Inference based analyses of functional programs: dead-code and strictness

Mario Coppo, Ferruccio Damiani and Paola Giannini
Dipartimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino (Italy),
Fax. (+39)-11-75.16.03, Tel. (+39)-11-74.29.111,
E-mail: {coppo,damiani,giannini}@di.unito.it

Abstract

We present a simple framework for "non-standard type inference based" analyses of functional programs and show how to apply it to dead-code and strictness analyses. A key feature of this framework is that is based directly on operational semantics.

1 Introduction

In this paper the λ-calculus (see [3, 22, 4]) is seen as a prototype *functional programming language* (see [23] for a survey) rather than a formalism for representing proofs as in constructive logic. This makes relevant peculiar aspects (as the evaluation strategies) which are typical of this view.

The aim of this paper is to present the main concepts of the type inference approach to program analysis and to illustrate it through a couple of relevant examples (dead-code and strictness). We do this by introducing a simple framework for defining formal systems for reasoning about properties of functional programs. A number of program analyses useful in the optimization of functional programs can be handled by instantiating our framework.

1.1 The type inference approach for reasoning about program properties

The Curry-Howard isomorphism shows that the construction of a λ-term M of a given type A is equivalent to prove A as a theorem in a suitable (propositional) logic. From the "λ-calculus" point of view, having type A can be seen as a property of the λ-term M, that can be proved using the usual machinery of propositional logic. The use of formal inference systems to prove program properties is also a basic feature of Curry's approach to typing lambda terms (see [4]), in which the

inference system to assign types to terms is defined on the top of the type-free λ-calculus. In Curry's approach different types can be assigned to the same λ-term, representing its different functional properties.

An essential feature of the inference approach is the existence, in many significant cases, of simple and (at least in the practical cases) efficient algorithms for automatic type inference. The inference approach has proved to be very useful in the definition of polymorphic type systems for programming languages. Starting from ML ([31]) most of the modern functional programming languages incorporate a polymorphic type system based on these ideas. In this paper we are interested in types from a "programming" point of view, in which the λ-terms are the basic objects and types are seen as tools for explaining their behaviour.

More recently it has been observed that, also from a "programming" point of view, the notion of type could be given a richer significance than the usual one, in which types simply represent collections of object with an uniform behaviour with respect to the basic ground domains. Indeed the usual inference rules for type assignment can be seen as the basic inference rules to prove properties of λ-terms, and having a certain type is only the most general one. If we take a more refined notion of "type" we can use the inference rules to prove more detailed properties. For instance if we distinguish, in the set of integer numbers, the properties of being an "even" or an "odd" number, we can try to prove if a given function which maps integer numbers into integer numbers maps, for instance, even integer numbers into odd ones. The successor, for instance, has this property. To prove properties like these we have to add to the basic inference rules also rules which are more specifically tailored to the properties we are interested in. In the case of even and odd numbers it could be useful, for example, to have an axiom that says that the property of being odd (or even) number implies that of being an integer number. So if we know that the function $F = \lambda x.x*2$ maps integer numbers into even numbers we should be able to deduce that F maps also odd (or even) numbers into integer numbers.

In this way we can get inference systems to prove program properties, most of which are useful in the optimization of functional programming languages. To this aim an essential requirement is the decidability and efficiency of the associated decision procedure. One main concern of this paper is to present applications for which a decision procedure exists and is feasible, even though we will not explicitly address here this aspect.

The inference approach could be applied to untyped λ-terms. However, the untyped setting does not seem, the most interesting environment in which to develop system for reasoning about programs. Indeed the structure of properties of untyped languages is very liberal but sometimes excessively general. The inference approach may, however, be developed for typed languages as well. In this case properties are defined for terms of a given type. So we will not speak only of terms of type "even" but rather of "even" terms of type int. We have to distinguish between "types" in the usual sense and "types" which represent properties of terms of a given type (in the previous sense). To keep clear this distinction we call *non-standard types* the second kind of types. This more stratified structure of

the inference systems has also the advantage that, if we start from a finite set of non-standard types at the *ground* types[1] , the set of non-standard types representing properties of terms of each type are finite, making typechecking decidable (at least in principle). Inference algorithms for these systems can be defined making possible practical applications.

A formal system to prove program properties is formed by two sets of rules. The basic assignment rules which are based on the structure of programs, λ-terms in our case, and the specific rules, which are determined by the properties we are representing. These can be divided, in turn, into two sets: the rules which represent the structure of the properties in themselves (for instance the property of "being odd" implies that of "being an integer number") and the rules that concern the way in which properties are assigned to λ-terms (for instance the axiom that the successor function maps even numbers in odd ones). We formalize the first rules by means of a formal inclusion relation ($\leq$) which represent the entailment structure of properties.

The idea of considering inference systems as a tool to prove program properties was introduced in the literature in [29] where a set of formal inference rules for the analysis of strictness properties was given (see also [12] where this idea was implicit). A first framework based on type inference for the proof of properties of untyped λ-calculus setting was given in [13]. For the simply typed λ-calculus the approach was pursued by a number of other authors, see for instance [25, 7, 51, 21, 48, 5, 33].

1.2 Type inference and abstract interpretation

A related approach for the investigation of program properties is based on abstract interpretation. Abstract interpretation was first introduced for imperative languages in [15] and then applied to functional languages by a number of authors, see [34, 35, 11, 36, 2, 24, 16]. In this approach programs are interpreted in abstract (usually finite) domains, instead of the standard denotational domains. Such abstract domains highlight the properties that are considered and are related to the standard denotational domains by suitable embedding-projection functions. The detection of properties then is obtained by giving an abstract form of the programming language constructors, and evaluating the meaning of a program over the abstract finite domains.

The abstract interpretation approach is related to the inference one, as it has been shown, for instance, in [14]. In fact an inference system can be seen as a kind of abstract interpretation. Similarly also abstract domains can be defined from inference systems via the notion of filter models ([12], [13]).

An explicit connection between the abstract interpretation approach and the type inference one is obtained by defining an inference system from the logical presentation of some abstract domain, along the lines of Abramsky in [2]. This approach has been followed by Jensen in [25] (see also Benton [7]) and subsequent

[1] We call *ground* the non-compound type of the language, like int (the type of integers) and bool (the type of booleans).

papers ([26, 27]) in which the logic of the abstract domains for strictness, introduced in [11], is presented via a type inference system which is proved to be sound and complete w.r.t. the abstract semantics. In our work we take a more direct approach: we connect directly the type inference system to operationally based semantics, namely the ground and the lazy observational equivalences, without the need of passing through an abstract interpretation.

1.3 Static analysis of functional programs

Depending on the semantics of function application, functional languages can be distinguished in two classes. *Strict* (or *eager*) languages, in which function application has a *call-by-value* semantics (i.e. the actual parameter is evaluated when the function is applied and the formal parameter is bound to that value). *Non-strict* (or *lazy*) languages, in which function application has a *call-by-name* semantics (i.e. the function's formal parameter is bound to the unevaluated actual parameter which is evaluated each time its value is required). For efficiency, application in non-strict languages is usually implemented using *call-by-need*, which is the same as call-by-name, except that after the first evaluation of the actual parameter its value is retained and used whenever the formal parameter is subsequently required. The evaluation of an application by call-by-value or call-by-need requires usually more machinery (and then more time) than the evaluation of the application by call-by-value.

According to the above definition, Scheme [1], ML [31] and Caml [30] are eager, whereas Miranda [50], Haskell [40] and Clean [43, 20] are lazy.

One of the main drawbacks of the functional paradigm is that the efficiency is often rather poor. Modern compilers for functional programming languages use various kinds of transformations to improve the efficiency of the code produced. Such transformations are based on static analyses of programs and must be proved to preserve their semantics. It is of practical interest, then, to find reliable methods for detecting properties useful to perform such optimizations. Various kinds of static analysis for functional languages have been proposed in literature.

Dead-code analysis aims to determine the parts of a program that are useless for the computation of the value of a program. This information allows the simplification of programs by removing useless parts (which are called dead-code). For example in the evaluation of an expression $(\lambda x^{\mathsf{int}}.3)P$ the value of P is discarded in the computation of the value of the expression (which is the integer number 3) and so its evaluation is useless. The expression P would be evaluated anyway in an eager language, and this can result is a considerable waste of time. P would not be evaluated, instead, in a lazy language, however the size of P could be big and its storage could determine a waste of time and memory. In both cases if we replace P with any constant d (of type int) which does not require evaluation we get a simpler expression $((\lambda x^{\mathsf{int}}.3)\mathsf{d})$ which is equivalent to the original one (in the sense that yields the same value[2]). In this case it is also possible to perform a

[2] In an eager language this transformation could change the termination behaviour of the program: if the evaluation of P does not terminate then the original program diverges while the

more substantial simplification by removing both the abstraction of the variable x^{int} on the term 3 which does not contain occurrences of it and the subsequent application of the resulting term to P, since the whole expression is equivalent to 3.

The information about dead-code can be propagated. We will consider a λ-calculus with pairs $\langle\ ,\ \rangle$ and projections proj_1 and proj_2. Let M be the term

$$\lambda x.\mathsf{proj}_1\,\langle P, Q\rangle N$$

for some term P, Q, and N. Since the projection proj_1 returns the first component of the pair the term Q is dead code. Moreover, assume that P does not contain any occurrence of x, in this case also N is not relevant to the computation of the final result. So M behaves like the term P, which is of course simpler.

Dead-code analysis has been mainly studied in the context of *logical frameworks*, like Coq [6] (see [41] for a short survey on logical frameworks), to remove redundant code from functional programs extracted from formal proofs (see [39, 38] for an introduction to the subject). In fact, programs extracted from proofs usually contain large parts that are useless for the computation of the final result, i.e. dead-code, and some sort of simplification is mandatory. Various simplification techniques have been developed in the last few years, see for instance [49, 8, 10, 9, 46, 19]. Dead-code can be found also in programs in which general-purpose functions, taken from standard libraries, are used in specialized contexts.

Strictness analysis aims to determine whether, in a lazy language, a function is strict in its arguments. A function expecting n arguments is *strict* in its i-th argument if diverges whenever its i-th argument (actual parameter) diverges, independently of the value of all other parameters. For example $\lambda x.\lambda y.x + y$, seen as a function of x and y is strict in both its first and second arguments while $\lambda f.\lambda y.(f\ y)$ is strict in its first argument but not in the second one. In fact if f is replaced by a function that does not evaluate its argument (like $\lambda x.3$) the value of the second argument is never used for the computation of the final result which is 3, which can be computed even if the second argument is undefined.

If a function is strict in one of its arguments then, for this argument, we can replace a call-by-need with a the more efficient call-by-value without affecting the result of the function. Other optimizations (like, for instance, the parallel evaluation of a function and its argument) are also possible for strict functions.

Other program analyses which can be expressed in our framework but which are not addressed in this paper are, for instance, *totality analysis*, which aims to determine whether the evaluation of an expression terminates (i.e. whether an expression has a weak head normal form) and *binding-time analysis*, which aims to determine the parts of a program that can be evaluated at compile-time.

---

simplified program could converge. However, when considering terminating programs (as well-written pure functional programs should be), the simplification preserves also the termination behaviour of the program.

1.4 About this paper

The functional language considered in this paper is a non-strict language, called PCFP, which is an extension of PCF(see [44]). The non-standard type inference framework that we propose exploits features common to different static analyses of functional languages. It provides a "core" system for type syntax, entailment ad assignment rules whose semantics is given in a term model of the language based on some operational equivalence. The particular equivalence considered as well as the entailment and inference rules can be tuned to focus on some specific analysis.

Section 2 introduces the programming language we are dealing with, its operational semantics, and some term models based on this semantics. Section 3 introduces our non-standard type inference framework, and Section 4 describes how to instantiate the framework to a particular analysis. In Sections 5 and 6 the framework is instantiated to dead-code and strictness analyses, respectively.

2 The language PCFP

In this section we introduce a simple functional programming language (basically the simply typed λ-calculus with cartesian product, if-then-else, fixpoint, and arithmetic constants) and its operational (call-by-name) semantics. We use the acronym PCFP for this language, standing for "Programming Computable Function with Pairs", since it is the dialect of the language PCF [44] obtained by adding a type constructor for pairs.

For a more comprehensive presentation of the material in Sections 2.1 and 2.2 see [42], where the language considered includes also lazy lists.

2.1 PCFP syntax and evaluation rules

The set of PCFP types is defined assuming as *ground* types int and bool: the set of natural numbers and the set of booleans. Types are ranged over by ρ, σ, τ.

Definition 2.1 (PCFP types) *The language of* types *(***T***) is defined by the following grammar:* $\rho ::= \iota \mid \rho \to \rho \mid \rho \times \rho$, *where* $\iota \in \{\mathsf{int}, \mathsf{bool}\}$.

PCFP terms are defined from a set of typed *term constants*

$$\begin{aligned}\mathcal{K} \;=\; \{\;& 0^{\mathsf{int}},\ 1^{\mathsf{int}},\ 2^{\mathsf{int}},\ \ldots,\ \mathsf{succ}^{\mathsf{int}\to\mathsf{int}},\ \mathsf{pred}^{\mathsf{int}\to\mathsf{int}},\ +^{\mathsf{int}\times\mathsf{int}\to\mathsf{int}},\ -^{\mathsf{int}\times\mathsf{int}\to\mathsf{int}},\\ & \mathsf{true}^{\mathsf{bool}},\ \mathsf{false}^{\mathsf{bool}},\ \mathsf{not}^{\mathsf{bool}\to\mathsf{bool}},\ \mathsf{and}^{\mathsf{bool}\times\mathsf{bool}\to\mathsf{bool}},\ \mathsf{or}^{\mathsf{bool}\times\mathsf{bool}\to\mathsf{bool}},\\ & =^{\mathsf{int}\times\mathsf{int}\to\mathsf{bool}}\;\}\ ,\end{aligned}$$

(ranged over by c), and a set $\mathcal{V}$ of typed *term variables* (ranged over by $x^{\rho}, y^{\sigma}, \ldots$). The type of a constant c is denoted by $\mathbf{T}(\mathsf{c})$. The choice of the term constants is not critical. Other operators on integer and booleans could be added without problems. PCFP terms, ranged over by M, N, ..., are defined as follows.

Definition 2.2 (PCFP terms) *We write* $\vdash_{\mathbf{T}} M : \rho$, *and say that* M *is a term of type* ρ, *if* $\vdash M : \rho$ *is derivable by the rules in Fig. 1.*

$$(\text{Var})\ \vdash x^{\rho} : \rho \qquad (\text{Con})\ \vdash \mathsf{c}^{\rho} : \rho$$

$$(\to \text{I})\ \frac{\vdash M : \sigma}{\vdash \lambda x^{\rho}.M : \rho \to \sigma} \qquad (\to \text{E})\ \frac{\vdash M : \rho \to \sigma \quad \vdash N : \rho}{\vdash MN : \sigma}$$

$$(\times \text{I})\ \frac{\vdash M_1 : \rho_1 \quad \vdash M_2 : \rho_2}{\vdash \langle M_1, M_2 \rangle : \rho_1 \times \rho_2} \qquad (\times \text{E}_i)\ \frac{\vdash M : \rho_1 \times \rho_2}{\vdash \mathsf{proj}_i M : \rho_i}\ i \in \{1,2\}$$

$$(\text{Fix})\ \frac{\vdash M : \rho}{\vdash \mathsf{fix}\, x^{\rho}.M : \rho} \qquad (\text{If})\ \frac{\vdash N : \mathsf{bool} \quad \vdash M_1 : \rho \quad \vdash M_2 : \rho}{\vdash \mathsf{if}\ N\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \rho}$$

Figure 1: Rules for PCFP term formation (system $\vdash_{\mathbf{T}}$)

Notation 2.3 *1. According to Definition 2.2, in a PCFP term M the types of variables and constants are explicitly mentioned. In the following we often omit to write types which are clear from the context.*

2. *The finite set of the* free variables *of a term M, denoted by* $\mathrm{FV}(M)$*, is defined in the standard way. In the following we take PCFP terms to be α-equivalence classes of syntax tree, i.e. we will identify terms modulo renaming of the bound variables.*

3. *We will often use, as syntactic sugar, the infix notation for the binary operators of the language, e.g. we will write "$3 + 5$" instead of "$+\langle 3, 5\rangle$".*

4. *As usual a substitution is a finite function mapping term variables to terms, denoted by $[x_1 := N_1, \ldots, x_n := N_n]$ ($[\vec{x} := \vec{N}]$ for short), which respects the types, i.e. each $x_i^{\rho_i}$ is substituted by a term N_i of the same type. Substitution acts on free variables, the renaming of the bound variables is implicitly assumed.*

Let $\Lambda_{\mathbf{T}}$ be the set of PCFP terms, i.e. $\Lambda_{\mathbf{T}} = \{M \mid \vdash_{\mathbf{T}} M : \rho \text{ for some type } \rho\}$, and $\Lambda_{\mathbf{T}}^{\mathbf{c}}$ be the set of the *closed* terms, i.e. $\Lambda_{\mathbf{T}}^{\mathbf{c}} = \{M \mid M \in \Lambda_{\mathbf{T}} \text{ and } \mathrm{FV}(M) = \emptyset\}$. The values of the terms in $\Lambda_{\mathbf{T}}^{\mathbf{c}}$ are defined via a standard call-by-name operational semantics (see [45, 28]) described by judgments of the form $M \Downarrow K$, where M is a closed term and K is a closed term in *weak head normal form* (*w.h.n.f.*), i.e. $K \in \mathbf{V}_{\mathbf{T}}$, where

$$\mathbf{V}_{\mathbf{T}} = \mathcal{K} \cup \{\lambda x^{\rho}.N \mid \lambda x^{\rho}.N \in \Lambda_{\mathbf{T}}^{\mathbf{c}}\} \cup \{\langle M_1, M_2\rangle \mid \langle M_1, M_2\rangle \in \Lambda_{\mathbf{T}}^{\mathbf{c}}\}.$$

The meaning of a functional constant c is given by a set $\mathbf{mean}(C)$ of pairs, i.e. if $(P_1, P_2) \in \mathbf{mean}(C)$ then CP_1 evaluates to P_2. For example $(\mathsf{true}, \mathsf{false}) \in \mathbf{mean}(\mathsf{not})$ and $(\langle 1, 3\rangle, 4) \in \mathbf{mean}(+)$.

Definition 2.4 (Value of a term) *Let $M \in \Lambda_{\mathbf{T}}^{\mathbf{c}}$. We write $M \Downarrow K$, and say that M evaluates to K, if this statement is derivable by using the rules in Fig. 2.*

$$(\text{CAN})\ \frac{K \in \mathbf{V_T}}{K \Downarrow K}$$

$$(\text{APP})\ \frac{M \Downarrow \lambda x.P \quad P[x := N] \Downarrow K}{MN \Downarrow K}$$

$$(\text{APP}_1)\ \frac{M \Downarrow \mathsf{c} \quad N \Downarrow \mathsf{c}_1}{MN \Downarrow \mathsf{c}_2}\ (\mathsf{c}_1, \mathsf{c}_2) \in \mathbf{mean}(\mathsf{c})$$

$$(\text{APP}_2)\ \frac{M \Downarrow \mathsf{c} \quad N \Downarrow \langle N_1, N_2\rangle \quad N_1 \Downarrow \mathsf{c}_1 \quad N_2 \Downarrow \mathsf{c}_2}{MN \Downarrow \mathsf{c}_3}\ (\langle \mathsf{c}_1, \mathsf{c}_2\rangle, \mathsf{c}_3) \in \mathbf{mean}(\mathsf{c})$$

$$(\text{PROJ}_i)\ \frac{P \Downarrow \langle M_1, M_2\rangle \quad M_i \Downarrow K}{\mathsf{proj}_i P \Downarrow K}\ i \in \{1,2\}$$

$$(\text{FIX})\ \frac{M[x := \mathsf{fix}\, x.M] \Downarrow K}{\mathsf{fix}\, x.M \Downarrow K}$$

$$(\text{IF}_1)\ \frac{N \Downarrow \mathsf{true} \quad M_1 \Downarrow K}{\mathsf{if}\ N\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \Downarrow K} \qquad (\text{IF}_2)\ \frac{N \Downarrow \mathsf{false} \quad M_2 \Downarrow K}{\mathsf{if}\ N\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \Downarrow K}$$

Figure 2: "Natural semantics" evaluation rules

Let $M \Downarrow$, to be read "M is convergent", mean that, for some K, $M \Downarrow K$, and let $M \Uparrow$, to be read "M is divergent", mean that, for no K, $M \Downarrow K$. It is easy to see that, for any types $\rho, \sigma \in \mathbf{T}$, $\mathsf{fix}\, x^\rho.x \Uparrow$ ($\mathsf{fix}\, x^\rho.x$ is the typical divergent term of type ρ) and that $(\mathsf{fix}\, x^{\rho\to\sigma}.x)P \Uparrow$ (for every closed term P of type ρ). Moreover, also $\mathsf{proj}_i(\mathsf{fix}\, x^{\rho\times\sigma}.x) \Uparrow$ (for $i \in \{1,2\}$).

2.2 Contextual equivalences

Following [42] we introduce two congruences on PCFP terms. The first is the congruence on terms induced by the contextual preorder that compares the behaviour of terms just at the ground type int (*ground contextual equivalence*), while the second is the one induced by the contextual preorder that compares the behavior of terms at every type (*lazy contextual equivalence*).

Let $(\mathcal{C}[\ ]^\rho)^\sigma$ denote a typed context of type σ with a hole of type ρ in it.

Definition 2.5 (Ground contextual equivalence) *Let M and N be terms of type ρ. Define $M \preceq^{\mathrm{gnd}}_{\mathrm{obs}} N$ whenever, for all closed contexts $(\mathcal{C}[\ ]^\rho)^{\mathsf{int}}$, if $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are closed terms, then $\mathcal{C}[M] \Downarrow$ implies $\mathcal{C}[N] \Downarrow$. The relation $\preceq^{\mathrm{gnd}}_{\mathrm{obs}}$ is the* ground contextual preorder *and the equivalence induced by $\preceq^{\mathrm{gnd}}_{\mathrm{obs}}$, denoted by $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$, is the* ground observational equivalence.

In [42] a co-inductive characterization of ground contextual equivalence in terms of bisimulation is given.

(bis 1a)	$(M\ \mathcal{B}_{\mathsf{bool}}\ N$ and $M \Downarrow \mathsf{b})$ implies $N \Downarrow \mathsf{b}$
(bis 1b)	$(M\ \mathcal{B}_{\mathsf{bool}}\ N$ and $N \Downarrow \mathsf{b})$ implies $M \Downarrow \mathsf{b}$
(bis 2a)	$(M\ \mathcal{B}_{\mathsf{int}}\ N$ and $M \Downarrow \mathsf{n})$ implies $N \Downarrow \mathsf{n}$
(bis 2b)	$(M\ \mathcal{B}_{\mathsf{int}}\ N$ and $N \Downarrow \mathsf{n})$ implies $M \Downarrow \mathsf{n}$
(bis 3)	$M\ \mathcal{B}_{\rho\to\sigma}\ N$ implies, for all P such that $\vdash_{\mathbf{T}} P : \rho$, $MP\ \mathcal{B}_\sigma\ NP$
(bis 4)	$M\ \mathcal{B}_{\rho\times\sigma}\ N$ implies $(\mathsf{proj}_1(M)\ \mathcal{B}_\rho\ \mathsf{proj}_1(N)$ and $\mathsf{proj}_2(M)\ \mathcal{B}_\sigma\ \mathsf{proj}_2(N))$

Figure 3: Bisimulation conditions for PCFP

We give here the definition of bisimulation for PCFP, and state the main result presented in [42]: Theorem 2.7. Such theorem, relating ground contextual equivalence and bisimilarity, is used in this paper to prove some equivalences between terms.

Definition 2.6 *A* PCFP bisimulation $\mathcal{B}$ *is a type indexed family of relations on closed terms,*

$$\mathcal{B}_\rho \subseteq \{\langle M, N\rangle \mid \vdash_{\mathbf{T}} M : \rho, \vdash_{\mathbf{T}} N : \rho \text{ and } \mathrm{FV}(M) \cup \mathrm{FV}(N) = \emptyset\}$$

($\rho \in \mathbf{T}$), satisfying the conditions in Fig. 3. PCFP bisimilarity *is the largest bisimulation and will be denoted by $\simeq$.*

The relation $\simeq$ can be extended to open terms by defining $M \simeq N$ if for all substitutions $[x_1 := N_1, \ldots, x_n := N_n]$ such that $\{x_1, \cdots, x_n\} \supseteq \mathrm{FV}(M) \cup \mathrm{FV}(N)$ and $\bigcup_{1\le n} \mathrm{FV}(N_i) = \emptyset$

$$M[x_1 := N_1, \ldots, x_n := N_n] \simeq N[x_1 := N_1, \ldots, x_n := N_n].$$

Theorem 2.7 (Operational extensionality for PCFP) *Ground contextual equivalence coincides with bisimilarity. That is, let M and N be terms of the same type:*

$$M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N \text{ if and only if } M \simeq N.$$

In the following we list some properties of $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$ which are used in the rest of this paper, and can be derived from Theorem 2.7. In particular property 1. says that the ground observational equivalence is extensional.

1. For all M and N such that $\vdash_{\mathbf{T}} M : \rho \to \sigma$ and $\vdash_{\mathbf{T}} N : \rho \to \sigma$, $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$ if and only if for all P such that $\vdash_{\mathbf{T}} P : \rho$, $MP \simeq^{\mathrm{gnd}}_{\mathrm{obs}} NP$, implies $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$.

2. For all M and N such that $\vdash_{\mathbf{T}} M : \rho$ and $\vdash_{\mathbf{T}} N : \rho$, if $M \Uparrow$ and $N \Uparrow$, then $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$.

3. $(\lambda x.M)N \simeq^{\mathrm{gnd}}_{\mathrm{obs}} M[x := N]$.

4. $\mathsf{proj}_i\langle M_1, M_2\rangle \simeq^{\mathrm{gnd}}_{\mathrm{obs}} M_i$ (for $i \in \{1,2\}$).

From the previous properties we derive that $\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x \simeq^{\mathrm{gnd}}_{\mathrm{obs}} \mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z$. Indeed for all P, $(\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x)P \Uparrow$ since from 3. $(\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x)P \simeq^{\mathrm{gnd}}_{\mathrm{obs}} \mathsf{fix}\, x^{\mathsf{int}}.x$. Moreover since $\mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z \Uparrow$ then for all P, $(\mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z)P \Uparrow$. Therefore, from 2., for all P, $(\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x)P \simeq^{\mathrm{gnd}}_{\mathrm{obs}} (\mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z)P$. From 1. so we can derive that $\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x \simeq^{\mathrm{gnd}}_{\mathrm{obs}} \mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z$.

We also define a contextual equivalence that observes terms at all types.

Definition 2.8 (Lazy contextual equivalence) *Let M and N be terms of type ρ. Define $M \preceq^{\mathrm{lazy}}_{\mathrm{obs}} N$ whenever, for all type σ and for all closed contexts $(C[\ ]^{\rho})^{\sigma}$, if $C[M]$ and $C[N]$ are closed terms, then $C[M] \Downarrow$ implies $C[N] \Downarrow$. The relation $\preceq^{\mathrm{lazy}}_{\mathrm{obs}}$ is the* lazy contextual preorder *and the equivalence induced by $\preceq^{\mathrm{lazy}}_{\mathrm{obs}}$, denoted by $\simeq^{\mathrm{lazy}}_{\mathrm{obs}}$, is the* lazy observational equivalence.

It is immediate to see that, $M \simeq^{\mathrm{lazy}}_{\mathrm{obs}} N$ implies $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$. The reverse is not true. For instance as previously shown $\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x \simeq^{\mathrm{gnd}}_{\mathrm{obs}} \mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z$ while $\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x \not\simeq^{\mathrm{lazy}}_{\mathrm{obs}} \mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z$, since $\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x \Downarrow$ and $\mathsf{fix}\, z^{\mathsf{int}\to\mathsf{int}}.z \Uparrow$. This example shows that property 1. above does not hold for $\simeq^{\mathrm{lazy}}_{\mathrm{obs}}$. Properties 2., 3., and 4., instead, holds also for $\simeq^{\mathrm{lazy}}_{\mathrm{obs}}$.

2.3 Closed term models of PCFP

In this section we introduce the term models induced by the ground and lazy equivalences. We also introduce a richer term model containing a "convergence to w.h.n.f." test, isdef. The entailment relation between non-standard types presented in Section 6.2 is complete w.r.t. this extended model.

2.3.1 The models $\mathcal{M}^{\mathrm{gnd}}$ and $\mathcal{M}^{\mathrm{lazy}}$

The *closed term model* $\mathcal{M}^{\mathrm{gnd}}$ of PCFP is defined by interpreting each type ρ as the set of the equivalence classes of the relation $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$ on the closed terms of type ρ in $\Lambda^{\mathbf{c}}_{\mathbf{T}}$. Let $\mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$ denote the interpretation of type ρ in this model, and let $[M]^{\mathcal{M}^{\mathrm{gnd}}}$ denote the equivalence class of the closed term M. The preorder $\preceq^{\mathrm{gnd}}_{\mathrm{obs}}$ is naturally extended to a partial order between equivalence classes by defining $[M]^{\mathcal{M}^{\mathrm{gnd}}} \preceq^{\mathrm{gnd}}_{\mathrm{obs}} [N]^{\mathcal{M}^{\mathrm{gnd}}}$ if $M \preceq^{\mathrm{gnd}}_{\mathrm{obs}} N$. For every type ρ, $[\mathsf{fix}\, x^{\rho}.x]^{\mathcal{M}^{\mathrm{gnd}}}$ (the equivalence class of the canonical divergent element, which represents the notion of endless computation), is the least element, w.r.t. $\preceq^{\mathrm{gnd}}_{\mathrm{obs}}$, of $\mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$. An *environment* is a mapping $e : \mathcal{V} \to \bigcup_{\rho\in\mathbf{T}} \mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$ which respects types, i.e. a mapping such that, for all x^{ρ}, $e(x^{\rho}) \in \mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$. The interpretation of a term M in an environment e is defined in a standard way by:

$$[\![M]\!]^{\mathcal{M}^{\mathrm{gnd}}}_{e} = [M[x_1 := N_1, \ldots, x_n := N_n]]^{\mathcal{M}^{\mathrm{gnd}}},$$

where $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$ and $[N_l]^{\mathcal{M}^{\mathrm{gnd}}} = e(x_l)$ $(1 \leq l \leq n)$. It is worth mentioning that for all environment e, if $\vdash_{\mathbf{T}} M : \rho$ then $[\![M]\!]_e^{\mathcal{M}^{\mathrm{gnd}}} \in \mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$.

The *closed term model* $\mathcal{M}^{\mathrm{lazy}}$ of PCFP is defined similarly by interpreting each type ρ as the set of the equivalence classes of the relation $\simeq_{\mathrm{obs}}^{\mathrm{lazy}}$ on the closed terms of type ρ in $\Lambda_{\mathbf{T}}^{\mathbf{c}}$. Note that $\mathcal{M}^{\mathrm{gnd}}$ is an extensional model, while $\mathcal{M}^{\mathrm{lazy}}$ is not.

2.3.2 The model $\mathcal{M}_{\mathsf{isdef}}^{\mathrm{lazy}}$.

To define this model of PCFP, we will refer to a set of terms extended by a new program constructor for testing whether a term has a w.h.n.f. (isdef). The resulting set of terms is denoted by $\Lambda_{\mathbf{T}}^{\mathsf{isdef}}$.

Extended terms. The term formation rule for the new constructor is the following:

$$(\mathrm{Isdef}) \quad \frac{\vdash M : \rho}{\vdash \mathsf{isdef}(M) : \mathsf{bool}} \quad ,$$

and the evaluation rule is as follows:

$$(\mathrm{ISDEF}) \quad \frac{N \Downarrow}{\mathsf{isdef}(N) \Downarrow \mathsf{true}} \quad .$$

The constructor isdef is just the extension of the isfn constructor of [42] page 280 to consider ground types and pairs. Note that (for non-ground types) isdef is not internally definable in PCFP, but it could be easily added.

The extended term model. The *closed term model* $\mathcal{M}_{\mathsf{isdef}}^{\mathrm{lazy}}$ of PCFP is defined in the obvious way by interpreting each type ρ as the set of the equivalence classes of the relation $\simeq_{\mathrm{obs}}^{\mathrm{lazy}}$ on the closed terms of type ρ in $\Lambda_{\mathbf{T}}^{\mathsf{isdef}}$. Note that for the language of extended terms, $\Lambda_{\mathbf{T}}^{\mathsf{isdef}}$, $\simeq_{\mathrm{obs}}^{\mathrm{lazy}}$ coincides with $\simeq_{\mathrm{obs}}^{\mathrm{gnd}}$. The constructor isdef is just a tool to make termination at higher types observable at the ground level.

3 A framework for non-standard type inference

In this section we present a simple framework for "non-standard type inference based" analyses of typed functional programs. The framework exploits common features of various static analyses by providing a "core" for non-standard type syntax, entailment, and assignment rules. A key feature of this framework is that it is based directly on operational semantics.

3.1 Non-standard types

For every PCFP type $\rho \in \mathbf{T}$ we consider a finite non-empty set $\mathbf{L}(\rho)$ of *non-standard types* (*ns-types* for short) expressing properties of terms of type ρ. Like "standard" types, ns-types are build from a set of basic ns-types (expressing "primitive" properties of values of type ρ). The properties expressed by basic ns-types are lifted at

$$(\mathbf{B})\ \frac{\phi \in \mathbf{B}(\rho)}{\phi \in \mathbf{L}(\rho)} \qquad (\rightarrow)\ \frac{\phi \in \mathbf{L}(\rho) \quad \psi \in \mathbf{L}(\sigma)}{\phi \rightarrow \psi \in \mathbf{L}(\rho \rightarrow \sigma)} \qquad (\times)\ \frac{\phi_1 \in \mathbf{L}(\rho_1) \quad \phi_2 \in \mathbf{L}(\rho_2)}{\phi_1 \times \phi_2 \in \mathbf{L}(\rho_1 \times \rho_2)}$$

Figure 4: "Core" non-standard types formation rules

higher types, following the type structure of the language, by the "$\rightarrow$" and "$\times$" constructors. For simplicity we do not do any formal distinction between type and ns-type constructors.

3.1.1 Syntax

For every type $\rho \in \mathbf{T}$, let $\mathbf{B}(\rho)$ be a finite set of *basic ns-types* (or *basic properties*) for type ρ. The sets $\mathbf{B}(\rho)$ depend on the particular analysis considered, however, in any case we require that

- for every PCFP ground type ι (i.e. for the ground types int and bool) there is a basic property $\top^\iota$ in $\mathbf{B}(\iota)$.

The set of ns-types for ρ $\mathbf{L}(\rho)$ is defined by adding to the set $\mathbf{B}(\rho)$ of basic ns-types for ρ the *structural* ns-types defined according to the structure of ρ as shown in Fig. 4. Note that owing to the above condition $\mathbf{L}(\rho)$ is not empty, for every type ρ. Let $\mathbf{L} = \cup_{\rho \in \mathbf{T}} \mathbf{L}(\rho)$.

Definition 3.1 *For every ns-type $\phi \in \mathbf{L}(\rho)$, ρ is called the* underlying type *of ϕ and it will be denoted by $\epsilon(\phi)$.*

The underlying type of a given ns-type ϕ is often understood. To simplify notations we will omit explicit mentions of underlying types (for instance the decorations of $\top$) except when these are not immediately clear from the context.

3.1.2 Semantics

For all the analyses considered in this paper the semantics of ns-types will be given according to the following principles.

Take a model $\mathcal{M}$ of the basic language (see e.g. [32] Section 4.5, for a general definition of model) which associates to any type ρ a suitable domain A^ρ. Non-standard types in $\mathbf{L}(\rho)$ are intended to represent properties of elements of A^ρ. This naturally suggests the interpretation of ns-types as subsets of A^ρ. We take however a more general approach (we will see soon a justification for that) and interpret ns-types in $\mathbf{L}(\rho)$ as *partial equivalence relations* (*p.e.r.s*) over A^ρ.

A partial equivalence relation B over a set A is a binary relation which is symmetric and transitive. The domain of B is $\{x \mid (x, y) \in B \text{ for some } y\}$. (Note that B is reflexive on its domain.)

Considering p.e.r.s instead of subsets makes possible to say, in addition to which elements have a given property (the elements that are in the domain of the p.e.r.),

$$
\begin{array}{lcl}
[\![\phi]\!]^{\mathcal{M}} & = & \mathbf{P}_{\rho}(\phi)^{\mathcal{M}}, \text{ if } \phi \in \mathbf{B}(\rho) \\
[\![\phi \to \psi]\!]^{\mathcal{M}} & = & \{\langle [M]^{\mathcal{M}}, [N]^{\mathcal{M}} \rangle \mid \\
 & & \quad \forall \langle [P]^{\mathcal{M}}, [Q]^{\mathcal{M}} \rangle \in [\![\phi]\!]^{\mathcal{M}}. \langle [MP]^{\mathcal{M}}, [NQ]^{\mathcal{M}} \rangle \in [\![\psi]\!]^{\mathcal{M}}\} \\
[\![\phi_1 \times \phi_2]\!]^{\mathcal{M}} & = & \{\langle [M]^{\mathcal{M}}, [N]^{\mathcal{M}} \rangle \mid \\
 & & \quad \forall i \in \{1,2\}.\ \langle [\mathsf{proj}_i M]^{\mathcal{M}}, [\mathsf{proj}_i N]^{\mathcal{M}} \rangle \in [\![\phi_i]\!]^{\mathcal{M}}\}
\end{array}
$$

Figure 5: P.e.r. semantics of "core" non-standard types

also which elements are indistinguishable w.r.t. the property (the elements in the same equivalence class). The interpretation of ns-types as subsets of the underlying domain will be considered as a particular case of the p.e.r. interpretation in which we are only interested in the elements satisfying a given property.

Since in this paper we are interested in the investigation of program properties in an operational settings, we will consider only closed term models induced by an operational equivalence between terms (which varies according to the considered properties). So in the following definition we assume that $\mathcal{M}$ is a term model, whose elements are equivalence classes of terms w.r.t. the considered operational equivalence.

Given a set A let $p.e.r.(A) = \{B \mid B \subseteq A \times A \text{ and } B \text{ is a p.e.r.}\}$. The interpretation of the basic ns-types of underlying type ρ is given by a family of functions (indexed over types):

$$\mathbf{P}_{\rho}(-)^{\mathcal{M}} : \mathbf{B}(\rho) \to p.e.r.(\mathbf{I}(\rho)^{\mathcal{M}})$$

which associates to each basic type in $\mathbf{B}(\rho)$ a non-empty p.e.r. over $\mathbf{I}(\rho)^{\mathcal{M}}$, the interpretation of type ρ in the model $\mathcal{M}$. For all ground type ι we require that

$$\mathbf{P}_{\iota}(\top^{\iota})^{\mathcal{M}} = \mathbf{I}(\iota)^{\mathcal{M}} \times \mathbf{I}(\iota)^{\mathcal{M}},$$

i.e. the ns-type $\top^{\iota}$ is always interpreted as the trivial p.e.r. over $\mathbf{I}(\iota)^{\mathcal{M}}$, representing the property that is true for all values of type ι.

The interpretation of ns-types is inductively defined from the interpretation of the basic ns-types in $\mathbf{B}(\rho)$ according to the set of clauses given in Fig. 5, which specify the semantics of the basic ns-types and of the ns-type constructors "$\to$" and "$\times$". The interpretations of the "$\to$" and "$\times$" ns-type constructors are as in standard type inference systems ([47]). In particular the interpretation of the "$\to$" ns-type constructor exhibits the well known monotonic-antimonotonic behaviour.

It is easy to see by induction on types that the interpretation $[\![\phi]\!]^{\mathcal{M}}$ of each ns-type ϕ is still a non empty p.e.r. over its underlying domain.

Definition 3.2 *Let $\mathbf{L}_{\top}$ be the subset of the ns-types defined by the following clauses:*

- $\top^{\iota} \in \mathbf{L}_{\top}$, *for all ground types ι*

- $\phi \to \psi \in \mathbf{L}_\top$, *for* $\phi \in \mathbf{L}$ *and* $\psi \in \mathbf{L}_\top$
- $\phi_1 \times \phi_2 \in \mathbf{L}_\top$, *for* $\phi_1, \phi_2 \in \mathbf{L}_\top$.

We can easily prove that the interpretation of all ns-types ϕ belonging to $\mathbf{L}_\top$ is $\mathbf{I}(\epsilon(\phi))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi))^{\mathcal{M}}$ (i.e. is trivial). Also the reverse implication is true, provided that to no other basic ns-type is explicitly given the trivial interpretation.

Proposition 3.3 *1. If* $\phi \in \mathbf{L}_\top$ *then* $[\![\phi]\!]^{\mathcal{M}} = \mathbf{I}(\epsilon(\phi))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi))^{\mathcal{M}}$.

2. Assume that, for all basic ns-types ϕ *different from* $\top^\iota$, $[\![\phi]\!]^{\mathcal{M}} \neq \mathbf{I}(\epsilon(\phi))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi))^{\mathcal{M}}$. *Then* $[\![\phi]\!]^{\mathcal{M}} = \mathbf{I}(\epsilon(\phi))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi))^{\mathcal{M}}$ *implies* $\phi \in \mathbf{L}_\top$.

Proof. 1. Immediate from the semantics.
2. Let $[\![\phi]\!]^{\mathcal{M}} = \mathbf{I}(\epsilon(\phi))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi))^{\mathcal{M}}$. The proof is by structural induction on $\phi \in \mathbf{L}$. The case $\phi \in \mathbf{B}(\rho)$, for some $\rho \in \mathbf{T}$, is immediate.
For $\phi = \phi_1 \to \phi_2$, assume that $\phi_1 \to \phi_2 \notin \mathbf{L}_\top$. Then $\phi_2 \notin \mathbf{L}_\top$ and so, by induction hypothesis there are $\langle [P]^{\mathcal{M}}, [Q]^{\mathcal{M}} \rangle \in \mathbf{I}(\epsilon(\phi_2))^{\mathcal{M}} \times \mathbf{I}(\epsilon(\phi_2))^{\mathcal{M}}$ such that $\langle [P]^{\mathcal{M}}, [Q]^{\mathcal{M}} \rangle \notin [\![\phi_2]\!]^{\mathcal{M}}$. But then, since $[\![\phi_1]\!]^{\mathcal{M}} \neq \emptyset$, we have that

$$\langle [\lambda z^{\epsilon(\phi_1)}.P]^{\mathcal{M}}, [\lambda z^{\epsilon(\phi_1)}.Q]^{\mathcal{M}} \rangle \notin [\![\phi_1 \to \phi_2]\!]$$

where z does not occur free in either P or Q, contradicting the hypothesis.
For $\phi = \phi_1 \times \phi_2$, the proof is similar. □

Non-standard types in $\mathbf{L}_\top$ represent then properties true for all terms of their underlying types. Such types are very useful from a technical point of view.

In some analysis all the basic properties are interpreted as p.e.r.s consisting of exactly one equivalence class. In this case we have that every non-standard type is interpreted as a p.e.r. consisting of exactly one equivalence class. We can then replace each p.e.r. by the subset corresponding to its equivalence class and switch to the (simpler) set semantics defined according to the clauses in Fig. 6, where the interpretation of each basic ns-type $\phi \in \mathbf{B}(\rho)$, is given by a family of functions $\mathbf{S}(-)^{\mathcal{M}}\rho$ (indexed over types ρ) such that:

$$\mathbf{S}_\rho(\phi)^{\mathcal{M}} = \{[M]^{\mathcal{M}} \mid \langle [M]^{\mathcal{M}}, [M]^{\mathcal{M}} \rangle \in \mathbf{P}(\phi)^{\mathcal{M}}\} \subseteq \mathbf{I}(\rho)^{\mathcal{M}} \ .$$

Note that, in this case, we have that, for all ground type ι, $[\![\top^\iota]\!]^{\mathcal{M}} = \mathbf{S}(\top^\iota)^{\mathcal{M}} = \mathbf{I}(\iota)^{\mathcal{M}}$. It is easy to see that a property similar to that stated in Proposition 3.3 holds for the set semantics of ns-types as well.

3.2 Entailment rules

The entailment relation between non-standard types (which represents a logical implication between properties) models the set-theoretic inclusion between their interpretations.

For each type ρ we define a formal relation $\leq_\rho$ between ns-types in $\mathbf{L}(\rho)$ which represent inclusion between ns-types of underlying type ρ. The formal judgements,

$$
\begin{aligned}
[\![\phi]\!]^{\mathcal{M}} &= \mathbf{S}_{\rho}(\phi)^{\mathcal{M}}, \text{ if } \phi \in \mathbf{B}(\rho) \\
[\![\phi \to \psi]\!]^{\mathcal{M}} &= \{[M]^{\mathcal{M}} \mid \forall [P]^{\mathcal{M}} \in [\![\phi]\!]^{\mathcal{M}}.[MP]^{\mathcal{M}} \in [\![\psi]\!]^{\mathcal{M}}\} \\
[\![\phi_1 \times \phi_2]\!]^{\mathcal{M}} &= \{[M]^{\mathcal{M}} \mid \forall i \in \{1,2\}.\ [\mathsf{proj}_i M]^{\mathcal{M}} \in [\![\phi_i]\!]^{\mathcal{M}}\}
\end{aligned}
$$

Figure 6: Set semantics of "core" non-standard types

$$
\text{(Ref) } \phi \leq \phi \quad \text{(Trans) } \frac{\phi_1 \leq \phi_2 \quad \phi_2 \leq \phi_3}{\phi_1 \leq \phi_3} \quad (\top) \ \frac{\psi \in \mathbf{L}_{\top} \quad \epsilon(\phi) = \epsilon(\psi)}{\phi \leq \psi}
$$

$$
(\to) \ \frac{\phi_1 \leq \phi_2 \quad \psi_1 \leq \psi_2}{\phi_2 \to \psi_1 \leq \phi_1 \to \psi_2} \quad (\times) \ \frac{\phi_1 \leq \psi_1 \quad \phi_2 \leq \psi_2}{\phi_1 \times \phi_2 \leq \psi_1 \times \psi_2}
$$

Figure 7: "Core" entailment rules for non-standard types

of the shape $\phi \leq \psi$, are derivable by a set of entailment rules. We will always omit the subscript ρ in $\leq_{\rho}$, since only ns-types of the same underlying type can be compared and then ρ is always understood. The basic rules for entailment are given in Fig. 7. These rules include reflectivity, transitivity, the basic rules for the "$\to$" and "$\times$" PCFP type constructors and a rule to represent the properties of the "trivial" basic ns-types in $\mathbf{L}_{\top}$. The intuition behind the entailment relation $\leq$ is that it corresponds to loss of information, i.e., it represents a logical implication between properties: $\phi_1 \leq \phi_2$ means that ϕ_2 is implied by ϕ_1, so it cannot contain more information than ϕ_1. Note the monotonic-antimonotonic behaviour of the arrow, typical of type inference systems. To these rules can be added other axiom schemes to reflect the meaning of specific basic properties. By $\cong$ we denote the equivalence relation induced by $\leq$.

As usual we say that the non-standard type entailment relation $\leq$ is *sound* w.r.t. the interpretation $[\![\cdot]\!]^{\mathcal{M}}$ if, for every $\phi, \psi \in \mathbf{L}$,

$$\phi \leq \psi \text{ implies } [\![\phi]\!]^{\mathcal{M}} \subseteq [\![\psi]\!]^{\mathcal{M}},$$

and *complete* w.r.t. the interpretation $[\![\cdot]\!]^{\mathcal{M}}$ if, for every $\phi, \psi \in \mathbf{L}$,

$$[\![\phi]\!]^{\mathcal{M}} \subseteq [\![\psi]\!]^{\mathcal{M}} \text{ implies } \phi \leq \psi.$$

It is just routine to prove a general soundness result for the core inclusion relation, saying that if $\phi \leq \psi$ holds then $[\![\phi]\!]^{\mathcal{M}} \subseteq [\![\psi]\!]^{\mathcal{M}}$ for all models and for all possible interpretations of basic ns-types. We do not do that since we are interested in soundness and completeness for the specific domains and inclusion axioms introduced in the next sections.

$$(\leq)\ \frac{\Sigma \vdash M : \phi \quad \phi \leq \psi}{\Sigma \vdash M : \psi}$$

$$\text{(Var)}\ \Sigma, x : \phi \vdash x : \phi \qquad \text{(Con)}\ \frac{\phi \in \mathbf{L}(\mathsf{c})}{\Sigma \vdash \mathsf{c} : \phi}$$

$$(\to \text{I})\ \frac{\Sigma, x : \phi \vdash M : \psi}{\Sigma \vdash \lambda x.M : \phi \to \psi} \qquad (\to \text{E})\ \frac{\Sigma \vdash M : \phi \to \psi \quad \Sigma \vdash N : \phi}{\Sigma \vdash MN : \psi}$$

$$(\times \text{I})\ \frac{\Sigma \vdash M_1 : \phi_1 \quad \Sigma \vdash M_2 : \phi_2}{\Sigma \vdash \langle M_1, M_2 \rangle : \phi_1 \times \phi_2} \qquad (\times \text{E}_i)\ \frac{\Sigma \vdash M : \phi_1 \times \phi_2}{\Sigma \vdash \mathsf{proj}_i M : \phi_i}\ i \in \{1,2\}$$

Figure 8: "Core" non-standard type assignment rules

3.3 Assignment rules

Non-standard types are assigned to terms by a set of inference rules.

If x^ρ is a term variable of type ρ an assumption for x^ρ is an expression of the shape $x^\rho : \phi$, or $x : \phi$ for short, where $\epsilon(\phi) = \rho$. A basis is a set Σ of non-standard type assumptions for term variables. $\Sigma, x : \psi$ denotes the basis $\Sigma \cup \{x : \psi\}$ where it is assumed that x does not appear in Σ.

For each constant c it is specified a finite non empty set $\mathbf{L}(\mathsf{c})$ of non-standard types for c. The set $\mathbf{L}(\mathsf{c})$ is such that every property that we want to derive for c is entailed by a property in $\mathbf{L}(\mathsf{c})$. That is, if ϕ is a property of c for some $\psi \in \mathbf{L}(\mathsf{c})$, $\psi \leq \phi$. Of course, for each $\phi \in \mathbf{L}(\mathsf{c})$, $\epsilon(\phi) = \mathbf{T}(\mathsf{c})$. The set $\mathbf{L}(\mathsf{c})$ is called the set of *minimal non-standard types for* c.

A non-standard typing statement is an expression $\Sigma \vdash M : \phi$ where M is a term of type $\epsilon(\phi)$ and Σ is a basis containing an assumption for each free variable of M. We write $\Sigma \vdash M : \phi$ to mean that this judgement can be derived by a set of assignment rules including the ones in Fig. 8, which represent the "core" ns-type assignment rules which are common to all analyses. They include

- rules for the basic program constructs variable, constant, λ-abstraction, application, pair and projection,
- a rule allowing the use of the entailment relation.

In a particular analysis other rules are added to those of Fig. 8 to take into account more specific properties. In any system there are, for instance, rules for the if and fix program constructors. General rules for if and fix constructors could be given but they would be too weak to be interesting. More interesting rules for these constructors can be given under specific interpretations.

Definition 3.4 *1. Two environments e_1, e_2 are Σ-related if and only if, for all $x : \psi \in \Sigma$, $\langle e_1(x), e_2(x) \rangle \in [\![\psi]\!]^{\mathcal{M}}$.*

2. *The ns-type assignment* $\vdash$ *is* sound *w.r.t. the interpretation* $[\![\cdot]\!]^{\mathcal{M}}$ *if, for all* e_1, e_2, *if* e_1 *and* e_2 *are* Σ*-related, then*

$$\Sigma \vdash M : \phi \text{ implies } \langle [\![M]\!]^{\mathcal{M}}_{e_1}, [\![M]\!]^{\mathcal{M}}_{e_2} \rangle \in [\![\phi]\!]^{\mathcal{M}}.$$

It is easy to prove that the assignment relation defined by the rules in Fig. 8 is sound with respect to any model $\mathcal{M}$ and for any interpretation of the basic ns-types. We do not state this result formally here since we are more interested in proving them for particular systems which include specific rules.

4 Instantiating the framework

To instantiate the framework described in Section 3 to a particular analysis we have to provide the following.

1. A specific term model defined via some operational program equivalence which include obviously β-convertibility (like $\mathcal{M}^{\mathrm{gnd}}$, $\mathcal{M}^{\mathrm{lazy}}$, ... in Section 2.3).
2. For every PCFP type $\rho \in \mathbf{T}$, the set of basic ns-types $\mathbf{B}(\rho)$ of underlying type ρ. Since $\top^{\iota} \in \mathbf{B}(\iota)$, for $\iota \in \{\mathsf{int}, \mathsf{bool}\}$ we will specify only the other ns-types in $\mathbf{B}(\iota)$.
3. The interpretation of the basic ns-types, given via the function $\mathbf{P}_{\rho}(\cdot)^{\mathcal{M}}$ mapping the basic ns-types in $\mathbf{B}(\rho)$ to p.e.r.s over $\mathbf{I}(\rho)^{\mathcal{M}}$.
4. The entailment rules for the basic properties (to be added to the rules in Fig. 7). These rules must obviously be sound with respect to the given semantics.
5. For each PCFP constant c the set of minimal non-standard types for c, $\mathbf{L}(\mathsf{c})$.
6. The additional assignment rules (to be added to the rules in Fig. 8) to take into account specific properties of the considered interpretation. Also in this case the new rules must be sound with respect to the considered semantics.

In the rest of this paper we show how the framework can be instantiated to dead-code and strictness analyses.

5 Dead-code analysis

We now consider a dead-code analysis w.r.t. ground contextual equivalence, i.e. an analysis such that, if a subterm N of a PCFP term M is proved to be useless (according to the analysis), then M is ground observational equivalent to the term obtained from M by replacing the useless subterm N with any term N' of the same type.

Considering the lazy contextual equivalence leads to the same entailment rules for properties and the inference rules for terms.

5.1 Dead-code types

5.1.1 Syntax

For every ground type $\iota \in \{\mathsf{int}, \mathsf{bool}\}$, the set of the basic properties is $\mathbf{B}^{\mathrm{d}}(\iota) = \{\delta^{\iota}\}$, while, for every non-ground type $\rho \in \mathbf{T} - \{\mathsf{int}, \mathsf{bool}\}$, we have $\mathbf{B}^{\mathrm{d}}(\rho) = \emptyset$. The corresponding set of non-standard types, called *dead-code types* (*d-types* for short), is denoted by $\mathbf{L}^{\mathrm{d}}$.

5.1.2 Semantics

Notation 5.1 *For every type $\rho \in \mathbf{T}$, let Δ_ρ be the diagonal p.e.r. over $\mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}$, i.e.*

$$\Delta_\rho = \{\langle [M]^{\mathcal{M}^{\mathrm{gnd}}}, [M]^{\mathcal{M}^{\mathrm{gnd}}}\rangle \mid [M]^{\mathcal{M}^{\mathrm{gnd}}} \in \mathbf{I}(\rho)^{\mathcal{M}^{\mathrm{gnd}}}\}.$$

For every $\iota \in \{\mathsf{int}, \mathsf{bool}\}$, let $\mathbf{P}(\delta^{\iota})^{\mathcal{M}^{\mathrm{gnd}}} = \Delta_\iota$.

This semantics says that, for every $\iota \in \{\mathsf{int}, \mathsf{bool}\}$, δ^{ι} characterizes terms whose values have a precise identity ("useful" terms), since their interpretation codifies the fact that an occurrence of a term having this non-standard type is allowed to be replaced only with an observational equivalent term. In the context of dead-code analysis $\top^{\iota}$ characterizes terms whose values do not have a precise identity ("useless" terms), since their interpretation codifies the fact that an occurrence of a term having this non-standard type is allowed to be replaced with any term of the same type. These properties are propagated to higher types. For instance a function F of type $\mathsf{int} \to \mathsf{int}$ which has the ns-types $\delta^{\mathsf{int}} \to \top^{\mathsf{int}}$ or $\top^{\mathsf{int}} \to \top^{\mathsf{int}}$ (which are indeed equivalent) yields a useless result whenever applied to any argument, and is so itself dead-code. To detect the dead code in a given term M we will assume that M is a useful term and try to assign the maximum amount of $\top$'s to its subterms (i.e. to detect as many useless subterms as possible).

Definition 5.2 *1. For every type ρ, let $\delta(\rho)$ be the d-type obtained by replacing each occurrence of any ground type ι by δ^{ι}, and similarly let $\top(\rho)$ be the d-type obtained by replacing each occurrence of any ground type ι by $\top^{\iota}$.*

2. Let $\mathbf{L}^{\mathrm{d}}_{\delta} = \{\delta(\rho) \mid \rho \in \mathbf{T}\}$ be the subset of the d-types containing only δ's.

Proposition 5.3 *For all $\rho \in \mathbf{T}$, $[\![\delta(\rho)]\!] = \Delta_\rho$.*

Proof. By induction on $\rho \in \mathbf{T}$. The case $\rho \in \{\mathsf{int}, \mathsf{bool}\}$ is trivial.

Let $\rho = \rho_1 \to \rho_2$. By induction hypothesis both $[\![\delta(\rho_1)]\!] = \Delta_{\rho_1}$ and $[\![\delta(\rho_2)]\!] = \Delta_{\rho_2}$. By definition of $\delta(\rho_1 \to \rho_2) = \delta(\rho_1) \to \delta(\rho_2)$ we have that

(*) $\langle [P], [Q]\rangle \in [\![\delta(\rho_1 \to \rho_2)]\!]$ if and only if, for all M and N such that $\langle [M], [N]\rangle \in [\![\delta(\rho_1)]\!]$, $\langle [P\,M], [Q\,N]\rangle \in [\![\delta(\rho_2)]\!]$.

We prove the result by showing that

$$P \simeq^{\mathrm{gnd}}_{\mathrm{obs}} Q \iff \langle [P], [Q]\rangle \in [\![\delta(\rho_1 \to \rho_2)]\!].$$

Let $P \simeq^{\mathrm{gnd}}_{\mathrm{obs}} Q$. For all M and N such that $\langle [M], [N] \rangle \in [\![\delta(\rho_1)]\!]$, by induction hypothesis $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$, and so $P\,M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} Q\,N$. Again by induction hypothesis $\langle [P\,M], [Q\,N] \rangle \in [\![\delta(\rho_2)]\!]$, and by $(*)$ $\langle [P], [Q] \rangle \in [\![\delta(\rho_1 \to \rho_2)]\!]$.

Let $P \not\simeq^{\mathrm{gnd}}_{\mathrm{obs}} Q$ then (by Theorem 2.7 of Section 2) $P \not\simeq Q$, so there would be an M such that $P\,M \not\simeq Q\,M$ and therefore $P\,M \not\simeq^{\mathrm{gnd}}_{\mathrm{obs}} Q\,M$ (again by Theorem 2.7). So (by induction hypothesis) $\langle [P\,M], [Q\,N] \rangle \notin [\![\delta(\rho_2)]\!]$ and therefore (from $(*)$) $\langle [P], [Q] \rangle \notin [\![\delta(\rho_1 \to \rho_2)]\!]$. The case $\rho = \rho_1 \times \rho_2$ is similar. □

Also the reverse implication holds, i.e. we have that $[\![\phi]\!] = \Delta_{\epsilon(\phi)}$ implies that $\phi \in \mathbf{L}^{\mathrm{d}}_{\delta}$. This can be be proved using Theorem 5.5.

5.2 Entailment rules

In the case of dead-code analysis the ns-type entailment relation is completely represented by the rules of Fig. 7. In fact we can always take any piece of code (also a useful one) and mark it as useless (rule $(\top)$). No other specific axiom is required, in this case, except the basic rules.

Definition 5.4 (Entailment relation for dead-code types) *Let $\phi, \psi \in \mathbf{L}^{\mathrm{d}}$. We write $\phi \leq_{\mathrm{d}} \psi$ to mean that $\phi \leq \psi$ is derivable by using the rules in Fig. 7. By $\cong_{\mathrm{d}}$ we denote the equivalence relation induced by $\leq_{\mathrm{d}}$.*

The $\leq_{\mathrm{d}}$ entailment relation between d-types is sound and complete w.r.t. the p.e.r. interpretation in $\mathcal{M}^{\mathrm{gnd}}$.

Theorem 5.5 (Soundness and completeness of $\leq_{\mathrm{d}}$) *$\phi \leq_{\mathrm{d}} \psi$ if and only if $[\![\phi]\!] \subseteq [\![\psi]\!]$.*

Proof. The proof of soundness is by induction on the derivation of the judgement $\phi \leq_{\mathrm{d}} \psi$. The proof of the completeness result, instead, is not straightforward. It can be found in [17] Chapter 7. □

5.3 Assignment rules

To give the assignment rules we define for each PCFP constant the set of minimal d-types as follows: for every $\mathsf{c} \in \mathcal{K}$, $\mathbf{L}^{\mathrm{d}}(\mathsf{c}) = \{\delta(\mathbf{T}(\mathsf{c}))\}$, i.e. the minimal d-type of a PCFP constant c of type ρ, is the δ-d-type $\delta(\rho)$. This says that if the output is useful then all the inputs are useful. Moreover since every PCFP constant c has a type that is either ι_1, or $\iota_1 \to \iota_2$, or $\iota_1 \times \iota_2 \to \iota_3$, for $\iota_1, \iota_2, \iota_3 \in \{\mathsf{int}, \mathsf{bool}\}$, we have that, for every d-type ϕ, $\delta(\mathbf{T}(\mathsf{c})) \leq_{\mathrm{d}} \phi$ implies either $\phi = \delta(\mathbf{T}(\mathsf{c}))$ or $\phi \in \mathbf{L}^{\mathrm{d}}_{\top}$, where $\mathbf{L}^{\mathrm{d}}_{\top}$ is the subset of the d-types ϕ identifying useless terms, inductively defined as in Definition 3.2.

Definition 5.6 (D-type assignment system $\vdash_{\mathrm{d}}$) *We write $\Sigma \vdash_{\mathrm{d}} M : \phi$ to mean that $\Sigma \vdash M : \phi$ can be derived by the rules in Fig.s 8 and 9, where in the rule $(\leq)$ of Fig. 8 it is used the entailment relation $\leq_{\mathrm{d}}$.*

$$(\text{Fix})\ \frac{\Sigma, x : \phi \vdash M : \phi}{\Sigma \vdash \mathsf{fix}\, x.x : \phi}$$

$$(\text{If}_\delta)\ \frac{\Sigma \vdash N : \delta^{\mathsf{bool}} \quad \Sigma \vdash M_1 : \phi \quad \Sigma \vdash M_2 : \phi}{\Sigma \vdash \mathsf{if}\, N\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2 : \phi}$$

$$(\text{If}_\top)\ \frac{\Sigma \vdash N : \top^{\mathsf{bool}} \quad \Sigma \vdash M_1 : \top(\rho) \quad \Sigma \vdash M_2 : \top(\rho)}{\Sigma \vdash \mathsf{if}\, N\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2 : \top(\rho)}$$

Figure 9: Rules for d-type assignment (system $\vdash_{\mathrm{d}}$)

To simplify the presentation of the soundness of the d-type assignment system and of the dead-code elimination procedure (in Section 5.4) w.r.t. the semantics $\mathcal{M}^{\mathrm{gnd}}$ we introduce the following notation.

Notation 5.7 *Let $\Sigma \vdash_{\mathrm{d}} M : \phi$ and $\Sigma \vdash_{\mathrm{d}} N : \phi$. We write $M \sim_\phi^\Sigma N$ to mean that for all e_1, e_2, if e_1 and e_2 are Σ-related, then $\langle [\![M]\!]_{e_1}, [\![N]\!]_{e_2} \rangle \in [\![\phi]\!]$.*

Soundness of the d-type assignment system $\vdash_{\mathrm{d}}$ is stated by the following theorem.

Theorem 5.8 (Soundness of $\vdash_{\mathrm{d}}$) *Let $\Sigma \vdash_{\mathrm{d}} M : \phi$. Then $M \sim_\phi^\Sigma M$.*

Proof. By induction on the derivation of $\Sigma \vdash_{\mathrm{d}} M : \phi$. □

In using d-type informations to optimize programs we need to examine not only the final type of a term but also the types assigned to its subtypes.

Example 5.9 *Take the term*

$$P = \lambda x^{\mathsf{int}}.(\lambda y^{\mathsf{int}}.x)x,$$

of type $\rho = \mathsf{int} \to \mathsf{int}$. By using the assignment rules in Fig. 8 it is possible to assign to P d-type

$$\delta^{\mathsf{int}} \to \delta^{\mathsf{int}} \ \textit{(useful term of type } \mathsf{int} \to \mathsf{int}\textit{)},$$

by a derivation which assigns d-type

$$\top^{\mathsf{int}} \ \textit{(useless term of type } \mathsf{int}\textit{)}$$

to the second occurrence of x in the body of the λ-abstraction P. In fact, for $\Sigma = \{x : \delta^{\mathsf{int}}, y : \top^{\mathsf{int}}\}$ and $\Sigma' = \{x : \delta^{\mathsf{int}}\}$, we have the following derivation.

$$(\to \text{E})\ \frac{(\to \text{I})\ \dfrac{(\text{Var})\ \Sigma \vdash_{\mathrm{d}} x : \delta^{\mathsf{int}}}{\Sigma' \vdash_{\mathrm{d}} \lambda y^{\mathsf{int}}.x : \top^{\mathsf{int}} \to \delta^{\mathsf{int}}} \qquad (\leq)\ \dfrac{(\text{Var})\ \Sigma' \vdash_{\mathrm{d}} x : \delta^{\mathsf{int}} \quad \delta^{\mathsf{int}} \leq \top^{\mathsf{int}}}{\Sigma' \vdash_{\mathrm{d}} x : \top^{\mathsf{int}}}}{(\to \text{I})\ \dfrac{\Sigma' \vdash_{\mathrm{d}} (\lambda y^{\mathsf{int}}.x)x : \delta^{\mathsf{int}}}{\emptyset \vdash_{\mathrm{d}} \lambda x^{\mathsf{int}}.(\lambda y^{\mathsf{int}}.x)x : \delta^{\mathsf{int}} \to \delta^{\mathsf{int}}}}$$

This means that the value of the second occurrence of x is not relevant in any computation involving the evaluation of P.

Since we are interested in finding the subterms to which $\top$-d-types are assigned, in the following we will consider a compact representation of a deduction of the form

$$M^{\phi},$$

where M^{ϕ} is a decorated term, i.e. a term in which each subterm is decorated by the d-type assigned to it in the considered deduction. Using this notation (and writing, for short, δ instead of δ^{int} and $\top$ instead of $\top^{\mathsf{int}}$) the derivation of Example 5.9 is represented by the decorated term

$$(\lambda x^{\delta}.((\lambda y^{\top}.x^{\delta})^{\top\to\delta}x^{\top})^{\delta})^{\delta\to\delta}.$$

When the $(\leq)$ rule is applied, as in

$$(\leq)\ \frac{\Sigma \vdash M : \phi \quad \phi \leq \psi}{\Sigma \vdash M : \psi},$$

N is decorated with the greatest d-type, i.e. ψ. Observe that the use of decorated terms allows us to see that the second occurrence of x in the body of the λ-abstraction P is dead-code just by inspecting the associated d-type. We write $\Sigma \vdash_{\mathrm{d}\star} M^{\phi}$ to mean that M^{ϕ} is a representation of a deduction of $\Sigma \vdash_{\mathrm{d}} M : \phi$. Note that the same terms can have different decorations. Let $\Lambda_{\mathbf{T}}^{\vdash_{\mathrm{d}\star}}$ be the set of $\vdash_{\mathrm{d}\star}$-decorated PCFP terms, i.e.,

$$\Lambda_{\mathbf{T}}^{\vdash_{\mathrm{d}\star}} = \{M^{\phi} \mid \phi \text{ is a d-type and } \Sigma \vdash_{\mathrm{d}\star} M^{\phi} \text{ for some basis } \Sigma\}.$$

Notation 5.10 *We extend the notations $\epsilon(\cdot)$, $\delta(\cdot)$, and $\top(\cdot)$ to terms. For every decorated term M^{ϕ}, $\epsilon(M^{\phi})$ is simply the term obtained from M^{ϕ} by erasing all the d-type decorations. For every term M, $\delta(M)$ ($\top(M)$) is the decorated term obtained from M by decorating each subterm of M of type σ by $\delta(\sigma)$ (resp. $\top(\sigma)$).*

The proof of the following fact is immediate.

Fact 5.11 *1. $\Sigma \vdash_{\mathrm{d}\star} M^{\phi}$ implies $\vdash_{\mathbf{T}} \epsilon(M^{\phi}) : \epsilon(\phi)$ and $\{x^{\epsilon(\phi)} \mid x^{\epsilon(\phi)} : \phi \in \Sigma\} \supseteq \mathrm{FV}(M)$.*

2. $\vdash_{\mathbf{T}} M : \rho$ implies $\{x^{\rho} : \delta(\rho) \mid x^{\rho} \in \mathrm{FV}(M)\} \vdash_{\mathrm{d}\star} \delta(M)$, and for all basis Σ such that $\Sigma = \{x^{\epsilon(\psi)} : \psi \mid x^{\epsilon(\psi)} \in \mathrm{FV}(M) \text{ and } \psi \in \mathbf{L}^{\mathrm{d}}\}$, $\Sigma \vdash_{\mathrm{d}\star} \top(M)$.

5.4 A dead-code elimination

In this section we introduce a dead-code elimination mapping $\mathbf{O}$ that, given an $\vdash_{\mathrm{d}\star}$-decorated term M^{ϕ}, defines an optimized version of it.

5.4.1 Dummy variables

For each type ρ, define d^{ρ}, d_1^{ρ}, d_2^{ρ}, ..., be *dummy variables* of type ρ. We remark that dummy variables are not present in the original programs: they are introduced by

the dead-code elimination mapping $\mathbf{O}$ as placeholders for the dead-code removed[3] . So in the following we assume that all the occurrences of the dummy variables in a program are free (i.e. there are no bound dummy variables) and distinct (i.e. each dummy variable occurs at most once in a program). For every term M define $\mathrm{DV}(M)$ to be the set of dummy variables of M.

5.4.2 The simplification mapping

Given $(\lambda x^{\delta}.((\lambda y^{\top}.x^{\delta})^{\top\to\delta}x^{\top})^{\delta})^{\delta\to\delta}$ that is the decorated version of the term $P = \lambda x^{\mathsf{int}}.(\lambda y^{\mathsf{int}}.x)x$ of Example 5.9, we can produce a simplified version of P by removing the dead-code showed by the decoration, i.e. the second occurrence of x in the body of the λ-abstraction. In particular we can replace this dead-code with some placeholder d and obtain

$$P' = \lambda x^{\mathsf{int}}.(\lambda y^{\mathsf{int}}.x)\mathsf{d}^{\mathsf{int}},$$

which is ground observationally equivalent to P.

Definition 5.12 (Simplification mapping O) *1. The function* $\mathbf{O} : \Lambda_{\mathbf{T}}^{\vdash_{\mathrm{d}\star}} \to \Lambda_{\mathbf{T}}^{\vdash_{\mathrm{d}\star}}$ *takes a decorated term M and replaces every largest subterm decorated by a d-type $\phi \in \mathbf{L}_{\top}^{\mathrm{d}}$ with a (fresh) dummy variable d^{ϕ}.*

2. If Σ is a basis then $\mathbf{O}(\Sigma) = \{x : \chi \mid x : \chi \in \Sigma$ and $\chi \notin \mathbf{L}_{\top}^{\mathrm{d}}\}$.

We have immediately that if $\Sigma \vdash_{\mathrm{d}\star} M^{\phi}$ then $\mathbf{O}(\Sigma) \subseteq \Sigma$ and $\Sigma' \vdash_{\mathrm{d}\star} \mathbf{O}(M^{\phi})$, where $\Sigma' = \mathbf{O}(\Sigma) \cup \{\mathsf{d}^{\sigma} : \top(\sigma) \mid \mathsf{d}^{\sigma} \in \mathrm{DV}(\epsilon(\mathbf{O}(M^{\phi}))\}$. Moreover we we have that the simplification mapping is correct w.r.t. the d-type semantics.

Proposition 5.13 (Correctness of O w.r.t. the d-types semantics) *If $\Sigma \vdash_{\mathrm{d}\star} M^{\phi}$ then $\epsilon(M^{\phi}) \sim_{\phi}^{\Sigma'} \epsilon(\mathbf{O}(M^{\phi}))$, where $\Sigma' = \Sigma \cup \{\mathsf{d}^{\sigma} : \top(\sigma) \mid \mathsf{d}^{\sigma} \in \mathrm{DV}(\epsilon(\mathbf{O}(M^{\phi})))\}$.*

Proof. By induction on the derivation $\Sigma \vdash_{\mathrm{d}\star} M^{\phi}$. □

In order to use the simplification mapping $\mathbf{O}$ to simplify terms while preserving their meaning (w.r.t. $\mathcal{M}^{\mathrm{gnd}}$) we identify a subset of $\vdash_{\mathrm{d}}$-typings for which the $\sim_{\phi}^{\Sigma}$ relations implies the $\simeq_{\mathrm{obs}}^{\mathrm{gnd}}$ relation.

Definition 5.14 (Faithful $\vdash_{\mathrm{d}}$-typing) $\Sigma \vdash_{\mathrm{d}} M : \phi$ *is a* faithful $\vdash_{\mathrm{d}}$*-type assignment statement if $\phi \in \mathbf{L}_{\delta}^{\mathrm{d}}$, and for all $x : \psi \in \Sigma$, $\psi \in \mathbf{L}_{\top}^{\mathrm{d}} \cup \mathbf{L}_{\delta}^{\mathrm{d}}$.*

The proof that the simplification performed by the mapping $\mathbf{O}$ on faithful decorated terms preserve $\simeq_{\mathrm{obs}}^{\mathrm{gnd}}$ rely on the following lemma.

Lemma 5.15 *Let $\Sigma \vdash_{\mathrm{d}} M : \phi$ and $\Sigma \vdash_{\mathrm{d}} N : \phi$ be faithful $\vdash_{\mathrm{d}}$-typings. Then $M \sim_{\phi}^{\Sigma} N$ implies $M \simeq_{\mathrm{obs}}^{\mathrm{gnd}} N$.*

[3] Dummy variables are just a tool for proving the soundness of the simplification mapping. In a real implementation we can can replace dummy variables by some polymorphic "dummy constant" d.

Proof. Let Σ' be the restriction of Σ to the variables $\{x_1, \ldots, x_n\} = \mathrm{FV}(M) \cup \mathrm{FV}(N)$ (observe that Σ' contains exactly the free variables of M and N). It is easy to see that, for any environment e, the fact that $\Sigma \vdash_{\mathrm{d}} M : \phi$ and $\Sigma \vdash_{\mathrm{d}} N : \phi$ are faithful, implies that e is Σ' related with e. Let now $M \sim_\phi^\Sigma N$, then (by Notation 5.7) $\langle [\![M]\!]_e, [\![N]\!]_e \rangle \in [\![\phi]\!]$.

We conclude the proof by observing that, from Proposition 5.3 and Theorem 2.7, we have that (since $\phi \in \mathbf{L}_\delta^{\mathrm{d}}$): if, for all environments e, $\langle [\![M]\!]_e, [\![N]\!]_e \rangle \in [\![\phi]\!]$ then $M \simeq_{\mathrm{obs}}^{\mathrm{gnd}} N$. □

We can now show how to use the system $\vdash_{\mathrm{d}\star}$ and the simplification mapping $\mathbf{O}$ to simplify terms in observationally equivalent ones. In particular we show that all the dummy variables introduced by applying the mapping $\mathbf{O}$ on a faithful $\vdash_{\mathrm{d}\star}$-typing are dead-code.

Theorem 5.16 ($\mathbf{O}$ on faithful $\vdash_{\mathrm{d}\star}$-typings preserves $\simeq_{\mathrm{obs}}^{\mathrm{gnd}}$) *Let $\Sigma \vdash_{\mathrm{d}\star} M^\phi$ be a faithful $\vdash_{\mathrm{d}\star}$-typing. Then $\epsilon(M^\phi) \simeq_{\mathrm{obs}}^{\mathrm{gnd}} \epsilon(\mathbf{O}(M^\phi))$.*

Proof. Let $\Sigma' = \Sigma \cup \{\mathsf{d}^\sigma : \top(\sigma) \mid \mathsf{d}^\sigma \in \mathrm{DV}(\epsilon(M^\phi))\}$. It easy to see that if $\Sigma \vdash_{\mathrm{d}\star} M^\phi$ is a faithful $\vdash_{\mathrm{d}\star}$-typing then also $\Sigma' \vdash_{\mathrm{d}\star} M^\phi$ and $\Sigma' \vdash_{\mathrm{d}\star} \mathbf{O}(M^\phi)$ are faithful $\vdash_{\mathrm{d}\star}$-typings. So the result follows immediately from Proposition 5.13 and Lemma 5.15. □

In [17] Chapter 7 it is proved that for every PCFP term M there is a faithful $\vdash_{\mathrm{d}\star}$-typing showing all the dead-code that can be proved by the d-type assignment system. This result is proved by giving an algorithm which computes such "optimal" $\vdash_{\mathrm{d}\star}$-typing.

Remark 5.17 *It is not difficult to define a clean-up operation which eliminates (some of) the place-holders* d *introduced by the simplification process. In this way the term* $P' = \lambda x^{\mathsf{int}}.(\lambda y^{\mathsf{int}}.x)\mathsf{d}^{\mathsf{int}}$, *considered at the beginning of this section, will be simplified to the equivalent (but simpler) term* x^{int}. *This final clean-up operation is important from a practical point of view, and should be considered when implementing program simplification tools based on the techniques described in this paper. For more details we refer to [17] Chapter 7, where an incremental dead-code detection and elimination algorithm based on the dead-code analysis presented in this section is given.*

5.5 An example from program extraction

In this section we show a non-trivial example of dead-code detection and elimination. This example, due to C. Mohring and further developed by S. Berardi and L. Boerio (see [8] and [10]), is based on an instance of dead-code that can arise in programs extracted from proofs (see [39, 38] for an introduction to the subject).

1. The term

$$\mathsf{rec}_\tau = \lambda m^\tau.\lambda f^{\mathsf{int}\to\tau\to\tau}.\mathsf{fix}\, r^{\mathsf{int}\to\tau}.\lambda n^{\mathsf{int}}.\mathsf{if}\ n = 0\ \mathsf{then}\ m\ \mathsf{else}\ f n(r(\mathsf{pred} n))$$

of type $\tau \to (\mathsf{int} \to \tau \to \tau) \to \mathsf{int} \to \tau$ represents the primitive recursor operator from natural numbers to τ since, for every M, F and N of the proper types:

- $\mathsf{rec}MF\mathsf{0} \simeq^{\mathsf{gnd}}_{\mathsf{obs}} M$, and
- if $N \Downarrow \mathsf{c}$ and $\mathsf{c} \neq \mathsf{0}$ then $\mathsf{rec}MFN \simeq^{\mathsf{gnd}}_{\mathsf{obs}} FN(\mathsf{rec}MF(\mathsf{pred}N))$.

2. The term

$$\mathsf{it}_\tau = \lambda m^{\mathsf{int}}.\lambda g^{\tau\to\tau}.\mathsf{fix}\, r^{\mathsf{int}\to\tau}.\lambda n^{\mathsf{int}}.\mathsf{if}\ n = \mathsf{0}\ \mathsf{then}\ m\ \mathsf{else}\ g(r(\mathsf{pred}n))$$

of type $\tau \to (\tau \to \tau) \to \mathsf{int} \to \tau$ instead represent a simplified form of the operator represented by rec_τ: it represents an "iterator" which allows to iterate n times a function g on an input m. In fact, for every M, G and N of the proper types:

- $\mathsf{it}MG\mathsf{0} \simeq^{\mathsf{gnd}}_{\mathsf{obs}} M$, and
- if $N \Downarrow \mathsf{c}$ and $\mathsf{c} \neq \mathsf{0}$ then $\mathsf{it}MGN \simeq^{\mathsf{gnd}}_{\mathsf{obs}} G(\mathsf{it}MG(\mathsf{pred}N))$.

Consider now the types $\rho = \mathsf{int} \to \mathsf{int}$, $\sigma = \mathsf{int} \to (\mathsf{int}\times\mathsf{int}) \to (\mathsf{int}\times\mathsf{int})$, $\tau = \mathsf{int}\times\mathsf{int}$. We will show how to use the system $\vdash_{\mathsf{d}\star}$ to detect and remove the dead-code in the term, of type ρ,

$$P \quad = \quad \lambda x^{\mathsf{int}}.\mathsf{proj}_1(\mathsf{rec}_\tau\ \langle M_1, M_2\rangle\, F\, x)$$

where

- $F = \lambda n^{\mathsf{int}}.\lambda m^{\mathsf{int}\times\mathsf{int}}.\langle h_1(\mathsf{proj}_1 m), h_2(\mathsf{proj}_1 m)(\mathsf{proj}_2 m)\rangle$ is a term of type σ, and
- M_1, M_2 are closed terms of type int,

(note that $\mathrm{FV}(P) = \mathrm{FV}(F) = \{h_1^{\mathsf{int}\to\mathsf{int}}, h_2^{\mathsf{int}\to\mathsf{int}\to\mathsf{int}}\}$). We have that $\Sigma \vdash_{\mathsf{d}\star} P'^{\phi}$ is a faithful $\vdash_{\mathsf{d}\star}$-typing, where (writing, for short, δ and $\top$ instead of δ^{int} and $\top^{\mathsf{int}}$, and omitting the d-type decoration which are clear from the context):

$$\begin{array}{l}
\Sigma = \{h_1 : \delta \to \delta, h_2 : \top \to \top \to \top\}\ , \\
P'^{\phi} = (\lambda x.\mathsf{proj}_1(\mathsf{rec}'_\tau\ \langle M_1, M_2\rangle^{\delta\times\top}\, F'^{\psi}\, x^{\delta}))^{\phi}\ , \\
\phi = \delta \to \delta\ , \\
\mathsf{rec}'_\tau = \lambda m.\lambda f.\mathsf{fix}\, r^{\delta\to\delta\times\top}.\lambda n.\mathsf{if}\ n = \mathsf{0}\ \mathsf{then}\ m\ \mathsf{else}\ f n^{\top}(r(\mathsf{pred}n^{\delta})^{\delta})^{\delta\times\top}\ , \\
F'^{\psi} = (\lambda n.\lambda m.\langle h_1(\mathsf{proj}_1 m^{\delta\times\top})^{\delta}, h_2(\mathsf{proj}_1 m)^{\top}(\mathsf{proj}_2 m)^{\top}\rangle)^{\psi}\ , \\
\psi = \top \to (\delta\times\top) \to (\delta\times\top)\ .
\end{array}$$

Let Q, ri_τ, and G be the terms obtained from P'^{ϕ}, rec'_τ, and F'^{ψ} by applying the $\mathbf{O}$ mapping and erasing all the d-type decorations, i.e. $Q = \epsilon(\mathbf{O}(P'^{\phi}))$, $\mathsf{ri}_\tau = \epsilon(\mathbf{O}(\mathsf{rec}'_\tau))$, and $G = \epsilon(\mathbf{O}(F'^{\psi}))$. It is not difficult to see that

$$Q \quad = \quad \lambda x^{\mathsf{int}}.\mathsf{proj}_1(\mathsf{ri}_\tau\ \langle M_1, \mathsf{d}_1^{\mathsf{int}}\rangle, G\, x)), \quad \text{where}$$

- $\mathsf{ri}_\tau = \lambda m^{\tau}.\lambda f^{\mathsf{int}\to\tau\to\tau}.\mathsf{fix}\, r^{\mathsf{int}\to\tau}.\lambda n^{\mathsf{int}}.\mathsf{if}\ n = \mathsf{0}\ \mathsf{then}\ m\ \mathsf{else}\ f\mathsf{d}_2^{\mathsf{int}}(r(\mathsf{pred}n))$, and

- $G = \lambda n^{\mathsf{int}}.\lambda m^{\mathsf{int}\times\mathsf{int}}.\langle h_1(\mathsf{proj}_2 m), \mathsf{d}_3^{\mathsf{int}}\rangle$.

Then, by eliminating the dummy placeholders (see Remark 5.17 above), we obtain as final result of the simplification the term:

$$R \quad = \quad \lambda x^{\mathsf{int}}.\mathsf{it}_{\mathsf{int}}\, M_1\, H\, x \ ,$$

where

- $\mathsf{it}_{\mathsf{int}}$ is as defined at the beginning of this section, and
- $H = \lambda m_1^{\mathsf{int}}.f m_1$.

6 Strictness analysis

In this section we consider strictness analysis, i.e. the problem of determining whether a function diverges whenever its input diverges. If a PCFP term F is proved to be strict (according to the analysis) then, for every possible argument M, the (value obtained from the) evaluation of FM by making the application of F to M with a call-by-need is observationally equivalent to the (value obtained from the) evaluation of FM by making the application of F to M with a call-by-value.

We first analyze the rules required to make the two applications equivalent w.r.t. the lazy observational equivalence, and then the ones required to make them equivalent w.r.t. the ground observational equivalence (but not necessarily w.r.t. the lazy one).

6.1 Strictness types

6.1.1 Syntax

For every $\rho \in \mathbf{T}$, the set of the basic properties is $\mathbf{B}^{\mathrm{s}}(\rho) = \{\bot^\rho\}$. The corresponding set of non-standard types, called *strictness types* (*s-types* for short), is be denoted by $\mathbf{L}^{\mathrm{s}}$.

6.1.2 Semantics

For every $\rho \in \mathbf{T}$, let $\mathbf{P}(\bot^\rho)^{\mathcal{M}^{\mathrm{lazy}}} = \{\langle [\mathsf{fix}\, x^\rho.x]^{\mathcal{M}^{\mathrm{lazy}}}, [\mathsf{fix}\, x^\rho.x]^{\mathcal{M}^{\mathrm{lazy}}}\rangle\}$.

Since every basic property is interpreted as a p.e.r. consisting of exactly one class, we can switch to the set semantics described at the end of Section 3.1.2 and define, for every $\rho \in \mathbf{T}$, $\mathbf{S}(\bot^\rho)^{\mathcal{M}^{\mathrm{lazy}}} = \{[\mathsf{fix}\, x^\rho.x]^{\mathcal{M}^{\mathrm{lazy}}}\}$. It is easy to see that $[\mathsf{fix}\, x^\rho.x]^{\mathcal{M}^{\mathrm{lazy}}} \in [\![\phi]\!]^{\mathcal{M}^{\mathrm{lazy}}}$ for all s-types ϕ such that $\epsilon(\phi) = \rho$. So $[\![\bot^\rho]\!]^{\mathcal{M}^{\mathrm{lazy}}}$ is included in the interpretation of all s-types in $\mathbf{L}^{\mathrm{s}}(\rho)$.

This semantics says that, for every $\rho \in \mathbf{T}$, $\bot^\rho$ (which is interpreted as the p.e.r. consisting of one class whose unique element is the $\simeq_{\mathrm{obs}}^{\mathrm{lazy}}$-class of the PCFP terms of type ρ without w.h.n.f.) is the property of being a divergent term of type ρ, characterizing the PCFP terms of type ρ without w.h.n.f. (lazy observational

equivalent to $\mathsf{fix}\, x^{\rho}.x$). Using s-types we can express strictness properties of functions. For instance the s-type $\bot^{\mathsf{int}} \to \top^{\mathsf{int}} \to \bot^{\mathsf{int}} \in \mathbf{L}^{\mathrm{s}}(\mathsf{int} \to \mathsf{int} \to \mathsf{int})$ represent the property of a function of two arguments which is strict in its first argument. In fact the functions having this s-type give a result belonging to (the interpretation of) $\bot^{\mathsf{int}}$ (i.e. undefined) whenever their first argument is undefined, independently of the value of the second argument (which can be anything in the proper type). Similarly a function of s-type $\top^{\mathsf{int}} \to \bot^{\mathsf{int}} \to \bot^{\mathsf{int}} \in \mathbf{L}^{\mathrm{s}}(\mathsf{int} \to \mathsf{int} \to \mathsf{int})$ is strict in its second argument.

6.2 Entailment rules

Let $\mathbf{L}^{\mathrm{s}}_{\top}$ denote the subset of "trivial" s-types of Definition 3.2.

Definition 6.1 (Entailment relation for strictness types) *Let $\phi, \psi \in \mathbf{L}^{\mathrm{s}}$. We write $\phi \leq_{\mathrm{s}} \psi$ to mean that $\phi \leq \psi$ is derivable by using the rules in Fig. 7 and the following axiom capturing the semantics of the basic s-types $\bot^{\rho}$:*

$$(\bot)\ \bot^{\epsilon(\phi)} \leq \phi\ .$$

By $\cong_{\mathrm{s}}$ we denote the equivalence relation induced by $\leq_{\mathrm{s}}$.

Soundness of entailment $\leq_{\mathrm{s}}$ (for both $\mathcal{M}^{\mathrm{lazy}}$ and $\mathcal{M}^{\mathrm{lazy}}_{\mathsf{isdef}}$) is stated by the following theorem.

Theorem 6.2 (Soundness of $\leq_{\mathrm{s}}$) *Let $\phi, \psi \in \mathbf{L}^{\mathrm{s}}$. $\phi \leq_{\mathrm{s}} \psi$ implies $[\![\phi]\!] \subseteq [\![\psi]\!]$.*

Proof. By induction on the derivation of the entailment judgment. □

The entailment rules of Definition 6.1 are not complete w.r.t. the semantics $\mathcal{M}^{\mathrm{lazy}}$, as the following example shows.

Example 6.3 (Incompleteness of $\leq_{\mathrm{s}}$ w.r.t. $\mathcal{M}^{\mathrm{lazy}}$) *In $\mathcal{M}^{\mathrm{lazy}}$ the functional terms with w.h.n.f. cannot be discriminated. For instance even though*

$$[\![\bot^{\mathsf{int}\to\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}} \neq [\![\top^{\mathsf{int}} \to \bot^{\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}}$$

(in particular $[\![\top^{\mathsf{int}} \to \bot^{\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}} = [\![\bot^{\mathsf{int}\to\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}} \cup \{[\lambda y^{\mathsf{int}}.\mathsf{fix}\, x^{\mathsf{int}}.x]^{\mathcal{M}^{\mathrm{lazy}}}\}$), we have that

$$[\![\bot^{\mathsf{int}\to\mathsf{int}} \to \bot^{\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}} = [\![(\top^{\mathsf{int}} \to \bot^{\mathsf{int}}) \to \bot^{\mathsf{int}}]\!]^{\mathcal{M}^{\mathrm{lazy}}}\ .$$

This is because there is no PCFP context of ground type *with a "hole" of type $\rho \to \sigma$ that converges if and only if filled with a term having a w.h.n.f., and, in particular, that converges when filled by $\lambda y^{\rho}.\mathsf{fix}\, x^{\sigma}.x$ and diverges when filled by $\mathsf{fix}\, z^{\rho\to\sigma}.z$. However*

$$\bot^{\mathsf{int}\to\mathsf{int}} \to \bot^{\mathsf{int}} \not\leq_{\mathrm{s}} (\top^{\mathsf{int}} \to \bot^{\mathsf{int}}) \to \bot^{\mathsf{int}}\ .$$

At the moment we do not know whether a sound (and decidable) extension of the $\leq_{\mathrm{s}}$ relation which is complete w.r.t. the model $\mathcal{M}^{\mathrm{lazy}}$ exists. The entailment $\leq_{\mathrm{s}}$ is indeed complete w.r.t. the term model $\mathcal{M}^{\mathrm{lazy}}_{\mathsf{isdef}}$ of Section 2.3.2.

$$\mathbf{L}^{\mathrm{s}}(\mathsf{n}) = \{\top^{\mathsf{int}}\} \qquad \mathbf{L}^{\mathrm{s}}(\mathsf{true}) = \mathbf{L}^{\mathrm{s}}(\mathsf{false}) = \{\top^{\mathsf{bool}}\}$$
$$\mathbf{L}^{\mathrm{s}}(\mathsf{succ}) = \mathbf{L}^{\mathrm{s}}(\mathsf{pred}) = \{\bot^{\mathsf{int}} \to \bot^{\mathsf{int}}\} \qquad \mathbf{L}^{\mathrm{s}}(\mathsf{not}) = \{\bot^{\mathsf{bool}} \to \bot^{\mathsf{bool}}\}$$
$$\mathbf{L}^{\mathrm{s}}(+) = \mathbf{L}^{\mathrm{s}}(-) = \{\bot^{\mathsf{int}} \times \top^{\mathsf{int}} \to \bot^{\mathsf{int}}, \top^{\mathsf{int}} \times \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}\}$$
$$\mathbf{L}^{\mathrm{s}}(\mathsf{and}) = \mathbf{L}^{\mathrm{s}}(\mathsf{or}) = \{\bot^{\mathsf{bool}} \times \top^{\mathsf{bool}} \to \bot^{\mathsf{bool}}, \top^{\mathsf{bool}} \times \bot^{\mathsf{bool}} \to \bot^{\mathsf{bool}}\}$$
$$\mathbf{L}^{\mathrm{s}}(=) = \{\bot^{\mathsf{int}} \times \top^{\mathsf{int}} \to \bot^{\mathsf{bool}}, \top^{\mathsf{int}} \times \bot^{\mathsf{int}} \to \bot^{\mathsf{bool}}\}$$

Figure 10: Strictness types for the PCFP constants

$$(\mathrm{Fix}_\bot)\ \frac{\Sigma, x:\phi_1 \vdash M:\phi_2 \ \cdots\ \Sigma, x:\phi_k \vdash M:\phi_1}{\Sigma \vdash \mathsf{fix}\, x.M : \phi_i}\ 1 \le i \le k$$

$$(\mathrm{If}_\bot)\ \frac{\Sigma \vdash N : \bot^{\mathsf{bool}} \quad \vdash_{\mathbf{T}} M_1 : \rho \quad \vdash_{\mathbf{T}} M_2 : \rho}{\Sigma \vdash \mathsf{if}\ N\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \bot^{\rho}}$$

$$(\mathrm{If}_\top)\ \frac{\Sigma \vdash N : \top^{\mathsf{bool}} \quad \Sigma \vdash M_1 : \phi \quad \Sigma \vdash M_2 : \phi}{\Sigma \vdash \mathsf{if}\ N\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \phi}$$

Figure 11: Rules for strictness type assignment (system $\vdash_{\mathrm{s}}$)

Theorem 6.4 (Completeness of $\leq_{\mathrm{s}}$ w.r.t. $\mathcal{M}^{\mathrm{lazy}}_{\mathrm{isdef}}$) *Let $\phi, \psi \in \mathbf{L}^{\mathrm{s}}$. $[\![\phi]\!]^{\mathcal{M}^{\mathrm{lazy}}_{\mathrm{isdef}}} \subseteq [\![\psi]\!]^{\mathcal{M}^{\mathrm{lazy}}_{\mathrm{isdef}}}$ implies $\phi \leq_{\mathrm{s}} \psi$.*

The proof of this result is quite long, it can be found in [17] Chapter 4.

6.3 Assignment rules

For each constant $\mathsf{c} \in \mathcal{K}$ the set of minimal s-types $\mathbf{L}^{\mathrm{s}}(\mathsf{c})$ is defined as in Fig. 10. The minimal types given to the unary and binary constants says that such constants are strict in their arguments. In particular for binary constants this is expressed by having two types in the set, the first says that the constants is strict in the first argument and the second that is strict in the second argument.

Definition 6.5 (Strictness type assignment system) *We write $\Sigma \vdash_{\mathrm{s}} M : \phi$ to mean that $\Sigma \vdash M : \phi$ can be derived by the rules in Fig.s 8 and 11, where in the rule $(\leq)$ of Fig. 8 it is used the entailment relation $\leq_{\mathrm{s}}$.*

Note that the same terms can have different s-types.

Example 6.6 *Take the term* $\mathrm{app} = \lambda f^{\mathsf{int}\to\mathsf{int}}.\lambda x^{\mathsf{int}}.fx$, *of type* $\rho = (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$. *By using the system $\vdash_{\mathrm{s}}$ it is possible to assign to* app *both the s-types*

- $\phi_1 = (\bot^{\mathsf{int}} \to \bot^{\mathsf{int}}) \to \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}$ *(strict function to strict function), and*
- $\phi_2 = (\top^{\mathsf{int}} \to \bot^{\mathsf{int}}) \to \top^{\mathsf{int}} \to \bot^{\mathsf{int}}$ *(function divergent on every argument to function divergent on every argument),*

as the following derivations show.

1. *Let* $\Sigma_1 = \{f : \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}, x : \bot^{\mathsf{int}}\}$, *and* $\Sigma_1' = \{f : \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}\}$.

$$(\to \mathrm{E})\ \frac{(\mathrm{Var})\ \Sigma_1 \vdash f : \bot^{\mathsf{int}} \to \bot^{\mathsf{int}} \qquad (\mathrm{Var})\ \Sigma_1 \vdash x : \bot^{\mathsf{int}}}{(\to \mathrm{I})\ \dfrac{\Sigma_1 \vdash fx : \bot^{\mathsf{int}}}{(\to \mathrm{I})\ \dfrac{\Sigma_1' \vdash \lambda x^{\mathsf{int}}.fx : \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}}{\emptyset \vdash \mathrm{app} : \phi_1}}}$$

2. *Let* $\Sigma_2 = \{f : \top^{\mathsf{int}} \to \bot^{\mathsf{int}}, x : \top^{\mathsf{int}}\}$, *and* $\Sigma_2' = \{f : \top^{\mathsf{int}} \to \bot^{\mathsf{int}}\}$.

$$(\to \mathrm{E})\ \frac{(\mathrm{Var})\ \Sigma_2 \vdash f : \top^{\mathsf{int}} \to \bot^{\mathsf{int}} \qquad (\mathrm{Var})\ \Sigma_2 \vdash x : \top^{\mathsf{int}}}{(\to \mathrm{I})\ \dfrac{\Sigma_2 \vdash fx : \bot^{\mathsf{int}}}{(\to \mathrm{I})\ \dfrac{\Sigma_2' \vdash \lambda x^{\mathsf{int}}.fx : \top^{\mathsf{int}} \to \bot^{\mathsf{int}}}{\emptyset \vdash \mathrm{app} : \phi_2}}}$$

The proof of the following fact is immediate.

Fact 6.7 *1.* $\Sigma \vdash_{\mathrm{s}} M : \phi$ *implies* $\vdash_{\mathbf{T}} M : \epsilon(\phi)$ *and* $\{x^{\epsilon(\phi)} \mid x^{\epsilon(\phi)} : \phi \in \Sigma\} \supseteq \mathrm{FV}(M)$.

2. $\vdash_{\mathbf{T}} M : \rho$ *implies, for all basis* Σ *such that* $\Sigma = \{x^{\epsilon(\psi)} : \psi \mid x^{\epsilon(\psi)} \in \mathrm{FV}(M)$ *and* $\psi \in \mathbf{L}^{\mathrm{s}}\}$, *for all* $\phi \in \mathbf{L}^{\mathrm{s}}_{\top}(\rho)$, $\Sigma \vdash_{\mathrm{s}} M : \phi$.

The previous fact says that we can always derive a non-informative s-type.

The language $\mathbf{L}^{\mathrm{s}}$ is a subset of the language of properties of the *conjunctive strictness logic* introduced in [25] and (independently) in [7]. This subset is obtained from the formulas of the conjunctive strictness logic, as presented in [7] page 36, by forbidding the use of the conjunction operator. This restriction greatly reduces the power of the logic. However, our rule ($\mathrm{Fix}_{\bot}$) introduces an implicit (and limited) use of conjunction, by allowing (in its antecedent) the derivation of more than one s-type for M.

The following example shows an application of the ($\mathrm{Fix}_{\bot}$) rule.

Example 6.8 *The function* $F = \mathsf{fix}\, f^{\rho}.M$, *where* $\rho = \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ *and*

$$M = \lambda x^{\mathsf{int}}.\lambda y^{\mathsf{int}}.\lambda z^{\mathsf{int}}.\mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ fyx(z - 1) \quad ,$$

was used in [29] to show the limitation of a type system without conjunction.

Using $\vdash_{\mathrm{s}}$ *we are able to prove that* F *is strict in each of its 3 arguments, i.e.* $\emptyset \vdash_{\mathrm{s}} F : \phi_1$, $\emptyset \vdash_{\mathrm{s}} F : \phi_2$, *and* $\emptyset \vdash_{\mathrm{s}} F : \phi_3$, *where*

- $\phi_1 = \bot^{\mathsf{int}} \to \top^{\mathsf{int}} \to \top^{\mathsf{int}} \to \bot^{\mathsf{int}}$,
- $\phi_2 = \top^{\mathsf{int}} \to \bot^{\mathsf{int}} \to \top^{\mathsf{int}} \to \bot^{\mathsf{int}}$, *and*
- $\phi_3 = \top^{\mathsf{int}} \to \top^{\mathsf{int}} \to \bot^{\mathsf{int}} \to \bot^{\mathsf{int}}$.

Note that to assign ϕ_3 to F one premise is sufficient, in fact $\{f : \phi_3\} \vdash_s M : \phi_3$, and so it is possible to apply the rule $(\mathrm{Fix}_\perp)$ *as follows:*

$$(\mathrm{Fix}_\perp)\ \frac{\{f : \phi_3\} \vdash_s M : \phi_3}{\emptyset \vdash_s \mathsf{fix}\, f^\rho.M : \phi_3}\ .$$

To assign ϕ_1 or ϕ_2 to F, instead, two premises are required. In fact, $\{f : \phi_1\} \vdash_s M : \phi_2$ and $\{f : \phi_2\} \vdash_s M : \phi_1$, and so two premises are sufficient, since (for $i \in \{1, 2\}$):

$$(\mathrm{Fix}_\perp)\ \frac{\{f : \phi_1\} \vdash_s M : \phi_2 \quad \{f : \phi_2\} \vdash_s M : \phi_1}{\emptyset \vdash_s \mathsf{fix}\, f^\rho.M : \phi_i}\ ,$$

but (as it is easy to check) they are also needed. So with the rule for fixpoint of [29] (that allows just one premise) neither ϕ_1 nor ϕ_2 can be assigned to F.

Soundness of the s-types assignment system (for both $\mathcal{M}^{\mathrm{lazy}}$ and $\mathcal{M}^{\mathrm{lazy}}_{\mathsf{isdef}}$) is stated by the following theorem.

Theorem 6.9 (Soundness of $\vdash_s$) *Let $\Sigma \vdash_s M : \phi$. Then $[\![M]\!]_e \in [\![\phi]\!]$ for all e such that, for all $x : \psi \in \Sigma$, $e(x) \in [\![\psi]\!]$.*

Proof. By induction on the derivation of $\Sigma \vdash_s M : \phi$. □

6.4 Strictness analysis and ground contextual equivalence

The entailment relation presented in Section 6.2 can be modified to fit better the ground contextual equivalence, $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$, instead of the lazy contextual equivalence, $\simeq^{\mathrm{lazy}}_{\mathrm{obs}}$. In this perspective an s-type ϕ is interpreted as a subset of the interpretation of the type $\epsilon(\phi)$ in the model $\mathcal{M}^{\mathrm{gnd}}$ of Section 2.3.1. For every s-type $\phi \in \mathbf{L}^{\mathbf{s}}$, let $[\![\phi]\!]^{\mathcal{M}^{\mathrm{gnd}}}$ be the interpretation of ϕ in the model $\mathcal{M}^{\mathrm{gnd}}$ (defined according to the clauses in Fig. 5). The ground observational equivalence, $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$, is such that $\mathsf{fix}\, x^{\rho\to\sigma}.x \simeq^{\mathrm{gnd}}_{\mathrm{obs}} \lambda y^\rho.\mathsf{fix}\, z^\sigma.z$, so the associated entailment relation between s-types will identify all the s-types corresponding to functions that diverge for all inputs.

The choice of the ground semantics would be justified, as explained in [7] page 19, by taking a view in which

(∗) *programs* are closed terms of basic type (int or bool) and the only behaviours that a program can exhibit are to diverge or to converge to an integer or boolean value.

So the only way in which we can observe terms of higher types is to plug them into a complete program. The semantics $\mathcal{M}^{\mathrm{lazy}}$, instead, does not identify $\mathsf{fix}\, x^{\mathsf{int}\to\mathsf{int}}.x$ and $\lambda y^{\mathsf{int}}.\ \mathsf{fix}\, z^{\mathsf{int}}.z$.

Since $M \simeq^{\mathrm{lazy}}_{\mathrm{obs}} N$ implies $M \simeq^{\mathrm{gnd}}_{\mathrm{obs}} N$, all optimizations preserving $\simeq^{\mathrm{lazy}}_{\mathrm{obs}}$ preserve also $\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$. The vice versa is not true. For instance, consider the term $M = \lambda g^{\mathsf{int}\to\mathsf{int}}.\lambda y^{\mathsf{int}}.g\, y$. We can assign to M the s-type $(\top^{\mathsf{int}} \to \perp^{\mathsf{int}}) \to \top^{\mathsf{int}} \to \perp^{\mathsf{int}}$ (the assignment does not require the use of $\leq$ so it can be done both for the ground

and the lazy system). Now if we consider the entailment for the ground semantics $\top^{\mathsf{int}} \to \bot^{\mathsf{int}}$ is equivalent to $\bot^{\mathsf{int}\to\mathsf{int}}$. So M, seen as a function of one argument, is strict in its argument and the application of M to P can be done "by value" (first evaluating P and then doing the replacement) preserving the ground observational equivalence. W.r.t. $\mathcal{M}^{\mathrm{lazy}}$ this is not the case, since observing MP at the type $\mathsf{int} \to \mathsf{int}$, when $P = \mathsf{fix}\, x^{\mathsf{int}\to\mathsf{int}}.x$, we have that $MP \Downarrow$ whereas the evaluation of the application MP "by value" leads to a divergent computation, i.e. the lazy observational equivalence is not preserved. However MP is still a function and it will result in a non terminating term whenever applied to any term. If P is undefined the whole program will produce a non-terminating computation. So if our ultimate goal is to obtain a result of ground type there is no harm in considering M as strict and to apply it by value.

The above considerations suggest that, under the assumption $(*)$, $\mathcal{M}^{\mathrm{gnd}}$ is indeed a more convenient model (rather than $\mathcal{M}^{\mathrm{lazy}}$) for proving the soundness of strictness analysis.

Remark 6.10 *Soundness of (a small variant of) system $\vdash_{\mathrm{s}}$ w.r.t. the model $\mathcal{M}^{\mathrm{gnd}}$ is proved in [18] (see also [17] Chapter 6), where a sound and complete inference algorithm for this strictness type assignment system is given.*

7 Final remarks

The research on program analysis is very active. In this paper we have focused on a specific approach to the problem (non-standard type inference) and on a specific class of languages (functional programming languages). A number of approaches to program analysis have been proposed and extensively studied in the literature. In Section 1 we have partially quoted the literature on non-standard type inference and abstract interpretation. An overview of some of the main approaches to program analysis can be found, for instance, in the forthcoming book [37].

References

[1] H. Abelson and G. J. Susman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–39, 1990.

[3] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[4] H. P. Barendregt. Lambda calculi with types. In S. Abramsky *et al*, editor, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Claredon Press, UK, 1992.

[5] E. Barendsen and S. Smetsers. A Derivation System for Uniqueness Typing . In *SEGRAGRA'95*. Elsevier, Electronic Notes in Theoretical Computer Science, 1995.

[6] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.2*. INRIA-Rocquencourt-CNRS-ENS Lyon, May 1998.

[7] P. N. Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, University of Cambridge, Pembroke College, 1992.

[8] S. Berardi. Pruning Simply Typed Lambda Terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

[9] S. Berardi and L. Boerio. Minimum Information Code in a Pure Functional Language with Data Types. In *TLCA'97*, LNCS 1210. Springer–Verlag, 1997.

[10] L. Boerio. *Optimizing Programs Extracted from Proofs*. PhD thesis, Università di Torino, 1995.

[11] G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[12] F. Cardone and M. Coppo. Two Extensions of Curry's Inference System. In P. Odifreddi, editor, *Logic and Computer Science*, pages 19–75. Accademic Press, 1990.

[13] M. Coppo and A. Ferrari. Type inference, abstract interpretation and strictness analysis. *Theoretical Computer Science*, 121:113–145, 1993.

[14] P. Cousot. Types as Abstract Interpretations. Invited paper. In *POPL'97*, pages 316–331. ACM, 1997.

[15] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximations of Fixedpoints . In *POPL'77*, pages 238–252. ACM, 1977.

[16] P. Cousot and R. Cousot. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages). Invited paper. In *ICCL'94*, pages 95–112. IEEE, 1994.

[17] F. Damiani. *Non-standard type inference for functional programs*. PhD thesis, Università di Torino, February 1998. Available from http://www.di.unito.it/~damiani.

[18] F. Damiani and P. Giannini. An Inference Algorithm for Strictness. In *TLCA'97*, LNCS 1210, pages 129–146. Springer–Verlag, 1997.

[19] F. Damiani and F. Prost. Detecting and Removing Dead Code using Rank 2 Intersection. In *International Workshop TYPES'96, Selected Papers*, LNCS, pages 66–87. Springer–Verlag, 1998.

[20] P. de Mast, J.-M. Jansen, D. Bruin, J. Fokker, P. Koopman, S. Smetsers, M. van Eekelen, and R. Plasmeijer. Functional Programming in Clean - draft, 1997. Available from http://www.cs.kun.nl/~clean.

[21] C. Hankin and D. Le Métayer. Lazy type inference and program analysis. *Science of Computer Programming*, 25:219–249, 1995.

[22] R. Hindley. *Basic Simple Type Theory.* Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.

[23] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[24] L.S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Partial Evaluation and Semantics-based Program Manipulation.* ACM, 1991.

[25] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, University of London, Imperial College, 1992.

[26] T. P. Jensen. Abstract Interpretation over Algebraic Datatypes. In *ICCL'94.* IEEE, 1994.

[27] T. P. Jensen. Conjunctive Type Systems and Abstract Interpretation of Higher-order Functional Programs. *Journal of Logic and Computation*, 5(4):397–421, 1995.

[28] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming Of Future Generation Computer.* Elsevier Sciences B.V. (North-Holland), 1988.

[29] T. M. Kuo and P. Mishra. Strictness analysis: a new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272. ACM, 1989.

[30] X. Leroy. The Caml Light System, documentation and users's guide, 1997. Available from http://paulliac.inria.fr/caml.

[31] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised.* MIT press, 1997.

[32] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[33] C. Mossin. *Flow analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, 1997. Revised version.

[34] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, LNCS 83, pages 269–281. Springer–Verlag, 1980.

[35] A. Mycroft. *Abstract Interpretation and Optimizing Transforming for Applicative programs.* PhD thesis, University of Edinburgh, Scotland, 1981.

[36] F. Nielson. Strictness Analysis and Denotational Abstract Interpretation. *Information and Computation*, 76(1):29–92, 1988.

[37] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* In preparation, http://www.daimi.au.dk/~hrn/PPA/ppa.html, 1999.

[38] C. Paulin-Mohring. Extracting F_ω's Programs from Proofs in the Calculus of Constructions. In *POPL'89.* ACM, 1989.

[39] C. Paulin-Mohring. *Extraction de Programme dans le Calcul des Constructions.* PhD thesis, Université Paris VII, 1989.

[40] J. Peterson, K. Hammond, L. Agustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Pejton Jones, A. Reis, and P. Wadler. Report on the Programming Language Haskell (version 1.4) - draft, 1997. Available from http://www-i2.informatik.rwth-aaken.de/Forshung/FP/Haskell.

[41] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[42] A. M. Pitts. Operationally-based theories of program equivalence. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, pages 241–298. Cambridge University Press, 1997.

[43] R. Plasmeijer and M. van Eekelen. The Cuncurrent Clean Language Report (version 1.2) - draft, 1997. Available from http://www.cs.kun.nl/~clean.

[44] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[45] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[46] F. Prost. Marking techniques for extraction. Technical Report RR95-47, Ecole Normale Supérieure de Lyon, Lyon, December 1995.

[47] D. S. Scott. Data Types as Lattices. *Siam J. Computing*, 5(3):522–587, 1976.

[48] K. L. Solberg. *Annotated Type Systems for Program Analysis.* PhD thesis, Aarhus University, Denmark, 1995. Revised version.

[49] Y. Takayama. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.

[50] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, LNCS 201, pages 1–16. Springer–Verlag, 1985.

[51] D. A. Wright. *Reduction Types and Intensionality in the Lambda-Calculus.* PhD thesis, University of Tasmania, 1992.