{ll}
\nabla_t(\gamma_t(n)) & \\
\quad = \nabla_t(\lambda z.\gamma_{t_2}(n)) & \text{by def. of } \gamma_t \\
\quad = \lambda z. \begin{cases} \nabla_{t_2}(\gamma_{t_2}(n)) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases} & \text{by def. of } \nabla_t \\
\quad = \lambda z.\gamma_{t_2}(n) & \text{by i.h.} \\
\quad = \gamma_t(n) & \text{by def. of } \gamma_t
\end{array}
$$

- $t = \forall\beta.t'$. It trivially holds by i.h. as $\nabla_t = \nabla_{t'}$ and $\gamma_t = \gamma_{t'}$.

□

The following proposition adds some results about the polymorphic functions. The first one tells us that we obtain the same result if we adapt an abstract value belonging to $D_{2t'}$ to the type *tinst* and then rise the result than if we first rise it and then adapt the result. In Figure 22 a diagram is shown. The second one tells us that the adaptation of an abstract value in $D_{1t'}$ to obtain an aproximation to the abstract value of an instance *tinst*, is basically equal to the adaptation made with $\gamma_{t'tinst}$ in the domains of the second and third analyses.

Proposition 16 *Given two types t, t' and a type variable β:*

(a) $\nabla_{tinst} \cdot \gamma_{t'tinst} = \gamma_{t'tinst} \cdot \nabla_{t'}$

(b) $\forall a \in D_{1t'}.\ a_{tinst} = \delta_{tinst}(\gamma_{t'tinst}(\eta_{t'}(a)))$

Proof. (Proposition 16) Item 16(a) can be proved by s.i. on t and propositions 2(a), 2(b), 3, 5(a), 6, and 15(b). Item 16(b) can be proved by s.i. on t and propositions 2(b), 6(d), 6(a) and 14.

First we prove item 16(a):

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'$ ($\neq \beta$). In these cases $D_{2tinst} = D_{2t'} = Basic$ and $\nabla_{tinst} = \gamma_{t'tinst} = \nabla_{t'} = id_{Basic}$, so it trivially holds.
- $t' = (t_1,\ldots,t_m)$. In this case $tinst = (tinst_1,\ldots,tinst_m)$ where $tinst_i = t_i[\beta := t]$, for $i = 1,\ldots,m$. By i.h. we have that for each $i = 1,\ldots,m$, $\nabla_{tinst_i} \cdot \gamma_{t_i tinst_i} = \gamma_{t_i tinst_i} \cdot \nabla_{t_i}$. Let $e_i \in D_{2t_i}$ where

$i = 1, \ldots, m$. Then:

$$
\begin{aligned}
&\nabla_{tinst}(\gamma_{t'\,tinst}(e_1, \ldots, e_m)) \\
&\quad = \nabla_{tinst}(\gamma_{t_1\,tinst_1}(e_1), \ldots, \gamma_{t_m\,tinst_m}(e_m)) \\
&\qquad \{\text{by definition of } \gamma_{t'\,tinst}\} \\
&\quad = (\gamma_{tinst_1}(\alpha_{tinst_1}(\gamma_{t_1\,tinst_1}(e_1))), \ldots, \\
&\qquad\qquad \gamma_{tinst_m}(\alpha_{tinst_m}(\gamma_{t_m\,tinst_m}(e_m)))) \\
&\qquad \{\text{by definition of } \nabla_t\} \\
&\quad = (\gamma_{tinst_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{tinst_m}(\alpha_{t_m}(e_m))) \\
&\qquad \{\text{by Proposition 6(c)}\} \\
&\quad = (\gamma_{t_1\,tinst_1}(\gamma_{t_1}(\alpha_{t_1}(e_1))), \ldots, \gamma_{t_m\,tinst_m}(\gamma_{t_m}(\alpha_{t_m}(e_m)))) \\
&\qquad \{\text{by Proposition 6(a)}\} \\
&\quad = \gamma_{t'\,tinst}(\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_1))) \\
&\qquad \{\text{by definition of } \gamma_{t'\,tinst}\} \\
&\quad = \gamma_{t'\,tinst}(\nabla_{t'}(e_1, \ldots, e_m)) \\
&\qquad \{\text{by definition of } \nabla_t\}
\end{aligned}
$$

- $t' = t_1 \to t_2$ or $t' = \mathit{Process}\ t_1\ t_2$. In this case $tinst = tinst_1 \to tinst_2$, respectively $tinst = \mathit{Process}\ tinst_1\ tinst_2$, where $tinst_i = t_i[\beta := t]$, for $i = 1, 2$. Let $f \in [D_{2\,t_1} \to D_{2\,t_2}]$. On one side we have:

$$
\begin{aligned}
&\nabla_{tinst}(\gamma_{t'\,tinst}(f)) \\
&\quad = \nabla_{tinst}(\lambda z.\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(z)))) \\
&\qquad \{\text{by definition of } \gamma_{t'\,tinst}\} \\
&\quad = \lambda z. \begin{cases} \nabla_{tinst_2}(\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))))) \\ \qquad\qquad\qquad\qquad \text{if } z \sqsubseteq \gamma_{tinst_1}(d) \\ \gamma_{tinst_2}(n) \text{ otherwise} \end{cases} \\
&\qquad \{\text{by definition of } \nabla_t\}
\end{aligned}
$$

On the other side:

$$
\begin{aligned}
&\gamma_{t'\,tinst}(\nabla_{t'}(f)) \\
&\quad = \gamma_{t'\,tinst}\left(\lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}\right) \\
&\qquad \{\text{by definition of } \nabla_t\} \\
&\quad = \lambda z. \begin{cases} \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) \\ \qquad\qquad\qquad\qquad \text{if } \alpha_{tinst_1\,t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) \text{ otherwise} \end{cases} \\
&\qquad \{\text{by definition of } \gamma_{t'\,tinst}\}
\end{aligned}
$$

We want to prove the equality of these two functions. Let $e \in D_{2\,tinst_1}$. We distinguish two cases::

- $e \sqsubseteq \gamma_{tinst_1}(d)$. Then by monotonicity of $\alpha_{tinst\,t'}$ (Proposition 5(a)), $\alpha_{tinst_1\,t_1}(e) \sqsubseteq \alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))$ and by Proposition 6(d) $\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d)) = \gamma_{t_1}(d)$, so

$$\alpha_{tinst_1\,t_1}(e) \sqsubseteq \gamma_{t_1}(d)$$

  So, in this case:

$$
\begin{aligned}
&(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e) \\
&\quad = \nabla_{tinst_2}(\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))))))
\end{aligned}
$$

  and

$$(\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d))))$$

So:

$$
\begin{aligned}
&(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e)\\
&\quad = \nabla_{tinst_2}(\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))))\\
&\quad = \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d)))))\\
&\qquad\quad \{\text{by i.h.}\}\\
&\quad = \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))))\\
&\qquad\quad \{\text{by Proposition 6(d)}\}\\
&\quad = \gamma_{t'\,tinst}(\nabla_{t'}(f)))(e)
\end{aligned}
$$

- $e \not\sqsubseteq \gamma_{tinst_1}(d)$. In this case by Lemma 3 $\alpha_{tinst_1}(e) = n$. Additionally, $(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e) = \gamma_{tinst_2}(n)$. And also we have that $\alpha_{tinst_1\,t_1}(e) \not\sqsubseteq \gamma_{t_1}(d)$. This is true because otherwise by Proposition 6(b) $\alpha_{tinst_1}(e) = \alpha_{t_1}(\alpha_{tinst_1\,t_1}(e))$, we would have that $\alpha_{tinst_1}(e) \sqsubseteq \alpha_{t_1}(\gamma_{t_1}(d))$ by monotonicity of α_t (Proposition 2(a)), what would mean that $\alpha_{tinst_1}(e) = d$ (by Proposition 2(b)), which is false as we have just seen that it is equal to n. So $(\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n))$. And:

$$
\begin{aligned}
&(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e)\\
&\quad = \gamma_{tinst_2}(n)\\
&\quad = \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) \qquad \text{by Proposition 6(a)}\\
&\quad = (\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e)
\end{aligned}
$$

– $t' = \beta$. In this case:

$$
\begin{aligned}
&\nabla_{tinst} \cdot \gamma_{t'\,tinst}\\
&\quad = \nabla_t \cdot \gamma_t && \text{by definition of } \nabla_t \text{ and } \gamma_t\\
&\quad = \gamma_t && \text{by Proposition 15(b)}\\
&\quad = \gamma_{t'\,tinst} \cdot \nabla_{t'} && \text{by definition of } \gamma_{t'\,tinst} \text{ and } \nabla_t
\end{aligned}
$$

– $t' = \forall\beta'.t_1$. It holds directly by i.h., as $\nabla_{tinst} = \nabla_{tinst_1}$, $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$ and $\nabla_{t'} = \nabla_{t_1}$.

We now prove item 16(b):

– $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'$ $(\neq \beta)$. In this case $D_{1\,tinst} = D_{1\,t'} = Basic$. Let $b \in Basic$. Then $b_{tinst} = b$. On the other side $\delta_{tinst} = \gamma_{t'\,tinst} = \eta_{t'} = id_{Basic}$, so it holds trivially.
– $t' = (t_1, \ldots, t_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$ where $tinst_i = t_i[\beta := t]$, for $i = 1, \ldots, m$. Let $b_i \in Basic$. Then

$$
\begin{aligned}
&\delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(b_1, \ldots, b_m)))\\
&= \delta_{tinst}(\gamma_{t'\,tinst}(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)))\\
&\qquad \{\text{by definition of } \eta_t\}\\
&= \delta_{tinst}(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b_1)), \ldots, \gamma_{t_1\,tinst_1}(\gamma_{t_m}(b_m)))\\
&\qquad \{\text{by definition of } \gamma_{t'\,tinst}\}\\
&= \delta_{tinst}(\gamma_{tinst_1}(b_1), \ldots, \gamma_{tinst_m}(b_m))\\
&\qquad \{\text{by Proposition 6(a)}\}\\
&= (\alpha_{tinst_1}(\gamma_{tinst_1}(b_1)), \ldots, \alpha_{tinst_m}(\gamma_{tinst_m}(b_m)))\\
&\qquad \{\text{by definition of } \delta_t\}\\
&= (b_1, \ldots, b_m)\\
&\qquad \{\text{by Proposition 2(b)}\}\\
&= (b_1, \ldots, b_m)_{tinst}\\
&\qquad \{\text{by def. of the adaptation}\}
\end{aligned}
$$

- $t' = t_1 \to t_2$ or $t' = \mathit{Process}\ t_1\ t_2$. In this case $\mathit{tinst} = \mathit{tinst}_1 \to \mathit{tinst}_2$, respectively $\mathit{tinst} = \mathit{Process}\ \mathit{tinst}_1\ \mathit{tinst}_2$, where $\mathit{tinst}_i = t_i[\beta := t]$, for $i = 1, 2$. Let $a \in D_{1\,t'} = D_{1\,t_2}$. Now we have:

$$\begin{aligned}
&\delta_{\mathit{tinst}}(\gamma_{t'\,\mathit{tinst}}(\eta_{t'}(a))) \\
&\quad = \delta_{\mathit{tinst}}\left(\gamma_{t'\,\mathit{tinst}}\left(\lambda z. \begin{cases} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}\right)\right) \\
&\qquad \{\text{by definition of } \eta_t\} \\
&\quad = \delta_{\mathit{tinst}}\left(\lambda z. \begin{cases} \gamma_{t_2\,\mathit{tinst}_2}(\eta_{t_2}(a)) \\ \qquad\qquad \text{if } \alpha_{\mathit{tinst}_1\,t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2\,\mathit{tinst}_2}(\gamma_{t_2}(n)) \text{ otherwise} \end{cases}\right) \\
&\qquad \{\text{by definition of } \gamma_{t'\,\mathit{tinst}}\} \\
&\quad = \delta_{\mathit{tinst}_2}(\gamma_{t_2\,\mathit{tinst}_2}(\eta_{t_2}(a))) \\
&\qquad \{\text{by def. of } \delta_t \text{ and Proposition 6(d)}\} \\
&\quad = a_{\mathit{tinst}_2} \\
&\qquad \{\text{by i.h.}\} \\
&\quad = a_{\mathit{tinst}} \\
&\qquad \{\text{by definition of adaptation}\}
\end{aligned}$$

- $t' = \beta$. Now $D_{1\,t'} = \mathit{Basic}$ and $D_{1\,\mathit{tinst}} = D_{1\,t}$. Let $b \in \mathit{Basic}$. Then:

$$\begin{aligned}
&\delta_{\mathit{tinst}}(\gamma_{t'\,\mathit{tinst}}(\eta_{t'}(b))) \\
&\quad = \delta_t(\gamma_t(b)) && \text{by def. of } \delta_t,\ \gamma_{t'\,\mathit{tinst}} \text{ and } \eta_t \\
&\quad = b_t && \text{by Proposition 14} \\
&\quad = b_{\mathit{tinst}}
\end{aligned}$$

- $t' = \forall \beta'.t_1$. It directly holds by i.h., as $\delta_{\mathit{tinst}} = \delta_{\mathit{tinst}_1}$, $\gamma_{t'\,\mathit{tinst}} = \gamma_{t_1\,\mathit{tinst}_1}$, $\eta_{\mathit{tinst}} = \eta_{\mathit{tinst}_1}$ and $b_{\mathit{tinst}} = b_{\mathit{tinst}_1}$.

□

A very important and useful property to prove the correctness is the semihomomorphic property of δ_t with respect to the application of a function. But as there are no functional domains in the first analysis, the property holds with respect to the pseudoapplication we use in such domains, that is, the way in which application of a function is interpreted: $f(x) = (\widehat{\sqcup} x) \sqcup f$ (see Figure 3).

Proposition 17 *Let $f \in [D_{2\,t_1} \to D_{2\,t_2}]$, $e \in D_{2\,t_1}$. Then: $\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f)$*

Proof. (Proposition 17) This proposition can be proved by propositions 3, 10(a) and 11. Let $f \in [D_{2\,t_1} \to D_{2\,t_2}]$, $e \in D_{2\,t_1}$. By definition $\delta_{t_1 \to t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d)))$, so we have to prove that $\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d)))$. We distinguish two cases::

- $e \sqsubseteq \gamma_{t_1}(d)$. As f is monotone, $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. By Proposition 10(a), δ_{t_2} is monotone, so

$$\delta_{t_2}(f(e)) \sqsubseteq \delta_{t_2}(f(\gamma_{t_1}(d)))$$

  which trivially is less than or equal to

$$(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d)))$$

- $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case, by Lemma 3 $\alpha_{t_1}(e) = n$, so by Proposition 11, $\widehat{\sqcup}(\delta_{t_1}(e)) = n$.
  It is obvious that $b \in \mathit{Basic}$, $n \sqcup b \sqsupseteq b'$ for any $b' \in \mathit{Basic}$. And if $a \in \mathit{Basic}^m$, then $n \sqcup a \sqsupseteq a'$, for any $a' \in \mathit{Basic}^m$. That is, the lub of any value and n takes us to the top of the corresponding domain.
  So

$$(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f) = n \sqcup \delta_{t_1 \to t_2}(f) \sqsupseteq \delta_{t_2}(f(e))$$

□

To prove the corollary of Theorem 19 we will need the following lemma.

Lemma 18 *For each type t: $\eta_t(d_t) = \gamma_t(d)$*

Proof. (Lemma 18) This is a direct consequence of propositions 14 and 15(b). By Proposition 14 we know that $b_t = \delta_t(\gamma_t(b))$. So:

$$\begin{aligned}\eta_t(b_t) &= \eta_t(\delta_t(\gamma_t(b))) \\ &= \nabla_t(\gamma_t(b)) && \text{by definition of } \nabla_t \\ &= \gamma_t(b) && \text{by Proposition 15(b)}\end{aligned}$$

□

The following theorem establishes that the first analysis is a safe (upper) approximation to the transformation by δ_t of those variants of the third analysis $[\![\cdot]\!]_3^{wop}$ such that $\delta_t \cdot wop_t = \delta_t$

Theorem 19 *Let $\mathcal{W}'_t$ be a widening operator for each type t, such that $\delta_t \cdot \mathcal{W}'_t = \delta_t$. If for each variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$ then: $\forall e :: t_e.\ [\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$. Or equivalently, by Proposition 10: $\forall e :: t_e.\ \eta_{t_e}([\![e]\!]_1\ \rho_1) \sqsupseteq \nabla_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$*

Proof. (Theorem 19) The theorem can be proved by structural induction on e and by propositions 2(a), 2(c), 5(a), 10(a), 11, 12, 14, 16 and 17:

- $v :: t$. In this case:

$$\begin{aligned}[\![v]\!]\ \rho_1 &= \rho_1(v) && \text{by definition of } [\![\cdot]\!]_1 \\ &\sqsupseteq \delta_t(\rho_3(v)) && \text{by hypothesis} \\ &= \delta_t([\![v]\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'}\end{aligned}$$

- $k :: K$.

$$\begin{aligned}[\![k]\!]_1\ \rho_1 &= d && \text{by definition of } [\![\cdot]\!]_1 \\ &= [\![k]\!]_3^{\mathcal{W}'}\ \rho_3 && \text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'} \\ &= \delta_K([\![k]\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by definition of } \delta_t\end{aligned}$$

- $C\ x_1 \ldots x_m :: T\ t'_1 \ldots t'_k$, where $x_i :: t_i$. By i.h. we know that for each $i = 1, \ldots, m$, the following holds:

$$[\![x_i]\!]_1\ \rho_1 \sqsupseteq \delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)$$

  As the operator $\widehat{\sqcup}$ is monotone, this means that for each $i = 1, \ldots, m$ the following holds

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3))$$

  By Proposition 11 we have that $\widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)) = \alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)$, so

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)\ (1)$$

  Then:

$$\begin{aligned}&[\![C\ x_1 \ldots x_m]\!]_1\ \rho_1 \\ &= \bigsqcup_i(\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1)) && \text{by definition of } [\![\cdot]\!]_1 \\ &\sqsupseteq \bigsqcup_i\ (\alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)) && \text{by (1)} \\ &= [\![C\ x_1 \ldots x_m]\!]_3^{\mathcal{W}'}\ \rho_3 && \text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'}\end{aligned}$$

– $(x_1, \ldots, x_m) :: (t_1, \ldots, t_m)$. Let us call t to $(t_1, \ldots, t_m)$. In this case we have on one side that:

$$\begin{aligned}
&[\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 \\
&\quad = (\widehat{\sqcup}([\![x_1]\!]_1\ \rho_1), \ldots, \widehat{\sqcup}([\![x_m]\!]_1\ \rho_1)) \\
&\qquad \{\text{by definition of } [\![\cdot]\!]_1\} \\
&\quad \sqsupseteq (\widehat{\sqcup}(\delta_{t_1}([\![x_1]\!]_3^{\mathcal{W}'}\ \rho_3)), \ldots, \widehat{\sqcup}(\delta_{t_m}([\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
&\qquad \{\text{by i.h. and monotonicity of } \widehat{\sqcup}\} \\
&\quad = (\alpha_{t_1}([\![x_1]\!]_3\ \rho_3), \ldots, \alpha_{t_m}([\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3) \\
&\qquad \{\text{by Proposition 11}\}
\end{aligned}$$

and on the other side:

$$\begin{aligned}
&\delta_t([\![(x_1, \ldots, x_m)]\!]_3^{\mathcal{W}'}\ \rho_3) \\
&\quad = \delta_t([\![x_1]\!]_3^{\mathcal{W}'}\ \rho_3, \ldots, [\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3) \\
&\qquad \{\text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'}\} \\
&\quad = (\alpha_{t_1}([\![x_1]\!]_3^{\mathcal{W}'}\ \rho_3), \ldots, \alpha_{t_m}([\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
&\qquad \{\text{by definition of } \delta_t\}
\end{aligned}$$

So

$$[\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 \sqsupseteq \delta_t([\![(x_1, \ldots, x_m)]\!]_3^{\mathcal{W}'}\ \rho_3)$$

– $\lambda v.e :: t_1 \to t_2$. The proof is exactly the same if the expression is *process* $v \to e$ :: *Process* $t_1\ t_2$. In this case we have on one side that:

$$[\![\lambda v.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \mapsto d_{t_1}]$$

Let us call ρ_1' to $\rho_1\ [v \mapsto d_{t_1}]$.
On the other side we have that:

$$[\![\lambda v.e]\!]_3^{\mathcal{W}'}\ \rho_3 = \lambda z.[\![e]\!]_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto z]$$

so that:

$$\begin{aligned}
&\delta_t([\![\lambda v.e]\!]_3^{\mathcal{W}'}\ \rho_3) \\
&\quad = \delta_t(\lambda z.[\![e]\!]_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto z]) \\
&\quad = \delta_{t_2}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto \gamma_{t_1}(d)]) \text{ by definition of } \delta_t
\end{aligned}$$

Let us call ρ_3' to $\rho_3\ [v \mapsto \gamma_{t_1}(d)]$.
If we proved that for each variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_3'(y))$, then by i.h. we would have that

$$[\![e]\!]_1\ \rho_1' \sqsupseteq \delta_{t_2}([\![e]\!]_3\ \rho_3')$$

that is what we wanted to prove.
So let us prove this. Let a variable $y :: t_y$.

- If $y \neq v$, it trivially holds by hypothesis over the environments ρ_1 and ρ_3.
- If $y = v$, then we have to prove that $d_{t_1} \sqsupseteq \delta_{t_1}(\gamma_{t_1}(d))$, which holds trivially by Proposition 14.

– $e\ x :: t_2$, where $e :: t_1 \to t_2$ and $x :: t_1$. Let us call t to $t_1 \to t_2$. The proof is similar in case the expression is $v\#x$ where v :: *Process* $t_1\ t_2$. By i.h. we know that

$$[\![x]\!]_1\ \rho_1 \sqsupseteq \delta_{t_1}([\![x]\!]_3^{\mathcal{W}'}\ \rho_3)$$

and

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$$

So we have that:

$$
\begin{aligned}
& [\![e\ x]\!]_1\ \rho_1 \\
& = (\widehat{\sqcup}([\![x]\!]_1\ \rho_1)) \sqcup ([\![e]\!]_1\ \rho_1) \\
& \qquad \{\text{by definition of } [\![\cdot]\!]_1\} \\
& \sqsupseteq (\widehat{\sqcup}(\delta_{t_1}([\![x]\!]_3^{\mathcal{W}'}\ \rho_3))) \sqcup \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) \\
& \qquad \{\text{by i.h. and monotonicity of } \sqcup \text{ and } \widehat{\sqcup}\} \\
& \sqsupseteq \delta_{t_2}(([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)([\![x]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
& \qquad \{\text{by Proposition 17}\} \\
& = \delta_{t_2}([\![e\ x]\!]_3^{\mathcal{W}'}\ \rho_3) \\
& \qquad \{\text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'}\}
\end{aligned}
$$

- **let** $v = e$ **in** $e' :: t$, where v and e have type t_e, and $e' :: t$. In this case, we have on one side that:

$$[\![\textbf{let } v = e \textbf{ in } e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1]$$

Let us call ρ_1' to $\rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1]$.
On the other side we have that

$$[\![\textbf{let } v = e \textbf{ in } e']\!]_3^{\mathcal{W}'}\ \rho_3 = [\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto [\![e]\!]_3^{\mathcal{W}'}\ \rho_3]$$

Let us call ρ_3' to $\rho_3\ [v \mapsto [\![e]\!]_1\ \rho_3]$.
If we proved that for each variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_3'(y))$, then by i.h. we would have that

$$[\![e']\!]_1\ \rho_1' \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3')$$

which is what we want to prove.
Let us then prove this. Let $y :: t_y$.

- If $y \neq v$, it trivially holds by hypothesis over the environments ρ_1 and ρ_3.
- If $y = v$, then we have to prove that

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$$

  which holds by i.h..

- **case** e **of** $(v_1, \ldots, v_m) \to e' :: t$, where $e :: t_e = (t_1, \ldots, t_m)$, $v_i :: t_i$ and $e' :: t$. In this case, we have on one side:

$$
\begin{aligned}
& [\![\textbf{case } e \textbf{ of } (v_1, \ldots, v_m) \to e]\!]_1\ \rho_1 \\
& = [\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]}
\end{aligned}
$$

And on the other side that

$$
\begin{aligned}
& [\![\textbf{case } e \textbf{ of } (v_1, \ldots, v_m) \to e]\!]_3^{\mathcal{W}'}\ \rho_3 \\
& = [\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ \overline{[v_i \mapsto \pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)]}
\end{aligned}
$$

If we proved that for each $i = 1, \ldots, m$ the following holds

$$(\pi_i([\![e]\!]_1\ \rho_1))_{t_i} \sqsupseteq \delta_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))$$

then by i.h. we would have that

$$
\begin{aligned}
& [\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]} \\
& \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ \overline{[v_i \mapsto \pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)]})
\end{aligned}
$$

which is what we want to prove.

So let us prove this. By i.h. we know that the following holds:

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$$

This can be rewritten in the following way:

$$\begin{array}{l}
(\pi_1([\![e]\!]_1\ \rho_1), \ldots, \pi_m([\![e]\!]_1\ \rho_1)) \\
\quad = [\![e]\!]_1\ \rho_1 \\
\qquad \{\text{as it is of tuple type}\} \\
\quad \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) \\
\qquad \{\text{by i.h.}\} \\
\quad = \delta_{t_e}(\pi_1([\![e]\!]_3^{\mathcal{W}'}\ \rho_3), \ldots, \pi_m([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
\qquad \{\text{as it is of tuple type}\} \\
\quad = (\alpha_{t_1}(\pi_1([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)), \ldots, \alpha_{t_m}(\pi_m([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
\qquad \{\text{by definition of } \delta_t\}
\end{array}$$

That is, for each $i = 1, \ldots, m$ the following holds:

$$\pi_i([\![e]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))\ (2)$$

So:

$$\begin{array}{l}
(\pi_i([\![e]\!]_1\ \rho_1))_{t_i} \\
= \delta_{t_i}(\gamma_{t_i}(\pi_i([\![e]\!]_1\ \rho_1))) \\
\qquad \{\text{by Proposition 14}\} \\
\sqsupseteq \delta_{t_i}(\gamma_{t_i}(\alpha_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)))) \\
\qquad \{\text{by (2) and monotonicity of } \delta_t \text{ and } \gamma_t\} \\
\sqsupseteq \delta_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
\qquad \{\text{by Proposition 2(c) and monotonicity of } \delta_t\}
\end{array}$$

so we have what we wanted to prove.

- **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i} :: t$, where $e :: t_e = T\ t'_1 \ldots t'_k$, $v_{ij} :: t_{ij}$ and $e_i :: t$. Let us call e' to **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i}$. In this case we have on one side that:

$$[\![e']\!]_1\ \rho_1 = \begin{cases} n_t \text{ if } [\![e]\!]_1\ \rho_1 = n \\ \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} \text{ otherwise} \end{cases}$$
$$\text{where } \rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto d_{t_{ij}}]}, v_{ij} :: t_{ij}, e_i :: t$$

and on the other side:

$$[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = n \\ \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i} \text{ otherwise} \end{cases}$$

$$\text{where } \rho_{3i} = \rho_3\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t$$

By i.h. we know that

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$$

that is, knowing that $\delta_{T\ t'_1,\ldots,t'_k} = id_{Basic}$:

$$[\![e]\!]_1\ \rho_1 \sqsupseteq [\![e]\!]_3^{\mathcal{W}'}\ \rho_3$$

We distinguish three cases:

- $[\![e]\!]_1\ \rho_1 = [\![e]\!]_3\ \rho_3 = n$. In this case:

$$\begin{aligned} [\![e']\!]_1\ \rho_1 &= n_t && \text{by definition of } [\![\cdot]\!]_1 \\ &= \delta_t(\gamma_t(n)) && \text{by Proposition 14} \\ &= \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'} \end{aligned}$$

- $[\![e]\!]_1\ \rho_1 = [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = d$. In this case:

$$[\![e']\!]_1\ \rho_1 = \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i}$$

where $\rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto d_{t_{ij}}]}$ and

$$[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}$$

where $\rho_{3i} = \rho_3\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}$.
If we proved for each $i = 1, \ldots, m$, that for each variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{3i}(y))$, then by i.h. we would have that for each i the following holds

$$[\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \delta_t([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i})$$

so, by continuity of δ_t:

$$\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \bigsqcup_i \delta_t([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}) = \delta_t(\bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i})$$

which is what we wanted to prove.
So, let us prove now that for each variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{3i}(y))$. If y is not any of the v_{ij}, this is trivially true by hypothesis on ρ_1 and ρ_3. If y is one of the v_{ij}, then it is also true by Proposition 14.

- $[\![e]\!]_1\ \rho_1 = n \sqsupset [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = d$. In this case $[\![e']\!]_1\ \rho_1 = n_t$, and $[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}$. Clearly, $n_t \sqsupseteq \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}$, as n_t is the top of D_{1t}.

Primitive *case* is very similar.

– **let rec** $\overline{\{v_i = e_i\}}$ **in** $e' :: t$, where $e' :: t$, and each v_i and e_i have type t_i. Let us call e to **let rec** $\overline{\{v_i = e_i\}}$ **in** e'. In this case:

$$[\![e]\!]_1\ \rho_1 = [\![e']\!]\ (\bigsqcup_{n \in \mathbb{N}} (\lambda \rho_1'.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho_1']})^n(\rho_{01}))$$

where ρ_{01} is the initial environment where each variable $y :: t_y$ has d_{t_y} as abstract value (that is, the infimum of the corresponding domain). Let us call F to the function between environments $\lambda \rho_1'.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho_1']}$. Let us call ρ_1^{fix} to $\bigsqcup_{n \in \mathbb{N}} F^n(\rho_{01})$.
On the other side:

$$\begin{aligned} &[\![e]\!]_3\ \rho_3 = \\ &\quad [\![e']\!]_3^{\mathcal{W}'}\ (\textstyle\bigsqcup_{n \in \mathbb{N}} (\lambda \rho_3'.\rho_3\ \overline{[v_i \mapsto \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_3')]})^n(\rho_{03})) \end{aligned}$$

where ρ_{03} is the initial environment where each variable $y :: t_y$ has $-_{2t_y}$ as abstract value (that is, the infimum of the corresponding domain). Let us call G to the function between environments $\lambda \rho_3'.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_3')]}$. Let us call ρ_3^{fix} to $\bigsqcup_{n \in \mathbb{N}} G^n(\rho_{03})$.
If we proved that for each variable $y :: t_y$, $\rho_1^{fix}(y) \sqsupseteq \delta_{t_y}(\rho_3^{fix}(y))$, then by i.h. we would have that

$$[\![e']\!]_1\ \rho_1^{fix} \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3^{fix})$$

which is what we wanted to prove.

Let us see that for each $n \geq 0$, the following holds

$$\forall y :: t_y.(F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^n(\rho_{03}))(y))$$

If this were true then

$$\forall y :: t_y.(\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{03}))(y))$$

by continuity of δ_t, and this is what we want to prove. It can be proved by induction on n:

- $n = 0$. This is a trivial case as $F^0(\rho_{01}) = \rho_{01}$, $G^0(\rho_{03}) = \rho_{03}$ and $d_t = \delta_t(-_{2t})$, by Proposition 12.
- $n = m + 1$. Then

$$\begin{aligned} & F^{m+1}(\rho_{01}) \\ & \quad = F(F^m(\rho_{01})) \\ & \quad = \rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ (F^m(\rho_{01}))]} \text{ by definition of } F \end{aligned}$$

and

$$\begin{aligned} & G^{m+1}(\rho_{03}) \\ & \quad = G(G^m(\rho_{03})) \\ & \quad = \rho_3\ \overline{[v_i \mapsto \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ (G^m(\rho_{03})))]} \\ & \qquad \{\text{by definition of } G\} \end{aligned}$$

Let $y :: t_y$. We want to prove that $(F^{m+1}(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^{m+1}(\rho_{03}))(y))$. We distinguish two cases. If y is not any of the v_i, then it holds by the hypothesis over the environments ρ_1 and ρ_3. If it is one of the v_i, then we have to prove that

$$[\![e_i]\!]_1\ (F^m(\rho_{01})) \sqsupseteq \delta_{t_i}(\mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ (G^m(\rho_{03})))) \quad (3)$$

By i.h. (internal on n)

$$\forall y :: t_y.\ (F^m(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^m(\rho_{03}))(y))$$

we obtain by i.h. (external on e) that

$$[\![e_i]\!]_1\ (F^m(\rho_{01})) \sqsupseteq \delta_{t_i}([\![e_i]\!]_3\ (G^m(\rho_{03})))$$

As $\delta_t \cdot \mathcal{W}'_t = \delta_t$ by hypothesis, we obtain (3).

– $\Lambda\beta.e :: \forall\beta.t$, where $e :: t$. In this case:

$$\begin{aligned} & [\![\Lambda\beta.e]\!]_1\ \rho_1 \\ & \quad = [\![e]\!]_1\ \rho_1 && \text{by definition of } [\![\cdot]\!]_1 \\ & \quad \sqsupseteq \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by i.h.} \\ & \quad = \delta_{\forall\beta.t}([\![\Lambda\beta.e]\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by def. of } [\![\cdot]\!]_3^{\mathcal{W}'} \text{ and } \delta_t \end{aligned}$$

– $e\ t :: tinst$ where $e :: \forall\beta.t'$ and $tinst = t'[\beta := t]$. In this case

$$
\begin{aligned}
&[\![e\ t]\!]_1\ \rho_1 \\
&= ([\![e]\!]_1\ \rho_1)_{tinst} \\
&\qquad \{\text{by definition of } [\![\cdot]\!]_1\} \\
&= \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}([\![e]\!]_1\ \rho_1))) \\
&\qquad \{\text{by Proposition 16(b)}\} \\
&\sqsupseteq \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(\delta_{t'}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)))) \\
&\qquad \{\text{by i.h. and monotonicity of } \delta_t,\ \gamma_{t'\,tinst} \text{ and } \eta_t\} \\
&= \delta_{tinst}(\gamma_{t'\,tinst}(\nabla_{t'}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
&\qquad \{\text{by definition of } \nabla_t\} \\
&= \delta_{tinst}(\nabla_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
&\qquad \{\text{by Proposition 16(a)}\} \\
&= \delta_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
&\qquad \{\text{by definition of } \nabla_t \text{ and Proposition 10(b)}\} \\
&= \delta_{tinst}([\![e\ t]\!]_3^{\mathcal{W}'}\ \rho_3) \\
&\qquad \{\text{by definition of } [\![\cdot]\!]_3^{\mathcal{W}'}\}
\end{aligned}
$$

□

As a corollary, we obtain the correctness of the first analysis with respect to these variants: If the first analysis tells us that an expression is deterministic then the corresponding variant of the third analysis also tell us that it is deterministic, probably with some additional detail as the independence of the output with respect to the input in a function/process.

Corollary 20 *Let $\mathcal{W}'_t$ be a widening operator for each type t, such that $\delta_t \cdot \mathcal{W}'_t = \delta_t$. If for each variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$ then: $\forall e :: t_e.\ [\![e]\!]_1\ \rho_1 = d_{t_e} \Rightarrow [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 \sqsubseteq \gamma_{t_e}(d)$*

Proof. (Corollary 20) The corollary is proved by Theorem 19, Lemma 18 and Proposition 10(c).

Let us assume that for each variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$. Let $e :: t_e$ such that $[\![e]\!]_1\ \rho_1 = d_{t_e}$. Then:

$$
\begin{aligned}
[\![e]\!]_3^{\mathcal{W}'}\ \rho_3 &\sqsubseteq \nabla_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) && \text{by Proposition 10(c)} \\
&\sqsubseteq \eta_{t_e}([\![e]\!]_1\ \rho_1) && \text{by Theorem 19} \\
&= \eta_{t_e}(d_{t_e}) && \text{by hypothesis} \\
&= \gamma_{t_e}(d) && \text{by Lemma 18}
\end{aligned}
$$

□

Proposition 22 asserts that the particular widening operator defined in Figure 15 has this property (the widening operator $\mathcal{W}_b$ previously mentioned also fulfills it). To prove it, we need the following one:

Proposition 21 *For each type t, $\alpha_t \cdot \mathcal{W}_t = \alpha_t$.*

Proof. (Proposition 21) This can be proved by s.i. on t.

– $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\alpha_t = \mathcal{W}_t = id_{Basic}$.
– $t = (t_1, \ldots, t_m)$. Let $(e_1, \ldots, e_m) \in D_{2t}$. In this case:

$$
\begin{aligned}
&\alpha_t(\mathcal{W}_t(e_1, \ldots, e_m)) \\
&\quad = \bigsqcup_i \alpha_{t_i}(\mathcal{W}_{t_i}(e_i)) && \text{by def. of } \alpha_t \text{ and } \mathcal{W}_t \\
&\quad = \bigsqcup_i \alpha_{t_i}(e_i) && \text{by i.h.} \\
&\quad = \alpha_t(e_1, \ldots, e_m) && \text{by def. of } \alpha_t
\end{aligned}
$$

- $t = t_1 \to t_2'$ or $t = \mathit{Process}\ t_1\ t_2'$. Let $f \in D_{2t}$. Let $m = \mathit{nArgs}(t)$, $[t_1, \ldots, t_m] = \mathit{aTypes}(t)$ and $t_r = \mathit{rType}(t)$. Then

$$\begin{array}{ll} \alpha_t(\mathcal{W}_t(f)) & \\ \quad = \alpha_{t_r}(\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d))) & \text{by def. of } \alpha_t \text{ and } \mathcal{W}_t \\ \quad = \alpha_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \text{by i.h.} \\ \quad = \alpha_t(f) & \text{by def. of } \alpha_t \end{array}$$

- $t = \forall\beta.t_1$. In this case it trivially holds by i.h. as $\alpha_t = \alpha_{t_1}$ and $\mathcal{W}_t = \mathcal{W}_{t_1}$.

□

Proposition 22 *For each type t, $\delta_t \cdot \mathcal{W}_t = \delta_t$.*

Proof. (Proposition 22) This can be proved by structural induction on t and by Proposition 21:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\delta_t = \mathcal{W}_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $(e_1, \ldots, e_m) \in D_{2t}$. In this case:

$$\begin{array}{l} \delta_t(\mathcal{W}_t(e_1, \ldots, e_m)) \\ \quad = (\alpha_{t_1}(\mathcal{W}_{t_1}(e_1)), \ldots, \alpha_{t_m}(\mathcal{W}_{t_m}(e_m))) \\ \qquad \{\text{by def. of } \delta_t \text{ and } \mathcal{W}_t\} \\ \quad = (\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) \\ \qquad \{\text{by Proposition 21}\} \\ \quad = \delta_t(e_1, \ldots, e_m) \\ \qquad \{\text{by def. of } \delta_t\} \end{array}$$

- $t = t_1 \to t_2'$ or $t = \mathit{Process}\ t_1\ t_2'$. Let $f \in D_{2t}$. Let $m = \mathit{nArgs}(t)$, $[t_1, \ldots, t_m] = \mathit{aTypes}(t)$ and $t_r = \mathit{rType}(t)$. Then

$$\begin{array}{ll} \delta_t(\mathcal{W}_t(f)) & \\ \quad = \delta_{t_r}(\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d))) & \text{by def. of } \delta_t \text{ and } \mathcal{W}_t \\ \quad = \delta_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \text{by i.h.} \\ \quad = \delta_t(f) & \text{by def. of } \delta_t \end{array}$$

- $t = \forall\beta.t_1$. It holds in this case by i.h. as $\delta_t = \delta_{t_1}$ and $\mathcal{W}_t = \mathcal{W}_{t_1}$.

□

The correctness of the first analysis with respect to the second one is trivially obtained from Theorem 19, as the second analysis is in fact a variant of the third analysis where the widening operator is the identity function.

8 Implementation of the third analysis

8.1 Introduction

In this section we describe the main aspects of the third 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 [Hug95]. In Section 9 we show some examples and the ouput produced for them by the analysis. It is important to annotate the subexpressions, even inside a lambda-abstraction. In [PS01] we explained that the full laziness transformation [JPS96] may change the semantics of a program when

$$\gamma'_t :: Basic \to S_t$$

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

Fig. 23. The signatures corresponding to $\gamma_t(n)$ and $\gamma_t(d)$.

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

Fig. 24. Implementation of functions $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$

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. It is not necessary to use a different notation for building abstract functions, but we can use instead the same language constructs. 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 23 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, the function $\gamma'_t = \wp_t \cdot \gamma_t$, defined in Figure 23, returns the signature of $\gamma_t(b)$.

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 8.3, 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 25. In Figure 24 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 27.

We also need to build a function when computing the lub of functions. In this case, we also use a new suspension $\boxed{\text{F}}[av_1, \ldots, av_m]$, see Figure 25. When the function represented by this suspension is applied, the lub will be computed, see Figure 27.

We now give some thoretical result that will be useful for the development of the implementation.

8.2 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. All of them can be proved by structural induction on t.

The following proposition 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.

Proposition 23 *For each type t, $\mathcal{W}_t \cdot \gamma_t = \gamma_t$.*

Proof. (Proposition 23) We prove this proposition by structural induction on t.

- $t = K, T\ t_1 \ldots t_m, \beta$. This case is trivial as $\mathcal{W}_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $b \in Basic$. Then

$$\begin{aligned}
\mathcal{W}_t(\gamma_t(b)) &= (\mathcal{W}_{t_1}(\gamma_{t_1}(b)), \ldots \mathcal{W}_{t_m}(\gamma_{t_m}(b))) && \{\text{by } \mathcal{W}_t \text{ and } \gamma_t \text{ definition}\} \\
&= (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) && \{\text{by induction hypothesis}\} \\
&= \gamma_t(b) && \{\text{by } \gamma_t \text{ definition}\}
\end{aligned}$$

- $t = t_1 \rightarrow t_2, Process\ t_1\ t_2$. Let $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$ and $t_r = rType(t)$. Let $z_i \in D_{2_{t_i}}$ $(i \in \{1..m\})$. We have to prove that $\mathcal{W}_t(\gamma_t(b))\ \overline{z_i} = \gamma_t(b)\ \overline{z_i}$. We distinguish two cases, $b = n$ and $b = d$. If $b = n$ then

$$\begin{aligned}
\mathcal{W}_t(\gamma_t(n))\ \overline{z_i} &= \mathcal{W}_{t_r}(\gamma_{t_r}(n)) && \{\text{by } \mathcal{W}_t \text{ and } \gamma_t \text{ definitions}\} \\
&= \gamma_{t_r}(n) && \{\text{by induction hypothesis}\} \\
&= \gamma_t(n) && \{\text{by } \gamma_t \text{ definition}\}
\end{aligned}$$

  If $b = d$ we have to distinguish two cases. If $\bigwedge_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$. It has been proved that α_t and γ_t are a Galois insertion, which in this case implies that (4) $\alpha_{t_i}(z_i) = d$ $(i \in \{1..m\})$. This means that (5) $\gamma_t(d)\ \overline{z_i} = \gamma_{t_r}(d)$ by definition of γ_t.

$$\begin{aligned}
\mathcal{W}_t(\gamma_t(d))\ \overline{z_i} &= \mathcal{W}_{t_r}(\gamma_{t_r}(d)) && \{\text{by (4) and } \mathcal{W}_t \text{ definition}\} \\
&= \gamma_{t_r}(d) && \{\text{by induction hypothesis}\} \\
&= \gamma_t(d)\ \overline{z_i} && \{\text{by (5)}\}
\end{aligned}$$

  Otherwise there is a $j \in \{1..m\}$ such that $z_j \not\sqsubseteq \gamma_{t_j}(d)$. This means that (6) $\alpha_{t_j}(z_j) = n$. This means that (7) $\gamma_t(d)\ \overline{z_i} = \gamma_{t_r}(n)$ by definition of γ_t.
  If $\bigwedge_{i=1, i \neq j}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$ then

$$\begin{aligned}
\mathcal{W}_t(\gamma_t(d))\ \overline{z_i} &= \mathcal{W}_{t_r}(\gamma_{t_r}(n)) && \{\text{by (6) and } \mathcal{W}_t \text{ definition }\} \\
&= \gamma_{t_r}(n) && \{\text{by induction hypothesis}\} \\
&= \gamma_t(d)\ \overline{z_i} && \{\text{by (7)}\}
\end{aligned}$$

  Otherwise

$$\begin{aligned}
\mathcal{W}_t(\gamma_t(d))\ \overline{z_i} &= \gamma_{t_r}(n) && \{\text{by } \mathcal{W}_t \text{ definition}\} \\
&= \gamma_t(d)\ \overline{z_i} && \{\text{by (7)}\}
\end{aligned}$$

- $t = \forall \beta . t_1$. This case is trivial by induction hypothesis as $\mathcal{W}_t = \mathcal{W}_{t_1}$ and $\gamma_t = \gamma_{t_1}$.

□

The following proposition 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 of the analysis, as we have to do this comparison very often. In the worst case the comparison is made in $\mathcal{H}_t$ steps.

Proposition 24 *For each type t, $\forall z \in D_{2t}.z \sqsubseteq \gamma_t(d) \Leftrightarrow \wp_t(z) \preceq \wp_t(\gamma_t(d))$.*

Proof. (Proposition 24)

- ($\Rightarrow$) This is trivial as by 1(a), $\wp_t$ is monotone.
- ($\Leftarrow$) Let us assume that $\wp_t(z) \preceq \wp_t(\gamma_t(d))$. Then

$$\begin{array}{ll} z \sqsubseteq \mathcal{W}_t(z) & \{\text{by Proposition 1(b)}\} \\ = \Re_t(\wp_t(z)) & \{\text{by definition of } \mathcal{W}_t\} \\ \sqsubseteq \Re_t(\wp_t(\gamma_t(z))) & \{\text{by hypothesis and Proposition 1(a)}\} \\ = \mathcal{W}_t(\gamma_t(z)) & \{\text{by definition of } \mathcal{W}_t\} \\ = \gamma_t(z) & \{\text{by Proposition 23}\} \end{array}$$

□

8.3 Polymorphism

We would like to have a property similar to Proposition 23, $\mathcal{W}_{tinst} \cdot \gamma_{t' tinst} = \gamma_{t' tinst}$ so that we could represent the instantiations' abstract value by its signature, but this is not true. As a counterexample, recall the example in Section 3. Let the polymorphic type $\forall\beta.t'$, where $t' = (\beta, \beta) \to \beta$, be applied to 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

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

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

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

8.4 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 25 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 25 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 $\sqcup_{\mathrm{F}}\,[av_1, \ldots, av_m]$ or a polymorphism suspension, i.e. $G\ t'\ t'[t/\beta]\ av$ or $A\ t'[t/\beta]\ t'\ av$.

In Figure 26 the lub operator between abstract values is defined. For basic values/signatures and tuples 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.

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

Fig. 25. Abstract values definition

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

Fig. 26. Lub operator definition

8.5 Abstract application of a function

In Figure 27 the definition of the application of an abstract function to an abstract argument is shown. In case the abstract function is a signature $<t_1 \rightarrow \ldots \rightarrow t_m \rightarrow t_r,\ aw_1 \ldots aw_m + aw>$ 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 24). If this happens, then we discard the first element aw_1 of the signature and return $<t_2 \rightarrow \ldots \rightarrow t_m \rightarrow t_r, aw_2 \ldots aw_m + aw>$, 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 $<t_2 \rightarrow \ldots \rightarrow t_m \rightarrow t_r, +aw_1>$ is returned. If the abstract function is a signature $<t_1 \rightarrow \ldots \rightarrow t_m \rightarrow t_r,\ +aw>$, 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. This is done by 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 first 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 definition of $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ (it was temporally suspended). The algorithm then proceeds by suspending and evaluating once and again.

8.6 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

$$
\begin{array}{l}
(\lambda v.(e,\rho))\ av = [\![e]\!]'\ \rho[v \mapsto (av, aw, b)] \text{ where } v :: t_v, \quad aw = \wp_{t_v}(av), \quad b = \alpha_{t_v}(av) \\
(\bigsqcup_{\mathrm{F}} [av_1, \ldots, av_m])\ av' = \bigsqcup\ [av_1\ av', \ldots, av_m\ av'] \\
(G\ t'\ t'[t/\beta]\ av)\ av' = \gamma'_{t_2 inst_2}(av\ (\alpha'_{inst_1 t_1}(av'))) \\
(A\ t'[t/\beta]\ t'\ av)\ av' = \alpha'_{inst_2 t_2}(av\ (\gamma'_{t_1 inst_1}(av'))) \\
<t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw>\ av'\ (m > 1) \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = <t_2 \to \ldots \to t_m \to t_r,\ aw_2 \ldots aw_m + aw> \\
\quad \mid \text{otherwise} = <t_2 \to \ldots \to t_m \to t_r,\ + aw_1> \\
<t_1 \to t_r,\ aw_1 + aw>\ av' \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw \\
\quad \mid \text{otherwise} = aw_1 \\
<t_1 \to \ldots \to t_m \to t_r,\ + aw>\ av'\ (m > 1) \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = <t_2 \to \ldots \to t_m \to t_r,\ + aw> \\
\quad \mid \text{otherwise} = \gamma'_{t_2 \to \ldots \to t_m \to t_r}(n) \\
<t_1 \to t_r,\ + aw>\ av' \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw \\
\quad \mid \text{otherwise} = \gamma'_{t_r}(n)
\end{array}
$$

Fig. 27. Application of abstract functions

to keep as much information as possible, except in the fixpoint calculation where it will be substituted by its corresponding signature. In Figure 28 only the algorithm for $[\![\cdot]\!]$ is shown in pseudocode. The one for $[\![\cdot]\!]'$ is very similar. Given an expression e and an environment ρ, $[\![e]\!]'\ \rho$ 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.

Let us describe the interpretation definition. The first component of the result av is 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.

The annotation of a variable is simple, as its signature is in the environment. A literal is annotated with a deterministic basic value d. To annotate a tuple, first we annotate the components; then, the annotation of the tuple is the tuple of the annotations. To annotate a constructed value, again first we annotate the components. Here we use the fact that in the environment we carry the corresponding basic value of each component (when it is a variable), so we just take their lub to obtain the final annotation.

To annotate a primitive operator, we first annotate each argument. As the abstract value of the primitive operator is a signature $\gamma'_{top}(d)$, the result of its application to several arguments will also be a signature, so in this case no additional probing is necessary. Additionally, the application of a signature to an argument implies the probing of the argument, remember Figure 27. In this point we know from the environment the signatures of the arguments, so we give them directly to the application.

To annotate a lambda abstraction, its body must be annotated first. But, to be able to do this, the argument variable must be given an abstract value. Its abstract value will depend on the function applications, but here we are annotating the function by itself and we do not know which applications there will be in the program. So we have to give the argument the pessimistic value $\gamma'_{t_v}(n)$. In an application $e\ x$, we first annotate the subexpressions e and x, and then annotate the expression with the probing of the resulting abstract value. Process abstractions and instantiations are similar to lambda abstraction and application.

A *let* expression is annotated with the same annotation as its main expression. If the binding $v = e$ is non-recursive, the variable v is added to the environment. If the binding is recursive, in the fixpoint

calculation we use $[\![\cdot]\!]'$ to obtain the abstract value of each right hand side and then probe it to obtain the signature. If $[\![\cdot]\!]$ were used instead, at each iteration all the e_i would be unusefully annotated. In this way, only when the fixpoint is obtained, we annotate the bindings' right hand sides.

A tuple *case* expression is annotated with the same annotation as its alternative. Each variable in the left hand side of the alternative is annotated with the corresponding component of the discriminant's annotation. In an algebraic *case*, we have to consider that the abstract value may be a lub suspension of the abstract values of the alternatives. The probing of this suspension is the lub of the probings of each alternative's abstract value, that is, we just have to take the alternatives' annotations and calculate their lub. The variables in the left hand sides are annotated according to the value of the discriminant. If a default alternative appears, the variable in the left hand side is annotated with the discriminant's abstract value. Primitive cases are like algebraic cases, where the constructors have no arguments, so we do not write this case. A type abstraction is annotated with the same annotation as its body. A type application is annotated with the probing of the corresponding abstract value.

9 Output examples

In Figure 29 we show some examples. This is not the source code given to the parser, but a more readable version. In figures 30 and 31 we show the output produced by the analysis for these examples as it is done by the pretty printer. If an expression `e` has been annotated with a signature `aw`, it is printed as `e::aw`. Functional signatures are written in braces. Tuples and lists are respectively shown with the constructor `Tup` and `Cons`.

First we present some simple examples. They are small functions that cover tuple types, higher order functions and recursion. Then, the polymorphism counterexample of Section 8.3 is shown. Here f is represented by `counter` and q corresponds to `pair`. The variable `cinstap` corresponds to $(\gamma_{t'\,tinst}(f)\ q)\ d$, so it should have a d annotation. In Figure 31 we can see that the signature given to `countinst` is $nn{+}d$. This corresponds to $\mathcal{W}_{tinst}(\gamma_{t'\,tinst}(f))$. The abstract value $\gamma_{t'\,tinst}(f)$ is however used to obtain the abstract value of `cinstap`, which is d, as it should.

Finally, a simplified version of a replicated workers topology [KPR00] is shown. We have a manager process `m` and $n = 2$ worker processes `w1` and `w2`. The manager provides the workers with tasks. When any of the workers finishes its task, it sends a message to the manager including the obtained results and asking for a new task. In order to enable the manager to assign new tasks to idle processes inmediately, even though the answers may be received from the workers in any order, a `merge` process is needed. An initial list of tasks `ts` is given to the manager process.

10 Cost of the Analysis

Analysing the cost of the interpretation algorithm has proved to be a very hard task. This is due to the fact that many of the functions involved —in particular $[\![\cdot]\!]$, $[\![\cdot]\!]'$, abstract application, γ'_{ttinst}, $\wp_t$, $\alpha'_{tinst\,t'}$, and α'_t— are heavily mutually recursive and also due to the creation of suspensions. The latter means that the interpretation of an expression stops as soon as a λ or a **process** is found (in what follows we will use the word 'lambda' for both). The interpretation is re-started when the lambda is applied to one argument, but then a new lambda may appear and the interpretation is suspended again. This stop-and-go process continues until the lambda is applied to all arguments, for instance when computing its probing.

Fortunately, there are small functions whose cost can be directly computed in terms of the size of the types involved. 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 two functions $s, s' : Expr \to Int$ respectively giving the 'size' of an expression e when interpreted by $[\![\cdot]\!]$ and by $[\![\cdot]\!]'$. We say then

that $[\![e]\!]'\ \rho \in O(s'(e))$ and that $[\![e]\!]\ \rho \in O(s(e))$. The complete definition of s and s' appears in Appendix 10.1. We can say here that most of the time, $s(e)$ and $s'(e)$ are linear with e using the 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:

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. The notion of lambda-body is a tricky one. It involves the replacement of free variables by its definition whenever the variables appear applied in the body. Assuming the notion of body is well defined and being e_λ the body of a lambda, the algorithm costs $O(s'(e_\lambda))$ each time the lambda is applied.

Probing a function Probing 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 $O((m+1)\ s'(e_\lambda))$.

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. Let us remember that $\mathcal{H}_t$ gives the depth of the signature domain. At each iteration, the signature of e is obtained, so we can say that 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 the cost of completely annotating e, which involves m probings more, each one with one parameter less. In total, the annotation will cost $O(m^2\ s'(e_\lambda))$.

Summarizing, we can say that 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 cuadratic because of probing—, and in fixpoints where it can reach a cubic cost.

10.1 Cost functions

In this section a more detailed study of the cost of the analysis is developed. Our aim in this cost analysis is to define the cost of the functions in terms of the original source program text, i.e. the program expressions or their types, and not in terms of their abstract values.

An equality or an inequality comparison between signatures in S_t is in $O(\mathcal{H}_t)$. The cost of a least upper bound between two abstract values of type t is in $O(\mathcal{H}_t)$. So the least upper bound between m abstract values of type t is in $O((m-1)\ \mathcal{H}_t)$. The cost of $\gamma'_t(b)$ is in $O(m + \mathcal{H}_{t_r})$, where $m = nArgs(t)$ and $t_r = rType(t)$.

Given an expression $e :: t$, whose abstract value is av, the cost of obtaining $\alpha_t(av)$ is in $O(a_t(e))$, being $a_t(e) = A_t + \sum_{i=1}^{m}(m_i + \mathcal{H}_{t_{ri}}) + oneap(e)$, where $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$, $t_r = rType(t)$, $m_i = nArgs(t_i)$ and $t_{ri} = rType(t_i)$. First we obtain the probing of av with deterministic arguments and then we flatten the result until we obtain a basic value, which costs A_t, see Figure 32.

Given a polymorphic type $\forall\beta.t'$, another type t, and an abstract value $av \in D_2{}'_t$, the cost of $\gamma'_{t'\,tinst}(av)$, where $tinst = t'[t/\beta]$, is in $O(\Gamma^{\beta}_{t'\ t})$, defined in Figure 33.

Given an expression e, $[\![e]\!]'\ \rho \in O(s'(e))$ and $[\![e]\!]\ \rho \in O(s(e))$, see Figure 39 and Figure 36. In Figure 34, we define an auxiliary function *comps*. Given an expression e of tuple type $(t_1, \ldots, t_m)$, with abstract value $(av_1, \ldots, av_m)$, we want to return in a list, the expressions e_i that originated the abstract value of each component av_i. As we have *case* expressions, there may be several expressions for each component. *comps* returns a pair of lists. The first one contains the lists of expressions corresponding to each component. The second one is a list of integers. When a variable is bound by a lambda, an algebraic *case* or a recursive *let* expression, its abstract value is a signature, and there are no components expressions. We just count how many of them there are for each component.

Let an expression $e :: t$ with abstract value av and an argument $x :: t_1$ with abstract value ax. To obtain the cost of applying $av\ ax$, we define a function *body*, see Figure 37. Given an expression e (of functional type) and its argument, it returns a list of expressions and an integer. The integer is an accumulated cost. The list elements are those expressions whose abstract value is the result of the abstract application. They may be more than one because of the *case* expressions.

In Figure 35 the cost of obtaining a signature is shown. We explain here the functional case, the rest are inmediate. Let an expression $e :: t$, where $t = t_1 \to t_2$ or $t = Process\ t_1\ t_2$, with abstract value av. The cost of obtaining the signature corresponding to av is denoted by $p_t(e)$. If av is a signature, the cost of its probing is just $\mathcal{H}_t$ (the time it takes to know it is a signature). If it is not, then $oneap(e)$ represents the cost of one probing of av, so $p_t(e) = \mathcal{H}_t + \sum_{i=1}^{m}(m_i + \mathcal{H}_{t_{r_i}}) + (m+1)\ oneap(e)$ represents the cost of the whole probing, where $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$, $t_r = rType(t)$, $m_i = nArgs(t_i)$ and $t_{ri} = rType(t_i)$.

11 Conclusions

We have presented and formally compared three non-determinism analyses for a functional language with second-order polymorphism and non-deterministic expressions. Although the main motivation for this work has been the correct compilation of our language Eden, everything 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. Related work concerning other abstract interpretation and type based analyses was already reported in previous papers [PS01,PS00]. We have not found any previous analyses for this problem in the literature. We have seen that the second and third analyses are based in abstract interpretation in the style of [BHA86], where functions are interpreted as abstract functions. There, a strictness analysis is presented where the basic domain **2** is also a two-point domain ($- \sqsubseteq \top$). However, the analyses are rather different. As an example, let $f :: (Int \to Int) \to Int$ be $f = \lambda g.g(head(merge\#[[0],[1]]))$. The strictness analysis tells us that f is strict in its argument, while the non-determinism analysis tells us that it may be non-deterministic. Also, the abstract interpretation of primitive operators, constructors and *case* expressions is different in each analysis.

The first paper on this subject [PS01] presented our preliminary ideas in the form of two not completely satisfactory analyses. The first one was efficient but not powerful enough, while the second one was powerful but not practical in terms of efficiency. This paper developes a third analysis that is both powerful enough and practical in implementation terms: 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. This paper has also presented the theoretical results supporting the correctness of that implementation. 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.

For our purposes, this closes the initial problem. A proof of correctness of the second analysis with respect to the standard semantics of the language remains to be done. Unfortunately there is still no such semantics available for Eden. Nevertheless, a simplified form of semantics could be used to prove part of the analysis correctness.

$[\![\cdot]\!]' :: Expr \to Env \to AbsVal$

$[\![e]\!]' \ \rho = \pi_1([\![e]\!] \ \rho)$

$[\![\cdot]\!] :: Expr(\ ) \to Env \to (AbsVal, Expr \ AbsVal)$

$[\![v]\!] \ \rho = (av, v@aw)$

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 $(av_i, x'_i) = [\![x_i]\!] \ \rho$; $aw = \sqcup_{i=1}^{m} \ b_i$; $b_i =$ if $isvar(x_i)$ then $(\pi_3(\rho(x_i)))$ else d

$[\![op \ x_1, \ldots, x_m]\!] \ \rho = (aw, op \ x'_1 \ldots x'_m@aw) \qquad \{op :: top\}$

where $(av_i, x'_i) = [\![x_i]\!] \ \rho$; $aw = \gamma'_{top}(d) \ aw_1 \ \ldots \ aw_m$; $aw_i =$ if $isvar(x_i)$ then $(\pi_2(\rho(x_i)))$ 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 $(\rho', bind') = [\![bind]\!]_B \ \rho$; $(a, e') = [\![e]\!] \ \rho'$ $e''@aw = e'$

$[\![\textbf{case} \ e \ \textbf{of} \ (v_1, \ldots, v_m) \to e']\!] \ \rho = (a, (\textbf{case} \ (e_1@awe) \ \textbf{of} \ (v'_1, \ldots, v'_m) \to (e'_1@aw))@aw) \qquad \{v_i :: t_i\}$

where $(ae, e1@awe) = [\![e]\!] \ \rho$; $aw_i = \pi_i(awe)$; $v'_i = v_i@aw_i$; $av_i = \pi_i(ae)$;

$b_i = \alpha_{t_i}(av_i)$; $\qquad (a, e'_1@aw) = [\![e']\!] \ \rho\overline{[v_i \mapsto (av_i, aw_i, b_i)]}$;

$[\![\textbf{case} \ e \ \textbf{of} \ \overline{alt_i}]\!] \ \rho = (av, (\textbf{case} \ e' \ \textbf{of} \ \overline{alt'_i})@aw) \qquad \{\textbf{case} \ e \ \textbf{of} \ \overline{alt_i} :: t\}$

where $(ae, e') = [\![e]\!] \ \rho$; $(av_i, alt'_i) = [\![alt_i]\!]_A \ ae \ \rho$; $C_i \ v_{i1} \ldots v_{im_i} \to (e_i@wa_i) = alt'_i$;

$aw =$ if $ae = n$ then $\gamma'_t(n)$ else $\sqcup_{i=1}^{m} \ aw_i$; $av =$ if $ae = n$ then $\gamma'_t(n)$ 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 $(av, e'@aw) = [\![e]\!] \ \rho$; $b = \alpha_{t_v}(av)$

$[\![\textbf{rec} \ \overline{v_i = e_i}]\!]_B \ \rho = (\rho_{fix}, \textbf{rec} \ \overline{v'_i = e''_i}) \qquad \{v_i :: t_i\}$

where $\rho_{fix} = fix \ f \ init$; $init = \rho\overline{[v_i \mapsto (aw_i, aw_i, d)]}$; $\qquad aw_i = \gamma'_{t_i}(d)$

$f\rho' = \rho'\overline{[v_i \mapsto (aw'_i, aw'_i, b_i)]}$ where $av'_i = [\![e_i]\!]' \ \rho'$; $\qquad aw'_i = \wp_{t_i}(av'_i)$; $b_i = \alpha_{t_i}(av'_i)$

$(_, e''_i) = [\![e_i]\!] \ \rho_{fix}$; $(_, aw_i, _) = \rho_{fix}(v_i)$; $v'_i = v_i@aw_i$

$[\![C \ v_1 \ldots v_m \to e]\!]_A \ avd \ \rho = (av, C \ v'_1 \ldots v'_m \to e') \qquad \{v_i :: t_i\}$

where $aw_i = \gamma'_{t_i}(avd)$; $(av, e') = [\![e]\!] \ \rho\overline{[v_i \mapsto (aw_i, aw_i, avd)]}$; $v'_i = v_i@aw_i$

$[\![v \to e]\!]_A \ avd \ \rho = (av, v' \to e')$

where $(av, e') = [\![e]\!] \ \rho[v \mapsto (avd, avd, avd)]$; $v' = v@avd$

Fig. 28. The expressions annotation algorithm

```
--Some lists
zero::[Int] = 0:[]
one::[Int] = 1:[]
zdnil::[[Int]] = zero:[]
zoxss::[[Int]] = one:zdnil

--A non-deterministic integer
mergeint::Process [[Int]] [Int] = merge Int
xs::[Int] = mergeint # coxss
nondet::Int = head xs

--Some functions with tuples as result
pf1::Int -> (Int,Int) = \pf01::Int.(pf01,2)
pf3::Int -> (Int,Int) = \pf03::Int.(pf03,pf03)
pf4::Int -> (Int,Int) = \pf04::Int.(pf04,nondet)

--Some functions with tuples as arguments
at1::(Int,Int) -> Int = \pat1::(Int,Int).case pat1 of (pat11::Int,pat12::Int) -> pat11
at3::(Int,Int) -> Int = \pat3::(Int,Int).case pat3 of (pat31::Int,pat32::Int) -> pat31 + pat32

--Some higher order functions
high1::(Int -> Int) -> Int = \f1::Int -> Int.f1 1
high2::(Int -> Int) -> Int -> Int = \f2::Int -> Int.\arg::Int.f2 arg
--The sum function
sum::[Int] -> Int = \l1::[Int].case l1 of
                                 [] -> 0
                                 x1:xs1 -> let rest1::Int = sum xs1
                                           in  x1 + rest1
--The polymorphism counterexample
id::Int -> Int = \v::Int.v
fn::Int -> Int = \z::Int.nondet
counter::\/ delta.(delta,delta) -> delta =
   /\delta.\cp::(delta,delta). case cp of (cp1::delta,cp2::delta) -> cp1
countinst::(Int -> Int, Int -> Int) -> Int -> Int = counter (Int -> Int)
pair::(Int -> Int,Int -> Int) = (id,fn)
cinstap::Int = (countinst pair) 6

--The replicated workers topology
replicated ::\/ alpha.\/ beta. Process ([alpha],[alpha]) (beta,beta,beta) ->
             Process beta [alpha] -> Process beta [alpha] -> [alpha] -> beta =
/\alpha./\beta.\m::Process ([alpha],[alpha]) (beta,beta,beta).
       \w1::Process beta [alpha].
       \w2::Process beta [alpha].
       \ts::[alpha]
           let rec
            t::([alpha],[alpha]) = (ts,is)
            om::(beta,beta,beta) = m # t
            px1::beta = case om of (u1::beta,u2::beta,u3::beta) -> u1
            px2::beta = case om of (v1::beta,v2::beta,v3::beta) -> v2
            px3::beta = case om of (q1::beta,q2::beta,q3::beta) -> q3
            o1::[alpha] = w1 # px1
            o2::[alpha] = w2 # px2
            lo1::[[alpha]] = o1:[]
            o1o2::[[alpha]] = o2:lo1
            mergealpha::Process [[alpha]] [alpha] = merge alpha
            is::[alpha] = mergealpha # o1o2
           in
             px3
```

Fig. 29. The program example

```
zero::d =  Cons (0 ::d) ([]::d) ::d
one::d =  Cons (1 ::d) ([]::d) ::d
zdnil::d =  Cons (zero::d) ([]::d) ::d
zoxss::d =  Cons (one::d) (zdnil::d) ::d
xs::n =  (mergeint::{n +n}) # (zoxss::d)::n
nondet::n =  head::{n +d} (xs::n) ::n
pf1::{(n, d) +(d, d)} =
    \ pf01::n. Tup (pf01::n) (2 ::d) ::(n, d)
    ::{(n, d) +(d, d)}
pf3::{(n, n) +(d, d)} =
    \ pf03::n. Tup (pf03::n) (pf03::n) ::(n, n)
    ::{(n, n) +(d, d)}
pf4::{(n, n) +(d, n)} =
    \ pf04::n. Tup (pf04::n) (nondet::n) ::(n, n)
    ::{(n, n) +(d, n)}
at1::{n +d} =
    \ pat1::(n, n).
        case (pat1::(n, n)) of
            Tup (pat11::n)  (pat12::n)  -> pat11::n
        ::n
    ::{n +d}
at3::{n +d} =
    \ pat3::(n, n).
        case (pat3::(n, n)) of
            Tup (pat31::n)  (pat32::n)  -> Plus::{n n +d} (pat31::n) (pat32::n) ::n
        ::n
    ::{n +d}
high1::{n +d} =
    \ f1::{n +n}. (f1::{n +n}) (3 ::d) ::n
    ::{n +d}
high2::{n n +d} =
    \ f2::{n +n}.
        \ arg::n. (f2::{n +n}) (arg::n) ::n
        ::{n +n}
    ::{n n +d}
sum::{n +d} =
    \ l1::n.
        case (l1::n) of
            Nil  -> 0 ::d
            Cons (x1::n)  (xs1::n)  ->
                  let
                      rest1::n =  (sum::{n +d}) (xs1::n) ::n
                  in
                     Plus::{n n +d} (x1::n) (rest1::n) ::n
                  ::n
        ::n
    ::{n +d}
```

Fig. 30. The output of the examples in Figure 29 (I)

```
id::{n +d} =
    \ v::n. v::n
    ::{n +d}
fn::{n +n} =
    \ z::n. nondet::n
    ::{n +n}
counter::{n +d} =
    /\ delta.
        \ cp::(n, n).
            case (cp::(n, n)) of
                Tup (cp1::n)  (cp2::n)  -> cp1::n
            ::n
        ::{n +d}
    ::{n +d}
countinst::{n n +d} =  (counter::{n +d}) (Int -> Int) ::{n n +d}
pair::({n +d}, {n +n}) =  Tup (if::{n +d}) (fn::{n +n}) ::({n +d}, {n +n})
cinstap::d =  ((countinst::{n n +d}) (pair::({n +d}, {n +n})) ::{n +d})
              (6 ::d) ::d
replicated::{n n n n +n} =
  /\ alpha.
      /\ beta.
         \ m::{(n, n, n) +(n, n, n)}.
             \ w1::{n +n}.
               \ w2::{n +n}.
                 \ ts::n.
                    let
                     t::(n, n) = Tup (ts::n) (is::n) ::(n,n)
                     om::(n, n, n) =  (m::{(n, n, n) +(n, n, n)}) # (t::(n, n))::(n, n, n)
                     px1::n =
                        case (om::(n, n, n)) of
                            Tup (u1::n)  (u2::n)  (u3::n)  -> u1::n
                        ::n
                     px2::n =
                        case (om::(n, n, n)) of
                            Tup (v1::n)  (v2::n)  (v3::n)  -> v2::n
                        ::n
                     px3::n =
                        case (om::(n, n, n)) of
                            Tup (q1::n)  (q2::n)  (q3::n)  -> q3::n
                        ::n
                     o1::n =  (w1::{n +n}) # (px1::n)::n
                     o2::n =  (w2::{n +n}) # (px2::n)::n
                     lo1::n =  Cons (o1::n) ([]::d) ::n
                     o1o2::n =  Cons (o2::n) (lo1::n) ::n
                     mergealpha::{n +n} =  (merge::{n +n}) (alpha) ::{n +n}
                     is::n =  (mergealpha::{n +n}) # (o1o2::n)::n
                    in
                     px3::n
                    ::n
                 ::{n +n}
               ::{n n +n}
             ::{n n n +n}
         ::{n n n n +n}
      ::{n n n n +n}
  ::{n n n n +n}
```

Fig. 31. The output for the examples in Figure 29 (II)

$$
\begin{aligned}
&\mathcal{A}_K = \mathcal{A}_{T\ t_1 \ldots t_m} = \mathcal{A}_\beta = 1 \\
&\mathcal{A}_{(t_1,\ldots,t_m)} = \textstyle\sum_{i=1}^m \mathcal{A}_{t_i} \\
&\mathcal{A}_t = \mathcal{A}_{resType(t)} \quad \text{if } t = t_1 \to t_2, Process\ t_1\ t_2 \\
&\mathcal{A}_{\forall \beta . t} = \mathcal{A}_t
\end{aligned}
$$

Fig. 32. The cost of flattening one probing

$$
\begin{aligned}
&\Gamma^\beta_{K,t} = \Gamma^\beta_{T\ t_1 \ldots t_m, t} = \Gamma^\beta_{\beta',t} = 1 \\
&\Gamma^\beta_{\beta,t} = nArgs(t) + \mathcal{H}_{resType(t)} \\
&\Gamma^\beta_{(t_1,\ldots,t_m),t} = \textstyle\sum_{i=1}^m \Gamma^\beta_{t_i,t} \\
&\Gamma^\beta_{t_1 \to t_2, t} = 1 \\
&\Gamma^\beta_{Process\ t_1\ t_2, t} = 1 \\
&\Gamma^\beta_{\forall \beta' . t', t} = \Gamma^\beta_{t',t}
\end{aligned}
$$

Fig. 33. The cost of $\gamma'_{t' tinst}$

$$
\begin{aligned}
&comps :: Expr \to ([[Expr]], [Int]) \\
&comps(x_1, \ldots, x_m) = ([[x_1], \ldots, [x_m]], [0, \ldots, 0]) \\
&comps(\textbf{let}\ bs\ \textbf{in}\ e) = comps\ (e) \\
&comps(\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e') = comps(e') \\
&comps(\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) = join(map\ comps\ [e_1, \ldots, e_m]) \quad \{i \in \{1..m\}\} \\
&comps(e\ x) = join(map\ comps\ es) \\
&\quad \text{where}\ \ (es, _) = body\ e\ x \\
&comps(v) = comps(e) \qquad \{\text{let-nonrec bound}\}\ \{v = e\} \\
&comps(v) = ([\,], [1, \ldots, 1]) \ \ \{\text{lambda, let-rec, alg. case bound}\} \\
&comps(v) = (ves, ncs) \qquad \{k\text{th tuple case bound, discriminant } e\} \\
&\quad \text{where}\ \ (es, cs) = comps(e), es_k = es!!(k-1), c_k = cs!!(k-1) \\
&\qquad\qquad (ves, vcs) = join(map\ comps\ es_k), ncs = zipWith\ (+)\ vcs\ [c_k, \ldots, c_k] \\
&\\
&join\ xs = foldl1\ g\ xs \\
&g\ (xs_1, cs_1)\ (xs_2, cs_2) = (zipWith\ (+\!\!+)\ xs_1\ xs_2, zipWith\ (+)\ cs_1\ cs_2)
\end{aligned}
$$

Fig. 34. The components function *comps*

$$
\begin{aligned}
&p_K(e) = p_{T\ t_1 \ldots t_m}(e) = p_\beta(e) = 1 \\
&p_{(t_1,\ldots,t_m)}(e) = \textstyle\sum_{i,j} p_{t_i}(e_{ij}) + \sum_i n_i\ \mathcal{H}_{t_i} \\
&\quad \text{where}\ \ (es, cs) = comps(e), i \in \{1..m\}, n_i = cs!!(i-1), \\
&\qquad\qquad j \in \{1..(length\ es_i)\}, e_{ij} = (es!!(i-1))!!(j-1) \\
&p_t(e) = \mathcal{H}_t + \textstyle\sum_{i=1}^m (m_i + \mathcal{H}_{t_{ri}}) + (m+1)\ oneap(e) \quad \{t = t_1 \to t_2, Process\ t_1\ t_2\} \\
&p_{\forall \beta . t}(e) = p_t(e)
\end{aligned}
$$

Fig. 35. The cost of obtaining a signature for an abstract value

$$
\begin{array}{l}
s'(k) = s'(v) = 1 \\
s'(x_1, \ldots x_m) = m \\
s'(C\ x_1 \ldots x_m) = m \\
s'(op\ x_1 \ldots x_m) = \sum_i \mathcal{H}_{t_i} \\
\qquad \{op :: top, m = nArgs(t), [t_1, \ldots, t_m] = aTypes(t), t_{m+1} = rType(top), i \in \{1..(m+1)\}\} \\
s'(\lambda v.e) = 1 \\
s'(e\ x) = s'(e) + s'(x) + (sum\ (map\ s'\ es) + c) \\
\qquad \text{where } (es, c) = body\ e\ x \\
s'(\textbf{let}\ v = e\ \textbf{in}\ e') = s'(e) + s'(e') + p_t(e) + a_t(e) \qquad \{e :: t\} \\
s'(\textbf{let rec}\ \overline{v_i = e_i}\ \textbf{in}\ e') = niter * \sum_i (s'(e_i) + p_{t_i}(e_i) + a_{t_i}(e_i) + 1) + s'(e') \\
\qquad \text{where } niter = \sum_{i=1}^{m} \mathcal{H}_{t_i} \\
\qquad\quad \{i \in \{1..m\}, e_i :: t_i\} \\
s'(\textbf{case}\ e\ \textbf{of}\ \overline{(v_1, \ldots v_m) \rightarrow e'}) = s'(e) + s'(e') + m + (p_t(e) + \sum_i A_{t_i}) \quad \{i \in \{1..m\}, v_i :: t_i, e :: t\} \\
s'(\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) = s'(e) + \sum_{i,j}(m_{ij} + \mathcal{H}_{t_{r_{ij}}}) + \sum_i s'(e_i) + (m-1)\ \mathcal{H}_t \\
\qquad \{i \in \{1..m\}, j \in \{1..m_i\}, e_i :: t, v_{ij} :: t_{ij}, t_{rij} = rType(t_{ij}), m_{ij} = nArgs(t_{ij})\} \\
s'(\Lambda\beta.e) = s'(e) \\
s'(e\ t) = s'(e) + \Gamma^{\beta}_{t'\ t} \\
\qquad \{e :: \forall\beta.t'\}
\end{array}
$$

Fig. 36. The cost of $[\![e]\!]'\ \rho$

$$
\begin{array}{l}
body :: Expr \rightarrow Var \rightarrow ([Expr], Int) \\
body\ e\ x = body'\ [e]\ [x]\ 0 \\
body' :: [Expr] \rightarrow [Var] \rightarrow Int \rightarrow ([Expr], Int) \\
body'\ [\,]\ xs\ c = ([\,], c) \\
body'\ (e : es)\ xs\ c = (es1 {+\!\!+} ess, c_2) \\
\quad \text{where } (es1, c_1) = lbody\ e\ xs\ c \\
\qquad\qquad (ess, c_2) = body'\ es\ xs\ c_1 \\
lbody :: Expr \rightarrow [Var] \rightarrow Int \rightarrow ([Expr], Int) \\
lbody\ e\ [\,]\ c = ([e], c) \\
lbody\ (\lambda v.e)\ (x : xs)\ c = lbody\ e[x/v]\ xs\ c \\
lbody\ (e\ x)\ xs\ c = lbody\ e\ (x : xs)\ c \\
lbody\ (\textbf{case}\ e\ \textbf{of}\ \overline{(v_1, \ldots, v_m) \rightarrow e'})\ xs\ c = lbody\ e'\ xs\ c \\
lbody\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i})\ xs\ c = body'\ [e_1, \ldots, e_m]\ xs\ (c + (m-1)\ \mathcal{H}_t) \quad \{i \in \{1..m\}, e_i :: t\} \\
lbody\ (\textbf{let}\ bs\ \textbf{in}\ e)\ xs\ c = lbody\ e\ xs\ c \\
lbody\ v\ xs\ c = lbody\ e\ xs\ c \quad \{\text{let-nonrec bound}\}\ \{v = e\} \\
lbody\ v\ xs\ c = ([v], c') \quad \{\text{lambda, let-rec or alg.case bound}\} \\
\quad \text{where } c' = c + \sum_{i=1}^{j} p_{t_i}(x_i) + (m-j) + \mathcal{H}_{t_r}, x_i = xs!!(i-1) \\
\qquad \{j = length\ xs\ , v :: t, t_r = rType(t)\} \\
lbody\ v\ xs\ c = body'\ es_k\ xs\ c' \quad \{k\text{th tuple case bound, discriminant } e\} \\
\quad \text{where} \\
\qquad (es, cs) = comps(e), es_k = es!!(k-1), n_k = cs!!(k-1), \\
\qquad c' = c + n_k\ \sum_{i=1}^{j} p_{t_i}(x_i) + (m-j) + \mathcal{H}_{t_r}, x_i = xs!!(i-1) \\
\qquad \{v :: t, m = nArgs(t), t_r = rType(t), j = length\ xs, i \in \{1..j\}, x_i :: t_i\}
\end{array}
$$

Fig. 37. Definition of *body*

$$
\begin{array}{l}
oneap\ (\lambda v.e) = skip\lambda(e) + 1 \\
oneap\ (e\ x) = c + sum\ (map\ oneap\ es) + (m-1)\ \mathcal{H}_t \quad \{e\ x :: t, m = length\ es\} \\
\quad \text{where } (es, c) = body\ e\ x \\
oneap\ (\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e') = oneap(e') \\
oneap\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) = \sum_i oneap\ (e_i) + (m-1)\ \mathcal{H}_t \quad \{i \in \{1..m\} \\
oneap\ (\textbf{let}\ bs\ \textbf{in}\ e) = oneap\ (e) \\
oneap\ (v) = oneap\ \ (e) \quad \{\text{let-nonrec bound}\}\ \{v = e\} \\
oneap\ (v) = \sum_{i=1}^{m} \mathcal{H}_{t_i} \quad \{\text{lambda, let-rec or alg.case bound}\} \\
\quad \text{where } \{v :: t, m = nArgs(t), [t_1, \ldots, t_m] = aTypes(t)\} \\
oneap\ (v) = sum\ (map\ oneap\ es) + c' \quad \{k\text{th tuple case bound, discriminant } e\} \\
\quad \text{where } \ (es, cs) = comps(e), es_k = es!!(k-1), n_k = cs!!(k-1), c' = n_k\ \sum_{j=1}^{m} \mathcal{H}_{t_j} \\
\qquad \{v :: t, [t_1, \ldots, t_m] = aTypes(t), j \in \{1..m\}\} \\
\\
skip\lambda\ (\lambda v.e) = skip\lambda\ (e) + 1 \\
skip\lambda\ (e\ x) = s'(e\ x) - sum\ (map\ s'\ es) + c' \\
\quad \text{where } c' = sum\ (map\ skip\lambda\ es) \\
\qquad (es, c) = body\ e\ x, m = length\ es \\
skip\lambda\ (\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e') = s'(\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e') - s'(e') + skip\lambda(e') \\
skip\lambda\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) = s'(\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) - \sum_i s'(e_i) + \sum_i skip\lambda\ (e_i) \\
skip\lambda\ (\textbf{let}\ bs\ \textbf{in}\ e) = s'(\textbf{let}\ bs\ \textbf{in}\ e) - s'(e) + skip\lambda(e) \\
skip\lambda(x_1, \ldots, x_m) = m + p_{(t_1, \ldots, t_m)}(x_1, \ldots, x_m) \quad \{x_i :: t_i\} \\
skip\lambda\ e = s'(e)
\end{array}
$$

Fig. 38. Cost of one probing *oneap*

$$
\begin{array}{l}
s(k) = s(v) = 1 \\
s(x_1, \ldots x_m) = m \\
s(C\ x_1 \ldots x_m) = m \\
s(op\ x_1 \ldots x_m) = \sum_i \mathcal{H}_{t_i} \\
\qquad \{op :: top, m = nArgs(t), [t_1, \ldots, t_m] = aTypes(t), t_r = rType(t), i \in \{1..(m+1)\}\} \\
s(\lambda v.e) = s(e) + p_t(\lambda v.e) + (m_1 + \mathcal{H}_{t_1}) \\
\qquad \{v :: t_1, m_1 = nArgs(t_1), \lambda v.e :: t\} \\
s(e\ x) = s(e) + s(x) + (sum\ (map\ s'\ es) + c) + p_t(e\ x) \quad \{e\ x :: t\} \\
\quad \text{where } (es, c) = body\ e\ x \\
s(\textbf{let}\ v = e\ \textbf{in}\ e') = s(e) + s(e') + a_t(e) \\
s(\textbf{let rec}\ \overline{v_i = e_i}\ \textbf{in}\ e') = niter * \sum_i (s'(e_i) + p_{t_i}(e_i) + a_{t_i}(e_i) + 1) + \sum_i s(e_i) + s(e') \\
\quad \text{where } \ niter = \sum_{i=1}^{m} \mathcal{H}_{t_i} \\
\qquad \{i \in \{1..m\}, e_i :: t_i\} \\
s(\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots v_m) \to e') = s(e) + s(e') + m + (p_t(e) + \sum_i A_{t_i}) \quad \{i \in \{1..m\}, v_i :: t_i, e :: t\} \\
s(\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) = s(e) + \sum_{i,j} (m_{ij} + \mathcal{H}_{t_{r_{ij}}}) + \sum_i s(e_i) + (m-1)\ \mathcal{H}_t \\
\qquad \{i \in \{1..m\}, j \in \{1..m_i\}, e_i :: t, v_{ij} :: t_{ij}, m_{ij} = nArgs(t_{ij}), t_{rij} = rType(t_{ij})\} \\
s(\Lambda\beta.e) = s(e) \\
s(e\ t) = s(e) + \Gamma^{\beta}_{t'\ t} + p_{tinst}(e\ t) \\
\qquad \{e :: \forall\beta.t', e\ t :: tinst\}
\end{array}
$$

Fig. 39. The cost of $[\![e]\!]\ \rho$

References

[Bar93] G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions.* PhD thesis, University of Glasgow, February 1993.

[BHA86] G. L. Burn, C. L. Hankin, and S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, October 1986.

[BLOP98] 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.

[CC77] 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, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CC79] 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, San Antonio, Texas, January 1979*, pages 269–282. ACM, 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CP85] C. Clack and S. L. Peyton Jones. Strictness analysis - A practical approach. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 35–49. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[HH92] 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 *Lecture Notes in Computer Science*, pages 219–232, Rennes, France, February 26–28, 1992. Springer, Berlin.

[HO90] 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, 21-23 August 1989*, pages 308–328, London, UK, 1990. Springer-Verlag.

[Hug95] J. Hughes. The design of a pretty-printing library. *Lecture Notes in Computer Science*, 925:53–96, 1995.

[JC87] 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.

[JPS96] S. L. Peyton Jones, W. Partain, and A. L. M. Santos. Let-floating: moving bindings to give faster programs. *International Conference on Functional Programming ICFP'96*, May 1996.

[KPR00] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. 2nd International Workshop on Constructive Methods for Parallel Programming (CMPP 2000). To be published by Nova Science, 2000.

[MH87] J. C. Martin and C. Hankin. Finding fixed points in finite lattices. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 426–445. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[PHH+93] 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.

[PP93] 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.

[PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Volume 2). Proceedings of 2nd Scottish Functional Programming Workshop, SFP'00*, pages 13–26. Intellect, 2000.

[PS98] 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.

[PS00] R. Peña and C. Segura. Two Non-determinism Analyses in Eden. Technical Report 108-00, 46 pages. Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, 2000. (It can be found in http://dalila.sip.ucm.es/miembros/ clara/publications.html).

[PS01] R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *Selected papers of the 12th International Workshop on Implementation of Functional Languages, IFL00*, pages 1–18. LNCS 2011, Springer-Verlag, 2001.

[San95] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Department of Computing Science. University of Glasgow, 1995.

[SS90] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, May 1990.

[SS92] H. Søndergaard and P. Sestoft. Non-Determinism in Functional Languages. *Computer Journal*, 35(5):514–523, October 1992.