Acta Informatica 27, 505–517 (1990)

Acta Informatica

Referential Transparency, Definiteness and Unfoldability

Harald Søndergaard[1] and Peter Sestoft[2]

[1] Department of Computer Science, University of Melbourne, Parkville 3052 Vic., Australia
[2] DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

Received November 30, 1987/January 4, 1990

Summary. The term "referential transparency" is frequently used to indicate that a programming language has certain useful substitution properties. We observe, however, that the formal and informal definitions given in the literature are not equivalent and we investigate their relationships. To this end, we study the definitions in the context of a simple expression language and show that in the presence of non-determinism, the differences between the definitions are manifest. We propose a definition of "referential transparency", based on Quine's, as well as of the related notions: definiteness and unfoldability. We demonstrate that these notions are useful to characterize programming languages.

1. Introduction

The notions of referential transparency and referential opacity are common in discussions of properties of programming languages. They were originally suggested by Quine and brought into computer science by Landin and Strachey. The notions however have changed during time and the formal or informal definitions found in the literature are not equivalent.

The present paper discusses referential transparency, referential opacity, and similar notions. Precise definitions are suggested in the context of a non-deterministic expression language, but the perspective is broader since some of the definitions extend to programming languages more generally. The purpose is to clarify the relations between a number of properties of a formal language. These are very fundamental properties concerning identity and substitutivity, and so should be treated with great care.

The observation that in everyday language one may not always substitute a term by an equivalent term is exemplified by Quine. The statements

$$\text{Cicero} = \text{Tully} \tag{1.1}$$

$$\text{'Cicero' contains six letters} \tag{1.2}$$

Offprint requests to: H. Søndergaard

are both true, but the replacement of the first name by the second will turn (1.2) false [13]. The point is that, owing to the quotes, 'Cicero' in (1.2) does not refer to the person Cicero, but to the word Cicero. In this way the quotes change or destroy reference, that is, the relation between a term and the object(s) it denotes.

The fact that words are sometimes used in this manner is noted by the medieval William of Sherwood, and Leibniz repeatedly mentions it (see Notes 1 and 2). Frege, in his discussion of sense and reference, gives the following formulation:

> "It can also happen, however, that one wishes to talk about the words themselves or their sense. This happens, for instance, when the words of another are quoted. One's own words then first designate words of the other speaker, and only the latter have their usual reference. We then have signs of signs. In writing, the words are in this case enclosed in quotation marks. Accordingly, a word standing between quotation marks must not be taken to have its ordinary reference" (see Note 3).

A thorough investigation of the phenomenon has been undertaken by Quine. He calls a context *referentially opaque* if it destroys reference. Otherwise it is *referentially transparent* [14].

We just saw how quoting is referentially opaque, and the example

Tegucigalpa = the capital of Honduras (1.3)

Philip believes that Tegucigalpa is in Nicaragua (1.4)

shows that "_ believes that _" is referentially opaque in place 2, since (1.3) and (1.4) may well hold, whereas replacing Tegucigalpa in (1.4) by the right-hand side of (1.3) presumably yields a falsehood [13]. In general modal contexts fail to preserve reference, and so are referentially opaque.

The usefulness of Quine's notions to the science of programming languages was realized by Landin and Strachey. Strachey refers to Quine and the notion of referential transparency which

> "means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression it its value" [18, page 16].

We shall henceforth refer to this principle as *extensionality* of the enclosing expression. Strachey also notes that

> "We tend to assume automatically that the symbol x in an expression such as $3x^2+2x+17$ stands for the same thing (or has the same value) on each occasion it occurs. This is the most important consequence of referential transparency" [18, page 22].

We shall refer to this principle as *definiteness* (of variables, which we thus call definite or indefinite depending on whether the principle applies or not).

Strachey's statement may be valid in the case of deterministic programming languages, but in a non-deterministic language it becomes a crucial question what is meant by "stands for the same thing". In a very natural interpretation it turns out that definiteness is *not* a consequence of extensionality. We demonstrate this by giving a language Q_1 which is referentially transparent in Quine's sense, but in which distinct occurrences of a variable (even within the same scope) may have different values.

Based on Strachey's observations, Stoy gives an informal *definition* of referential transparency in this textbook on denotational semantics. Definiteness, which Strachey regards a *consequence* of referential transparency, becomes *part* of Stoy's definition:

> "We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs". [17, page 5]

This definition has three components. The first states the extensionality principle. The second is the principle that "any subexpression can be replaced by any other equal in value" which we refer to as *Leibniz's law*, or *substitutivity of identity*. The third expresses the principle of definiteness, and perhaps even some kind of *determinacy*. The qualification "within certain limits" is presumably to allow for the notion of *scope* of variables. Thus, in arguing that the lambda calculus is referentially transparent in his sense, Stoy notes that

> "for $\ulcorner\lambda$ I.E$\urcorner$ all *free* occurrences of I in E denote the same value". [17, page 190, italics ours]

Stoy's book does not discuss mathematical semantics for non-deterministic languages, a topic that was just about to find its proper treatment at the time the book was written. So the above definition is perfectly adequate for Stoy's purpose. In general, however, it is advantageous to separate the component notions, because each provides a useful dimension in the characterization of a formal language, and because they are fundamentally distinct, as will be shown.

Languages with assignment lack many useful substitutivity properties. Consider

$$x := x - 1; \quad y := x. \tag{1.5}$$

All occurrences of x in (1.5) are in the same scope. A discussion of reference is complicated by the fact that an occurrence of x on the left-hand side of $:=$ is interpreted rather differently to an occurrence of x on the right-hand side. This is the well-known distinction between L-values and R-values of variables [17, 18]. In (1.5), the first occurrence of x is taken to denote a "location" which in turn "holds" a value, whereas the other occurrences denote values. Whether such ambiguity should count as destruction of reference may be a matter of taste. In any case variables are not definite, since even the two right-hand side occurrences of x have different values.

In functional programming languages, the use of substitutivity of identity as a criterion for referential transparency has an added twist. Namely, one kind of substitution that may or may not apply is *unfolding* of a function application (corresponding to β-reduction in the lambda calculus). If unfolding is equivalence preserving, we say that *unfoldability* applies, or that β-reduction is admissible. The problem is that it is common to use the equality symbol " $=$ " when writing function definitions, as in

$$f\,x = \ldots x \ldots \tag{1.6}$$

This may be misleading, since it may well be the case that substitutivity of identity holds and yet unfoldability does not apply, so that it is not admissible to replace the function application $f\,e$ by the instantiated right-hand side $\ldots e \ldots$ of (1.6). This

will be exemplified when we investigate unfoldability and its relation to the notions of referential transparency and definiteness.

Textbooks on programming languages offer a variety of suggestions as to what referential transparency means. The following is a selection: definiteness (Note 4), absence of side-effects (Note 5), determinacy (Note 6), unfoldability (Note 7), extensionality (Note 8), and applicability of Leibniz's law (Note 9). This variety is justified to some extent because textbooks often teach more efficiently by simplifying issues slightly. It must however confuse a student seeking precise knowledge: certainly not all of the above notions are equivalent. The intention with this paper is to clarify some of their relations.

After some preliminary remarks on notation in Sect. 2, we present in Sect. 3 a simple expression language called Q_0. This language will serve as a basis for the discussion of concepts. In Sect. 4 we define the notions of referential transparency, referential opacity, definiteness, and unfoldability, and we use these to characterize the language Q_0. In Sect. 5 we apply some distortions to the semantics of the language, thus obtaining a series of slightly different languages $Q_1 - Q_4$ that exhibit varying combinations of the properties introduced in Sect. 4. We summarize in Sect. 6 and draw conclusions as to the relations between the notions discussed in the paper.

2. Preliminaries

The present note is concerned with issues of reference. We should therefore state exactly which devices the paper itself employs to indicate referential use. We use italics to indicate identifiers such as x or *exp*. We use the pair of brackets $[\![...]\!]$ for quasi-quotation of expressions: meta-language variables that appear between the brackets denote whatever they are bound to, whereas all other symbols denote themselves. This is the usual convention of denotational semantics. Thus the brackets are nothing but "Quine corners" [12].

To distinguish various kinds of equivalence, we use two equivalence signs: a "syntactic" and a "semantic" one. The symbol $\equiv$ is used between expressions to denote syntactic identity, that is, the relation in which an expression stands to itself and to no other expression. Trivial as it may seem, there is good use for $\equiv$, because we use variables and expressions that range over syntactic objects (expressions); thus it makes sense to write, for instance, $e_1 \equiv e_2$, where e_1 and e_2 are meta-language variables.

The equality symbol, $=$, is used for a number of purposes: (1) between elements of a set, (2) between sets, with the usual meaning: each is a subset of the other, (3) between functions to denote strong pointwise equality, and (4) between expressions to denote codenotation: both sides denote the same object.

In the following, some knowledge of denotational semantics will be beneficial [17]. A note concerning the modeling of non-determinism and the interpretation of equality is in order. Consider the non-deterministic expressions e_1 and e_2. We follow the standard approach and let the denotation of an expression be the *set* of values it may evaluate to (including possible error values). So $e_1 = e_2$ means that whatever e_1 may evaluate to, e_2 may evaluate to, and *vice versa*. This, however, does not mean that e_1 and e_2 will necessarily evaluate to the same thing. In this way the introduction of non-determinism highlights a difference between *denotation* and *intended meaning*: expression e may denote a set, but the intended meaning is that e stands for some member of the set, we just cannot know which.

3. A Simple Expression Language

This section introduces the language Q_0 which we shall use for defining and illustrating various substitution properties. Q_0 is a very restricted expression language and certainly useless as a programming language. Even so, we feel justified in basing our analysis on it, because it incorporates constructs known from existing programming languages, albeit only those considered relevant for our discussion: Occam's razor has been applied whenever possible, to bring the issues under investigation into focus. The syntax of Q_0 is given in Table 1. In Sect. 5 we discuss a series of languages $Q_1 - Q_4$, all having the same syntax as Q_0 but with differing semantics. Each of these semantics is defined by changing the definition of the preceding language very slightly. Nevertheless they will be shown to lead to rather different substitution properties.

The "full form" displayed will help certain definitions, but for readability we normally use the "short form" of expressions, using parentheses to remove ambiguity. However, we use $\lambda_x e\, e'$ simply, for $(\lambda_x e)\, e'$, in the understanding that variables in e' are not bound by the occurrence of λ_x. Since x is the only available variable, the standard notation "$\lambda x.e$" for a lambda abstraction would be pleonastic, and the use of "λe" would disturb intuition, hence the notation "$\lambda_x e$".

Table 2 gives a denotational definition of Q_0. The set of values of expressions is

$$V = \{0, 1, \#\},$$

Table 1. Syntax for the Q family of languages

	Full form	Short form
exp →	0	0
	| 1	1
	| *x*	*x*
	| ⟨*minus exp exp*⟩	*exp* − *exp*
	| ⟨*numeral? exp*⟩	@ *exp*
	| ⟨*choose exp exp*⟩	*exp* ⊓ *exp*
	| ⟨*apply fun exp*⟩	*fun exp*
fun →	⟨*lambda exp*⟩	λ_x *exp*

Table 2. Semantics of Q_0

Below, $e, e_1, e_2 \in exp$; $f \in fun$; $v_1, v_2 \in V$; $u \in PV$.

$$D[\![e]\!] = E[\![e]\!]\{\#\}$$

$$E[\![0]\!]u = \{0\}$$

$$E[\![1]\!]u = \{1\}$$

$$E[\![x]\!]u = u$$

$$E[\![e_1 - e_2]\!]u = \{\text{if } v_1 = \# \text{ or } v_2 = \# \text{ or } v_1 < v_2 \text{ then } \# \text{ else } v_1 - v_2 \mid v_1 \in E[\![e_1]\!]u \text{ and } v_2 \in E[\![e_2]\!]u\}$$

$$E[\![@\, e]\!]u = \text{if } e \equiv 0 \text{ or } e \equiv 1 \text{ then } \{1\} \text{ else } \{0\} \quad \text{(E1)}$$

$$E[\![e_1 \sqcap e_2]\!]u = E[\![e_1]\!]u \cup E[\![e_2]\!]u \quad \text{(E2)}$$

$$E[\![f\, e]\!]u = F[\![f]\!](E[\![e]\!]u) \quad \text{(E3)}$$

$$F[\![\lambda_x e]\!]u = \text{if } u = \{\#\} \text{ then } \{\#\} \text{ else } (E[\![e]\!]u) \cup (\{\#\} \cap u) \quad \text{(F1)}$$

where # signifies failure of evaluation. The non-determinism of expressions is modeled by having the semantic equations define *sets of possible results*. We therefore use a semantic domain

$$PV=(\mathscr{P}\, V)\backslash\{\varnothing\},$$

that is, the set of non-empty subsets of V. The empty set is excluded because an expression must have a value (if evaluation fails, then its "value" is #). The valuation function for expressions is

$$D: exp \to PV,$$

which in turn is defined in terms of the functions

$$E:\ exp \to PV \to PV$$
$$F:\ fun \to PV \to PV.$$

An expression is evaluated given an element from PV (namely the set of possible values of x), to yield an element from PV (namely the set of possible result values).

Let us informally describe evaluation of compound expressions. If e_1 and e_2 may evaluate to v_1 and v_2, respectively, then e_1-e_2 may evaluate to v_1-v_2 (which fails in case $v_1<v_2$). If e is a numeral (0 or 1), then @ e must evaluate to 1, otherwise it must evaluate to 0. The choice expression $e_1 \sqcap e_2$ may evaluate to v if either e_1 may or e_2 may, and it may fail if either e_1 may or e_2 may. This amounts to *erratic* non-determinism [4]. Finally, let u be the set of values to which e' may evaluate. The expression $\lambda_x e\, e'$ may evaluate to whatever e may evaluate to, given that an occurrence of x in e must evaluate to some $v \in u$. There are, however, two provisos, given by (F1): (1) if evaluation of e' must fail, then so must evaluation of $\lambda_x e\, e'$, and (2) if evaluation of e' may fail, then so may evaluation of $\lambda_x e\, e'$. This yields the natural counterpart to applicative order evaluation in a deterministic language.

We can now define semantic equivalence of expressions as codenotation. Expressions are *semantically equivalent* iff they have the same denotation. By definition,

$$e_1=e_2 \quad \text{iff } D[\![e_1]\!]=D[\![e_2]\!]$$

for all $e_1, e_2 \in exp$. Note that D is a parameter in this definition, in the sense that the definition applies throughout the paper, even when D (or E and F, on which D depends) is changed.

It is a crucial feature of Q_0 that x becomes bound to a (non-empty) *set* of values. As a consequence, the denotation of, say, $\lambda_x(x-x)(0 \sqcap 1)$ is $\{0, 1, \#\}$ rather than $\{0\}$. In other words, even within the same scope, distinct occurrences of a variable need not evaluate to the same result, witness the subexpression $(x-x)$ in $\lambda_x(x-x)(0 \sqcap 1)$. This type of semantics is known as *plural* [5] or *run time choice* [8] semantics. The last term hints at the operational view that a value for x is chosen anew every time x is met during evaluation.

The language Q_0 lacks a number of substitutivity properties which are introduced in the next section. In Sect. 5 we investigate the consequences of various changes in the semantic definition with respect to such properties, and in Sect. 6 we summarize and compare the properties. Elsewhere we have discussed further (algebraic) properties of languages akin to Q_0-Q_4, but with recursion [15, 16].

4. Referential Transparency and Substitution Properties

Quine defines referential transparency using the concept of a *purely referential position* in a sentence: "the position must be subject to the substitutivity of identity" [14, page 142]. He then defines *referential transparency*:

> "I call a mode of containment ϕ referentially transparent if, whenever an occurrence of a singular term t is purely referential in a term or sentence $\psi(t)$, it is purely referential also in the containing term or sentence $\phi(\psi(t))$" [14, page 144].

So ϕ is referentially transparent if it preserves pure referentiality, that is, preserves substitutivity of identity. This amounts to Leibniz's law, and to extensionality. However, it is independent of the other important substitutivity properties we have met. We now define the various substitutivity concepts more precisely and check whether they apply to Q_0.

A *position* p is a (possibly empty) sequence of natural numbers, $p \in N^*$. The empty sequence is denoted by ε. The sequence constructor is denoted by "$\cdot$". Let Ω be an operator. Expression e with e' *inserted at position* p, $e[e'/p]$, is defined by

$$e[e'/\varepsilon] \equiv e'$$
$$e[e'/i \cdot p] \equiv \langle \Omega\, e_1 \ldots e_i[e'/p] \ldots e_n \rangle \quad \text{if } e \equiv \langle \Omega\, e_1 \ldots e_i \ldots e_n \rangle, \quad \text{else undefined.}$$

Position p is *purely referential* in expression e iff $e_1 = e_2 \Rightarrow e[e_1/p] = e[e_2/p]$ for all $e_1, e_2 \in exp$. That is, a position is purely referential in expression e iff it is subject to the substitutivity of identity. Note that for every expression e, position ε is purely referential in e. An operator Ω is *referentially transparent* in place i iff for every expression $e \equiv \langle \Omega e_1 \ldots e_i \ldots e_n \rangle$, whenever position p is purely referential in e_i, position $i \cdot p$ is purely referential in e. Otherwise Ω is *referentially opaque* in place i. We shall also say that Ω is referentially transparent, simply, iff it is referentially transparent in all places. Similarly, Ω is referentially opaque iff it is referentially opaque in some place. Finally, we call a formal language referentially transparent if all its operators are referentially transparent; otherwise we call it referentially opaque.

Examples of referentially opaque operators are @ as introduced in Sect. 3, Pascal's quotation marks used for character strings, and Lisp's *quote* (a few other operators from the Lisp category of "fexpr" are referentially opaque, though not all: *cond*, for example, is referentially transparent). Common to these operators is that they depend on the *form* of their argument rather than its *value*. Note that a position may well be purely referential globally and yet not locally: if (as in Lisp) we had had an *eval* and a *quote* operator such that *eval (quote e)* $= e$, then the position of e in *eval (quote e)* would be purely referential, in spite of the fact that *quote* is referentially opaque.

A language has the *definiteness* property iff the denotation of a variable necessarily is a single value. For a language in our family, the implication of this is that $x - x$ will always evaluate to 0 (or possibly #), even though x may be bound (by a function application) to an expression such as $0 \sqcap 1$. To see that this is to give x a special treatment, consider replacing x by what it is bound to. This yields $(0 \sqcap 1) - (0 \sqcap 1)$, which may evaluate to either 0 or 1, or fail. If definiteness applies, we say that the variables are *definite*, otherwise we call them *indefinite*.

Let $e[e']$ denote the expression e with all free occurrences of x replaced by e' (even when enclosed by semantic brackets as in $[\![e[e']]\!]$). We say that *unfoldability* applies (for a language in our family) iff $\lambda_x\, e\, e' = e[e']$ for all $e, e' \in exp$.

Let us analyse Q_0 in terms of the introduced notions. First, Q_0 is not referentially transparent, because the operator @ destroys reference. As an example, $1-0=1$ since both sides denote $\{1\}$. However, $@(1-0) \neq @\,1$ since the left-hand side denotes $\{0\}$, whereas the right-hand side denotes $\{1\}$. All other operators are referentially transparent, as will be shown in Sect. 5.1. Second, variables in Q_0 are indefinite. Thus the Q_0 expression

$$\lambda_x(x-x)(0 \sqcap 1) \tag{4.1}$$

denotes $\{0, 1, \#\}$ rather than $\{0\}$. Finally, unfoldability does not apply in the case of Q_0. To see this, consider the expression

$$\lambda_x\, 0(0-1). \tag{4.2}$$

Unfolding this application yields 0, but $D[\![\lambda_x\, 0(0-1)]\!] = \{\#\}$, while $D[\![0]\!] = \{0\}$.

5. Variations on a Theme

In this section various changes are made to the semantics of Q_0. We thereby obtain a series of languages $Q_1 - Q_4$ that exhibit quite different substitutivity properties. In this way the languages illustrate how the properties interrelate.

5.1. Obtaining Referential Transparency

We have seen that @ is referentially opaque. The definition of the semantic function E is now changed slightly to make @ void of effect. We replace (E 1) by

$$E[\![@\, e]\!]\, u = E[\![e]\!]\, u. \tag{E 1'}$$

The resulting language is called Q_1.

Example. The Q_1 expression $@(@\,1)$ denotes $\{1\}$, whereas *qua* Q_0 expression it denotes $\{0\}$. □

Clearly, by (E 1'), @ is no longer referentially opaque. In fact the following proposition holds.

Proposition. *Q_1 is referentially transparent.*

Proof. Consider the semantic definition in Table 2. The result of applying E to expressions of the form $(\lambda_x e_1)\, e_2$, $e_1 \sqcap e_2$, or $e_1 - e_2$ only depends on the meaning $E[\![e_i]\!]\, u$ of the subexpressions e_i for $i \in \{1, 2\}$. We demonstrate this in the case of e_1 in expression $e_1 \sqcap e_2$. If $e_1 = e_1'$, then $E[\![e_1]\!]\, u = E[\![e_1']\!]\, u$, by definition of "$=$" on expressions. So for $u \in PV$, $E[\![e_1 \sqcap e_2]\!]\, u = E[\![e_1]\!]\, u \cup E[\![e_2]\!]\, u = E[\![e_1' \sqcap e_2]\!]\, u$, and so $e_1 \sqcap e_2 = e_1' \sqcap e_2$. This proves that $(\_ \sqcap \_)$ is purely referential in place 1, and since e_1 and e_2 were arbitrary, it follows that it is referentially transparent in place 1. Similar arguments hold for all other cases. □

Although no operator thus destroys reference, variables are still not definite. Namely, example (4.1) *qua* Q_1 expression evaluates exactly like before, since it does not contain

the @ operator. Furthermore, as example (4.2) *qua* Q_1 expression shows, unfoldability still does not apply.

This proves that a non-deterministic programming language may well be referentially transparent in the sense of Sect. 4 (and of Quine) and yet violate the principle that "a variable stands for the same thing on each occasion it occurs". It also shows that referential transparency does not imply unfoldability.

5.2. *Obtaining Unfoldability*

We now modify Q_1 to make unfolding equivalence preserving. This is done by replacing (F 1) in the definition of Q_1 by

$$F[\![\lambda_x e]\!]u = E[\![e]\!]u. \tag{F 1'}$$

The resulting language is called Q_2. Evaluation in Q_2 resembles normal order evaluation in deterministic languages.

Example. The Q_2 expression $\lambda_x 0(0-1)$ denotes $\{0\}$, whereas *qua* Q_0 or Q_1 expression it denotes $\{\#\}$, that is, evaluation must fail. □

Like Q_1, Q_2 is referentially transparent, and variables in Q_2 are still indefinite, as witnessed by (4.1) *qua* Q_2 expression. However, Q_2 allows for unfolding as the following proposition shows.

Proposition. *Let e and e' be Q_2 expressions. Then $\lambda_x e\, e' = e[e']$.*

Proof. Let $u \in PV$ and let $u' = E[\![e']\!]u$. Since by (E 3) and (F 1'),

$$E[\![\lambda_x e e']\!]u = F[\![\lambda_x e]\!](E[\![e']\!]u) = E[\![e]\!]u',$$

it suffices to show that $E[\![e]\!]u' = E[\![e[e']]\!]u$ for all $e, e' \in exp$. We show this by induction on the structure of e. The cases $e \equiv 0$ and $e \equiv 1$ are trivial. For $e \equiv x$ we have: $E[\![x]\!]u' = E[\![e']\!]u = E[\![x[e']]\!]u$. The case $e \equiv e_1 - e_2$ is similar to $e_1 \sqcap e_2$ below.

Case $e \equiv @\, e_1$:

$$\begin{aligned} E[\![@\, e_1]\!]u' &= E[\![e_1]\!]u' && \text{by (E 1')} \\ &= E[\![e_1[e']]\!]u && \text{by the induction hypothesis} \\ &= E[\![@(e_1[e'])]\!]u && \text{by (E 1')} \\ &= E[\![(@\, e_1)[e']]\!]u && \text{q.e.d.} \end{aligned}$$

Case $e \equiv e_1 \sqcap e_2$:

$$\begin{aligned} E[\![e_1 \sqcap e_2]\!]u' &= E[\![e_1]\!]u' \cup E[\![e_2]\!]u' && \text{by (E 2)} \\ &= E[\![e_1[e']]\!]u \cup E[\![e_2[e']]\!]u && \text{by the induction hypothesis} \\ &= E[\![e_1[e'] \sqcap e_2[e']]\!]u && \text{by (E 2)} \\ &= E[\![(e_1 \sqcap e_2)[e']]\!]u && \text{q.e.d.} \end{aligned}$$

Case $e \equiv \lambda_x e_1\, e_2$:

$$\begin{aligned} E[\![\lambda_x e_1\, e_2]\!]u' &= F[\![\lambda_x e_1]\!](E[\![e_2]\!]u') && \text{by (E 3)} \\ &= F[\![\lambda_x e_1]\!](E[\![e_2[e']]\!]u) && \text{by the induction hypothesis} \\ &= E[\![\lambda_x e_1(e_2[e'])]\!]u && \text{by (E 3)} \\ &= E[\![(\lambda_x e_1\, e_2)[e']]\!]u && \text{q.e.d.} \end{aligned}$$

By structural induction, $E[\![e]\!]u' = E[\![e[e']]\!]u$, and the proposition follows. □

5.3. *Obtaining Definiteness at the Expense of Unfoldability*

All of the languages so far had indefinite variables: different occurrences of a variable might evaluate differently, so for instance $x-x$ might evaluate to 1. We now want to make variables definite. This can be done by making the semantics *singular* [5], that is, by guaranteeing that at any instant, a variable denotes exactly one value (rather, a singleton set of values). This type of semantics is found also under the label *call time choice* semantics, reflecting the operational view that the value for a (lambda) variable is determined once: at the time of application [8]. More precisely we replace line (E3) in the definition of Q_2 by

$$E[\![f\,e]\!]\,u = \cup\{F[\![f]\!]\{v\} \mid v \in E[\![e]\!]\,u\}. \tag{E3'}$$

Here $\cup$ is the distributed union operator. The resulting language is called Q_3.

Example. The Q_3 expression

$$\lambda_x(x-x)(0 \sqcap 1) \tag{5.1}$$

denotes $\{0\}$, whereas *qua* Q_0, Q_1, or Q_2 expression it denotes $\{0, 1, \#\}$. □

It should be clear that x is now definite. However, as compared to Q_2, Q_3 has lost unfoldability. Unfolding the call in (5.1) yields $(0 \sqcap 1)-(0 \sqcap 1)$ which denotes $\{0, 1, \#\}$.

In the presence of non-determinism, we cannot obtain both definiteness and unfoldability. If we retain (F1) rather than (F1'), the resulting language (let us call it Q_3') will still have definite variables and lack unfoldability, since the above example still applies. Usually designers of non-deterministic languages choose to renounce unfoldability in order to keep variables definite. This is the case, for example, in the wide-spectrum language CIP-L whose transformation system explicitly requires a function to be determinate for any of its applications to be unfolded [3, page 99].

5.4. *Giving Up Non-Determinism*

The only way to achieve both definiteness and unfoldability is to give up non-determinism. This can be done for instance by replacing (E2) in the definition of Q_3 by

$$E[\![e_1 \sqcap e_2]\!]\,u = E[\![e_1]\!]\,u. \tag{E2'}$$

The resulting language is called Q_4.

Example. The Q_4 expression $0 \sqcap 1$ denotes $\{0\}$, whereas *qua* Q_0, Q_1, Q_2, or Q_3 expression it denotes $\{0, 1\}$. □

Thus $\sqcap$ is now almost void of effect. All sets that are returned by E are singleton sets, and it should be clear that both definiteness and unfoldability apply. Note that Q_4 is still referentially transparent, as is Q_1, Q_2, and Q_3. The operator $\sqcap$ does not destroy reference by ignoring e_2. It is still true that "all that matters about e_2 is its value", although this value is not being used for anything.

6. Conclusion

We have suggested a definition of referential transparency in programming languages which is in accordance with Quine's notion [14]. By this, an operator is referentially transparent if it preserves applicability of Leibniz's law, or substitutivity of identity: the principle that any subexpression can be replaced by any other equal in value.

Even though definiteness (the principle that within the same scope, distinct occurrences of a variable evaluate to the same) is sometimes taken as the definition of referential transparency, a language may well lack definiteness and yet be referentially transparent. The same goes for unfoldability, or admissibility of β-reduction. Neither of these properties are implied by referential transparency. This has been exemplified by a series of languages $Q_0 - Q_4$. It was also argued that in the presence of non-determinism it is impossible to obtain both definiteness and unfoldability at the same time. However, a non-deterministic language may well be referentially transparent.

Table 3 summarizes these findings. The table does not present an exhaustive list of the possible combinations of "yes" and "no". Many more (and indeed more contrived) languages are possible. So one should not deduce from the table that definiteness implies referential transparency or that determinacy implies unfoldability, for example. Such implications fail to hold in general. We conclude that it is useful to separate the issues: determinacy, definiteness, unfoldability, and referential transparency, because they cover different aspects of languages, and they all provide useful dimensions along which to characterize programming languages.

We have avoided such classical terms for evaluation strategies as "call by value" and "call by name", simply because they were not defined for non-deterministic languages. Table 4 suggests a natural way to generalize the notions (we find the nomenclature more natural than Clinger's [5], as well as in better agreement with current use for deterministic languages). In the absence of non-determinism, the two rows of Table 4 collapse into one: at least in a language without side-effects, there is no difference in the results obtained using call by name and call by need.

Table 3. Properties of the languages

	Q_0	Q_1	Q_2	Q_3	Q_4
Determinacy	no	no	no	no	yes
Definiteness	no	no	no	yes	yes
Unfoldability	no	no	yes	no	yes
Referential transparency	no	yes	yes	yes	yes

Table 4. Relations to operational notions

	"Applicative" order	"Normal" order
Plural semantics (run time choice)	Q_0 and Q_1	Q_2 Call by name
Singular semantics (call time choice)	(Q_3') Call by value	Q_3 and Q_4 Call by need

We have not discussed side-effects in the present paper, since this phenomenon does not come up in purely applicative languages. In a language like Algol 60, whose semantics can only be explained in terms of a notion of "state" [17], the expression $x-x$ (where x is an integer variable) may well evaluate to 6, say, owing to side-effects of the evaluation of x. Clearly non-determinism is not the only obstacle for definiteness.

Acknowledgements. We have found the referees' critical comments very helpful when revising this paper. In particular, the present simplified and improved example language is basically due to one of the referees. Thanks also go to Rodney Topor and Phil Wadler. The first author has been supported in part by the Danish Research Academy, the second by the Danish Natural Science Research Council.

Notes

1. "A∞B significat A et B esse *idem*, seu ubique sibi posse substitui. (Nisi prohibeatur, quod fit in iis, ubi terminus aliquis certo respectu considerari declaratur ver. g. licet trilaterum et triangulum sint idem, tamen si dicas triangulum, quatenus tale, habet 180 gradus; non potest substitui trilaterum. Est in eo aliquid materiale)." [6, page 261].
2. "Au reste il arrive quelques fois que nos idées et pensées sont la matiere de nos discours et font la chose même qu'on veut signifier, et les notions reflexives entrent plus qu'on ne croit dans celles des choses. On parle mêmes quelque fois des mots materiellement sans que dans cet endroit là precisement, on puisse substituer à la place du mot la signification, ou le rapport aux idées ou aux choses" [11, page 287].
3. „Es kann aber auch vorkommen, daß man von den Worten selbst oder von ihrem Sinne reden will. Jenes geschieht, z.B., wenn man die Worte eines Andern in gerader Rede anführt. Die eigenen Worte bedeuten dann zunächst die Worte des Andern und erst diese haben die gewöhnliche Bedeutung. Wir haben dann Zeichen von Zeichen. In der Schrift schließt man in diesem Falle die Wortbilder in Anführungszeichen ein. Es darf also ein in Anführungszeichen stehendes Wortbild nicht in der gewöhnlichen Bedeutung genommen werden" [7, page 28].
4. "Program variables necessarily violate Quine's (1960) request for "referential transparency": that a "variable"(=a designation) has the same value at every position in the text" [2, page 353].
5. "The purpose of an expression is to compute a value. Expressions are composed of operands such as constants and variables, operators and possibly function calls. The process of evaluation is to substitute values for the operands and perform the operations. The values which are associated with the variables of a program form the *state space* or *environment* of the program. Evaluation of an expression should only produce a value and not change the environment. This idea is called by the grandiose name *referential transparency*" [10, page 93].
6. "One essential ingredient of functional programming is that the value of a function is determined solely by the values of its arguments. Thus, calls of the function using the same arguments will always produce the same value. This property is called *referential transparency*" [10, page 357].
7. "...consider a logic program containing two statements

   S1: **parent** (x, z) **if mother** (x, z)
   S2: **grandparent** (x, y) **if parent** (x, z), **parent** (z, y)

We may ... substitute for **parent**(x, z) in S2 its referent **mother**(x, z)... to obtain

S3: **grandparent**(x, y) **if mother**(x, z), **parent**(z, y)

... Logic and other pure *declarative languages* therefore enjoy what is called *referential transparency*; what any of their constituents refer to in a program can be ascertained by exploiting the soundness of *substitutivity*, taking no account of the irrelevant run-time context" [9, page 240–241].

8. "... a principle known as *referential transparency* [Quine 1960] that holds in mathematics: The value of an expression can be determined solely from the values of its subexpressions, and if any subexpression is replaced by an arbitrary expression with the same value then the value of the entire expression remains unchanged" [19, page 29].
9. "A language that supports the concept that "equals can be substituted for equals" without changing the values of expressions is said to be *referentially transparent*" [1, page 178].

References

1. Abelson, H., Sussman, G.J.: Structure and interpretation of computer programs. Cambridge, Mass.: MIT Press 1985
2. Bauer, F.L., Wössner, H.: Algorithmic language and program development. Berlin Heidelberg New York: Springer 1982
3. Bauer, F.L. et al. The Munich project CIP, vol. I: The wide spectrum language CIP-L (Lect. Notes Comput. Sci., vol. 183). Berlin Heidelberg New York: Springer 1985
4. Broy, M.: A theory for nondeterminism, parallelism, communication, and concurrency. Theor. Comput. Sci. **45**, 1–61 (1986)
5. Clinger, W.: Nondeterministic call by need is neither lazy nor by name. Proc. 1982 Symp. Lisp Funct. Progr., Pittsburgh, Pennsylvania pp. 226–234, 1982
6. Couturat, L.: Opuscules et fragments inédits de Leibniz. Hildesheim: Georg Olm 1961
7. Frege, G.: Über Sinn und Bedeutung. Z. Philosophie philosophische Kritik **100**, 25–50 (1892). English translation in: Geach, P., Black, M. (eds.) Translations from the philosophical writings of Gottlob Frege, pp. 56–78. Oxford: Blackwell 1952
8. Hennessy, M.C.B., Ashcroft, E.A.: Parameter-passing mechanisms and nondeterminism. Proc. Ninth ACM Symp. Theory Comput., pp. 306–311. Boulder, Colorado 1977
9. Hogger, C.J.: Introduction to logic programming. London: Academic Press 1984
10. Horowitz, E.: Fundamentals of programming languages. Berlin Heidelberg New York: Springer 1983
11. Leibniz, G.W.: Nouveaux essais sur l'entendement humain, III, 2. In: Gottfried Wilhelm Leibniz – Sämtliche Schriften und Briefe VI, 6. Berlin (GDR): Akademie-Verlag 1962
12. Quine, W.V.O.: Mathematical logic. New York: Norton 1940
13. Quine, W.V.O.: Reference and modality. In: From a logical point of view, pp. 139–159. Cambridge, Mass.: Harvard University Press 1953
14. Quine, W.V.O.: Word and object. Cambridge, Mass: MIT Press 1960
15. Søndergaard, H., Sestoft, P.: Non-determinism in functional languages. Comput. J. (to appear)
16. Søndergaard, H., Sestoft, P.: Referential transparency and allied notions. University of Copenhagen, Denmark, DIKU Research Report 88/7, 1988
17. Stoy, J.E.: Denotational semantics. Cambridge, Mass.: MIT Press 1977
18. Strachey, C.: Fundamental concepts in programming languages. Int. Summer School in Computer Programming, Copenhagen, Denmark 1967 (unpublished)
19. Waite, W.M., Goos, G.: Compiler construction. Berlin Heidelberg New York: Springer 1984