Privé
Werkexemplaar

Some thoughts on nondeterminism

Maarten Fokkinga, 11 April 1986.

A collection of private opinions and thoughts on the desirability, necessity and managability of nondeterminism is presented. Together they constitute the slightly adapted and extended contents of two previous notes of mine (in Dutch). I am fully aware of the fact that all topics treated here are presumably treated much better in the literature; I welcome any specific or general references to the literature and any comments or critisism the reader may have. These notes have been written hastily on special request.

* * *

Desirability of nondeterminism

1. Avoiding overspecification.

Avoiding overspecification is advantageous for the process of program development. Hence the program notation should not force any overspecification. Sometimes one can think of a set of possibilities, each of which is as good as any other for the purpose under consideration. One may then express this

(i) by using sets of values, rather than one single value.

(ii) by using a choice operator with the various possibilities as the alternatives from which to choose, possibly each alternative guarded with some condition.

For the purpose of program development both the set of possibilities and the set of alternatives for the choice operator may be infinite! However, for executability it seems clear that such sets must be finite: a machine cannot make a choice out of an infinite number of possibilities in finite time (at least, not with a fair chance for each of them).

Note that this point of view gives rise to the notion of "semantic refinement", denoted by $\gg$; (besides "semantic equality" as usual denoted by $=$): $e = e'$ means that both expresions may deliver the same results (including nontermination), whereas $e \gg e'$ means that each possible result of e' (including nontermination) is also a possible result of e. Although nondeterminism may be of advantageous for the program development, the ultimate goal is of course to construct a program which is just a refinement of the specification: it needs only ~~to be able to~~ deliver one ~~possible~~ result.

2. Ease of programming.
Programming is made easier when the programmer may delegate "choosing the right alternative" to the machine, rather than programming the right

choice himself. Thus, some kind of choice operators have been proposed (called angelic choice and divergence avoiding ambiguity) which free the programmer of some work but burden the machine with "backtracking" (in a global or local way). Personally I do not like these constructs.

3. Necessity.
When one wants to describe concurrently operating (hardware) systems in a functional (or imperative!) way, one should be aware of influence of the operating speed of the various components. It seems quite impossible, however, to keep track of the time (at which messages are send and/or recieved); it may be possible in theory, but it seems quite unfeasible to do so in practise. The most promising thing to do seems to abstract from time (absolute and relative) altogether and instead let the speeds be left undetermined. Thus nondeterminism enters the picture; (not really as a necessity but rather as a promising approach). More precisely, the idea is to model the actual merge of two message streams (so that the order in the merged stream is completely determined by the respective arrival times) by a nondeterministic merge (which only respects

the ~~order~~ order within both streams separately). Moreover, the following observation of reality (which is what we want to describe functionally) leads to the requirement of fairness of the merge operator: the actual production of a message takes some time and at least 1 time_unit say, so the actual merge of message streams will produce in the merged stream each ~~item~~ message eventually (remember: the order in the result stream coincides with the arrival orders of the messages and these times are ever increasing by at least 1 time_unit).

We shall now show another way of avoiding some overspecification (other than "choosing among a set of possibilities"). It turns out that the fair nondeterministic merge is again the needed construct. Consider a ~~list~~ set A and the set B of all finite lists over A; mathematically we may define B as the smallest set satisfying either of the following equations

$$B = \{[]\} \cup \{a:b \mid a \in A,\ b \in B\}$$
$$B = \{[]\} \cup \{a:b \mid b \in B,\ a \in A\}$$
$$B = \{a:b \mid a \in A,\ b \in B\} \cup \{[]\}$$
$$B = \{a:b \mid b \in B,\ a \in A\} \cup \{[]\}$$

$$B = \{[]\} \cup \{[a] \mid a \in A\} \cup \{b ++ b' \mid b, b' \in B\}$$
$$\vdots$$

One may ~~be tempted to~~ represent sets by some enumeration of their elements and one may be tempted to expect that a simple transcription of the set-notation into the list-notation would give correct definitions for a representation of B:

$$B = [[]] ++ [a:b \mid a \leftarrow A;\ b \leftarrow B]$$
$$B = [[]] ++ [a:b \mid b \leftarrow B;\ a \leftarrow A]$$
$$B = [a:b \mid a \leftarrow A;\ b \leftarrow B] ++ [[]]$$
$$B = [a:b \mid b \leftarrow B;\ a \leftarrow A] ++ [[]]$$
$$B = [[]] ++ [[a] \mid a \leftarrow A] ++ [b ++ b' \mid b \leftarrow B;\ b' \leftarrow B]$$
$$\vdots$$

However most definitions will give an incorrect value to B; various possibilities are (for $A = [0,1]$):

$$B = [\,[],\ [0],\ [1],\ [0,0],\ [1,0],\ \cdots\ ]$$
$$B = [\,[],\ [0],\ [0,0],\ [0,0,0],\ \cdots\ ]$$
$$B = [\,[0,0,0,0,\cdots$$
$$B = [\,\bot\ = \bot$$
$$B = [\,[],[0],[1],[],[0],[1],[0],[0],[1],\ldots.\,]$$

And even neither of the definitions is correct if A is

infinite. However, all original mathematical definitions are correct and executable provided that

1. sets are made a programming language datatype, implemented by, say, some enumeration of their elements;
2. the union operator ∪ behaves as a fair merge of its two operands: given the two lists which represent sets it should merge them so that each element eventually appears in the result list.

Of course, in $\{\ldots \mid \ldots a \in A \ldots\}$ the epsilon is to be read as a generator ($\leftarrow$) rather than a membership test, and "$a \leftarrow A, b \leftarrow B$" stands for $(a,b) \leftarrow A * B$ where

$$\{\} * B = \{\}$$
$$(a:A) * B = \{(a,b) \mid b \leftarrow B\} \cup A * B$$

and similarly $b, b' \leftarrow B$ stands for $(b, b') \leftarrow B * B$.

Thus even outside the scope of concurrent systems the fair nondeterministic merge enters the picture as union. It should be noted, however, that union is completely deterministic as regards to sets: as long as the implementation of

representation of sets is not accessible by the programmer he cannot detect any nondeterminacy. (And of course, when there is an choice operator which takes any element of a set, or when sets can be converted to lists, then nondeterminism is clearly present!)

2. Problems with nondeterminism

The introduction of nondeterminism gives several problems, namely

(i) how to implement it;
(ii) how to give a denotational semantics for it;
(iii) how to reason about it in practical programs.

We can be very brief with respect to (i): the kind of nondeterminism we are in favour of is easily implemented. ((Remember also that we propose to use nondeterminism only*) to avoid overspecification, hence it is not necessary for an implementation to yield all possibilities.))
We shall pay no attention to (ii): we are convinced that in due time anything can be described in a denotational fashion, even gotos, semaphores

*) even abstracting from timing aspects is a way to avoid overspecification.

and angelic nondeterminism or what have you. Much more important is point (iii): how can we reason about the correctness of programs involving nondeterminism, and what further restrictions should we impose on nondeterminism in order that this reasoning becomes practical (while still achieving the goals for which nondeterminism has been introduced). We shall only consider this point below, and more specifically we shall only deal with "equational reasoning".

Proving a program P correct with respect to a specification s means, under equational reasoning, that s itself is a "program" (at a very high level of abstraction and very clearly satisfying the needs of the principal) and that one shows that $p = s$ (p is semantically equal to s) or at least $p \ll s$ (p is semantically a refinement of s). In order that this is practical the relation $=$ should obey the "laws of equality", namely

reflexivity: $x = x$

symmetry: $x = y$ implies $y = x$

transitivity: $x = y = z$ implies $x = z$

congruency: $x = y$ implies $\mathcal{C}[x] = \mathcal{C}[y]$ for any program context $\mathcal{C}[.]$

substitutivity: $x = y$ implies $\mathbb{P}(x) \Leftrightarrow \mathbb{P}(y)$ for any predicate $\mathbb{P}$ about programs (built up from $p = q$ and $p \gg q$ as elementary predicates)

Moreover, two further laws should give the $=$ relation the intended interpretation, namely

definition: for any program definition, lhs = rhs

"continuity": $x = y$ if $BT(x)$ and $BT(y)$ are the same (see below)

Actually, the definition-law is the beta-rule in a disguised form. In the "continuity"-law BT stands for Böhm Tree, that is: the (possibly infinite) head-normal form. For really infinite Böhm Trees the programmer should (presumably) use induction to prove equality of the Böhm Trees; induction for $=$ is thus avoided. One may however derive from this law specific induction laws such as Scott's Induction or Turner's Partial Object Induction. As an example of the use of the "continuity"-law, consider list X defined by $X = 1 : 1 : X$ and then try to show that $X = 1 : X$. (Note that by law (def) one always has an even number of ones at one side and an odd number at the other side! $BT(X) = 1:1:1:\ldots = 1:(1:\ldots)$.)

For semantic refinement it is not clear what laws to expect; the following seem unavoidable.

(refl') $x \gg x$

(trans') $x \gg y \gg z$ implies $x \gg z$

(congr') $x \gg y$ implies $C[x] \gg C[y]$

I cannot think of an obvious (true) analogue of the substitutively law: both $e \gg e' \Rightarrow (P(e) \Rightarrow P(e'))$ and $e \gg e' \Rightarrow (P(e') \Rightarrow P(e))$ are false in general.

Let us now consider nondeterminism more specifically. ~~For simplicity we only look at the demonic~~ Recall that the semantic equality relation is to mean that both sides have the same possible (determinate) results. Based on this intention, we conjecture that the laws given above are true. A formal proof requires that we give a formal definition of the intended semantic equality. We shall not attempt to give such a definition, let alone prove the conjecture; the details are not clear to us, at the moment. (I would welcome any hints or references to the literature where this is already done!)

Nevertheless we shall now argue that equational reasoning based on the above laws is not practical. To this end we consider the nondeterministic demonic choice, denoted by $\sqcap$, and operationally defined by

$x \sqcap y \to x$

$x \sqcap y \to y$

Define a function f by $f\,x = x + x$. Then we have

(each result of) $f\,0$ is even;

(each result of) $f\,1$ is even;

so we would like that

(each result of) $f\,(0 \sqcap 1)$ is even.

This kind of reasoning is quite similar to the reasoning about nondeterministic imperative programs, so as Dijkstra's guarded commands. Compare the valid inference rule

$$\frac{\{P\}\,S\,\{Q\} \qquad \{P\}\,S'\,\{Q\}}{\{P\}\;S \sqcap S'\;\{Q\}}$$

Unfortunately, the reasoning above about $f\,(0 \sqcap 1)$ is invalid, for we find $f\,(0 \sqcap 1) = 0 \sqcap 1 \sqcap 1 \sqcap 2$ (by operational reasoning) rather than only $f\,(0 \sqcap 1) = 0 \sqcap 2$.

The cause of the invalidity of the law

$$f\,(x \sqcap y) = f\,x \sqcap f\,y$$

in contrast to the validity of its imperative analog, is the duplication of the choice through the beta-reduction. One might conclude from this that

(i) for the sake of validity of $f(x \,[]\, y) = fx \,[]\, fy$ the evaluation should be in applicative order (call-by-value),

whereas

(ii) for the sake of convenience in programming normal order evaluation has proved to be beneficial.

Fortunately, there exists an evaluation method which combines the "good" properties of normal order and applicative order: lazy evaluation. This is so if "good" means "(optimal) with respect to number of reduction steps", but it now turns out that "good" may also mean: "with respect to the validity of $f(x \,[]\, y) = fx \,[]\, fy$". Indeed, under lazy evaluation argument evaluations are not duplicated; they are shared instead. Care should be taken to indicate this sharing property of the semantics in the axiomatization. Our proposal to do so reads as follows.

1. Sharing may be indicated in an expression by labelling a subexpression by a prefix superscript α (or $\beta, \gamma, \ldots$) and using these labels elsewhere as a subexpression. Thus, eg. $^{\alpha}(1:\alpha)$ denotes an infinite list of ones.
2. Within one definition or expression, different occurrences of a variable (with the same binding occurrence) are assumed to be shared by default. Thus, $fx = {}^{\alpha}x + \alpha$ may be written $fx = x+x$. (Also, the semantic equation $2*x = x+x$ really means $2 * {}^{\alpha}x = \alpha + \alpha$.)
3. The sharing may not get lost when instantiating a definition (applying beta-conversion). Thus $fx = x+x$ may be instantiated to $f\,{}^{\alpha}(0[]1) = \alpha + \alpha$ or $f\alpha = {}^{\alpha}(0[]1) + \alpha$ but not to $f(0[]1) = {}^{\beta}(0[]1) + (0[]1)$.

We have now a kind of nondeterministic choice that we consider practically managable; the law

(□ distr) $\quad f(x \,[]\, y) = fx \,[]\, fy$

is valid, its implementation is easy (use call-by-value, call-time-choice or the sharing facility of lazy evaluation), and the notational overhead of indicating sharing is very small indeed.

Remark. It would be much easier if the law

$$\text{(full D-distr)} \qquad e[x \,[\!]\, y] = e[x] \,[\!]\, e[y]$$

would be valid. To require so, would mean to require a compile-time choice rather than call-time choice. This kind of nondeterminism is too poor to be usefull for avoiding overspecification. Consider for example the specification "any number in -8..+8" which we might wish to write as

$$\begin{aligned} &\text{any } [1..9] \;-\; \text{any } [1..9] \\ &\text{where} \\ &\text{any } [x_1, \ldots, x_n] = x_1 \,[\!]\, x_2 \,[\!]\, \ldots \,[\!]\, x_n \end{aligned}$$

Admittedly, the specification could be written more clearly, but worse, it is wrong too because under compile-time choice the expression is semantically equal to zero! Rather than avoiding overspecification we have introduced it! (End of remark.)

It remains to be seen, however, whether the demonic call-time choice really satisfies our needs as regards to avoiding overspecification. The function

$$\text{any } (x{:}X) = x \,[\!]\, \text{any } X$$

is unsuitable to express "choose any member of an infinite list", because it has a possibility for divergence. One way out is to consider there for some kind of divergence avoiding choice; another way out is to introduce sets and nondeterministic pattern matching, so that we may define

$$\text{any } (x{:}X) = x$$

(where the pattern $x{:}X$ now means $\{x\} \cup X$.) This latter approach seems the most promising: it seems to me that both infinite sets and pattern matching are easily axiomatizable, and that these constructs exactly suit our needs!

3 The nondeterministic fair merge

Let us denote the nondeterministic fair merge by $\oplus$. Operationally, it is defined by

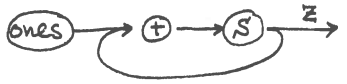
1. $x{:}X \oplus Y \rightarrow x{:}(X \oplus Y)$
2. $X \oplus y{:}Y \rightarrow y{:}(X \oplus Y)$
3. $X \oplus [] \rightarrow X$
4. $[] \oplus Y \rightarrow Y$
5. "rules 1 and 2 should get a fair chance, i.e. may not be by-passed forever at a certain occurrence of $\oplus$) if they are applicable"

It is tempting to axiomatize merge by

$$x{:}X \oplus Y \gg x{:}(X \oplus Y)$$
$$X \oplus (y{:}Y) \gg y{:}(X \oplus Y)$$
$$X \oplus [] = X$$
$$[] \oplus Y = Y$$

Clearly, what we then miss is the effect of the 5th operational rule: fairness. Remarkably, however, nothing is wrong with these axioms! It is the continuity axiom that gets invalid, or in other words even though $e \gg e_1 \gg e_2 \gg e_3 \gg \ldots$ and $e_1, e_2, e_3 \ldots$ converge to a well determined limit e_∞, one may in general not conclude that $e \gg e_\infty$. (Here, "the limit" is some Böhm tree.)

As an example of the way to reason about programs when nondeterminism and sharing is involved, we treat one version of the Brock-Ackermann anomaly. Consider the following system



$$ones = 1{:}\ ones$$
$$S(x{:}X) = (x+1){:}\ SX$$
$$Z = S(ones \oplus Z)$$

According to our calculus we find

$$\begin{aligned} Z &= \{\text{by definition}\} \\ &\quad S(ones \oplus Z) \\ &= \{\text{by notational convention}\} \\ &\quad \alpha(S(ones \oplus \alpha)) \\ &= \alpha(S((1{:}\ ones) \oplus \alpha)) \\ &= \alpha(S(1{:}(ones \oplus \alpha))) \\ &= \alpha(2 : S(ones \oplus \alpha)) \end{aligned}$$

$= \{$sharing of constants is not relevant$\}$

$2 : {}^{\alpha}(S\ (ones \oplus 2 : \alpha))$

which proves that anyway a "2" is the head of the list Z (assuming that the axioms are consistent). Any attempt to derive that some number exceeding 2 may be at the head of Z will fail, because one may not replace α by its definition as that would duplicate a shared expression. More formally, one should prove that the axioms, when used in this way, are consistent!; this is beyond the scope of this essay (and my capabilities?).

The next thing is to provide a nonrecursive closed form for Z and show it to be equal to Z semantically. I can not think of any better closed form than

(*) any list X such that there exists some $f : (1..) \to (1..)$ with

- $\forall i,j.\ i<j \Rightarrow f\,i < f\,j$
- $\forall i.\ X\hat{}i = 1 + X\hat{}f\,i$
- $\forall j.\ (\neg\exists i.\ f\,i = j) \Rightarrow X\hat{}j = 2$

(and concerning the fairness aspects

- $\forall K.\ \neg\exists N.\ \forall i>N.\ f\,i = i+K$
- $\forall i.\ f\,i$ is well defined $< \infty$.)

Here f is "the feed-back function": the i-th element of X, $X\hat{}i$, is fed back into the merge and then appears again, incremented by one, as the $(f\,i)$-th element, $X\hat{}f\,i$. So the constraints on f formalize, in turn, that

- in the feed-back the elements may not overtake each other;
- each cycle in the feed-back process increments the elements by one;
- elements not got by feed-back must be got from the ones-process via one increment;

(and

- it must not be the case that eventually only feed-back elements turn up;
- each element eventually is fed back).

We shall now sketch a proof that $Z \gg X$ and that these X are the only possible refinements of Z. To this end consider the values

(**) any list $X +\!\!+ {}^{\alpha}(Y +\!\!+ S\ (ones \oplus \alpha))$ such that there exists some $f : (1..\#X) \to (1..\#X+Y)$ satisfying

- $\forall i,j.\ i<j \Rightarrow f\,i < f\,j$

- $\forall i: (1..\#X).\quad X\hat{}i = 1 + (X{+\!\!+}Y)\hat{}fi$
- $\forall j: (1..\#X{+\!\!+}Y).\quad (\neg\exists i.\ fi = j) \Rightarrow (X{+\!\!+}Y)\hat{}j = 2$

Here, f plays the same role as before; part X of these lists have already been fed back and consumed by merge, part α is still waiting to be consumed, part Y of α are the elements already computed and explicitly available. We shall show by induction on $\#(X{+\!\!+}Y)$ that these values are all and only refinements of Z. First, for $X{+\!\!+}Y = []$ this amounts to showing that $Z = {}^{\alpha}(S\ (ones \oplus \alpha))$ which is true by definition. Second, consider such a list $X {+\!\!+} {}^{\alpha}(Y {+\!\!+} S\ (ones \oplus \alpha))$ with a suitable f; we derive

(a)
$$\begin{aligned}
& X {+\!\!+} {}^{\alpha}(Y {+\!\!+} S\ (ones \oplus \alpha)) && \text{with } f \text{ s.th. } \dots\\
&= X {+\!\!+} {}^{\alpha}(Y {+\!\!+} S\ (1{:}ones \oplus \alpha)) && "\\
&\gg X {+\!\!+} {}^{\alpha}(Y {+\!\!+} S\ (1{:}(ones \oplus \alpha))) && "\\
&= X {+\!\!+} {}^{\alpha}(Y {+\!\!+} 2{:}\ S\ (ones \oplus \alpha)) && "\\
&= X {+\!\!+} {}^{\alpha}((Y {+\!\!+} [2]) {+\!\!+} S\ (ones \oplus \alpha)) && \text{with } f' := f
\end{aligned}$$

(b)
$$\begin{aligned}
& X {+\!\!+} {}^{\alpha}(y{:}Y' {+\!\!+} S\ (ones \oplus \alpha))\\
&= X {+\!\!+} {}^{\beta y{:}\alpha}(Y' {+\!\!+} S\ (ones \oplus \beta{:}\alpha))\\
&= \{\text{sharing of constants } y \text{ is not relevant}\}\\
& X {+\!\!+} y{:}\,{}^{\alpha}(Y' {+\!\!+} S\ (ones \oplus y{:}\alpha))
\end{aligned}$$

$$\begin{aligned}
&\gg (X {+\!\!+} [y]) {+\!\!+} {}^{\alpha}(Y' {+\!\!+} S\ (y{:}(ones \oplus \alpha)))\\
&= (X {+\!\!+} [y]) {+\!\!+} {}^{\alpha}(Y' {+\!\!+} (y{+}1){:}\ S\ (ones \oplus \alpha))\\
&= (X {+\!\!+} [y]) {+\!\!+} {}^{\alpha}((Y' {+\!\!+} [y{+}1]) {+\!\!+} S\ (ones \oplus \alpha))\\
&\quad \text{where } f' := f \cup (i \mapsto j)\\
&\qquad \underline{\text{where}}\ \ i = \#(X {+\!\!+} [y])\\
&\qquad\qquad j = \#(X {+\!\!+} [y] {+\!\!+} Y' {+\!\!+} [y{+}1])
\end{aligned}$$

In (a) and (b) we have employed the two possible refinement axioms for $\oplus$; no other refinement axioms are applicable, thus (a) and (b) list all possibilities. (Actually, what we have done is proving by induction on n that up to depth n Z ~~equals~~ refines to any of the Böhm Trees (= infinite head-normal-forms characterized by (**).)

Without the fairness taking into account we may now conclude that Z ~~equals~~ refines to any of those Böhm Trees completely -- instead of only to depth n --. But in the context of fairness this conclusion is invalid. We shall not attempt to deal with fairness while keeping the spirit of our equational reasoning.

We conclude our treatment of the fair merge with two conjectures, although they might already be folklore --I don't know--. First, Barendregt has proved, in his famous book about the λ-calculus, that the ~~to~~ divergence avoiding choice is not expressible in the λ-calculus, although it is definitely algorithmically implementable; see

his book, Section 14.4. In the same vein we state the following conjecture.

Conjecture Fair merge can not be expressed in the lambda calculus.

Secondly, we conjecture that any fair merge is necessarily nondeterministic.

Conjecture Let merge be such that (merge X Y) = some enumeration (list) of all elements of X and Y, even for partial lists X and Y. (A list is partial if it has the form $x_1 : x_2 : \ldots : x_n : \bot$ for some n.) Then there exist expressions E1, E1', E2, E2' such that

E1 and E1' are semantically equal, E1 = E1',
E2 and E2' are semantically equal, E2 = E2',
but nevertheless
(merge E1 E2) ≠ (merge E1' E2')

That is, semantically merge is not a function: it is nondeterministic. (Operationally it means that the evaluator must necessarily take into account the syntactic form of the computations/reductions of both arguments of merge.)

I don't think that the Brock_Ackermann anomaly proves this conjecture. Rather I think that a proof will be based on computability theory, and will run as follows.

By the assumption of the conjecture we have

merge [x] ⊥ = [x ...

By computability theory, any algorithmic process is monotone, so that from ⊥ ⊑ [y] we find

merge [x] [y] ∃ merge [x] ⊥ = [x ...

So merge [x] [y] ≫ [x, y ...

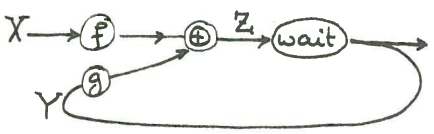
Similarly, merge [x] [y] ≫ [y, x ---

Hence there are two different possible results, that is merge is necessarily nondeterministic.

This sketch is far from complete, and far from convincing!

Addendum Another variant of the B_A_ anomaly.

Consider the system



f (x:X) = 0: f X

g (y:Y) = 1: g Y

wait [] = []

wait (x:[]) = (x: [])

wait (x: y: Z) = x: y: wait Z

Y_0 = wait (f X ⊕ g Y_0) ... (*)

It is easy to show that for all finite or infinite total lists Z (i.e not ending in ⊥),

Z = wait Z

It is tempting to apply the congruency law "x=y ⇒ C[x] = C[y]" (in all contexts) and conclude that wait may be replaced by the identity function. This would give the erroneous claim that

Y_0 = f X_0 ⊕ g Y_0 cfr (*).

The claim is erroneous because

if $X_0 = []$ then $Y_0 = \bot = f\,X_0 \oplus g\,Y_0$
if $X_0 = [x]$ then $Y_0 = \bot \neq 0 : (\eta : \alpha) = f\,X_0 \oplus g\,Y_0$
if $\#X_0 = n \geq 2$ then
$Y_0 = 0 : 0$: fair merge of $n-2$ zeros and infinite ones
$\neq f\,X_0 \oplus g\,Y_0 = 0$: fair merge of $n-1$ zeros and inf. ones

Fortunately, the congruency cannot be applied (so that the erroneous equality disappears) because the law "$x = y \Rightarrow e[x] = e[y]$" requires to show that, in this case, $Z = \text{wait}\,Z$ for <u>all</u> lists Z, not just the total ones but also the partial ones. Clearly, we cannot show this for $Z = z : \bot$, for we find

$$Z = z : \bot \neq \bot = \text{wait}\,(z : \bot) = \text{wait}\,Z\ .$$

(<u>End of addendum</u>.)

<u>Addendum</u>: guarded alternatives

Here is a sketchy attempt to formulate an inference rule for guarded alternatives (with lazy evaluation or rather environmental transparancy: all occurrences of the same variable denote a single (nondeterministic) value).

$$\text{(i)}\quad e_1 \vee e_2 = \text{true}$$

$$\text{(ii)}\quad \frac{e_1 \gg \text{true} \supset \mathbb{P}(e_1') \;\&\; e_2 \gg \text{true} \supset \mathbb{P}(e_2')}{\mathbb{P}\big((e_1 \to e_1') \,[\!]\, (e_2 \to e_2')\big)}$$

Note that sharing takes places in the entire first premiss and separately from it also in the entire second premiss. Premiss (i) demands that anyway one of the guards must evaluate to true. The condition $e_1 \gg \text{true}$ says "some evaluation of e_1 must ~~may~~ yield ~~false~~ true" but doesnot exclude that some evaluation of e_1 may yield false; divergence is impossible because of premiss (i).

Now one may prove $\mathbb{P}(x, y, \max\,x\,y)$ where

$\max\,x\,y = (x \leq y \to y \;[\!]\; y \leq x \to x)$ (note: sharing!)
$\mathbb{P}(x, y, z) = (x \leq z \wedge y \leq z \wedge (x \equiv z \vee y \equiv z)) = \text{true}$ (sharing!)

Hence $\mathbb{P}(\alpha(0[\!]1), \beta(0[\!]1), \max\,\alpha\,\beta)$ is true as well. (<u>End of Addendum</u>)