Correctness of Non-determinism Analyses in a Parallel-Functional Language*

Clara Segura and Ricardo Peña

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

Abstract. The presence of non-determinism in the parallel-functional language Eden creates some problems. Several non-determinism analyses have been developed to determine when an Eden expression is sure to be deterministic, and when it may be non-deterministic. The correctness of these analyses had not been proved yet. In this paper we define a "maximal" denotational semantics for Eden in the sense that the set of possible values produced by an expression is bigger than the actual one. This semantics is enough to prove the correctness of the analyses. We provide the abstraction and concretisation functions relating the concrete and abstract values so that the determinism property is adequately captured. Finally we prove the correctness of the analyses with respect to the previously defined semantics.

1 Introduction

The presence of non-determinism in the parallel-functional language Eden creates some problems: It affects the referential transparency of programs [11] and invalidates some optimizations done in the Glasgow Haskell Compiler (GHC) [10]. Three non-determinism abstract interpretation based analyses have been defined to determine when an Eden expression is sure to be deterministic, and when it may be non-deterministic [7, 8]. They have been formally related and compared with respect to expresiveness and efficiency [5].

However the correctness of these analyses had not been proved yet as there was no appropriate denotational semantics for Eden including non-determinism. Very recently it has been published in our group a complete denotational semantics [3] for Eden based on continuations. There, non-determinism is expressed by the fact that, after evaluating an expression, a process may arrive to *a set* of different states, so that several continuations are possible. Unfortunately this semantics is not still appropriate for our purposes: On the one hand it provides lots of details that would obscure the proof of correctness. On the other, the set of states a process may arrive to do not constitute a mathematical domain and this is essential when abstract interpretation is used.

So, the first contribution of this paper is the definition of an appropriate denotational semantics, in one sense simpler and in another sense more complex than that of [3]. Moreover, as concurrency and parallelism aspects are abstracted away, the non-determinism analyses would also be correct for any non-deterministic functional language whose semantics is (upper) approximated by this one. It

* Work partially supported by the Spanish project TIC 2000-0738.

is a *plural* semantics in the style of [12] but with higher order and algebraic types incorporated. The domains of values are defined by means of Hoare powerdomains considering that the behaviour of the non-deterministic operator is near to angelic non-determinism. To our knowledge, this is the first time that a powerdomain-based non-deterministic semantics including higher-order values is defined. It is not the actual semantics of Eden but an upper approximation to it in the sense that, if an Eden expression e may evaluate to value v, then v is included in the set s denoted by e in the semantics, but s may include values that the implementation will never arrive to. However, this semantics is enough to prove the correctness of the analyses.

The second contribution of the paper is the proof of correctness itself. We provide the abstraction and concretisation functions relating the concrete and abstract values so that the determinism property is adequately captured. We prove that they form a Galois connection and then we prove the correctness of the analyses with respect to the semantics. The techniques we use are rather standard in the abstract interpretation area but the problem addressed —non-determinism analysis with functional domains, denotational semantics with Hoare higher-order powerdomain— and the proof itself are new.

The plan of the paper is as follows. In Section 2 we describe Eden and the non-determinism analyses that have been defined for it. In Section 3 we present the denotational semantics including non-determinism. Finally, in Section 4 correctness of the analyses is formally proved.

2 Non-determinism Analyses for Eden

2.1 Eden in a nutshell

The parallel-functional language Eden extends the lazy functional language Haskell by constructs to explicitly define and communicate processes. The three main new concepts are *process abstractions*, *process instantiations* and the non-deterministic process abstraction `merge`.

A *process abstraction* expression `process x -> e` of type `Process a b` defines the behaviour of a process having the formal parameter `x::a` as input and the expression `e::b` as output. An instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Process instantiations can be compared to function applications: Each time an expression `e1 # e2` is evaluated, a new parallel process is created to evaluate `(e1 e2)`.

The evaluation of an expression `e1 # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* will evaluate first the expression `e1` until a process abstraction `process x -> e` is obtained and then the application `(\ x -> e) e2`, returning the result via another implicitly generated channel. The instantiation protocol deserves some attention: (1) Closure `e1`

together with the closures of all the free variables referenced there (its whole environment) are *copied*, in the current evaluation state (possibly unevaluated), to a new processor, and the child process is created there to evaluate the expression `(\ x -> e) e2`, where `e2` must be remotely received. (2) Expression `e2` is eagerly evaluated in the parent process to normal form. The result is communicated to the child process as its input argument. (3) The normal form of the value `(\ x -> e) e2` is sent back to the parent. Normal forms are full, except for lambdas where they are weak ones. For input or output tuples, independent concurrent threads are created to evaluate each component.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access input which is not available yet, are temporarily suspended. This is the only way in which Eden processes synchronize.

Lazy evaluation is changed to eager evaluation in two cases: Processes are eagerly instantiated, and instantiated processes produce their output even if it is not demanded. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation. The rest of the language is as lazy as Haskell is. In general, a process is implemented by several threads concurrently running in the same processor, so that different values can be produced independently. The concept of a virtually shared global graph does not exist. Each process evaluates its outputs autonomously.

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. Its implementation immediately copies to the output list any value appearing at any of the input lists. So, `merge` can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. This feature is essential in reactive systems and very useful in some deterministic parallel algorithms. Eden is aimed at both types of applications.

2.2 A simplified language

In the next section a denotational semantics is defined for a simplified version of Eden, see Figure 1, in order to prove the correctness of several non-determinism analyses. The language is an extended simplification of Core-Haskell [9], a simple functional language with second-order polymorphism. As Eden is an extension of Haskell, it is obviously polymorphic. But in order to simplify the rest of the paper, we have removed this aspect of the language. So there are neither type abstractions nor type applications.

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 type $(t_1, \ldots, t_m)$, an algebraic (sum) type T[1], or a functional type $t_1 \to t_2$.

[1] Defined by **data** $T = C_1\ t_{11} \ldots t_{1n_1} \mid \ldots \mid C_m\ t_{m1} \ldots t_{mn_m}$.

$$
\begin{array}{lll}
\mathit{prog} \rightarrow \mathit{bind}_1; \ldots; \mathit{bind}_m & & \\
\mathit{bind} \rightarrow \quad v = \mathit{expr} & & \{\text{non-recursive binding}\} \\
\quad\quad | \ \mathbf{rec}\ v_1 = \mathit{expr}_1; \ldots; v_m = \mathit{expr}_m & & \{\text{recursive binding}\} \\
\mathit{expr} \rightarrow \quad \mathit{expr}\ x & & \{\text{application to an atom}\} \\
\quad\quad | \ \lambda v.\mathit{expr} & & \{\text{lambda abstraction}\} \\
\quad\quad | \ \mathbf{case}\ \mathit{expr}\ \mathbf{of}\ \mathit{alts} & & \{\mathit{case}\ \text{expression}\} \\
\quad\quad | \ \mathbf{let}\ \mathit{bind}\ \mathbf{in}\ \mathit{expr} & & \{\mathit{let}\ \text{expression}\} \\
\quad\quad | \ (x_1, \ldots, x_m) & & \{\text{tuple}\} \\
\quad\quad | \ C\ x_1 \ldots x_m & & \{\text{saturated constructor application}\} \\
\quad\quad | \ x & & \{\text{atom: variable } v \text{ or literal } k\} \\
\quad\quad | \ \mathit{merge}_t & & \{\text{non-determinism operator}\} \\
\mathit{alts} \rightarrow \quad \mathit{Calt}_1; \ldots; \mathit{Calt}_m; & m \geq 0 & \\
\quad\quad | \ \mathit{TAlt} & & \\
\mathit{TAlt} \rightarrow (v_1, \ldots, v_m) \rightarrow \mathit{expr} & m \geq 0 & \{\text{tuple alternative}\} \\
\mathit{Calt} \rightarrow C\ v_1 \ldots v_m \rightarrow \mathit{expr} & m \geq 0 & \{\text{algebraic alternative}\}
\end{array}
$$

Fig. 1. A simplified version of a parallel functional language

Process abstractions **process** $v \rightarrow e$ and process instantiations $e \ \# \ x$ do not appear in the language. This simplification is motivated by an approximation to the semantics explained in Section 3.2. When an unevaluated non-deterministic free variable is duplicated in two different processes, it may happen that the actual value computed by each process is different. However, within the same process, a variable is evaluated at most once and its value is shared thereafter. Consequently this means that variables are *definite* (each occurrence denotes the same single value) within the same process and are not definite (different occurrences may denote different values) within different processes. In general, in Eden the *unfoldability* property does not hold (a variable cannot be replaced by its definition, i.e. $[\![(\lambda x.e)\ e']\!]\ \rho \neq [\![e[e'/x]]\!]\ \rho$), except in the case that the unfolded expression is deterministic. This is a consequence of having definite variables within a process.

So, there are some occurrences that surely have the same value but others may have different values. The following example illustrates this situation. Assume *ne* is a non-deterministic expression in

$$\mathbf{let}\ v = ne\ \mathbf{in}\ (p_1\ v)\#v + (p_2\ v)\#v$$

The second and fourth occurrences of v necessarily have the same value as they are evaluated in the parent process. However the first and third occurrences may have different values as v is copied twice and evaluated in two children processes. So, an upper approximation is obtained by considering that

- All the occurrences of each variable may have a different value, i.e. all the variables are non-definite.
- All functions behave as processes, and all function applications behave as process instantiations. Consequently, we will only have syntactical lambda abstractions and function applications with the semantics of process abstractions and process instantiations.

The semantics defined in Section 3.2 will make these assumptions.

As polymorphism is omitted, the `merge` operator is monomorphic, so we consider the existence of an instance $merge_t$ for every type t. Additionally we simplify this operator so that it merges just two lists of values: $merge_t : [t] \to [t] \to [t]$. Eden's `merge` is more convenient since it may receive as arguments any finite number of lists, but it can be simulated by the simplified one, $merge_t$.

2.3 Motivation for the analyses

The non-deterministic process `merge` may be used to create non-deterministic expressions and to define non-deterministic functions. Subsection 2.4 introduces several analyses to detect at compile time these non-deterministic expressions. The analyses annotate the expressions with a mark which, in the simplest case is just d or n. The first one means that the expression is *sure* to be deterministic, while the second one means that it *may be* non-deterministic. So, a possible better name for these analyses would be *determinism* analyses because the sure value is the deterministic one. We found at least three motivations for developing these analyses.

On the one hand, to annotate the places in the text where equational reasoning may be lost due to the presence of non-determinism. This is important in an optimizing compiler such as that of Eden built on top of GHC [9]. A lot of internal transformations such as *inlining* or *full laziness* are done on the assumption that it is always possible to replace equals by equals. This is not true when the expressions involved are non-deterministic. For instance, the full laziness transformation moves a binding out of a lambda when it does not depend on the lambda argument. So, the expression

$$\begin{array}{l}\textbf{let } f = \lambda x.\ \textbf{let } y = e_1 \textbf{ in } e_2\\ \textbf{in } \ e_3\end{array}$$

when e_1 does not depend on x is transformed to

$$\begin{array}{l}\textbf{let } y = e_1\\ \textbf{in } \ \textbf{let } f = \lambda x.e_2 \textbf{ in } e_3\end{array}$$

If e_1 is non-deterministic, this transformation restricts the set of values the expression may evaluate to, as now expression e_1 is evaluated only once instead of many times.

A second motivation is to be able to implement in the future a semantics for Eden, different from the currently implemented one, in which all variables will be guaranteed to be definite, i.e. they will denote the same value in all the processes. To this aim, when a non-deterministic binding is to be copied to a newly instantiated process, the runtime system will take care of previously evaluating the binding to normal form. Doing this evaluation for all bindings would make Eden more eager than needed and would decrease the amount of parallelism as more work would be done in parent processes. So, it is important to do this evaluation only when it is known that the binding is possibly non-deterministic.

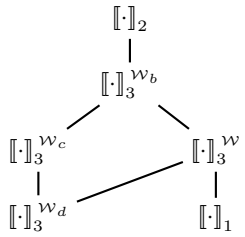

Fig. 2. A hierarchy of analyses

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

Fig. 3. Abstract domains for the second analysis

A third motivation could be to be able to inform the programmer of the deterministic expressions of the program. In this way, the part of the program where equational reasoning is still possible would be clearly determined. A first step towards this aim is doing the analysis at the core language level. A translation of the annotations to source level would also be required in order to provide the programmer with meaningful information. For the moment we have not implemented this translation.

2.4 A hierarchy of analyses

Three non-determinism analyses have been developed to determine when an Eden expression is sure to be deterministic and when it may be non-deterministic. In [7], two different abstract interpretation based analyses were presented and compared with respect to expressiveness 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). In [8] an intermediate analysis $[\![\cdot]\!]_3$ and its implementation (written in Haskell) were described. Such analysis is a compromise between power and efficiency (cubic). 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 $[\![\cdot]\!]_3^{wop}$. That paper described one particular variant $[\![\cdot]\!]_3^{\mathcal{W}}$ in detail.

In [5], the three analyses were formally related so that they become totally ordered by increasing cost and precision. It was shown that all variants of the third analysis are safe approximations to the second analysis and that the first analysis is only a safe aproximation to those variants of the third analysis satisfying a particular property. An example was given to show the differences in precision between $[\![\cdot]\!]_1$, $[\![\cdot]\!]_2$ and $[\![\cdot]\!]_3^{\mathcal{W}}$. In Figure 2 we show the relation between the first and second analyses, and some variants of the third one.

In this paper we only summarize the second analysis as we are going to prove its correctness with respect to the Eden semantics. The previous results lead us to correctness of the whole hierarchy of analyses with respect to it.

In Figure 3 the abstract domains for $[\![\cdot]\!]_2$ are shown. There is a domain *Basic* with two values: d represents *determinism* and n *possible non-determinism*, with

$$
\begin{array}{l}
[\![v]\!]_2\ \rho_2 = \rho_2(v) \\
[\![k]\!]_2\ \rho_2 = d \\
[\![(x_1,\ldots,x_m)]\!]_2\ \rho_2 = ([\![x_1]\!]_2\ \rho_2,\ldots,[\![x_m]\!]_2\ \rho_2) \\
[\![C\ x_1 \ldots x_m]\!]_2\ \rho_2 = \bigsqcup_i \phi_{t_i}([\![x_i]\!]_2\ \rho_2) \text{ where } x_i :: t_i \\
\\
[\![e\ x]\!]_2\ \rho_2 = ([\![e]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \text{ where } v :: t_v \\
[\![merge_t]\!]_2\ \rho_2 = \lambda z_1 \in Basic.\lambda z_2 \in Basic.n \\
[\![\textbf{let } v = e \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \mapsto [\![e]\!]_2\ \rho_2] \\
[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ (\mathit{fix}\ (\lambda\rho'_2.\rho_2\ \overline{[v_i \mapsto [\![e_i]\!]_2\ \rho'_2]})) \\
[\![\textbf{case } e \textbf{ of } (v_1,\ldots,v_m) \to e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ \overline{[v_i \mapsto \pi_i([\![e]\!]_2\ \rho_2)]} \\
[\![\textbf{case } e \textbf{ of } \overline{C_i\ \overline{v_{ij}} \to e_i};]\!]_2\ \rho_2 = \begin{cases} \mu_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i} \text{ otherwise} \end{cases} \\
\qquad\qquad \text{where } \rho_{2i} = \rho_2\ \overline{[v_{ij} \mapsto \mu_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t
\end{array}
$$

Fig. 4. Abstract interpretation $[\![\cdot]\!]_2$

$$
\begin{array}{ll}
\phi_t : D_{2t} \to Basic & \mu_t : Basic \to D_{2t} \\
\phi_K = \phi_T = id_{Basic} & \mu_K = \mu_T = id_{Basic} \\
\phi_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \bigsqcup_i \phi_{t_i}(e_i) & \mu_{(t_1,\ldots,t_m)}(b) = (\mu_{t_1}(b),\ldots,\mu_{t_m}(b)) \\
\phi_{t_1 \to t_2}(f) = \phi_{t_2}(f(\mu_{t_1}(d))) & \mu_{t_1 \to t_2}(b) = \begin{cases} \lambda z \in D_{2t_1}.\mu_{t_2}(n) \text{ if } b = n \\ \lambda z \in D_{2t_1}.\mu_{t_2}(\phi_{t_1}(z)) \text{ if } b = d \end{cases}
\end{array}
$$

Fig. 5. Functions ϕ_t and μ_t

the ordering $d \sqsubseteq n$. This is the abstract domain corresponding to basic types and algebraic types. The abstract domains corresponding to a tuple type and a function/process type are respectively the cartesian product of the components' domains and the domain of continuous functions between the domains of the argument and the result. In [7] polymorphism was also included, but in this paper we do not treat it.

In Figure 4 the analysis is shown. It is an abstract interpretation based analysis in the style of [1]. We outline here only some cases. The interpretation of a tuple is the tuple of the abstract values of the components. Functions are interpreted as abstract functions. So, applications are interpreted as abstract functions applications. The interpretation of a constructor application belongs to *Basic*, obtained as the least upper bound (lub) of the components' abstract values. But each component $x_i :: t_i$ has an abstract value belonging to D_{2t_i}, that must be first *flattened* to a basic abstract value. This is done by a function called the *flattening function* $\phi_t : D_{2t} \to Basic$, defined in Figure 5. The idea is to flatten the tuples (by applying the lub operator) and to apply the functions to deterministic arguments.

In a recursive **let** expression the fixpoint can be calculated by using Kleene's ascending chain. We have two different kinds of *case* expressions (for tuple and algebraic types). The more complex one is the algebraic *case*. Its abstract value is non-deterministic if either the discriminant or any of the expressions in the alternatives is non-deterministic. Note that the abstract value of the discriminant e, let us call it b, belongs to *Basic*. That is, when it was interpreted, the

$A_K = \mathcal{P}([\![K]\!])$ where $[\![Int]\!] = \mathbb{Z}_\perp$
$A_{(t_1,\ldots,t_m)} = A_{t_1} \times \ldots \times A_{t_m}$
$A_T = \mathcal{P}([\![T]\!])$
where $[\![T]\!] = \oplus_{i=1}^{m}(C_i \times \times_{j=1}^{n_i} A_{t_{ij}})_\perp$, **data** $T = C_1\ t_{11} \ldots t_{1n_1} \mid \ldots \mid C_m\ t_{m1} \ldots t_{mn_m}$
$A_{t_1 \to t_2} = [A_{t_1} \to A_{t_2}]$

Fig. 6. Domain of values

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 *unflattening function* $\mu_t : Basic \to D_{2t}$ defined in Figure 5. Given a type t, it *unflattens* a basic abstract value and produces an abstract value in D_{2t}. The idea is to obtain the best safe approximation both to d and n in a given domain. The flattening and unflattening functions are mutually recursive. In [7] they were explained in detail and an example was given to illustrate their definitions. They have some interesting properties (e.g. they are a Galois insertion pair [2]), studied in [5]. Tuples are treated separately from algebraic types because we want the analysis to be more precise here due to the use of tuples in Eden as input or output channels of processes.

3 A Denotational Semantics for Non-determinism

3.1 The domain of values

To capture the idea of a non-deterministic value, the traditional approach is to make an expression to denote a *set* of values. This is obvious for basic types such as integers, but things get more complex when we move to structured types such as functions or tuples. Should a functional expression denote a set of functions or a function from sets to sets? Should a tuple expression denote a set of tuples or a tuple of sets? Additionally, the denoted values should constitute a domain. In the literature, three powerdomains with different properties have been proposed: Hoare, Smyth and Plotkin powerdomains [12]. The first one models *angelic* or bottom-avoiding nondeterminism (in which bottom is never chosen unless it is the only option), the second one models *demonic* non-determinism (it chooses bottom whenever it is a possible option) and the third one models *erratic* non-determinism (in which bottom is an option as the other ones).

Regarding structured domains we have chosen a functional expression to denote a single function from sets to sets. In this sense, the following two bindings

$$f_1 = head(merge_{Int \to Int}[\lambda x.0][\lambda x.1])$$
$$f_2 = \lambda x.head(merge_{Int}[0][1])$$

will both denote the function $\lambda x.\{0, 1, \perp\}$. That is, the information whether the non-deterministic decision is taken at binding evaluation time or at function application time is lost. Non-deterministic decisions are deferred as much as

possible; in this example to function application time. This is consistent with the plural semantics we have adopted for our language in Section 3.2: Several occurrences of the same variable (let us say f_1) may represent different values.

Regarding the selection of powerdomain, we have decided to use Hoare's one. This is consistent with the implementation of `merge` in Eden: If one of the input lists is blocked (i.e., it denotes $\perp$), `merge` will still produce an output list by copying values from the non-blocked list. Only if both lists are blocked will the output list be blocked. Nevertheless, `merge` will terminate only when both input lists terminate. This behaviour is very near to angelic non-determinism. If D is a domain, $\mathcal{P}(D)$ will denote the Hoare powerdomain of D. First, a preorder relation is defined in $P(D)$ (all subsets of D) as follows:

$$A \sqsubseteq_{P(D)} B \quad \textit{iff} \quad \forall a \in A. \exists b \in B. a \sqsubseteq_D b$$

This preorder relation induces an equivalence relation $\equiv \stackrel{\text{def}}{=} \sqsubseteq \cap \sqsupseteq$ identifying sets such as $\{0, 1, \perp\}$ and $\{0, 1\}$. The Hoare powerdomain is the quotient $\mathcal{P}(D) \stackrel{\text{def}}{=} (P(D) - \emptyset)/\equiv$. A property enjoyed by all elements of a Hoare powerdomain is that they are downwards closed, i.e. $\forall x \in A. y \sqsubseteq_D x \Rightarrow y \in A$.

In Figure 6, the domains of semantic values for every type are defined. Notice that, for basic and constructed types, the domains consist of sets of values while for tuples and functions, the domains consist of single values. In the definition for constructed types, $\oplus$ denotes the coalesced sum of (lifted) domains. Sets of values are needed for the constructed types because non-deterministic values of such types may contain several different constructors. However, those with only one constructor could be treated as tuples.

If the constructed type is recursive, notice that the recursive occurrences denote sets of values. For instance, a non-deterministic list would consist of a set of lists. A non-empty list of this set would consist of a head value and a tail value formed by a set of lists. Note also that the domain allows the existence of infinite values as limits of their finite approximations.

3.2 A maximal semantics: Non-definite variables

In Figure 7 a denotational semantics for Eden is given. There $\{v\}^*$ denotes the downwards closure of a value, i.e. a set of values containing all values below v. The environment ρ maps variables of type t to values of their corresponding non-deterministic domains A_t. The semantic function $[\![.]\!]$ maps an expression of type t and an environment ρ to a value in A_t. The only expression introducing sets of values is $merge_t$. Its behaviour is that of a lambda abstraction returning all the possible interleavings of all pairs of input lists. The detail of the auxiliary function *mergeS* is given in Figure 8.

These decisions configure a plural semantics for Eden as every occurrence of the same variable within an expression is mapped to *all* possible values for that variable (see definitions for **let** and lambda in Figure 7). This is not the actual semantics of Eden, but just a safe upper approximation to it in the sense that the set of possible values denoted by an expression is bigger than the actual one. As

$[\![v]\!]\ \rho = \rho(v)$
$[\![k]\!]\ \rho = \{k\}^*$
$[\![(x_1,\ldots,x_m)]\!]\ \rho = ([\![x_1]\!]\ \rho,\ldots,[\![x_m]\!]\ \rho)$
$[\![C\ x_1 \ldots x_m]\!]\ \rho = \{C\ [\![x_1]\!]\ \rho \ldots [\![x_m]\!]\ \rho\}^*$
$[\![\lambda v.e]\!]_2\ \rho = \lambda s \in A_{t_v}.[\![e]\!]\ \rho\ [v \mapsto s]$ where $v :: t_v$
$[\![e\ x]\!]\ \rho = ([\![e]\!]\ \rho)\ ([\![x]\!]\ \rho)$
$[\![merge_t]\!]\ \rho = \lambda s_1 \in A_{[t]}.\lambda s_2 \in A_{[t]}.\bigcup\{mergeS\ l_1\ l_2 \mid l_1 \in s_1, l_2 \in s_2\}$
$[\![\textbf{let}\ v = e\ \textbf{in}\ e']\!]\ \rho = [\![e']\!]\ \rho\ [v \mapsto [\![e]\!]\ \rho]$
$[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]\ \rho = [\![e']\!]\ (\mathit{fix}\ (\lambda\rho'.\rho\ \overline{[v_i \mapsto [\![e_i]\!]\ \rho']}))$
$[\![\textbf{case}\ e\ \textbf{of}\ (v_1,\ldots,v_m) \to e']\!]\ \rho = [\![e']\!]\ \rho\ \overline{[v_i \mapsto \pi_i([\![e]\!]\ \rho)]}$
$[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i};]\!]\ \rho = \begin{cases} \bot_{A_t} \text{ if } [\![e]\!]\ \rho = \bot_{A_T} \\ \bigsqcup_{A_t}\{[\![e_k]\!]\ \rho\overline{[v_{kj} \mapsto s_{kj}]}^{m_k} \mid C_k\ \overline{s_{kj}}^{m_k} \in [\![e]\!]\ \rho\} \text{ otherwise} \end{cases}$

Fig. 7. A denotational semantics for Eden

$mergeS\ \bot\ \bot = \{\bot\}$ $\quad mergeS\ \bot\ l_2 = \{l_2 +\!\!+ \bot\}^*$ $\quad mergeS\ l_1\ \bot = \{l_1 +\!\!+ \bot\}^*$
$mergeS\ [\,]\ [\,] = \{[\,]\}^*$ $\quad mergeS\ [\,]\ l_2 = \{l_2\}^*$ $\quad mergeS\ l_1\ [\,] = \{l_1\}^*$
$mergeS\ (s_1 : ls_1)\ (s_2 : ls_2) = \{s_1 : (\bigcup_{l' \in ls_1} mergeS\ l'\ (s_2 : ls_2)), s_2 : (\bigcup_{l' \in ls_2} mergeS\ (s_1 : ls_1)\ l')\}^*$
where $\bot +\!\!+ \bot = \bot$
$[\,] +\!\!+ \bot = \bot$
$(xs : xss) +\!\!+ \bot = xs : \{xss' +\!\!+ \bot \mid xss' \in xss\}$

Fig. 8. Non-determinism semantics

an example, the expression **let** $f = head(merge_{Int \to Int}[\lambda x.0]\ [\lambda x.1])$ **in** $(f\ 3) + (f\ 4)$ in fact may only produce the values 0 or 2 while the approximated semantics will say that it may also produce the value 1. It is *maximal* in the sense that all variables are considered non definite, while in the actual semantics only those variables duplicated in different processes may be non definite if they are non-deterministic. Notice that with this approximated semantics unfoldability holds although in the actual semantics this is not true. The denotation given to $merge_t$ is also an upper approximation as the actual one only produces fair interleavings.

The reason for this maximal semantics is that, if we are able to show the correctness of the analysis with respect to it, then the analysis will be correct with respect to the actual semantics. We remind the reader that the sure value is the deterministic one. If the analysis detects an expression as deterministic then it should be semantically deterministic.

An exception is the algebraic **case** expression where the variables in the right hand side of the alternatives are definite. The discriminant's value is a set that may contain different constructors, so we have to take the lub of all the alternatives' values that match them. As the discriminant is immediately evaluated, the non-deterministic decision is immediately taken so that all the occurrences of the same variable in the right hand side have the chosen value.

For example, let a type **data** $Fool = C\ Int \mid C'\ Int$ and the values $s_1 = \{\bot, C\{0,\bot\}, C'\{0,\bot\}\}$, $s_2 = \{\bot, C\{1,\bot\}, C\{0,\bot\}\}$ and $s_2' = \{\bot, C\{1,\bot\}, C\{0,\bot\}, C\{0,1,\bot\}\}$. Let an expression $e' =$ **case** e **of** $C\ v \to v + v; C'\ v' \to v' + 4$. If $[\![e]\!]\ \rho = s_1$, then $[\![e']\!]\ \rho = \{0, 4, \bot\}$. Notice that s_2 and s_2' are different: If $[\![e]\!]\ \rho = s_2$

$$
\begin{array}{l}
det_K(s) = unit(s) \\
\quad \text{where} \;\; unit(\{\bot\}) = true \qquad unit(\{z, \bot\}) = true \qquad unit\_ = false \\
det_{(t_1,\ldots,t_m)}((s_1,\ldots,s_m)) = \bigwedge_{i=1}^{m} det_{t_i}(s_i) \\
det_T(s) = \begin{cases} \bigwedge_{i=1}^{m} det_{t_i}(\sqcup\{s_i \mid C\ s_1 \ldots s_m \in s, s_i :: t_i\}) \text{ if } one(s) \\ false \quad \text{otherwise} \end{cases} \\
\quad \text{where} \;\; one(s) = (s = \{\bot\}) \vee (\exists C. \forall s' \in s. s' \neq \bot \Rightarrow s' = C\ s_1 \ldots s_m) \\
det_{t_1 \to t_2}(f) = \forall s \in A_{t_1}. det_{t_1}(s) \Rightarrow det_{t_2}(f(s))
\end{array}
$$

Fig. 9. Semantic definition of determinism

then $[\![e']\!]\ \rho = \{0, 2, \bot\}$, but if $[\![e]\!]\ \rho = s_2'$, then $[\![e']\!]\ \rho = \{0, 1, 2, \bot\}$. This is because the variables in the right hand side of a **case** alternative are definite. We could have chosen another option when building the environments for the right hand sides (see [6]) but this is nearer to the actual semantics. The rest of the rules are self-explanatory.

4 Capturing the Determinism Meaning

4.1 Deterministic values

In this section we are proving that $[\![\cdot]\!]_2$ is correct with respect to the denotational semantics presented in the previous section (see Theorem 1). In order to establish the correctness predicate we need first to define the semantic property we want to capture, that is the determinism of an expression. In Figure 9 the boolean functions det_t are defined. Given $s \in A_t$, $det_t(s)$ tells us whether s is a deterministic value or not. A value of type K is deterministic if it is a set with at most one element different from $\bot$ (as $\bot$ belongs to each $s \in A_K$), which is established by the function *unit*. A tuple is deterministic if each component is deterministic. A constructed value $s \in A_T$ is deterministic if its elements different from $\bot$ (again $\bot$ belongs to each $s \in A_T$) have the same constructor, which is established by the function *one*, and additionally the least upper bound of the values in each component is deterministic. For example, values s_1, s_2 and s_2' defined in Section 3.2 are non-deterministic: The first one because it has two different constructors, and the other two because the least upper bound of the first component, $\{0, 1, \bot\}$, is non-deterministic. The definition of det_t in Figure 9 and the propositions below assume that there are not algebraic infinite values. This is not a severe restriction as processes communicating infinite values will not terminate and Hoare powerdomains ignores non-termination ($\bot$ is included in all values).

Finally, a function is deterministic if given a deterministic argument it produces a deterministic result.

Let us note that this semantical definition of determinism characterizes a possibly non-terminating single value expression as being deterministic. This is in accordance with the Hoare powerdomain semantics we have adopted producing Scott-closed sets: Where the actual semantics produces a single value, our approximate semantics produces a non-singleton set because it always includes

$$\alpha_t : A_t \to D_{2t}$$
$$\alpha_K(s) = \begin{cases} d & \text{if } det_K(s) \\ n & \text{otherwise} \end{cases}$$
$$\alpha_{(t_1,\ldots,t_m)}((s_1,\ldots,s_m)) = (\alpha_{t_1}(s_1),\ldots,\alpha_{t_m}(s_m))$$
$$\alpha_T(s) = \begin{cases} d & \text{if } det_T(s) \\ n & \text{otherwise} \end{cases}$$
$$\alpha_{t_1 \to t_2}(f) = \lambda z \in D_{2t_1} \cdot \bigsqcup_{s_1 \in \Gamma_{t_1}(z)} \alpha_{t_2}(f(s_1))$$

$$\Lambda_t : \mathcal{P}(A_t) \to D_{2t}$$
$$\Lambda_t(S) = \bigsqcup_{s \in S} \alpha_t(s)$$

Fig. 10. Abstraction function

$\bot$. That is, predicate det_t characterizes determinism up to non-termination. Notice also that, if we eliminate $\bot$ in the definitions of *unit* and *one*, then predicate det_t characterizes real singleton sets in the basic type, tuples and algebraic type cases, and functions mapping single values into single values in the functional type case. Predicates det_t have some properties (see [6]) we do not show here.

4.2 Abstraction and concretisation functions

In this section we define the abstraction Λ_t and concretisation Γ_t functions that relate the abstract and concrete domains, following the ideas in [1]. We will prove that they are a Galois connection, a crucial property in the correctness proof.

The function Λ_t is just an extension of a function α_t to Hoare sets by applying it to each element of the set and taking the lub. So α_t will also be called abstraction function. With this function, defined in Figure 10, we want to abstract the determinism behaviour of the concrete values. It loses information, i.e. several concrete values may have the same abstract value. In Figure 11 function Γ_t is defined. For each abstract value, it returns all the concrete values that can be approximated by that abstract value. They are mutually recursive.

A value of type K or T is abstracted to d only if it is deterministic. The abstraction of a tuple is the tuple of the abstractions. The abstraction of a function f of type $t_1 \to t_2$ is a little more involved. It is an abstract function taking an argument $z \in D_{2t_1}$. Such z represents several concrete values $s_1 \in \Gamma_t(z)$ whose abstract images are $\alpha_{t_2}(f(s_1))$. So the abstraction of the result is the lub of these abstract images.

The concretisation function is defined so that it builds a Galois connection with Λ_t, which implies that for each concrete value there may be several abstract approximations but there exists only one best (least) approximation.

It can easily be proved that Γ_t is well defined, i.e. it produces downwards closed sets of concrete values. It can also be proved that for each type t, functions α_t, Λ_t and Γ_t are continuous. Both things are shown in [6].

The most important result in this section is that Λ_t and Γ_t are a Galois connection (i.e. $\Lambda_t \cdot \Gamma_t \sqsubseteq id_{D_{2t}}$ and $\Gamma_t \cdot \Lambda_t \sqsupseteq id_{\mathcal{P}(A_t)}$), which is equivalent to the following proposition, that will be intensively used in the correctness proof.

$$\Gamma_t : D_{2t} \to \mathcal{P}(A_t)$$

$$\Gamma_K(b) = \begin{cases} \{s \in A_K \mid unit(s)\} & \text{if } b = d \\ \mathcal{P}(A_K) & \text{if } b = n \end{cases}$$

$$\Gamma_{(t_1,\ldots,t_m)}((z_1,\ldots,z_m)) = \{(s_1,\ldots,s_m) \mid \alpha_{t_i}(s_i) \sqsubseteq z_i \forall i \in \{1..m\}\}$$

$$\Gamma_T(b) = \begin{cases} \{s \in A_T \mid det_T(s)\} & \text{if } b = d \\ \mathcal{P}(A_T) & \text{if } b = n \end{cases}$$

$$\Gamma_{t_1 \to t_2}(f^\#) = \{f \in A_{t_1 \to t_2} \mid \forall s \in A_{t_1}.\alpha_{t_2}(f(v)) \sqsubseteq f^\#(\alpha_{t_1}(s))\}$$

Fig. 11. Concretisation function

Proposition 1 *For each type t, $z \in D_{2t}$, and $s \in A_t$: $s \in \Gamma_t(z) \Leftrightarrow \alpha_t(s) \sqsubseteq z$.*

This proposition can be proved by structural induction on t (see [6]).

Finally we present an interesting property that only holds when the concrete domains of basic and algebraic types have at least two elements different from $\perp$. In the following proposition we show that α_t is surjective, i.e. each abstract value is the abstraction of a concrete value, which in particular belongs to the concretisation of that abstract value. This means that Λ_t and Γ_t are a Galois insertion ($\Lambda_t \cdot \Gamma_t = id_{D_{2t}}$).

Proposition 2 *If all $[\![K]\!]$ and $[\![T]\!]$ have at least two elements different from $\perp$, then for each type t and $z \in D_{2t}$, there exists $s \in \Gamma_t(z)$ such that $\alpha_t(s) = z$.*

This can be proved by structural induction on t (see [6]). If the proposition hypothesis about $[\![K]\!]$ and $[\![T]\!]$ does not hold then it is easy to see that all the concrete values are abstracted to d and none to n. In fact we are avoiding the *Unit* type. However this property is not necessary in the correctness proof.

4.3 A proof of partial correctness

In this subsection we prove that $[\![\cdot]\!]_2$ is correct with respect to the denotational semantics: When the analysis tells that an expression is deterministic, then the concrete value produced by the denotational semantics is semantically deterministic. Otherwise we do not know anything about it. We have to formally describe this intuition. On the one hand, we said in Section 2.4 that $\mu_t(d)$ is the best safe approximation to d in a given domain, so the analysis tells us that an expression is deterministic when its abstract value is less than or equal to $\mu_t(d)$. On the other hand the semantical determinism of a concrete value is established by the predicate det_t. So, the main correctness result is expressed as follows.

Theorem 1. *Let ρ and ρ_2 be two environments, such that for each variable $x :: t_x$, $\alpha_{t_x}(\rho(x)) \sqsubseteq \rho_2(x)$. Then for each $e :: t$: $[\![e]\!]_2\ \rho_2 \sqsubseteq \mu_t(d) \Rightarrow det_t([\![e]\!]\ \rho)$.*

Notice that this only proves the partial correctness of the analysis with respect to the actual semantics of Eden. This (not formally defined) semantics only produces non-singleton sets when the expression e contains at least one occurrence of `merge`. If expression e completely terminates, then we can ignore the undefined

values in $[\![e]\!]\ \rho$ and then $det_t([\![e]\!]\ \rho)$ amounts to saying that $[\![e]\!]\ \rho$ consists of a single value, i.e. e is deterministic in the actual semantics sense.

The theorem is proved in two parts written as Propositions 3 and 4, shown below. The first one tells us that all the values whose abstraction is below $\mu_t(d)$ are semantically deterministic. The second one asserts that the analysis is an upper approximation to the abstraction of the concrete semantics. The proofs use intensively some properties of ϕ_t and μ_t already shown in [5].

Proposition 3 *For each type t, and $s \in A_t$: $\alpha_t(s) \sqsubseteq \mu_t(d) \Leftrightarrow det_t(s)$.*

Proof 1 *We use structural induction on t. The interesting case is the function type, $t = t_1 \to t_2$. The rest are straightforward.*

- $(\Rightarrow)$. *We have to prove that $\forall s \in A_{t_1}.det_{t_1}(s) \Rightarrow det_{t_2}(f(s))$. So, let $s \in A_{t_1}$ such that $det_{t_1}(s)$. We have that*

$$\begin{array}{ll} \alpha_{t_2}(f(s)) \sqsubseteq \bigsqcup_{s_1 \in \Gamma_{t_1}(\alpha_{t_1}(s))} \alpha_{t_2}(f(s_1)) & \{s \in \Gamma_{t_1}(\alpha_{t_1}(s))\} \\ \sqsubseteq \mu_{t_2}(\phi_{t_1}(\alpha_{t_1}(s))) & \{\alpha_t(f) \sqsubseteq \mu_t(d)\} \\ \sqsubseteq \mu_{t_2}(\phi_{t_1}(\mu_{t_1}(d))) & \{\text{by i.h. on } t_1 \text{ and monotonicity}\} \\ = \mu_{t_2}(d) & \{\phi_t \cdot \mu_t = id_{Basic}, \text{ by Prop. 2(b) in [5]}\} \end{array}$$

  Consequently, by i.h. on t_2 we have $det_{t_2}(f(s))$.
- $(\Leftarrow)$. *We have to prove that $\forall z \in D_{2t_1}.\bigsqcup_{s_1 \in \Gamma_{t_1}(z)} \alpha_{t_2}(f(s_1)) \sqsubseteq \mu_{t_2}(\phi_{t_1}(z))$. Let $z \in D_{2t_1}$. We distinguish two cases.*
  - *$z \sqsubseteq \mu_{t_1}(d)$. Then*

$$\begin{array}{ll} s_1 \in \Gamma_{t_1}(z) & \\ \Rightarrow \alpha_{t_1}(s_1) \sqsubseteq z & \{\text{by Proposition 1}\} \\ \Rightarrow \alpha_{t_1}(s_1) \sqsubseteq \mu_{t_1}(d) & \{z \sqsubseteq \mu_{t_1}(d)\} \\ \Rightarrow det_{t_2}(f(s_1)) & \{\text{by i.h. on } t_1 \text{ and } det_t(f)\} \\ \Rightarrow \alpha_{t_2}(f(s_1)) \sqsubseteq \mu_{t_2}(d) & \{\text{by i.h. on } t_2\} \\ \Rightarrow \alpha_{t_2}(f(s_1)) \sqsubseteq \mu_{t_2}(\phi_{t_1}(z)) & \{z \sqsubseteq \mu_{t_1}(d) \text{ and } \phi_t \cdot \mu_t = id_{Basic}\} \end{array}$$

  - *$z \not\sqsubseteq \mu_{t_1}(d)$. In this case $\phi_{t_1}(z) = n$ (by Proposition 3 in [5]). The proposition holds trivially as $\mu_{t_2}(n)$ is the top element in D_{2t_2} (by Proposition 2(d) in [5]).*

□

Proposition 4 *Let ρ and ρ_2 be two environments, such that for each variable $x :: t_x$, $\alpha_{t_x}(\rho(x)) \sqsubseteq \rho_2(x)$. Then for each expression $e :: t$: $\alpha_t([\![e]\!])\ \rho \sqsubseteq [\![e]\!]_2\ \rho_2$.*

Proof 2 *We use structural induction on e. We show here only two interesting cases. In the* **letrec** *case a double induction is necessary (see [6]).*

- *$e = C\ x_1 \ldots x_m :: T$. We distinguish two cases. If $\alpha_T([\![C\ x_1 \ldots x_m]\!]\ \rho) = d$ then it is trivial, as d is the bottom element in Basic.*
  If $\alpha_T([\![C\ x_1 \ldots x_m]\!]\ \rho) = n$, then $\neg det_T(\{C\ ([\![x_1]\!]\ \rho) \ldots ([\![x_m]\!]\ \rho)\}^*)$ by definition of α_t and $[\![\cdot]\!]$. In $\{C\ ([\![x_1]\!]\ \rho) \ldots ([\![x_m]\!]\ \rho)\}^*$ there is just one constructor, so the only possibility for it to be non-deterministic, is that there exists

$i \in \{1..m\}$ *such that* $\neg det_{t_i}(\sqcup\{s_j \mid C\ s_1 \ldots s_m \in \{C\ (\llbracket x_1 \rrbracket\ \rho) \ldots (\llbracket x_m \rrbracket\ \rho)\}^*\})$, *i.e. such that* $\neg det_{t_i}(\llbracket x_i \rrbracket\ \rho)$. *By Proposition 1, this implies that* $\alpha_{t_i}(\llbracket x_i \rrbracket\ \rho) \not\sqsubseteq \mu_{t_i}(d)$ *and consequently* $\phi_{t_i}(\alpha_{t_i}(\llbracket x_i \rrbracket\ \rho)) = n$ *(by Proposition 3 in [5]), so*

$$\begin{array}{lll} \llbracket C\ x_1 \ldots x_m \rrbracket_2\ \rho_2 = \bigsqcup_{j=1}^{m} \phi_{t_j}(\llbracket x_j \rrbracket_2\ \rho_2) & \{\text{by definition of } \llbracket\cdot\rrbracket_2\} \\ \quad \sqsupseteq \bigsqcup_{j=1}^{m} \phi_{t_j}(\alpha_{t_j}(\llbracket x_j \rrbracket\ \rho)) & \{\text{by i.h. on } t_j \text{ and monotonicity}\} \\ \quad = n & \{\phi_{t_i}(\alpha_{t_i}(\llbracket x_i \rrbracket\ \rho)) = n\} \end{array}$$

– $e = \lambda v.e' :: t_1 \to t_2$. *By definition of* $\llbracket\cdot\rrbracket$ *and* α_t *we have to prove that* $\bigsqcup_{s_1 \in \Gamma_{t_1}(z)} \alpha_{t_2}(\llbracket e' \rrbracket\ \rho[v \mapsto s_1]) \sqsubseteq \llbracket e' \rrbracket_2\ \rho_2[v \mapsto z]$.
If $s_1 \in \Gamma_{t_1}(z)$ *then* $\alpha_{t_1}(s_1) \sqsubseteq z$ *by Proposition 1, so* $\rho[x \mapsto s_1]$ *and* $\rho_2[v \mapsto z]$ *satisfy the theorem hypothesis about the environments. We can then apply induction hypothesis on* e' *and obtain* $\alpha_{t_2}(\llbracket e' \rrbracket\ \rho[v \mapsto s_1]) \sqsubseteq \llbracket e' \rrbracket_2\ \rho_2[v \mapsto z]$. □

5 Conclusions and Future Work

We have proved the correctness of a whole hierarchy of non-determinism analyses for the parallel-functional language Eden. In order to do this, we have defined first a denotational semantics for Eden where non-determinism is represented. We have chosen to use a plural semantics in which non-deterministic choices for variables are deferred as much as possible. A semantics nearer to the actual one (within a single process) would have been a singular one in which environments map variables to single values. This would reflect the fact that non-deterministic choices are done at binding evaluation time instead of at each variable occurrence. For instance, a let-bound variable will get its value the first time it is evaluated and this value will be shared thereafter by all its occurrences. In order to consider all the possible values the variable can have, we build one environment for each of them:

$$\llbracket \textbf{let } v = e \textbf{ in } e' \rrbracket\ \rho = \bigsqcup_{z \in \llbracket e \rrbracket\ \rho} \llbracket e' \rrbracket\ \rho[v \mapsto z]$$

The same would be true for case-bound and lambda-bound variables. We have tried to define this singular semantics and things go wrong when trying to give semantics to mutually recursive definitions. The traditional fixpoint computation by using Kleene's ascending chain gives a semantics more plural than expected. For instance, in the definition

$$\begin{array}{ll} \textbf{letrec} & f = head(merge_{Int \to Int}\ [g]\ [\lambda x.0]) \\ & g = head(merge_{Int \to Int}\ [f]\ [\lambda x.1]) \\ \textbf{in } (f, g) & \end{array}$$

Kleene's ascending chain will compute the following set of possible environments:

$$\begin{array}{l} \overline{\rho} = \{\ \{f \mapsto \lambda x.\{\bot\},\ g \mapsto \lambda x.\{\bot\}\},\quad \{f \mapsto \lambda x.\{0\}^*,\ g \mapsto \lambda x.\{1\}^*\}, \\ \qquad \{f \mapsto \lambda x.\{0\}^*,\ g \mapsto \lambda x.\{0\}^*\},\quad \{f \mapsto \lambda x.\{1\}^*,\ g \mapsto \lambda x.\{1\}^*\}, \\ \qquad \{f \mapsto \lambda x.\{1\}^*,\ g \mapsto \lambda x.\{0\}^*\}\ \} \end{array}$$

However, the lazy evaluation of the expression will never produce the fifth possibility. In [12] a singular semantics for a small non-deterministic recursive functional language was defined. The problem with fixpoints did not arise there because the language was extremely simple: Only one recursive binding was allowed in the program and this had to be a lambda abstraction. Additionally, the language was only first-order. The problem arises when there are at least two mutually recursive bindings to non normal-form expressions. In order to define a real singular semantics, we think that an operational approach should be taken, similar to that of [4]. In this way, the actual lazy evaluation with its updating of closures and sharing of expressions could be appropriately modeled. We forsee to do it as future work.

Another extension of the present work is to include polymorphism in the language, in the semantics and in the proof of correctness. The analyses originally presented in [7, 8] already included this aspect.

References

1. 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, 1986.
2. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282. ACM, 1979.
3. M. Hidalgo and Y. Ortega. Continuation Semantics for Parallel Haskell Dialects. In *APLAS'03*, volume 2895 of *LNCS*, pages 303–321. Springer-Verlag, 2003.
4. J. Hughes and A. Moran. Making Choices Lazily. In *FPCA'95*, pages 108–119. ACM Press, 1995.
5. R. Peña and C. Segura. Three Non-determinism Analyses in a Parallel-Functional Language. Technical Report 117-01, Univ. Complutense de Madrid, Spain, 2001. (http://dalila.sip.ucm.es/miembros/clara/publications.html).
6. R. Peña and C. Segura. Correctness of Non-determinism Analyses in a Parallel-Functional Language. Technical Report 131-03, Univ. Complutense de Madrid, Spain, 2003. (http://dalila.sip.ucm.es/miembros/clara/publications.html).
7. R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *IFL'00*, volume 2011 of *LNCS*, pages 1–18. Springer-Verlag, 2001.
8. R. Peña and C. Segura. A Polynomial Cost Non-Determinism Analysis. In *IFL'01*, volume 2312 of *LNCS*, pages 121–137. Springer-Verlag, 2002.
9. 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.
10. 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.
11. H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, May 1990.
12. H. Søndergaard and P. Sestoft. Non-Determinism in Functional Languages. *Computer Journal*, 35(5):514–523, October 1992.