Redundant-code detection and elimination for PCF with algebraic Datatypes

Ferruccio Damiani

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: damiani@di.unito.it

Abstract. We present a non-standard type assignment system and simplifications mappings for detecting and removing redundant-code in simply typed functional programs with algebraic datatypes and recursive function definitions. We characterize two classes of redundant code: the *dead-code*, that is code that is never executed under the lazy-call-by-name evaluation strategy, and the *minimum-information-code*, that is code (in general non-constant) that contributes to the computation only with a minimum amount of constant information. The soundness of the analysis and of the induced program transformations is proved via a partial equivalence relation semantics of the non-standard types, showing that the simplified programs are observationally greater than the original ones. The main application of the technique is the optimization of programs extracted from proofs in logical frameworks but it could be used as well in the elimination of redundant-code determined by program specialization.

1 Introduction

Redundant-code analysis for typed functional programming languages has been mainly studied in the context of *logical frameworks*, like Coq [1] (see [14] for a short survey on logical frameworks), to remove redundant-code from functional programs extracted from formal proofs (see [13,12] 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 and some sort of simplification is mandatory. To this aim various simplification techniques have been proposed in the last ten years (e.g. [19,2,5,18,3,6,8,4]). More in general redundant-code elimination is worthwhile in the same situations in which is useful to perform partial evaluation.

Let us look at a couple of examples of redundant-code detection and elimination.

Example 1. Let $M = (\lambda x^{\mathsf{int}}.3)P$ where P is a term of type int. Since x is never used in the body of the λ-abstraction $F = \lambda x^{\mathsf{int}}.3$, we have that the value of M can be computed without using P, which is therefore redundant-code. In fact, in a *lazy-call-by-name* language[1], M behaves like the term $M' = (\lambda x^{\mathsf{int}}.3)\mathsf{d}$, where d is a place-holder for the redundant-code removed.

Note that it is indeed possible to simplify the redundant-code in a more substantial way, i.e. by removing the useless pair $\langle$formal parameter x, actual parameter $\mathsf{d}\rangle$ and replacing M with the body of the λ-abstraction (the constant 3).

In the following we will call *dead-code* the redundant-code that, like P above, is never executed under the lazy-call-by-name evaluation strategy.

The next (more complex) example introduces another class of redundant-code, called *minimum-information-code*, that has been characterized by Berardi and Boerio in [4].

Example 2. Let $\mathsf{intList} = \mathsf{data}\,X.\mathsf{Nil} \;[\!]\; \mathsf{Cons}(\mathsf{int}, X)$ be the datatype of the list of integers, and $\mathsf{bool} = \mathsf{data}\,X.\mathsf{True} \;[\!]\; \mathsf{False}$ be the datatype of booleans. Consider the term $N = \lambda x^{\mathsf{int}}.GPA$, where

$$\begin{array}{lll} G : \mathsf{int} \to \mathsf{intList} \to \mathsf{int} & = \lambda y^{\mathsf{int}}.\lambda z^{\mathsf{intList}}.\mathsf{case}\, z \,\mathsf{of}\, \{\ \mathsf{Nil}\ \mathsf{to}\ Q_1 \;[\!]\; \mathsf{Cons}(h,t)\ \mathsf{to}\ Q_2\ \}\ , & \text{and} \\ A : \mathsf{intList} & = \mathsf{case}\, B \,\mathsf{of}\, \{\ \mathsf{True}\ \mathsf{to}\ \mathsf{Cons}(E_1, L_1) \;[\!]\; \mathsf{False}\ \mathsf{to}\ \mathsf{Cons}(E_2, L_2)\ \} & \end{array}$$

[1] Like Miranda, Haskell and Clean.

for some terms P, Q_1, Q_2, B, E_1, L_1, E_2 and L_2. It is easy to see that Q_1 is redundant-code[2], in particular dead-code, so it could be replaced by a place-holder d. Let z, h and t do not occur in Q_2, then we have that E_1, L_1, E_2, L_2 are dead-code, and can therefore replaced by place-holders $\mathsf{d}_1', \mathsf{d}_1'', \mathsf{d}_2', \mathsf{d}_2''$. Suppose also that the variable y does not occur in Q_2, but only in Q_1 (that has been removed), then also P is dead-code and it can be replaced by a place-holder d_3.

We have removed a lot of *dead-code*, but in the term N there is other redundant-code, since the subexpression A is an example of *minimum-information-code*: it contributes to the computation only by providing the Cons data-constructor of $\mathsf{intList}$. Quoting from Berardi and Boerio [4] we say that the subexpression A is minimum-information-code since *"we use only the first symbol of its output, and this first symbol is always the same"*. Minimum-information-code can be simplified by exploiting the (minimum) information that it provides. For the program N above this amounts to

1. replace the minimum-information-code A by the term $A' = \mathsf{Cons}(\mathsf{d}_1, \mathsf{d}_2)$, which provides the same (minimum) information, and
2. replace the term G (which uses the minimum-information-code A) by the term $G' = \lambda y^{\mathsf{int}}.\lambda z^{\mathsf{intList}}.Q_2$,

obtaining the simplified term $N' = \lambda x^{\mathsf{int}}.G'\mathsf{d}_3 A'$.

Note that replacing A by A' may have changed the termination behaviour of N (even w.r.t. the lazy-call-by-name evaluation strategy[3]): minimum-information-code is redundant-code, but is *not* dead-code, and simplifying it may change the termination property of the program[4] (i.e. it may happen that the simplified program converges while the original one diverges).

Moreover after turning G to G' we have that A' is dead-code[5] and it can be replaced by a place-holder d_4. Note that A is not dead-code in N, but: *by simplifying the minimum-information-code A in N (i.e. by turning G to G' and A to A') we have obtained a term N' containing* new *dead-code.* By this second simplification step we obtain the term: $N'' = \lambda x^{\mathsf{int}}.G'\mathsf{d}_3\mathsf{d}_4$, that does not contain redundant-code (supposing that Q_2 does not).

Also in this case (as in Example 1) it is possible to simplify the term in a clever way by eliminating the place-holders for the dead-code removed. I.e. the application $G'\mathsf{d}_3\mathsf{d}_4$ can be replaced by the body, Q_2 of the function G'. After this last "cosmetic" simplification[6] we obtain the term: $N''' = \lambda x^{\mathsf{int}}.Q_2$, which is significant more simple than the original one. We remark that the transformation of N in N''' cannot be performed by (partially) evaluating the program N, since the value of the subexpression B may depend on the formal parameter x.

In Section 2 of this paper we introduce the programming language we are dealing with and its operational semantics. Section 3 briefly shows how program properties can be represented by partial equivalence relations on a term model of the programming language.

In Section 4 we describe the language of non-standard types (that we call *evaluation types*) and its semantics, we also give a complete axiomatization of the logical implication between evaluation types. Section 5 presents an evaluation type assignment system and a program simplification based on the information provided by the evaluation types assigned to a program and to its subexpressions. Related work is considered in Section 6.

---

[2] Since A (the actual value of the formal parameter z) either diverges or converges to a term of the form $\mathsf{Cons}(E, L)$ (for some E, L), so Q_1, which is the first branch of the case examining A, is never executed.

[3] Suppose for instance that, for some values of the parameter x, the term B is divergent.

[4] In Berardi and Boerio [4] the simplification of minimum-information-code preserves the termination behaviour since the language considered is strongly normalizing.

[5] Remember that the variable z does not occur in Q_2.

[6] The elimination of (some of) the place-holders introduced by the simplification process is quite easy to do, the difficult part of program simplification is in the previous steps in which dead-code and minimum-information-code are detected and simplified by introducing place-holders. However this final clean-up operation is important from a practical point of view, and should be considered when implementing a program simplification tool based on the technique described in this paper.

2 The language PCFD

In this section we introduce a simple functional programming language and its operational semantics. The acronym PCFD stands for "Programming Computable Function with lazy algebraic Datatypes", since this language is a dialect of the language PCF [16] obtained by adding algebraic datatypes.

This section and the next one (Section 3) are routine work: the original contributions of the paper are in Sections 4 and 5. For a more comprehensive presentation of the material in Sections 2.1 and 2.2 see, for instance, Pitts [15] and Gordon [9,10].

2.1 PCFD syntax and evaluation rules

The set of PCFD types is defined assuming as *ground* type the set of integers, int. Types are ranged over by ρ, σ, τ (with superscripts and subscript when necessary), and *algebraic datatypes* (*datatypes* in the following) are ranged over by μ.

Definition 3 (PCFD types). The language of *types* ($\mathbf{T}$) is defined by the grammar in Fig. 1 where, in a datatype $\mu = \mathsf{data}\, X.\mathbf{dc}_1 [\![\cdots [\![\mathbf{dc}_m$, the type variable X is bound by the data-binder and, for each clause $\mathbf{dc}_i = \mathsf{C}_i(\diamondsuit_{i,1}, \ldots, \diamondsuit_{i,a_i})$, each $\diamondsuit_{i,j}$ is either a type $\rho \in \mathbf{T}$ or the type variable X representing the datatype being defined. The function $Con(\mathbf{dc})$ is defined as: $Con(\mathsf{C}(\diamondsuit_1, \ldots, \diamondsuit_a)) = \mathsf{C}$.

ρ	$::= \mathsf{int}$	(integers)
	$\mid\ \rho \to \rho$	(functions)
	$\mid\ \mu$	(dataypes)
μ	$::= \mathsf{data}\, X.\mathbf{dc}_1 [\![\cdots [\![\mathbf{dc}_m$	(datatype, $m \geq 0$, $Con(\mathbf{dc}_i) = Con(\mathbf{dc}_j)$ iff $i = j$, X bound in each $\mathbf{dc}_i$)
$\mathbf{dc}$	$::= \mathsf{C}(\diamondsuit_1, \ldots, \diamondsuit_a)$	(data-clause, $a \geq 0$, each $\diamondsuit_i$ is either a type or the variable X)

Fig. 1. PCFD types

Sometimes we will use, as meta-notation, parametrized definitions like $\mathsf{list}(X_1) = \mathsf{data}\, X.\mathsf{Nil} [\![\mathsf{Cons}(X_1, X)$, and $\mathsf{pair}(X_1, X_2) = \mathsf{data}\, X.\mathsf{Pair}(X_1, X_2)$. For every type $\rho, \rho_1, \rho_2 \in \mathbf{T}$, $\mathsf{list}(\rho) \in \mathbf{T}$ and $\mathsf{pair}(\rho_1, \rho_2) \in \mathbf{T}$ are the types "list of ρ elements" and "pair having a ρ_1 as first element and a ρ_2 as second element", respectively. We remark that in our language it is not possible to define polymorphic type constructors like $\mathsf{list}(X_1)$ above: type parameters are only a convenient notation, and we need a separate definition for each particular instance of the parameter. Note that the datatype with no constructors, $\mathsf{void} = (\mathsf{data}\, X.)$, is a legal datatype[7].

PCFD terms are defined from a set of typed *term constants* ($\mathcal{K}$) including the usual operations involving the datatype of booleans, $\mathsf{bool} = \mathsf{data}\, X.\mathsf{True} [\![\mathsf{False}$.

$$\begin{aligned}\mathcal{K} = \{\ & \mathsf{0}^{\mathsf{int}},\ \mathsf{1}^{\mathsf{int}},\ \ldots\ -^{\mathsf{pair(int,int)\to int}},\ +^{\mathsf{pair(int,int)\to int}},\ *^{\mathsf{pair(int,int)\to int}},\ \ldots \\ & =^{\mathsf{pair(int,int)\to bool}},\ <^{\mathsf{pair(int,int)\to bool}},\ \ldots \\ & \mathsf{not}^{\mathsf{bool\to bool}},\ \mathsf{and}^{\mathsf{pair(bool,bool)\to bool}},\ \mathsf{or}^{\mathsf{pair(bool,bool)\to bool}} \qquad \}\ ,\end{aligned}$$

(ranged over by k), and a set $\mathcal{V}$ of typed *term variables* (ranged over by $x^\rho, y^\sigma, \ldots$). The type of a constant k is denoted by $\mathbf{T}(\mathsf{k})$. PCFD terms, ranged over by M, N, $\ldots$, are defined as follows.

[7] Although it has no use in programming. In fact (as it will be clear at the end of the section) the "typical" term of type void is $\mathsf{fix}\, x^{\mathsf{void}}.x$, and every closed term of type void is always divergent.

Definition 4 (PCFD 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. 2.

Note that the tested term in a case-expression cannot be of void type (i.e. it must have at least a data-constructor).

$$(\mathrm{Var}) \vdash x^{\rho} : \rho \qquad (\mathrm{Con}) \vdash \mathsf{k}^{\rho} : \rho \qquad (\mathrm{Fix}) \frac{\vdash M : \rho}{\vdash \mathsf{fix}\, x^{\rho}.M : \rho}$$

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

$$(\mathrm{Data}_{1 \le i \le m}) \frac{\vdash P_1 : \sigma_1 \quad \cdots \quad \vdash P_{a_i} : \sigma_{a_i}}{\vdash \mathsf{C}_i^{\mu}(P_1, \ldots, P_{a_i}) : \mu}$$

where $\mu = \mathsf{data}\, X.\mathsf{C}_1(\diamondsuit_{1,1}, \ldots, \diamondsuit_{1,a_1}) \,[\!]\, \cdots \,[\!]\, \mathsf{C}_m(\diamondsuit_{m,1}, \ldots, \diamondsuit_{m,a_m})$

$$\forall j \in \{1, \ldots, a_i\}.\sigma_j = \begin{cases} \mu, & \text{if } \diamondsuit_{i,j} = X \\ \diamondsuit_{i,j}, & \text{if } \diamondsuit_{i,j} \in \mathbf{T} \end{cases}$$

$$(\mathrm{Case}) \frac{\vdash P : \mu \quad \vdash Q_1 : \tau \quad \cdots \quad \vdash Q_m : \tau}{\vdash \mathsf{case}\, P\, \mathsf{of}\, \{\mathsf{C}_1(x_{1,1}, \ldots, x_{1,a_1})\ \mathsf{to}\ Q_1 \,[\!]\, \cdots \,[\!]\, \mathsf{C}_m(x_{m,1}, \ldots, x_{m,a_m})\ \mathsf{to}\ Q_m\} : \tau}$$

where $m \ge 1$

$\mu = \mathsf{data}\, X.\mathsf{C}_1(\diamondsuit_{1,1}, \ldots, \diamondsuit_{1,a_1}) \,[\!]\, \cdots \,[\!]\, \mathsf{C}_m(\diamondsuit_{m,1}, \ldots, \diamondsuit_{m,a_m})$

$\forall i \in \{1, \ldots, m\}$, the program variables $x_{i,1}, \cdots, x_{i,a_i}$

- have the types specified by the data-clause $\mathsf{C}_i(\diamondsuit_{i,1}, \ldots, \diamondsuit_{i,i_m})$
- may occur free in Q_i
- are bound by the left-hand-side of the case-clause $\mathsf{C}_i(x_{i,1}, \ldots, x_{i,a_i})$ to Q_i

---

Fig. 2. Rules for PCFD term formation (system $\vdash_{\mathbf{T}}$)

1. According to Definition 4, in a PCFD term M the types of variables, constants, and data constructors (C^{μ}) 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 PCFD terms to be α-equivalence classes of syntax tree, i.e. we will identify terms modulo renaming of the bound[8] variables.
3. As usual a substitution is a finite function mapping term variables to terms, denoted by $[x_1 := N_1, \ldots, x_n := N_n]$ 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 supposed.

Let $\Lambda_{\mathbf{T}}$ be the set of PCFD terms, i.e. $\Lambda_{\mathbf{T}} = \{M \mid \vdash_{\mathbf{T}} M : \rho \text{ for some type } \rho\}$, and $\Lambda_{\mathbf{T}}^{\mathsf{c}}$ be the set of the *closed* terms, i.e. $\Lambda_{\mathbf{T}}^{\mathsf{c}} = \{M \mid M \in \Lambda_{\mathbf{T}} \text{ and } \mathrm{FV}(M) = \emptyset\}$. The process of evaluating a program is specified in a standard way by giving a *structural operational semantics* (see [17,11]) in the form of an inductively defined *evaluation relation*, $M \Downarrow K$, where M is a closed term and K is a closed term in *weak head normal form* (*w.h.n.f.*), i.e. an element of the set of *values*

$$\mathbf{V}_{\mathbf{T}} = \mathcal{K} \cup \{\lambda x^{\rho}.N \mid \lambda x^{\rho}.N \in \Lambda_{\mathbf{T}}^{\mathsf{c}}\} \cup \{\mathsf{C}^{\mu}(P_1, \ldots, P_a)^{\rho} \mid \mathsf{C}^{\mu}(P_1, \ldots, P_a)^{\rho} \in \Lambda_{\mathbf{T}}^{\mathsf{c}}\} \ .$$

We assume that any functional constant has a type of the shape either $\rho_1 \to \rho_2$ or $\mathsf{pair}(\rho_1, \rho_2) \to \rho_3$, for some $\rho_1, \rho_2, \rho_3 \in \{\mathsf{int}, \mathsf{bool}\}$. So the meaning of a functional constant k can be given by a set $\mathbf{mean}(\mathsf{k})$ of pairs, i.e. if $(K_1, K_2) \in \mathbf{mean}(\mathsf{k})$ then $\mathsf{k}K_1$ evaluates to K_2. For example $(\mathsf{True}, \mathsf{False}) \in \mathbf{mean}(\mathsf{not})$ and $(\mathsf{Pair}(1,3), 4) \in \mathbf{mean}(+)$.

Definition 5 (Evaluation relation). Let $M \in \Lambda_{\mathbf{T}}^{\mathsf{c}}$. We write $M \Downarrow K$, and say that M evaluates to K, if this statement is derivable by using the rules in Fig. 3.

$$(\mathrm{CAN})\ \frac{K \in \mathbf{V_T}}{K \Downarrow K} \quad (\mathrm{FIX})\ \frac{M[x := \mathsf{fix}\, x.M] \Downarrow K}{\mathsf{fix}\, x.M \Downarrow K} \quad (\mathrm{APP})\ \frac{M \Downarrow \lambda x.P \quad P[x := N] \Downarrow K}{MN \Downarrow K}$$

$$(\mathcal{K}\mathrm{APP}_1)\ \frac{M \Downarrow \mathsf{k} \quad N \Downarrow K_1}{MN \Downarrow K_2}\ (K_1, K_2) \in \mathbf{mean}(\mathsf{k})$$

$$(\mathcal{K}\mathrm{APP}_2)\ \frac{M \Downarrow \mathsf{k} \quad N \Downarrow \langle N_1, N_2\rangle \quad N_1 \Downarrow K_1 \quad N_2 \Downarrow K_2}{MN \Downarrow K_3}\ (\mathsf{Pair}(K_1, K_2), K_3) \in \mathbf{mean}(\mathsf{k})$$

$$(\mathrm{CASE}_{1 \le i \le m})\ \frac{P \Downarrow \mathsf{C}_i(P_1, \ldots, P_{a_i}) \quad Q_i[x_{i,1} := P_1, \ldots, x_{i,a_i} := P_{a_i}] \Downarrow K}{\mathsf{case}\, P\, \mathsf{of}\, \{\mathsf{C}_1(x_{1,1}, \ldots, x_{1,a_1})\ \mathsf{to}\ Q_1 \,[\!]\, \cdots \,[\!]\, \mathsf{C}_m(x_{m,1}, \ldots, x_{m,a_m})\ \mathsf{to}\ Q_m\} \Downarrow K}$$

Fig. 3. "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, $M \Downarrow K$. For every type $\rho \in \mathbf{T}$, let $\perp_\rho = \mathsf{fix}\, x^\rho.x$. It is easy to check that $\perp_\rho \Uparrow$, i.e. $\perp_\rho$ is the "typical" divergent computation of type ρ.

2.2 Ground contextual equivalence

Roughly speaking, two expressions M and N of a programming language are contextually equivalent if any occurrence of M in a complete program can be replaced by N, and vice versa, without changing the results of program execution. To formalize this for a particular programming language is necessary to specify which are the observable results of program execution.

Following [15] we introduce the *ground contextual equivalence* on PCFD terms, which is the congruence on terms induced by the contextual preorder that compares the termination behaviour of programs just at the ground type int. This amounts to assume that *complete PCFD programs* are closed terms of type int, and that the only observable behaviour of a complete program P is its divergence or converge to some integer number.

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

Definition 6 (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*.

2.3 A closed term model of PCFD

The *closed term model* $\mathcal{M}^{\mathrm{gnd}}$ of PCFD 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 $\mathbf{I}(\rho)$ denote the interpretation of type ρ in this model, and let $[M]$ denote the equivalence class of the closed term M. For every type ρ, $[\perp_\rho]$ (the equivalence class of the divergent elements, which represents the notion of endless computation), is the least element, w.r.t. $\preceq^{\mathrm{gnd}}_{\mathrm{obs}}$, of $\mathbf{I}(\rho)$. An *environment* is a mapping $e : \mathcal{V} \to \bigcup_{\rho \in \mathbf{T}} \mathbf{I}(\rho)$ which respects types, i.e. a mapping such that, for all x^ρ, $e(x^\rho) \in \mathbf{I}(\rho)$. The interpretation of a term M in an environment e is defined in a standard way by:

$$[\![M]\!]_e = [M[x_1 := N_1, \ldots, x_n := N_n]],$$

where $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$ and $[N_i] = e(x_i)$ $(1 \le i \le n)$.

[8] Variables can be bound either by λ-abstraction or by a clause in a case-expression.

3 P.e.r.s over $\mathcal{M}^{\mathrm{gnd}}$ as program properties

The language of program properties $\mathbf{L}$ (introduced in Section 4 as a language of non-standard types over $\mathbf{T}$) which is at the basis of the program analysis and transformation techniques proposed in this paper, will be interpreted as a subset of the *partial equivalence relations*[9] over the interpretation, $\mathbf{I}(\rho)$, of the types $\rho \in \mathbf{T}$ in the closed term model $\mathcal{M}^{\mathrm{gnd}}$. In the following let "p.e.r. over a type ρ" mean "p.e.r. over $\mathbf{I}(\rho)$". This section is devoted to explain how some properties of PCFD programs P of type ρ can be represented as p.e.r.s over the type $\rho \in \mathbf{T}$. For every type $\rho \in \mathbf{T}$,

- the diagonal p.e.r. over $\mathbf{I}(\rho)$, $\Delta^{\mathsf{int}} = \{([M],[M]) \mid [M] \in \mathbf{I}(\mathsf{int})\}$, which equates each element of $\mathbf{I}(\rho)$ with only itself, can be seen as the property satisfied by any term M of type ρ that[10] *can be used*. Note that any closed term M of type ρ satisfies this property.
- The trivial p.e.r. over $\mathbf{I}(\rho)$, $\Omega^{\rho} = \{([M],[N]) \mid [M],[N] \in \mathbf{I}(\rho)\}$, which equates all the elements of $\mathbf{I}(\rho)$, can be seen as the "true" property (satisfied by every term of type ρ) giving *no information* about the use of the term.
- Given a p.e.r. $\mathcal{R}_1$ over ρ_1 and a p.e.r. $\mathcal{R}_2$ over ρ_2, let $\mathcal{R}_1 \twoheadrightarrow \mathcal{R}_2$ be the p.e.r. over $\rho_1 \to \rho_2$ defined as:

$$\mathcal{R}_1 \twoheadrightarrow \mathcal{R}_2 \quad = \quad \{([F],[G]) \mid \forall([M],[N]) \in \mathcal{R}_1.([FM],[GN]) \in \mathcal{R}_2\}.$$

  The intuition behind this definition is that $\mathcal{R}_1 \twoheadrightarrow \mathcal{R}_2$ is the property of the programs F such that, for every program M having the property $\mathcal{R}_1$, the program FM has the property $\mathcal{R}_2$. For instance $\Omega^{\mathsf{int}} \to \Delta^{\mathsf{int}}$ is the property (satisfied by all the closed terms of type $\mathsf{int} \to \mathsf{int}$ which represent, necessary constant, functions which do not use their argument) which says that the the application of the function can be used without using the argument.

The following definition formally explains what is meant by "a program P of type ρ satisfies the property (p.e.r.) $\mathcal{R}$ over ρ".

Definition 7 (P satisfies $\mathcal{R}$). Let $\mathcal{R}, \mathcal{R}_1, \ldots, \mathcal{R}_n$ ($n \geq 0$) be p.e.r. over types $\rho, \rho_1, \ldots, \rho_n$, respectively. We say that a program P of type ρ with free variables $x_1^{\rho_1} \ldots, x_n^{\rho_n}$ satisfies the property $\mathcal{R}$ under the assumptions $\mathcal{R}_i$ for $x_i^{\rho_i}$ ($1 \leq i \leq n$) if, for all the environments e and e' such that $(e(x_i^{\rho_i}), e'(x_i^{\rho_i})) \in \mathcal{R}_i$, we have that $([\![M]\!]_e, [\![M]\!]_{e'}) \in \mathcal{R}$.

The set-theoretic inclusion between p.e.r.s over a type ρ represents a logical implication between properties, i.e., if $\mathcal{R}_1 \subseteq \mathcal{R}_2$ and a program P has the property $\mathcal{R}_1$, then P has also the property $\mathcal{R}_2$. For instance, for every p.e.r. $\mathcal{R}$ over ρ, $\mathcal{R} \subseteq \Omega^{\rho}$, according to the fact that Ω^{ρ} represents the "true" property. We also have that $\Omega^{\rho_1} \twoheadrightarrow \Delta^{\rho_2} \subseteq \Delta^{\rho_1} \twoheadrightarrow \Delta^{\rho_2}$, according to fact that if the application of a function can be used without using the argument, then it can still be used if the value of the argument can be used.

4 Evaluation types

In this section we introduce a language of non-standard types over $\mathbf{T}$, the language of the *evaluation types* (*e-types* for short), which is at the basis of the program analyses and transformations technique proposed in this paper.

[9] A *partial equivalence relation* (*p.e.r.* for short) over a set A is a symmetric and transitive binary relation over A.

[10] Under some assumption on the free variables of the term. In the following we will omit to mention the assumptions on the free variables till Definition 7 which formally explains what is meant by "the program P satisfies the p.e.r. $\mathcal{R}$".

4.1 E-types syntax and semantics

Let ϕ range over e-types and ϕ^ρ range over e-types with *underlying type* ρ (i.e. expressing properties of terms of type ρ). In the following we will often omit the superscript ρ when it is either not relevant or clear from the context.

There are two e-types denoting properties for terms of type int:

- δ^{int}, which is the property of the terms of type int such that their value *can be used*, and
- ω^{int}, which is the "true" property, satisfied by every term of type int.

Others e-types are built following the standard type construction.

- Given two e-types ϕ_1 and ϕ_2 the e-type $\phi_1 \to \phi_2$ says that the application of the function to every argument of e-type ϕ_1 has e-type ϕ_2.
- Given a datatype $\mu = \mathsf{data}\, X.\mathsf{C}_1(\diamondsuit_{1,1}, \ldots, \diamondsuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m(\diamondsuit_{m,1}, \ldots, \diamondsuit_{m,a_m})$ the e-type ω^μ is the "true" property, satisfied by every term of type μ. There also e-types that provide information about which constructors of the datatype may be used: two *constructor annotations*, ∂ and ℓ (ranged over by $\mathtt{u}, \mathtt{v}$), are introduced to represent the fact that a datatype constructor *is not used* (i.e. it is ∂*ead*) or *may be used* (i.e. it is ℓ*ive*), respectively. In particular the syntax of the e-types ϕ^μ different from ω^μ is as follows:

$$\phi^\mu = \mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1}, \ldots, \spadesuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1}, \ldots, \spadesuit_{m,a_m}),$$

where for all $i \in \{1, \ldots, m\}$ and for all $j \in \{1, \ldots, a_i\}$

$$\spadesuit_{i,j} = \begin{cases} \text{either } X \text{ or } \omega^\mu, & \text{if } \mathtt{u}_i = \ell \text{ and } \diamondsuit_{i,j} = X \\ \text{an e-type } \phi^{\diamondsuit_{i,j}}, & \text{if } \mathtt{u}_i = \ell \text{ and } \diamondsuit_{i,j} \in \mathbf{T} \\ \diamondsuit_{i,j}, & \text{if } \mathtt{u}_i = \partial\ . \end{cases}$$

If a datatype Q of type μ has the e-type ϕ^μ above, then the constructors having annotation ∂ *are not used* while those having annotation ℓ *may be used* (with arguments having the specified e-types). For instance there are 11 e-types (listed in Fig. 4) for the type $\mathsf{list}(\mathsf{int})$.

The 11 e-types for $\mathsf{list}(\mathsf{int})$ are:

$$\begin{array}{ll} \psi_0 = \omega^{\mathsf{list}(\mathsf{int})} & \psi_1 = \mathsf{data}\, X.\mathsf{Nil}^\partial \parallel \mathsf{Cons}^\partial(\mathsf{int}, X) \\ \psi_2 = \mathsf{data}\, X.\mathsf{Nil}^\partial \parallel \mathsf{Cons}^\ell(\delta^{\mathsf{int}}, X) & \psi_3 = \mathsf{data}\, X.\mathsf{Nil}^\partial \parallel \mathsf{Cons}^\ell(\omega^{\mathsf{int}}, X) \\ \psi_4 = \mathsf{data}\, X.\mathsf{Nil}^\partial \parallel \mathsf{Cons}^\ell(\delta^{\mathsf{int}}, \omega^{\mathsf{list}(\mathsf{int})}) & \psi_5 = \mathsf{data}\, X.\mathsf{Nil}^\partial \parallel \mathsf{Cons}^\ell(\omega^{\mathsf{int}}, \omega^{\mathsf{list}(\mathsf{int})}) \end{array}$$

plus the 5 e-types (say $\psi'_1, \ldots, \psi'_5$) obtained from $\psi_1, \ldots, \psi_5$ by replacing Nil^∂ with Nil^ℓ.

Fig. 4. The e-types for terms of type $\mathsf{list}(\mathsf{int})$

The "typical" terms having e-types ψ_1, ψ'_1, and ψ_5 are $\mathsf{fix}\, x^{\mathsf{list}(\mathsf{int})}.x$, Nil, and $\mathsf{Cons}(N, M)$ (for any $N : \mathsf{int}$ and $M : \mathsf{list}(\mathsf{int})$), respectively. $\mathsf{Cons}(N, M)$ is also the "typical" term of e-type ψ_4. Moreover $\mathsf{fix}\, x^{\mathsf{list}(\mathsf{int})}.x$ has every e-type of $\mathsf{list}(\mathsf{int})$, while Nil has none of the e-types $\psi_1, \ldots, \psi_5$, and $\mathsf{Cons}(N, M)$ has neither e-type ψ_1 nor ψ'_1. Both the e-types ψ_2 and ψ_3 specify that the constructor Nil is not used while the second component of the constructor Cons (which may be used) has (recursively) the same e-type, so the "typical" terms having such e-types are of the form $\mathsf{Cons}(N_1, \mathsf{Cons}(N_2, \cdots \mathsf{Cons}(N_p, \mathsf{fix}\, x^{\mathsf{list}(\mathsf{int})}.x) \cdots))$, for some $N_1, \ldots, N_p$ ($p \geq 0$).

We remark that the e-type syntax (see Definition 8 below) has been designed in such a way that (syntactically) different e-types denote (w.r.t. the semantics in Definition 9) different p.e.r.s. This observation justifies some choices that might appear quite arbitrary, e.g. not having e-types of the form δ^ρ where $\rho \neq \mathsf{int}$, and of the form $\phi \to \omega^\sigma$ (for any e-type ϕ and type σ).

Definition 8 (Evaluation types). The language of the e-types, $\mathbf{L}$, is defined by: $\mathbf{L} = \cup_{\rho \in \mathbf{T}} \mathbf{L}(\rho)$, where the sets $\mathbf{L}(\rho)$ are defined by the rules in Fig. 5.

$$(\delta)\ \delta^{\mathsf{int}} \in \mathbf{L}(\mathsf{int}) \qquad (\omega)\ \omega^{\rho} \in \mathbf{L}(\rho) \qquad (\to)\ \frac{\phi \in \mathbf{L}(\rho) \quad \psi \in \mathbf{L}(\sigma)}{\phi \to \psi \in \mathbf{L}(\rho \to \sigma)}\ \psi \neq \omega^{\sigma}$$

$$(\mathsf{data})\ \frac{\forall i \in \{1,\dots,m\}.\forall j \in \{1,\dots,a_i\}.\spadesuit_{i,j} \in \begin{cases} \{X, \omega^{\mu}\}, & \text{if } \mathtt{u}_i = \ell \text{ and } \diamondsuit_{i,j} = X \\ \mathbf{L}(\diamondsuit_{i,j}), & \text{if } \mathtt{u}_i = \ell \text{ and } \diamondsuit_{i,j} \in \mathbf{T} \\ \{\diamondsuit_{i,j}\}, & \text{if } \mathtt{u}_i = \partial \end{cases}}{\mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1},\dots,\spadesuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1},\dots,\spadesuit_{m,a_m}), \in \mathbf{L}(\mu)}$$
$$\text{where } \mu = \mathsf{data}\, X.\mathsf{C}_1(\diamondsuit_{1,1},\dots,\diamondsuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m(\diamondsuit_{m,1},\dots,\diamondsuit_{m,a_m})$$

Fig. 5. Evaluation types

Each e-type ϕ^{ρ} is interpreted as a p.e.r., $[\![\rho]\!]$, over $\mathbf{I}(\rho)$ (see Section 3).

Definition 9 (Semantics of e-types). The semantic function $[\![\cdot]\!]$ which maps e-types ϕ^{ρ} to p.e.r.s over $\mathbf{I}(\rho)$ is defined by the clauses in Fig. 6.

$$\begin{array}{lll} [\![\delta^{\mathsf{int}}]\!] & = \Delta^{\mathsf{int}}, & \\ [\![\omega^{\rho}]\!] & = \Omega^{\rho} & \\ [\![\phi \to \psi]\!] & = [\![\phi]\!] \twoheadrightarrow [\![\psi]\!], & \text{where } \psi \text{ is not an } \omega\text{-e-type} \\ [\![\phi^{\mu}]\!] & = \bigcup_{p \geq 0} [\![\phi^{\mu}]\!]^{p}, & \text{where} \end{array}$$

$$\phi^{\mu} = \mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1},\dots,\spadesuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1},\dots,\spadesuit_{m,a_m})$$

$$\begin{array}{rl} [\![\phi^{\mu}]\!]^{0} & = \{([\mathsf{fix}\, x^{\mu}.x], [\mathsf{fix}\, x^{\mu}.x])\} \\ [\![\phi^{\mu}]\!]^{p+1} & = [\![\phi^{\mu}]\!]^{p} \cup \\ & \quad \bigcup_{i \in \{1 \leq i \leq m \text{ and } \mathtt{u}_i = \ell\}} \{([\mathsf{C}_i(P_{i,1},\dots,P_{i,a_i})], [\mathsf{C}_i(Q_{i,1},\dots,Q_{i,a_i})]) \mid \\ & \qquad \forall j \in \{1,\dots,a_i\}.([P_{i,j}],[Q_{i,j}]) \in \begin{cases} [\![\phi^{\mu}]\!]^{p}, & \text{if } \spadesuit_{i,j} = X \\ [\![\spadesuit_{i,j}]\!], & \text{if } \spadesuit_{i,j} \in \mathbf{L} \end{cases} \} \end{array}$$

Fig. 6. Evaluation types semantics

The well foundness of our definition of $[\![\phi^{\mu}]\!]$ is guaranteed by the following property.

Theorem 10. *For all the e-types ϕ^{μ} of the form $\phi^{\mu} = \mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1},\dots,\spadesuit_{1,a_1}) \parallel \cdots \parallel \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1},\dots,\spadesuit_{m,a_m})$, we have that*

$$\begin{array}{rl} [\![\phi^{\mu}]\!] = & \{([\mathsf{fix}\, x^{\mu}.x], [\mathsf{fix}\, x^{\mu}.x])\} \cup \\ & \bigcup_{i \in \{1 \leq i \leq m \text{ and } \mathtt{u}_i = \ell\}} \{([\mathsf{C}_i(P_{i,1},\dots,P_{i,a_i})], [\mathsf{C}_i(Q_{i,1},\dots,Q_{,a_i})]) \mid \\ & \quad \forall j \in \{1,\dots,a_i\}.([P_{i,j}],[Q_{i,j}]) \in \begin{cases} [\![\phi^{\mu}]\!], & \text{if } \spadesuit_{i,j} = X \\ [\![\spadesuit_{i,j}]\!], & \text{if } \spadesuit_{i,j} \in \mathbf{L} \end{cases} \}. \end{array}$$

According to the e-type semantics (Definition 9) an e-type ω^{ρ} denotes the trivial p.e.r. Ω^{ρ}. Others "special" e-types are the δ-e-types, which represents the property of being a term that can be used, i.e. a δ-e-type ϕ^{ρ} denotes the diagonal p.e.r. Δ^{ρ}.

Definition 11 (ω-e-types and δ-e-types).
1. Let the ω-e-types be the e-types in the set $\mathbf{L}_{\omega} = \{\omega^{\rho} \mid \rho \in \mathbf{T}\}$.
2. The set of the δ-e-types ($\mathbf{L}_{\delta}$) is the subset of the e-types which do not contain subexpressions of the form ω^{ρ} (for some ρ) and do not contain the annotation ∂. For every type $\rho \in \mathbf{T}$, let $\delta(\rho)$ be the corresponding unique δ-e-type[11].

[11] It is immediate to see that for every type $\rho \in \mathbf{T}$ there is exactly one δ-e-type ϕ^{ρ}. For instance $\delta(\mathsf{list}(\mathsf{int}))$ is the e-type ψ'_2 in Fig. 4.

Two other subclasses of e-types which are useful for program simplification are the ∂-e-datatypes that characterize datatypes in which no constructor is used (and which are therefore, according to Definition 9[12], ground observational equivalent to a divergent computation), and the $\ell 1$-e-datatypes that characterize datatypes in which only one constructor is used and its arguments are not used (i.e. they characterize the minimum-information-code[13]).

Definition 12 (∂-e-datatypes and $\ell 1$-e-datatypes). 1. The set of the ∂-e-datatypes, $\mathbf{L}_\partial$, is the set of the e-types of the form:

$$\mathsf{data}\, X.\mathsf{C}_1{}^{\partial}(\spadesuit_{1,1},\ldots,\spadesuit_{1,a_1}) \,[\!]\cdots[\!]\, \mathsf{C}_m{}^{\partial}(\spadesuit_{m,1},\ldots,\spadesuit_{m,a_m}),$$

where $m \geq 0$. For every datatype μ, let $\partial(\mu)$ be the corresponding unique ∂-e-datatype[14].

2. The set of the $\ell 1$-e-datatypes, $\mathbf{L}_{\ell 1}$, is the set of the e-types of the form:

$$\mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1},\ldots,\spadesuit_{1,a_1}) \,[\!]\cdots[\!]\, \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1},\ldots,\spadesuit_{m,a_m}),$$

where $m \geq 1$ and exists $i \in \{1,\ldots,m\}$ such that $\mathtt{u}_i = \ell$, $\spadesuit_{i,1},\ldots,\spadesuit_{i,a_i} \in \mathbf{L}_\omega$, and $j \neq i$ implies $\mathtt{u}_j = \partial$. For every datatype μ with $m \geq 1$ data-constructors, let $\ell 1_{\mathsf{C}}(\mu)$ be the corresponding unique $\ell 1$-e-datatype in which the data-constructor C is the live one[15].

4.2 A complete entailment relation for e-types

In this section we introduce an entailment relation between e-types, $\leq$. This relation models the set-theoretic inclusion between the interpretation of e-types, and so it represents the logical implication between properties.

Definition 13 (Entailment relation $\leq$). Let $\phi,\psi \in \mathbf{L}$. We write $\phi \leq \psi$ to mean that $\phi \leq \psi$ is derivable by the rules in Fig. 7. By $\cong$ we denote the equivalence relation induced by $\leq$.

$$(\mathrm{Ref})\ \phi \leq \phi \qquad (\omega)\ \phi^\rho \leq \omega^\rho \qquad (\rightarrow)\ \frac{\phi_1 \leq \phi_2 \quad \psi_1 \leq \psi_2}{\phi_2 \rightarrow \psi_1 \leq \phi_1 \rightarrow \psi_2}\ \psi_2 \notin \mathbf{L}_\omega$$

$$(\mathsf{data})\ \frac{\forall i \in \{h \mid 1 \leq h \leq m \text{ and } \mathtt{u}_h = \ell\}.\forall j \in \{1,\ldots,a_i\}.\spadesuit_{i,j} \preceq \clubsuit_{i,j}}{\phi^\mu \leq \psi^\mu}$$

where $\phi^\mu = \mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{u}_1}(\spadesuit_{1,1},\ldots,\spadesuit_{1,a_1}) \,[\!]\cdots[\!]\, \mathsf{C}_m{}^{\mathtt{u}_m}(\spadesuit_{m,1},\ldots,\spadesuit_{m,a_m})$
$\psi^\mu = \mathsf{data}\, X.\mathsf{C}_1{}^{\mathtt{v}_1}(\clubsuit_{1,1},\ldots,\clubsuit_{1,a_1}) \,[\!]\cdots[\!]\, \mathsf{C}_m{}^{\mathtt{v}_m}(\clubsuit_{m,1},\ldots,\clubsuit_{m,a_m})$
$\forall i \in \{1,\ldots,m\}.(\mathtt{u}_i = \partial \text{ or } \mathtt{v}_i = \ell)$

$$\spadesuit \preceq \clubsuit \text{ is short for: } \begin{cases} \clubsuit \in \{X, \omega^\mu\}, & \text{if } \spadesuit = X \\ \clubsuit \in \mathbf{L} \text{ and } \spadesuit \leq \clubsuit, & \text{if } \spadesuit \in \mathbf{L} \end{cases}$$

Fig. 7. Entailment rules for e-types (system $\leq$)

Note that $\leq$ is reflexive and transitive. Moreover, for any type $\rho \in \mathbf{T}$, $(\mathbf{L}(\rho),\leq)$ is a complete lattice with top ω^ρ and bottom b^ρ (inductively defined by: $b^{\mathsf{int}} = \delta^{\mathsf{int}}$, $b^{\sigma\rightarrow\tau} = \omega^\sigma \rightarrow b^\tau$, and $b^\mu = \partial(\mu)$). For instance the lattice $(\mathbf{L}(\mathsf{list}(\mathsf{int})),\leq)$ (whose elements are the 11 e-types listed in Fig. 4) is showed in Fig. 8.

The $\leq$ entailment relation between e-types is sound and complete w.r.t. the p.e.r. interpretation of Definition 9 (the completeness result can be proved by using the same technique used in [7] Chapter 7, where an analogous result for the subsystem with pairs but without others datatypes is presented).

[12] Since an ∂-e-datatype ϕ^μ denotes the singleton p.e.r. $[\![\phi^\mu]\!] = \{([\bot_\mu],[\bot_\mu])\}$.

[13] Since an $\ell 1$-e-datatype ϕ^μ denotes the p.e.r. (with exactly two classes): $[\![\phi^\mu]\!] = \{([\bot_\mu],[\bot_\mu])\} \cup \{([\mathsf{C}(M_1,\ldots,M_a)],[\mathsf{C}(N_1,\ldots,N_a)]) \mid \mathsf{C}(M_1,\ldots,M_a),\mathsf{C}(N_1,\ldots,N_a) \in \mathbf{I}(\mu)\}$, where C is the unique ℓive constructor of μ.

[14] For instance $\partial(\mathsf{list}(\mathsf{int}))$ is the e-type ψ_1 in Fig. 4.

[15] For instance $\ell 1_{\mathsf{Nil}}(\mathsf{list}(\mathsf{int}))$ and $\ell 1_{\mathsf{Cons}}(\mathsf{list}(\mathsf{int}))$ are the e-types ψ_1' and ψ_5 in Fig. 4, respectively.

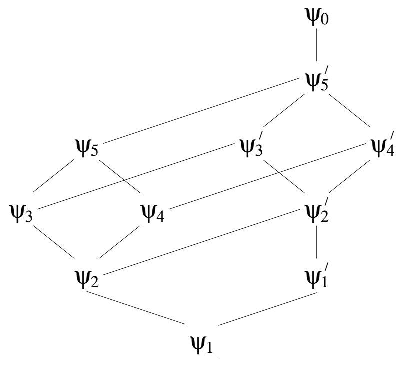

Fig. 8. The lattice $(\mathbf{L}(\mathsf{list}(\mathsf{int})), \leq)$

Theorem 14 (Soundness and completeness of $\leq$). *$\phi \leq \psi$ if and only if $[\![\phi]\!] \subseteq [\![\psi]\!]$.*

5 Detecting and removing redundant-code

In this section we first introduce an e-type assignment system for detecting dead-code and minimum-information-code in PCFD programs, then we present simplification mappings for removing from a program the redundant-code that can detected with the e-type assignment system. We also explain how to compute the best simplification of a program (w.r.t. the informations provided by the system $\vdash_{\mathbf{L}}$) and discuss the application of the technique to non lazy-call-by-name functional programming languages.

The basic idea for using e-types to detect the redundant-code in a given PCFD term M of type ρ is the following: if we can prove that (under suitable assumptions on its free variables) the term has a δ-e-type while some of its subexpressions have ω-e-types, then (according to the e-type semantics):

- the value of M can be computed without using these subexpressions, so they are indeed dead-code that can be removed without changing the semantics of the program w.r.t. the model $\mathcal{M}^{\mathrm{gnd}}$,
- all the subexpression having an ∂-e-datatype ϕ^{μ} are ground observational equivalent to the divergent computation $\perp_{\mu}$[16], and
- all the subexpression having an $\ell 1$-e-datatype are minimum-information-code.

5.1 An e-type assignment system for detecting redundant-code

If x^{ρ} is a term variable of type ρ an assumption for x^{ρ} is an expression of the shape $x^{\rho} : \phi^{\rho}$, or $x : \phi^{\rho}$ for short. A basis is a set Σ of e-types assumptions for term variables.

E-types are assigned to PCFD terms by a set of inference rules for judgments of the form $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ where M^{ϕ} is a *decorated* term, i.e., it has written in it (some of) the e-types assigned to its subterms. Such a decorated term can then be processed by a transformation procedure (like those described in Sections 5.2 and 5.3) that simplifies programs according to the information supplied by the e-types. For any e-type $\phi \in \mathbf{L}(\rho)$ let $\epsilon(\phi)$ denote the underlying type, ρ, of ϕ, and for any decorated term M^{ϕ}, define $\epsilon(M^{\phi})$ be the term obtained from M^{ϕ} by erasing all the e-type decorations. For each constant k it is specified a finite non-empty subset of e-types, $\mathbf{L}(\mathsf{k})$.

[16] We do not expect to find always divergent subexpressions when simplifying programs (indeed, when such an expression is discovered, a warning should be reported to user). By the way we have defined a simplification procedure for them.

- For all integers n, $\mathbf{L}(\mathsf{n}) = \{\delta^{\mathsf{int}}\}$.
- For any binary operator $\odot^{\mathsf{pair(int,int)} \to \mathsf{int}}$, $\mathbf{L}(\odot) = \{\delta(\mathsf{pair(int,int)} \to \delta^{\mathsf{int}}\}$.
- The constants involving the boolean datatype have more than one (non ω-) e-type, see Fig. 9[17]. For instance the 4 e-types associated to not say that: if the result can be used then also the argument can be used, if in the argument the constructor True is ℓive and the constructor False is ∂ead then in the result False is ℓive and True is ∂ead (and vice versa), and if the argument diverges then also the result diverges.

Let $\mathtt{t}^{\mathsf{bool}} = \mathsf{data}X.\mathsf{True}^{\ell} [\![\mathsf{False}^{\partial}$, $\mathtt{f}^{\mathsf{bool}} = \mathsf{data}X.\mathsf{True}^{\partial} [\![\mathsf{False}^{\ell}$ and, for any $\phi, \psi \in \mathbf{L}$, $\mathsf{pair}(\phi, \psi) = \mathsf{data}\ X.\mathsf{Pair}^{\ell}(\phi, \psi)$. Then

- for any binary operator $\odot^{\mathsf{pair(int,int)} \to \mathsf{bool}}$,
  $\mathbf{L}(\odot) = \{\delta(\mathsf{pair(int,int)}) \to \delta(\mathsf{bool}),\ \partial(\mathsf{pair(int,int)}) \to \partial(\mathsf{bool})\}$.
- $\mathbf{L}(\mathsf{not}) = \{\delta(\mathsf{bool}) \to \delta(\mathsf{bool}),\ \mathtt{t}^{\mathsf{bool}} \to \mathtt{f}^{\mathsf{bool}},\ \mathtt{f}^{\mathsf{bool}} \to \mathtt{t}^{\mathsf{bool}},\ \partial(\mathsf{bool}) \to \partial(\mathsf{bool})\}$
- $\mathbf{L}(\mathsf{and}) = \{\mathsf{pair}(\mathtt{t}^{\mathsf{bool}}, \mathtt{t}^{\mathsf{bool}}) \to \mathtt{t}^{\mathsf{bool}},$
  $\mathsf{pair}(\phi_1, \mathtt{f}^{\mathsf{bool}}) \to \mathtt{f}^{\mathsf{bool}},\ \mathsf{pair}(\mathtt{f}^{\mathsf{bool}}, \phi_2) \to \mathtt{f}^{\mathsf{bool}},$
  $\partial(\mathsf{pair(bool,bool)}) \to \partial(\mathsf{bool}),$
  $\mathsf{pair}(\partial(\mathsf{bool}), \psi) \to \partial(\mathsf{bool}),\ \mathsf{pair}(\psi, \partial(\mathsf{bool})) \to \partial(\mathsf{bool}),$
  $\mathsf{pair}(\phi_1, \phi_2) \to \delta(\mathsf{bool})\ \mid \phi_1, \phi_2 \in \{\mathtt{t}^{\mathsf{bool}}, \mathtt{f}^{\mathsf{bool}}\}$ and $\psi \in \mathbf{L}(\mathsf{bool})\}$
- $\mathbf{L}(\mathsf{or}) = \{\mathsf{pair}(\mathtt{f}^{\mathsf{bool}}, \mathtt{f}^{\mathsf{bool}}) \to \mathtt{f}^{\mathsf{bool}},$
  $\mathsf{pair}(\phi_1, \mathtt{t}^{\mathsf{bool}}) \to \mathtt{t}^{\mathsf{bool}},\ \mathsf{pair}(\mathtt{t}^{\mathsf{bool}}, \phi_2) \to \mathtt{t}^{\mathsf{bool}},$
  $\partial(\mathsf{pair(bool,bool)}) \to \partial(\mathsf{bool}),$
  $\mathsf{pair}(\partial(\mathsf{bool}), \psi) \to \partial(\mathsf{bool}),\ \mathsf{pair}(\psi, \partial(\mathsf{bool})) \to \partial(\mathsf{bool}),$
  $\mathsf{pair}(\phi_1, \phi_2) \to \delta(\mathsf{bool})\ \mid \phi_1, \phi_2 \in \{\mathtt{t}^{\mathsf{bool}}, \mathtt{f}^{\mathsf{bool}}\}$ and $\psi \in \mathbf{L}(\mathsf{bool})\}$

Fig. 9. E-types of the PCFD constants

Definition 15 (E-type assignment system $\vdash_{\mathbf{L}}$). A $\vdash_{\mathbf{L}}$-typing statement is an expression $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ where Σ is a basis containing an assumption for each free variable of M, and M^{ϕ} is a decorated version of M. $\Sigma, x : \psi$ denotes the basis $\Sigma \cup \{x : \psi\}$ where it is assumed that x does not appear in Σ. We write $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ to mean that $\Sigma \vdash M^{\phi}$ can be derived by the rules in Fig. 10[18].

1. For any finite set Γ of term variables let $\delta(\Gamma)$ and $\omega(\Gamma)$ denote the bases $\{x^{\sigma} : \delta(\sigma) \mid x^{\sigma} \in \Gamma\}$ and $\{x^{\sigma} : \omega^{\sigma} \mid x^{\sigma} \in \Gamma\}$, respectively. For any basis Σ let $\epsilon(\Sigma)$ denote the set of variables in Σ, i.e. $\epsilon(\Sigma) = \{x^{\rho} \mid x^{\rho} : \phi^{\rho} \in \Sigma,$ for some $\phi^{\rho} \in \mathbf{L}\}$.
2. Let $\Lambda_{\mathbf{T}}^{\vdash_{\mathbf{L}}}$ be the set of $\vdash_{\mathbf{L}}$-decorated PCFD terms, i.e.,
$$\Lambda_{\mathbf{T}}^{\vdash_{\mathbf{L}}} = \{M^{\phi} \mid \Sigma \vdash_{\mathbf{L}} M^{\phi} \text{ for some e-type } \phi \text{ and basis } \Sigma\}.$$
   Note that a same term can have different decorations. For all term $M \in \Lambda_{\mathbf{T}}$, define $\delta(M)$ be the decorated term obtained from M by decorating each subterm with the corresponding δ-e-type.

The proof of the following proposition is immediate.

Proposition 16. *1. $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ implies $\vdash_{\mathbf{T}} \epsilon(M^{\phi}) : \epsilon(\phi)$ and $\epsilon(\Sigma) \supseteq \mathrm{FV}(M)$.*

---

[17] The e-type inference algorithm (see Section 5.4) represents these sets of e-types in a compact way, by using an *e-type scheme*, that is a *pattern* (i.e. a sort of e-type containing variables ranging over $\{\delta, \omega, \ell, \partial\}$) with a set of *constraints* involving the variables in the pattern.

[18] The system in Fig. 10 could be expressed in slightly different way (indeed equivalent to our purposes, since it characterizes the same class of redundant code) by removing the use of entailment in rule (Var) and by adding an entailment rule saying that $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ and $\phi \leq \psi$ imply $\Sigma \vdash_{\mathbf{L}} M^{\psi}$.

$$(\omega)\ \frac{\Sigma \supseteq \mathrm{FV}(M) \quad \vdash_{\mathbf{T}} M : \rho}{\Sigma \vdash M^{\omega^\rho}} \qquad (\mathrm{Fix})\ \frac{\Sigma, x : \phi \vdash M^\psi}{\Sigma \vdash (\mathsf{fix}\, x^\phi.M)^\psi}\ \psi \notin \mathbf{L}_\omega \text{ and } \phi \in \{\psi, \omega^{\epsilon(\psi)}\}$$

$$(\mathrm{Var})\ \frac{\phi \leq \psi}{\Sigma, x : \phi \vdash x^\psi}\ \psi \notin \mathbf{L}_\omega \qquad (\mathrm{Con})\ \Sigma \vdash \mathsf{k}^\phi\ ,\ \ \phi \in \mathbf{L}(\mathsf{k})$$

$$(\to \mathrm{I})\ \frac{\Sigma, x : \phi \vdash M^\psi}{\Sigma \vdash (\lambda x.M)^{\phi \to \psi}}\ \psi \notin \mathbf{L}_\omega \qquad (\to \mathrm{E})\ \frac{\Sigma \vdash M^{\phi \to \psi} \qquad \Sigma \vdash N^\phi}{\Sigma \vdash (MN^\phi)^\psi}\ \psi \notin \mathbf{L}_\omega$$

$$(\mathrm{Data}_{1 \leq i \leq m})\ \frac{\Sigma \vdash {P_1}^{\psi_1} \quad \cdots \quad \Sigma \vdash {P_{a_i}}^{\psi_{a_i}}}{\Sigma \vdash (\mathsf{C}_i(P_1, \ldots, P_{a_i}))^{\phi^\mu}}$$

$$\begin{array}{ll} \text{where} & \phi^\mu = \mathsf{data}\, X.{\mathsf{C}_1}^{\mathtt{u}_1}(\spadesuit_{1,1}, \ldots, \spadesuit_{1,a_1}) \mathrel{[\!]} \cdots \mathrel{[\!]} {\mathsf{C}_m}^{\mathtt{u}_m}(\spadesuit_{m,1}, \ldots, \spadesuit_{m,a_m}) \\ & \mathtt{u}_i = \ell \\ & \forall j \in \{1, \ldots, a_i\},\ \psi_j = \begin{cases} \phi^\mu, & \text{if } \spadesuit_{i,j} = X \\ \spadesuit_{i,j}, & \text{if } \spadesuit_{i,j} \in \mathbf{L} \end{cases} \end{array}$$

$$(\mathrm{Case})\ \frac{\begin{array}{l} \Sigma \vdash P^{\phi^\mu} \qquad\qquad\qquad\qquad \forall i \in \{1, \ldots, m\}.\ \vdash_{\mathbf{T}} Q_i : \tau \\ \forall i \in \{h \mid 1 \leq h \leq m \text{ and } \mathtt{u}_i = \ell\}.\Sigma \cup \{x_{i,1} : \psi_{i,1}, \ldots, x_{i,a_i} : \psi_{i,a_i}\} \vdash {Q_i}^{\chi^\tau} \end{array}}{\Sigma \vdash (\mathsf{case}\, P^{\phi^\mu}\, \mathsf{of}\, \{\mathbf{cc}_1 \mathrel{[\!]} \cdots \mathrel{[\!]} \mathbf{cc}_m\})^{\chi^\tau}}$$

$$\begin{array}{ll} \text{where} & \chi^\tau \notin \mathbf{L}_\omega \\ & m \geq 1 \\ & \phi^\mu = \mathsf{data}\, X.{\mathsf{C}_1}^{\mathtt{u}_1}(\spadesuit_{1,1}, \ldots, \spadesuit_{1,a_1}) \mathrel{[\!]} \cdots \mathrel{[\!]} {\mathsf{C}_m}^{\mathtt{u}_m}(\spadesuit_{m,1}, \ldots, \spadesuit_{m,a_m}) \\ & \forall i \in \{h \mid 1 \leq h \leq m \text{ and } \mathtt{u}_i = \ell\}.\forall j \in \{1, \ldots, a_i\}.\psi_{i,j} = \begin{cases} \phi^\mu, & \text{if } \spadesuit_{i,j} = X \\ \spadesuit_{i,j}, & \text{if } \spadesuit_{i,j} \in \mathbf{L} \end{cases} \\ & \forall i \in \{1, \ldots, m\}.\mathbf{cc}_i = \begin{cases} \mathsf{C}_i(x_{i,1}, \ldots, x_{i,a_i})\ \mathsf{to}\ Q_i, & \text{if } \mathtt{u}_i = \ell \\ \mathsf{C}_i(x_{i,1}, \ldots, x_{i,a_i})\ \mathsf{to}\ \epsilon(Q_i), & \text{if } \mathtt{u}_i = \partial \end{cases} \end{array}$$

Fig. 10. Rules for e-type assignment (system $\vdash_{\mathbf{L}}$)

2. $\vdash_{\mathbf{T}} M : \rho$ *implies* $\delta(\mathrm{FV}(M)) \vdash_{\mathbf{L}} \delta(M)$ *and* $\Sigma \vdash_{\mathbf{L}} M^{\omega^\rho}$*, for all basis* Σ *such that* $\epsilon(\Sigma) \supseteq \mathrm{FV}(M)$*.*

To state the soundness of the e-type assignment system w.r.t. the semantics we introduce the following definition.

Definition 17. 1. Two environments e_1, e_2 are Σ-related if, for all $x^\psi \in \Sigma$, $(e_1(x), e_2(x)) \in [\![\psi]\!]$.
2. Let $\Sigma \vdash_{\mathbf{L}} M^\phi$ and $\Sigma \vdash_{\mathbf{L}} N^\phi$. We write $\epsilon(M^\phi) \sim^\Sigma_\phi \epsilon(N^\phi)$ to mean that for all e_1, e_2, if e_1 and e_2 are Σ-related, then $([\![\epsilon(M^\phi)]\!]_{e_1}, [\![\epsilon(N^\phi)]\!]_{e_2}) \in [\![\phi]\!]$.

Now we can state the soundness of the system $\vdash_{\mathbf{L}}$ w.r.t. Definitions 9 and 7.

Theorem 18 (Soundness of $\vdash_{\mathbf{L}}$). *Let* $\Sigma \vdash_{\mathbf{L}} M^\phi$*. Then* $\epsilon(M^\phi) \sim^\Sigma_\phi \epsilon(M^\phi)$*.*

5.2 A dead-code simplification

In this section we introduce a dead-code elimination mapping $\mathbf{O}$ that takes a $\vdash_{\mathbf{L}}$-decorated term M^ϕ and returns a simplified version of it in which the dead-code showed by the e-type decorations has been replaced by "dummy variables".

For each type ρ, define d^ρ, d^ρ_1, d^ρ_2, ..., 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 place-holders for the dead-code removed[19]. So in the following we assume that all the

[19] Dummy variables are simply a tool for proving the soundness of the simplification mapping $\mathbf{O}$. Indeed, since in a simplified program every dummy variable is dead-code, in a real implementation we can replace them with some polymorphic "dummy constant" d.

occurrences of 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, let $\mathrm{DV}(M)$ be the set of the dummy variables in M.

Definition 19 (Simplification mapping O). 1. The function $\mathbf{O} : \Lambda_{\mathbf{T}}^{\vdash_{\mathbf{L}}} \to \Lambda_{\mathbf{T}}^{\vdash_{\mathbf{L}}}$ is defined by the clauses in Fig. 11, where the occurrence of "d" in the second row denotes a fresh dummy variable of the proper type.
2. If Σ is a basis then define $\mathbf{O}(\Sigma) = \{x : \chi \mid x : \chi \in \Sigma \text{ and } \chi \notin \mathbf{L}_\omega\}$.

$\mathbf{O}(M^\psi) \;=\; (\mathbf{o}(M,\psi))^\psi, \quad$ where

1. $\mathbf{o}(M,\psi) \;=\; \mathsf{d}, \quad \text{if } \psi \in \mathbf{L}_\omega$
2. $\mathbf{o}(M,\psi) \;=\; \mathsf{fix}\, x^\psi.x, \quad \text{if } \psi \in \mathbf{L}_\partial$
3. Otherwise:

$$\mathbf{o}(\mathsf{k},\psi) = \mathsf{k}$$

$$\mathbf{o}(x,\psi) = x$$

$$\mathbf{o}(MN^\phi,\psi) = \mathbf{o}(M,\phi\to\psi)(\mathbf{o}(N,\phi))^\phi$$

$$\mathbf{o}(\lambda x.M,\psi_1\to\psi_2) = \lambda x.\mathbf{o}(M,\psi_2)$$

$$\mathbf{o}(\mathsf{fix}\,x^\phi.M,\psi) = \begin{cases} \mathbf{o}(M,\psi) & \text{if } \phi\in\mathbf{L}_\omega \\ \mathsf{fix}\,x^\psi.\mathbf{o}(M,\psi) & \text{if } \phi=\psi \end{cases}$$

$$\begin{array}{l} \mathbf{o}(\mathsf{C}_i(P_1,\dots,P_{a_i}),\phi^\mu) = \mathsf{C}_i(\mathbf{o}(P_1,\psi_1),\dots,\mathbf{o}(P_{a_i},\psi_{a_i})) \\ \quad\text{where } \phi^\mu = \mathsf{data}\,X.\mathbf{dc}_1^{\mathtt{u}_1} \parallel \cdots \parallel \mathbf{dc}_m^{\mathtt{u}_m} \\ \qquad\qquad \mathbf{dc}_i^{\mathtt{u}_i} = \mathsf{C}_i^\ell(\spadesuit_{i,1},\dots,\spadesuit_{i,a_i}) \\ \qquad\qquad \forall j\in\{1,\dots,a_i\}.\psi_j = \begin{cases} \phi^\mu & \text{if } \spadesuit_{i,j}=X \\ \spadesuit_{i,j} & \text{if } \spadesuit_{i,j}\in\mathbf{L} \end{cases} \end{array}$$

$$\mathbf{o}(\mathsf{case}\,P^{\phi^\mu}\,\mathsf{of}\,\{\mathbf{cc}_1 \parallel \cdots \parallel \mathbf{cc}_m\},\chi) \;=\mathsf{fix}\,x^\chi.x \qquad \text{if } \phi^\mu\in\mathbf{L}_\partial$$

$$\begin{array}{l} \mathbf{o}(\mathsf{case}\,P^{\phi^\mu}\,\mathsf{of}\,\{\mathbf{cc}_1 \parallel \cdots \parallel \mathbf{cc}_m\},\chi) = \mathsf{case}\,\mathbf{o}(P,\phi^\mu)^{\phi^\mu}\,\mathsf{of}\,\{\mathbf{cc}'_1 \parallel \cdots \parallel \mathbf{cc}'_m\} \\ \quad\text{where } \phi^\mu = \mathsf{data}\,X.\mathbf{dc}_1^{\mathtt{u}_1} \parallel \cdots \parallel \mathbf{dc}_m^{\mathtt{u}_m} \notin \mathbf{L}_\partial \\ \qquad \forall i\in\{1,\dots,m\}.\mathbf{dc}_i = \mathsf{C}_i^\ell(\spadesuit_{i,1},\dots,\spadesuit_{i,a_i}) \\ \qquad\qquad\qquad \mathbf{cc}_i = \mathsf{C}_i(x_{i,1},\dots,x_{i,a_i})\;\mathsf{to}\;Q_i \\ \qquad\qquad\qquad \mathbf{cc}'_i = \begin{cases} \mathsf{C}_i(x_{i,1},\dots,x_{i,a_i})\;\mathsf{to}\;\mathbf{o}(Q_i,\chi) & \text{if } \mathtt{u}_i=\ell \\ \mathsf{C}_i(x_{i,1},\dots,x_{i,a_i})\;\mathsf{to}\;\mathsf{d} & \text{if } \mathtt{u}_i=\partial \end{cases} \end{array}$$

Fig. 11. Dead-code simplification mapping **O**

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

Proposition 20 (Correctness of O w.r.t. the e-types semantics). *If* $\Sigma \vdash_{\mathbf{L}} M^\phi$ *then* $\epsilon(M^\phi) \sim_\phi^{\Sigma'} \epsilon(\mathbf{O}(M^\phi))$*, where* $\Sigma' = \Sigma \cup \{\mathsf{d}^\sigma : \omega^\sigma \mid \mathsf{d}^\sigma \in \mathrm{DV}(\epsilon(\mathbf{O}(M^\phi)))\}$.

In order to use the simplification mapping **O** to simplify terms while preserving their meaning (w.r.t. $\mathcal{M}^{\mathrm{gnd}}$) we identify a subset of $\vdash_{\mathbf{L}}$-typings (that we call *faithful*) for which the $\sim_\phi^\Sigma$ relations implies the $\simeq_{\mathrm{obs}}^{\mathrm{gnd}}$ relation[20].

[20] A faithful $\vdash_{\mathbf{L}}$-typing simply says that the term can be used (it has e-type $\in \mathbf{L}_\delta$) and that any its free variable is either used (it has e-type $\in \mathbf{L}_\delta$) or not used at all (it has e-type $\in \mathbf{L}_\omega$).

Definition 21 (Faithful $\vdash_{\mathbf{L}}$-typing). $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ is a *faithful* $\vdash_{\mathbf{L}}$-type assignment statement if $\phi \in \mathbf{L}_{\delta}$, and for all $x : \psi \in \Sigma$, $\psi \in \mathbf{L}_{\delta} \cup \mathbf{L}_{\omega}$.

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

Lemma 22. *Let $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ and $\Sigma \vdash_{\mathbf{L}} N^{\phi}$ be faithful $\vdash_{\mathbf{L}}$-typings. Then $\epsilon(M^{\phi}) \sim^{\Sigma}_{\phi} \epsilon(N^{\phi})$ implies $\epsilon(M^{\phi}) \simeq^{\text{gnd}}_{\text{obs}} \epsilon(N^{\phi})$.*

We can now show how to use the system $\vdash_{\mathbf{L}}$ 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 $\vdash_{\mathbf{L}}$-typing are dead-code.

Theorem 23 ($\mathbf{O}$ on faithful $\vdash_{\mathbf{L}}$-typings preserves $\simeq^{\text{gnd}}_{\text{obs}}$). *Let $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ be a faithful $\vdash_{\mathbf{L}}$-typing. Then $\epsilon(M^{\phi}) \simeq^{\text{gnd}}_{\text{obs}} \epsilon(\mathbf{O}(M^{\phi}))$.*

5.3 A dead-code and minimum-information-code simplification

In this section we show how the simplification mapping $\mathbf{O}$ can be extended to remove the minumum-information-code. The improved mapping, $\mathbf{O}'$, performs more simplifications then $\mathbf{O}$, however in general it does not preserve the meaning (w.r.t. $\mathcal{M}^{\text{gnd}}$) of terms since it may give simplified terms which are strictly more observationally greater ($\succeq^{\text{gnd}}_{\text{obs}}$) than the original ones (see Example 2).

Definition 24 (Simplification mapping $\mathbf{O}'$). The function $\mathbf{O}' : \Lambda^{\vdash_{\mathbf{L}}}_{\mathbf{T}} \to \Lambda^{\vdash_{\mathbf{L}}}_{\mathbf{T}}$ is defined as the function $\mathbf{O}$ (see Definition 19) by adding to the last clause for the mapping $\mathbf{o}(\cdot,\cdot)$ in Fig. 11 the condition $\phi^{\mu} \notin \mathbf{L}_{\ell 1}$, and by adding the following two clauses.

- Between the clauses 2 and 3:
  2′. $\mathbf{o}(M, \psi) \; = \; \mathsf{C}_i(\mathsf{d}^{\rho_1}, \ldots, \mathsf{d}^{\rho_{a_i}})$,
  if ψ is the $\ell 1$-datatype $\mathsf{data}\, X.\mathbf{dc}^{\partial}_1 \; [\![\; \cdots \; [\![\; \mathsf{C}_i(\omega^{\rho_1}, \ldots, \omega^{\rho_{a_i}}) \; [\![\; \cdots \; [\![\; \mathbf{dc}^{\partial}_m$.
- As last clause:
  $\mathbf{o}(\mathsf{case}\; P^{\phi^{\mu}} \;\mathsf{of}\; \{\mathbf{cc}_1 \; [\![\; \cdots \; [\![\; \mathbf{cc}_m\}, \chi) = \mathbf{o}(Q_i, \chi)$,
  where ϕ^{μ} is the $\ell 1$-datatype $\mathsf{data}\, X.\mathbf{dc}^{\partial}_1 \; [\![\; \cdots \; [\![\; \mathsf{C}_i(\omega^{\rho_1}, \ldots, \omega^{\rho_{a_i}}) \; [\![\; \cdots \; [\![\; \mathbf{dc}^{\partial}_m$
  and $\mathbf{cc}_i = \mathsf{C}_i(x_{i,1}, \ldots, x_{i,a_i})\; \mathsf{to}\; Q_i$.

Theorem 25 ($\mathbf{O}'$ on faithful $\vdash_{\mathbf{L}}$-typings). *Let $\Sigma \vdash_{\mathbf{L}} M^{\phi}$ be a faithful $\vdash_{\mathbf{L}}$-typing. Then $\epsilon(M^{\phi}) \preceq^{\text{gnd}}_{\text{obs}} \epsilon(\mathbf{O}'(M^{\phi}))$.*

Note that, however, when dealing with terminating programs[21] (or with programs that are trusted to terminate[22]) we have that, the simplified programs are (or are trusted to be) ground observationally equivalent to the original ones.

5.4 Computing the best simplification of a program

Using the same technique of [6] (see also [7] Chapter 7) we can prove that for every PCFD term M there is a faithful $\vdash_{\mathbf{L}}$-typing (see Definition 21) showing the maximum amount of reduntant-code. This is done by reducing the e-type inference problem to the solution of a set of contraint involving two kinds of variables, ranging over $\{\delta, \omega\}$ and over $\{\ell, \partial\}$, respectively. Each set of constraints admits a solution that maximizes the number of variables that takes values ω and ∂, and such a maximal solution determines a faithful $\vdash_{\mathbf{L}}$-typing showing the maximum amount of redundant-code. In this way we also provide a complete (w.r.t. the system $\vdash_{\mathbf{L}}$) redundant-code detection algorithm.

[21] Like programs extracted from proofs.

[22] As well written pure functional programs should be.

As showed in Example 2, redundant-code elimination may "rise" new redundant-code, so we have to apply the detection algorithm and the simplification mapping $\mathbf{O}'$ repeatedly, until no new redundant-code is discovered. At the end we can apply (once) a ($\simeq^{\mathrm{gnd}}_{\mathrm{obs}}$ preserving) simplification mapping that removes some[23] of the place-holders d introduced in the previous simplification step and that does no introduce new redundant-code.

We leave as an exercise to the reader the application of the system $\vdash_{\mathbf{L}}$ and of the simplification mapping $\mathbf{O}'$ to the term N of Example 2. Assuming that Q_2 does not contain redundant-code, two successive applications of the mapping $\mathbf{O}'$ on faithful $\vdash_{\mathbf{L}}$-typing showing the maximal (w.r.t. $\vdash_{\mathbf{L}}$) amount of redundant-code suffice to remove all the redundant-code present in N.

5.5 About functional languages with other evaluation strategies

PCFD is a lazy-call-by-name language, however the redundant-code detection and elimination techniques described in this paper can be applied independently from the evaluation strategy of the language (an so also to lazy-call-by-value languages[24]): it still holds that the simplified term is observationally greater ($\succeq^{\mathrm{gnd}}_{\mathrm{obs}}$) than the original one[25]. In particular, for every term in which a maximum amount of redundant-code (w.r.t. a faithful $\vdash_{\mathbf{L}}$-typing) has been simplified using either $\mathbf{O}$ or $\mathbf{O}'$:

- the evaluation under the lazy-call-by-name evaluation strategy never requires the evaluation of a dummy variable, and
- the evaluation under any evaluation strategy never requires the evaluation of an application of a dummy variable (i.e., $\mathsf{d}M_1 \cdots M_n$ with $n \geq 1$) or of a case-choice over a dummy variable (i.e., $\mathsf{case}\,\mathsf{d}\,\mathsf{of}\ \cdots$).

6 Related work

The class of redundant-code characterized in this paper strictly includes the redundant-code characterized by the technique presented by Berardi an Boerio in [4]. The extra power is due to the use of subtyping (not used in [4]). For a discussion about the power gained by adding the e-type entailment relation we refer to [3] and [5] Chapter 4 (where a dead-code analysis with type entailment for a simply typed lambda calculus[26] is presented), see also [6] and [7] Chapter 7 (where a constraint-based inference algorithm for the analysis of [3] is presented). We remark that in presence of datatypes the advantage of having the e-type entailment relation is even greater. Take for instance the term $P = (\lambda z^{\mu}. \cdots (\mathsf{case}\,B\,\mathsf{of}\,\{\ \mathsf{True}\ \mathsf{to}\ z \mid \mathsf{False}\ \mathsf{to}\ \mathsf{C}_i(\cdots)\ \}) \cdots)Q$, where the case-expression is not redundant-code. Suppose that we can assign to Q an e-type in which the constructor C_i is ∂ead. If we try to assign an e-type to P without using entailment, we are forced to assign to every (non-dead) occurrence of z in P (and also to Q) an e-type ϕ^{μ}_1 in which C_i is ℓive[27]. By using entailment, instead, we can assign such an e-type ϕ^{μ}_1 to the occurrence of z in the True branch of the case-expression, but we can assign e-types $\phi^{\mu}, \phi^{\mu}_2, \phi^{\mu}_3, \ldots$ in which C_i is ∂ead to Q and to the other occurrences of z in P, respectively (provided that $\phi^{\mu} \leq \phi^{\mu}_1$, $\phi^{\mu} \leq \phi^{\mu}_2, \ldots$).

Beside the use of the e-type entailment relation, the main differences between our approach and that of [4] are in the programming language considered and in the algorithm that finds the redundant-code in a given term. The language considered in [4] is strongly normalizing and it can be seen as the language obtained from PCFD by removing the constructor fix and adding, for every type $\tau \in \mathbf{T}$, an operator it_{τ} for primitive recursion over datatypes and from datatypes to every type.

[23] As explained in [5] Chapter 4 and [7] Chapter 7, when using an e-type entailment relation it is not possible, in general, to remove all the place-holders d.

[24] Like ML, Caml and Scheme.

[25] Notice that Theorem 23 does not extend to all the evaluation strategies, but in this case also the mapping $\mathbf{O}$ may produce simplified programs that are strictly $\succeq^{\mathrm{gnd}}_{\mathrm{obs}}$ than the original ones.

[26] With pairs and primitive recursion over natural numbers, but without other datatypes.

[27] Since ℓive branches of a case-expression must have the same (non-ω) e-type.

The algorithm described in [4] is a kind of "data flow" algorithm that analyzes a term by implicitly building a directed graph which represents the input-output relation between the subterms of a given term. The principal advantage of our constraint-based algorithm is that (as shown in [6] and [7] Chapter 7) it is naturally compositional while that of [4] is not. In particular the same set of constraints can be carried on along the iterations corresponding to applications of the mapping $\mathbf{O}'$ (see Section 5.4), while with the algorithm of [4] it is necessary to rebuild completely, at every iteration, the complex data structure that represents symbolically the computation flow in the term being analyzed.

Acknowledgements. I would thank Stefano Berardi, Mario Coppo and Paola Giannini for valuable remarks and suggestions. I would also to thank Jean-Christophe Filliâtre for stimulating discussions on the topic of program extraction.

During the elaboration of this paper I was visiting the Laboratoire d'Informatique de l'École Polytechnique (LIX). I would like to thank my host Radhia Cousot and the whole LIX for the ideal working conditions they provided.

References

1. 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.
2. S. Berardi. Pruning Simply Typed Lambda Terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.
3. S. Berardi and L. Boerio. Using Subtyping in Program Optimization. In *TLCA'95*, LNCS 902. Springer–Verlag, 1995.
4. S. Berardi and L. Boerio. Minimum Information Code in a Pure Functional Language with Data Types. In *TLCA'97*, LNCS 1210. Springer–Verlag, 1997.
5. L. Boerio. *Optimizing Programs Extracted from Proofs.* PhD thesis, Università di Torino, 1995.
6. M. Coppo, F. Damiani, and P. Giannini. Refinement Types for Program Analysis. In *SAS'96*, LNCS 1145, pages 143–158. Springer–Verlag, 1996.
7. F. Damiani. *Non-standard type inference for functional programs.* PhD thesis, Università di Torino, February 1998. Available at `htpp://www.di.unito.it/~damiani`.
8. F. Damiani and F. Prost. Detecting and Removing Dead Code using Rank 2 Intersection. In *International Workshop TYPES'96, Selected Papers*, LNCS, to appear. Springer–Verlag, 1997.
9. A. D. Gordon. *Functional Programming and Input/Output.* Distinguished Disserdations in Computer Science. Cambridge University Press, 1994.
10. A. D. Gordon. Bisimilarity as a Theory of Functional Programs. Mini-Course. Technical Report NS-95-3 BRICS Notes Series, Computer Science Department of Aarhus University, 1995.
11. G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming Of Future Generation Computer.* Elsevier Sciences B.V. (North-Holland), 1988.
12. C. Paulin-Mohring. Extracting F_ω's Programs from Proofs in the Calculus of Constructions. In *POPL'89.* ACM, 1989.
13. C. Paulin-Mohring. *Extraction de Programme dans le Calcul des Constructions.* PhD thesis, Université Paris VII, 1989.
14. 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.
15. 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.
16. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.
17. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
18. F. Prost. Marking techniques for extraction. Technical Report RR95-47, Ecole Normale Supérieure de Lyon, Lyon, December 1995.
19. Y. Takayama. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.