J. Functional Programming **7** (1): 73–101, January 1997 © 1997 Cambridge University Press

First-order functional languages and intensional logic

P. RONDOGIANNIS and W. W. WADGE
Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6
(*e-mail:* {prondo,wwadge}@lucy.uvic.ca)

Abstract

The purpose of this paper is to demonstrate that first-order functional programs can be transformed into intensional programs of nullary variables, in a semantics preserving way. On the foundational side, the goal of our study is to bring new insights and a better understanding of the nature of functional languages. From a practical point of view, our investigation provides a formal basis for the tagging mechanism that is used in the implementation of first-order functional languages on dataflow machines.

Capsule Review

Intensional programming languages such as Lucid treat certain parameters – such as time – implicitly. An intensional interpretation called eduction is then given to programs which introduces the implicit parameters more explicitly. Although intensional languages are clearly declarative, their precise connection to functional languages is worth investigating. This paper describes a translation from first-order functional programs into intensional programs in which the formal parameters to functions are eliminated, leaving them with 'nullary variables'. This approach is interesting as an implementation method for first-order functional languages, and also demonstrates some useful connections to dataflow architectures.

1 Introduction

In 1984 A. Yaghi, in his PhD dissertation (Yaghi, 1984), presented for the first time a transformation algorithm from functional to intensional programs. Motivated by Montague's intensional logic (Thomason, 1974; Dowty *et al.*, 1981), Yaghi first defined a simple intensional programming language whose syntax only allowed nullary variable definitions. He then proposed an algorithm for transforming first-order functional programs into programs of this language.

The practical significance of his work is that the resulting intensional programs can be implemented (in fact, evaluated) using *eduction*, a simple tagged (dynamic) demand-driven dataflow model which does not require closures, pointers or heaps. In fact, since its inception the algorithm has been used as the core implementation technique for a number of first-order languages (Du and Wadge, 1990a, 1990b; Faustini *et al.*, 1991). Moreover, the algorithm is closely related to the *colouring* technique that has been extensively used for implementing first-order functional programs on dataflow architectures (Kirkham *et al.*, 1985; Arvind and Nikhil, 1990).

Yaghi's work, although ground-breaking, was incomplete in the sense that it lacked a precise formulation, did not include a correctness proof, and applied only to first-order programs. In this paper we resolve the above first two issues, and lay the basis for the extension to higher-order programs, the subject of a separate forthcoming paper (Rondogiannis and Wadge, 1996).

The first sections of the paper concentrate on giving a self-contained, intuitive introduction to the research area under consideration and to its main underlying concepts. The overall structure of the rest of the paper has as follows: section 2 introduces intensional logic and intensional programming languages. The computational model of *eduction* that can be used in order to interpret intensional languages, is presented in section 3. Section 4 introduces Yaghi's transformation algorithm. Sections 5, 6 and 7 present mathematical preliminaries, and introduce the functional language $FOFL$ and the intensional language $NVIL$ that are the source and target languages of the transformation algorithm. In sections 8 and 9 we give a precise formulation of the transformation algorithm, and illustrate it by examples. The correctness proof for the transformation is given in section 10, and the paper concludes by discussing the connections and implications of our work in the area of dataflow computation as well as implementation issues.

2 Intensional logic and intensional languages

Intensional logic (Thomason, 1974; Dowty *et al.*, 1981; van Benthem, 1988) is a mathematical formal system for describing entities whose values depend upon implicit contexts. The need for such a logic became apparent when the study of natural languages was undertaken by linguists and logicians. Consider for example the following natural language expression (Thomason, 1974):

Iceland is covered with a glacier

The truth value of this expression varies according to an implicit time context: at the present time the above expression is false; however, there are times in the past when the expression was true. Therefore, the semantic value of the expression is really a function from time-points to truth values.

One can easily think of other expressions whose truth value depends on more than one coordinates, such as *spatial position*, *speaker*, or *audience*. In general, the semantic value of an expression is a function from a set of contexts (also called *possible worlds*) to a set of values. This function is called the *intension* of the expression. The value of the intension at a particular context is called the *extension* of the expression at that particular context.

Consider now the following two expressions:

The exchange rate of the Canadian dollar per US dollar
Yesterday's exchange rate of the Canadian dollar per US dollar

Table 1. *The intension of the first expression*

Date	⋯	8/2/94	8/3/94	8/4/94	⋯
Rate	⋯	1.378	1.379	1.380	⋯

Table 2. *The intension of the second expression*

Date	⋯	8/3/94	8/4/94	8/5/94	⋯
Rate	⋯	1.378	1.379	1.380	⋯

The intension of the first expression is a function which, given a date, returns the exchange rate on that day. This intension can be visualized as shown in Table 1.

On the other hand, the intension of the second expression is a function which, given a date, returns the exchange rate on the previous day. This intension is represented in Table 2.

Obviously, there exists a relationship between the two intensions. In fact, the word 'Yesterday's' in the second expression above can be thought of as an operator that transforms the intension of the first expression into the intension of the second; it simply increases all the dates by one day.

The above examples illustrate how the meaning of some natural language expressions can be captured using intensions and context switching operators (like 'Yesterday's'). Notice that these operators allow information from different contexts to be compared and combined without explicit context manipulation. This is the main reason that intensional logic has proved to be an effective tool in the study of the semantics of natural languages.

Intensional programming is a programming paradigm that is based on intensional logic. The distinguishing characteristic of intensional languages is that they have context switching operators. One such intensional language is Lucid (Wadge and Ashcroft, 1985), in which the value of an expression depends on a hidden time parameter. The Lucid context-switching operators are **first**, **next** and **fby**.

The time domain is the set of natural numbers; therefore, the value of a Lucid expression is a *stream* (infinite sequence) of ordinary data values. In particular, the value of a Lucid constant is a constant stream.

The statements of a Lucid program are equations defining individual and function variables, required to be true at every context (time point). Ordinary data operations (such as +, * and **if-then-else**) are referentially transparent. This means, for example, that the value of $\mathbf{x} + \mathbf{y}$ at timepoint t is the sum of the values of **x** and **y** at the same timepoint t.

The basic context-switching operators are **first**, **next** and **fby**. The operator **first** switches us to time point 0, **next** takes us from t to $t + 1$. The operator **fby** takes us back from $t+1$ to t (giving us the value of its second operand at that point) or from 0 to 0, giving us the value of its first operand.

Let x and y be streams. Then, the above ideas are formalized by the following equations:

$$\begin{array}{lcl} (x+y)_t & = & x_t + y_t \\ first(x)_t & = & x_0 \\ next(x)_t & = & x_{t+1} \\ (x\ fby\ y)_t & = & \begin{cases} x_0 & \text{if } t=0 \\ y_{t-1} & \text{if } t>0 \end{cases} \end{array}$$

These equations constitute what we call the *indexical semantics* of the operations; they define the extensions of the result of an operation in terms of the extensions of the operands.

A Lucid intension x can be thought of as a value which is varying over time, for example, a loop variable in an iterative computation. Thus x_0 is the initial value of x, x_1 is the value after the first iteration step, and x_2, x_3, x_4 etc are the values after subsequent steps. (It is also possible to think of intensions as streams in a pipeline dataflow model.)

The **fby** operator allows us to express many iterative algorithms concisely; the first operand of the **fby** gives the initial value, and the second operand specifies the way in which each succeeding value is determined by the current value. For example, the following program computes the stream $\langle 1,1,2,3,5,\ldots\rangle$ of all Fibonacci numbers:

```
result  ≐  fib
fib     ≐  1 fby (fib+g)
g       ≐  0 fby fib
```

Notice that the above Lucid program is simply a set of nullary variable definitions (that is, it does not use any user-defined functions). In the next section we present a very simple technique for interpreting Lucid programs such as the above.

3 The computational model of eduction

Lucid programs (such as those given in the last section) are usually implemented using a computational model known as eduction (Wadge and Ashcroft, 1985; Faustini and Wadge, 1987). An eductive computation proceeds by propagating demands for the extensions (values) of specific variables at specific contexts. Demands for a variable are converted into demands for its defining expression, and these generate demands for subexpressions and, eventually, for the variables occurring in these expressions – not necessarily at the same contexts.

We illustrate the main idea of eduction with an example. Suppose we want to calculate the second Fibonacci number, as defined by the program given above. To do so, we demand the value of `result` at time 2. This generates a demand for `fib` at time 2, which creates a demand for `(1 fby (fib+g))` at time 2. But now, according to the semantics of **fby**, this will generate a demand for `(fib+g)` at time 1. The overall execution by an eductive evaluator $EVAL$, is given in Figure 1. The ways in which these demands are propagated through the operators, and the ways

$$
\begin{aligned}
& EVAL(\texttt{result}, 2) = \\
= \quad & EVAL(\texttt{fib}, 2) \\
= \quad & EVAL(\texttt{(1 fby (fib+g))}, 2) \\
= \quad & EVAL(\texttt{(fib+g)}, 1) \\
= \quad & EVAL(\texttt{fib}, 1) + EVAL(\texttt{g}, 1) \\
= \quad & EVAL(\texttt{(1 fby (fib+g))}, 1) + EVAL(\texttt{(0 fby fib)}, 1) \\
= \quad & EVAL(\texttt{(fib+g)}, 0) + EVAL(\texttt{fib}, 0) \\
= \quad & EVAL(\texttt{fib}, 0) + EVAL(\texttt{g}, 0) + EVAL(\texttt{(1 fby (fib+g))}, 0) \\
= \quad & EVAL(\texttt{(1 fby (fib+g))}, 0) + EVAL(\texttt{(0 fby fib)}, 0) + 1 \\
= \quad & 1 + 0 + 1 \\
= \quad & 2
\end{aligned}
$$

Fig. 1. Execution of intensional code.

in which the resulting values are combined by the operators, are determined by the intensional semantics of these operators.

The eductive approach can clearly be extended to any family of data structures whose members are (or can be seen as) collections of simple data objects indexed by simple values. For example, we could add arrays to the language by introducing an extra *space* dimension. We could then write programs with streams of arrays and evaluate them using essentially the same procedure. The only difference is that the $EVAL$ function has an extra parameter, and the tags on items in the warehouse have an extra field. Also, we might need to perform a dimensionality analysis on the program to identify variables that are time or space invariant (so as to avoid duplication of computations).

The defining characteristic of eduction is that it computes the value of expressions with respect to contexts. The *dataflow* class of machine architectures (Kirkham *et al.*, 1985; Arvind and Nikhil, 1990) is based on exactly this idea – tagged tokens are really extensions, labelled (tagged) with their contexts. In other words, dataflow machines should make ideal platforms on which eduction is implemented (this point is further elaborated in section 11). This suggests the triangle given in Figure 2, in which:

- Intensional logic provides the language framework in which programs are written or compiled to.
- Eduction provides the conceptual execution model for implementing the intensional programs.
- Dataflow architectures provide the appropriate hardware on which eduction can be executed in an efficient way.

The above description suggests that we can implement a programming language on a dataflow architecture by first compiling source programs into semantically equivalent intensional ones (that contain only nullary variable definitions). The next section discusses how this can be done for the case of first-order functional languages. In other words, it describes how first-order functional programs can be transformed into equivalent, eductible Lucid-like programs *without* any user-defined functions.

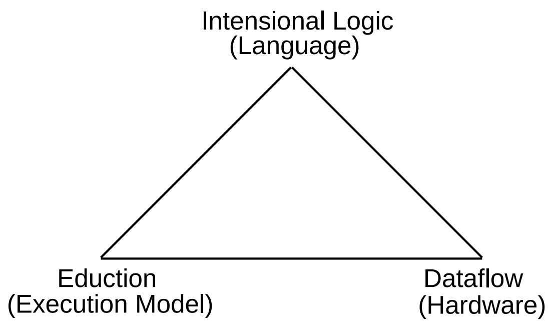


Fig. 2. Intensional logic, eduction and dataflow.

4 Intensional logic and functional languages

Yaghi's PhD dissertation (Yaghi, 1994) was the first work to establish a relationship between intensional and functional languages. In this dissertation, Yaghi used intensional logic to formalize an implementation technique for first-order functional languages that was originally invented by C. Ostrum at the University of Waterloo. Yaghi first defined a simple intensional programming language that only supported nullary variable definitions. He then showed that Ostrum's implementation could be understood as involving an implicit translation of the source functional program into a function-free program of this intensional language.

The main idea behind Yaghi's work is that functions (and their formal parameters) can be understood as intensions, varying over the space of calls or invocations. The extension of a formal parameter at one of these contexts is (the value of) the appropriate actual parameter, and the value of the function itself at a context is the value returned by the call in question.

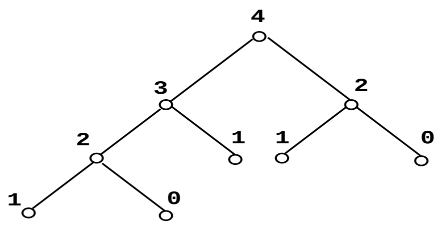
We can illustrate his ideas with an example. Consider the following first-order functional program that computes the fourth Fibonacci number:

```
result  ≐  fib(4)
fib(n)  ≐  if (n<2) then 1 else fib(n-1)+fib(n-2)
```

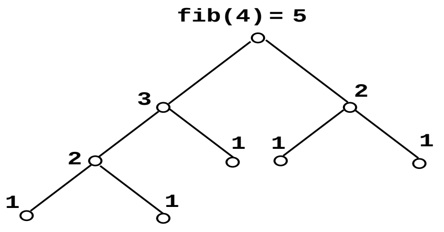
To compute `fib(4)`, we need to know `fib(3)` and `fib(2)`. Similarly, `fib(3)` requires `fib(2)` and `fib(1)`, and so on. Therefore, one can actually think of the formal parameter `n` as being a labelled tree of the form shown in Figure 3(a).

Similarly, the function `fib` can be thought of as a labelled tree that has been created by 'consulting' the tree for `n`. Figure 3(b) illustrates the corresponding tree. The bottom labels of the tree for `fib` are all equal to 1, because this is the value that `fib` takes when the corresponding value of `n` is less than 2. As we move up the tree for `fib`, the label on each node is formed by adding the values of the right and left children of the node. The initial program can be transformed into a new one that reflects the above ideas:

$$\begin{array}{lcl} \texttt{result} & \doteq & \texttt{call}_0\texttt{(fib)} \\ \texttt{fib} & \doteq & \texttt{if (n<2) then 1 else call}_1\texttt{(fib)+call}_2\texttt{(fib)} \\ \texttt{n} & \doteq & \texttt{actuals(4,n-1,n-2)} \end{array}$$



(a)



(b)

Fig. 3. (a) Tree for the parameter `n`; (b) Tree for the function `fib`.

Notice that the above program is a Lucid-like one, the only difference being that it is manipulating tree intensions and not just stream ones. The definition of `n` in terms of **actuals** expresses the fact that `n` is a tree with root labelled 4; the root of the left subtree is equal to the current root minus one, and the root of the right subtree is equal to the current root minus two. Clearly, one can proceed in this way and create the whole tree for `n` as given in Figure 3(a). The operators $\mathbf{call}_i$ are used to create the tree for `fib`. The definition for `fib` can be read as follows:

"The value of a node of the tree for `fib` is equal to 1 if the value of the corresponding node of the tree for `n` is less than 2; otherwise, it is equal to the sum of the values found at the roots of the left and right subtrees of the node."

Informally speaking, $\mathbf{call}_1$ selects the root of the left subtree of the current node of `fib`, while $\mathbf{call}_2$ the root of the right subtree. The operator $\mathbf{call}_0$ returns the root of the tree for `fib`.

The above description presents at an intuitive level the relationship between first-order functional programs and intensional ones. Yaghi's algorithm for performing the transformation in a systematic way can be summarized as follows:

1. Let **f** be a function defined in the source functional program. Number the textual occurrences of calls to **f** in the program, starting at 0 (including calls in the body of the definition of **f**).
2. Replace the *i*th call of **f** in the program by $\mathbf{call}_i(\mathbf{f})$. Remove the formal parameters from the definition of **f**, so that **f** is defined as an ordinary individual variable.

3. Introduce a new definition for each formal parameter of **f**. The right-hand side of the definition is the operator **actuals** applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered.

To illustrate the algorithm, consider the following simple first-order functional program:

```
result  ≐  f(4)+f(5)
f(x)    ≐  g(x+1)
g(y)    ≐  y
```

Yaghi's translation algorithm produces the following intensional program:

```
result  ≐  call₀(f)+call₁(f)
f       ≐  call₀(g)
g       ≐  y
x       ≐  actuals(4,5)
y       ≐  actuals(x+1)
```

The eductive interpretation of the latter program can be understood in terms of the trees described above. A node in a tree is determined by a finite list of natural numbers, each number in the list specifying which child branch to follow as we descend from the root. (By contrast, an element in a Lucid stream can be identified by a single natural number.) In other words, the set of possible worlds (contexts) for this intensional language is the set of lists of natural numbers. (If the source program already has, say, temporal operators, then they are retained. The tree space is added as an *extra* dimension, in much the same way that a linear space dimension with its own operators could be added.)

An execution model is established by considering the **call**$_i$ and **actuals** as context switching operators. Intuitively, **call**$_i$ augments a list w by prefixing it with i. On the other hand, **actuals** takes the head i of a list, and uses it to select its ith argument. More formally, given intensions $a, a_1, \ldots, a_n$, and letting ':' denote the consing operation on lists, the semantic equations as introduced in Yaghi (1984) are:

$$\begin{aligned}(call_i(a))(w) &= a(i : w)\\ (actuals(a_0, \ldots, a_{n-1}))(i : w) &= (a_i)(w)\end{aligned}$$

Following the above semantic rules, the intensional program obtained above can be interpreted as shown in Figure 4.

It is the purpose of this paper to formalize the ideas presented in the previous sections, precisely define Yaghi's algorithm, and, most importantly, demonstrate its correctness.

5 Mathematical notation

The set of natural numbers is denoted by ω. The set of functions from A to B is denoted by $A \to B$ or B^A. For simplicity, in certain cases we use the subscript notation for function application, writing for example f_a instead of $f(a)$.

$$
\begin{array}{rl}
& EVAL(\texttt{call}_0\texttt{(f)+call}_1\texttt{(f)},[\,]) \\
= & EVAL(\texttt{call}_0\texttt{(f)},[\,]) + EVAL(\texttt{call}_1\texttt{(f)},[\,]) \\
= & EVAL(\texttt{f},[0]) + EVAL(\texttt{f},[1]) \\
= & EVAL(\texttt{call}_0\texttt{(g)},[0]) + EVAL(\texttt{call}_0\texttt{(g)},[1]) \\
= & EVAL(\texttt{g},[0,0]) + EVAL(\texttt{g},[0,1]) \\
= & EVAL(\texttt{y},[0,0]) + EVAL(\texttt{y},[0,1]) \\
= & EVAL(\texttt{actuals(x+1)},[0,0]) + EVAL(\texttt{actuals(x+1)},[0,1]) \\
= & EVAL(\texttt{x+1},[0]) + EVAL(\texttt{x+1},[1]) \\
= & EVAL(\texttt{x},[0]) + EVAL(\texttt{1},[0]) + EVAL(\texttt{x},[1]) + EVAL(\texttt{1},[1]) \\
= & EVAL(\texttt{x},[0]) + 1 + EVAL(\texttt{x},[1]) + 1 \\
= & EVAL(\texttt{actuals(4,5)},[0]) + 1 + EVAL(\texttt{actuals(4,5)},[1]) + 1 \\
= & EVAL(\texttt{4},[\,]) + 1 + EVAL(\texttt{5},[\,]) + 1 \\
= & 4+1+5+1 \\
= & 11
\end{array}
$$

Fig. 4. Execution of intensional code.

Given two sets I and S, an I-indexed sequence is any function $s : I \to S$, and is denoted by $\langle s_i \rangle_{i \in I}$. The set I is called the index set of s. When $I = \{0, \dots, n-1\}$, we usually denote s by $\langle s_0, s_1, \dots, s_{n-1} \rangle$. Notice that sequences are simply functions and will therefore often be represented in the usual way (i.e. as sets of ordered pairs). We adopt the following general notion of set product: if I is any set and A_i is a set for every $i \in I$ then

$$\prod_{i \in I} A_i = \{f : I \to \bigcup_{i \in I} A_i \mid \forall i \in I, f(i) \in A_i\}.$$

The perturbation of a function with respect to another function, is defined as follows:

Definition 5.1
Let $f : A \to B$ and $g : S \to B$, where $S \subseteq A$. Then, the perturbation $f \oplus g$ *of f with respect to g is defined as:*

$$(f \oplus g)(x) = \begin{cases} g(x) & \textit{if } x \in S \\ f(x) & \textit{otherwise.} \end{cases}$$

Given a function $g = \{\langle x_0, b_0 \rangle, \dots, \langle x_{n-1}, b_{n-1} \rangle\}$, we often write $f[x_0/b_0, \dots, x_{n-1}/b_{n-1}]$ instead of $f \oplus g$.

Let L be a given set. We write *List*(L) for the set of lists of elements of L. The usual list operations *head*, *tail* and *cons* are adopted. The infix notation ':' will often be used instead of *cons*.

In the rest of this paper, we assume familiarity with the basic notions of domain theory and denotational semantics (Stoy, 1977; Tennent, 1991; Gunter, 1992). Given a domain D, the partial order and the least element of D are represented by $\sqsubseteq_D$ and $\perp_D$, respectively. The subscript D will often be omitted when it is obvious. If A, B are domains, $[A \to B]$ is the set of all continuous functions from A to B.

Finally, we adopt certain typographic conventions which are outlined below. Elements of the object language, such as for example the code of programs or function names in such programs, are represented using typewriter font (e.g. `f`, `x`, ...).

Elements of the meta-language are divided in two classes: those that are used to represent usual mathematical objects such as functions, sets, and so on, and for which we adopt the italics and the calligraphic fonts (e.g. $f, x, \mathscr{E}, \mathscr{A}, \ldots$), and those that are used to talk about the syntax of the object language, for which we adopt the boldface font (e.g. $\mathbf{f}, \mathbf{x}, \mathbf{P}, \mathbf{E}, \ldots$).

6 The First-Order Functional Language (FOFL)

In this section, we define the syntax and denotational semantics of the First-Order Functional Language $FOFL$. In the following, $FOFL$ will also be referred as an *extensional* language, to distinguish it from the *intensional* language $NVIL$ that will be defined later on in this paper.

We assume the existence of a set Σ of constant symbols, whose elements are denoted by $\mathbf{c}$. We write Σ_n for the subset of Σ of constants of arity n. We also assume the availability of a set Var of variable symbols of various arities. We will use $\mathbf{f}$ and $\mathbf{g}$ to denote elements of Var. In particular, we often use $\mathbf{x}, \mathbf{y}, \mathbf{x}_0, \mathbf{y}_0$, and so on, to denote nullary variables. We write Var_n for the subset of Var of variables of arity n.

Definition 6.1
The syntax of the first-order functional language $FOFL$ is recursively defined by the following rules, in which $\mathbf{E}, \mathbf{E}_i$ *denote* expressions*,* $\mathbf{F}, \mathbf{F}_i$ *denote* definitions *and* $\mathbf{P}$ *denotes a* program*:*

$$\begin{array}{lcl}
\mathbf{E} & ::= & \mathbf{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}),\ \mathbf{c} \in \Sigma_n \\
 & | & \mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}),\ \mathbf{f} \in Var_n \\
\mathbf{F} & ::= & (\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{E}),\ \mathbf{f} \in Var_n, \mathbf{x}_0, \ldots, \mathbf{x}_{n-1} \in Var_0 \\
\mathbf{P} & ::= & \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}.
\end{array}$$

Given a definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{E}$, the variables $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ are the *formal parameters* or *formals* of $\mathbf{f}$, and $\mathbf{E}$ is the *defining expression* or the *body* of $\mathbf{f}$. We often use $\mathbf{B}$ instead of $\mathbf{E}$ to denote the body of a function. The set of variables defined in a program $\mathbf{P}$ is denoted by $func(\mathbf{P})$.

Definition 6.2
Let $\mathbf{P} = \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}$ *be a program. Then the following assumptions are adopted:*

1. *Exactly one of the* $\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}$ *defines the nullary variable* **result***, which does not appear in the body of any of the definitions in* $\mathbf{P}$.
2. *Every variable symbol in* $\mathbf{P}$ *is defined or appears as a formal parameter in a function definition, at most once in the whole program.*
3. *The formal parameters of a function definition in* $\mathbf{P}$ *can only appear in the body of that definition.*
4. *The only variables that can appear in* $\mathbf{P}$ *are the ones defined in* $\mathbf{P}$ *and their formal parameters.*

Let D be a domain. Then, the semantics of constant symbols of $FOFL$ with respect to D, are obtained by a given interpretation function $\mathscr{C}$, which assigns to every n-ary constant $\mathbf{c}$ a function in $[D^n \to D]$.

Let Env be the set of *extensional environments* defined by $Env = \prod_{n\in\omega} Var_n \to [D^n \to D]$. Then, the semantics of $FOFL$ are defined as follows:

Definition 6.3
The semantics of expressions of FOFL with respect to $u \in Env$, are recursively defined as follows:

$$[\![\mathbf{c}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})]\!](u) = \mathscr{C}(\mathbf{c})([\![\mathbf{E}_0]\!](u),\dots,[\![\mathbf{E}_{n-1}]\!](u))$$
$$[\![\mathbf{f}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})]\!](u) = u(\mathbf{f})([\![\mathbf{E}_0]\!](u),\dots,[\![\mathbf{E}_{n-1}]\!](u)).$$

Theorem 6.1
(Tennent, 1991, p. 97) For all expressions $\mathbf{E}$, $[\![\mathbf{E}]\!]$ is continuous and therefore monotonic.

Definition 6.4
The semantics of the program $\mathbf{P} = \{\mathbf{F}_0,\dots,\mathbf{F}_{n-1}\}$ of FOFL with respect to $u \in Env$, is defined as $\widetilde{u}(\mathbf{result})$, where $\widetilde{u}$ is the least *environment such that:*

1. *For every $\mathbf{f} \in Var$ with $\mathbf{f} \notin func(\mathbf{P})$, $\widetilde{u}(\mathbf{f}) = u(\mathbf{f})$.*
2. *For every $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, and for all $d_0 \in D,\dots,d_{n-1} \in D$,*
   $\widetilde{u}(\mathbf{f})(d_0,\dots,d_{n-1}) = [\![\mathbf{B}]\!](\widetilde{u}[\mathbf{x}_0/d_0,\dots,\mathbf{x}_{n-1}/d_{n-1}])$.

The above definition does not specify how the least environment $\widetilde{u}$ can be constructed. The following theorem suggests that $\widetilde{u}$ is the least upper bound of a chain of environments:

Theorem 6.2
(Tennent, 1991, p. 96) Let $\mathbf{P}$ and $\widetilde{u}$ be as in Definition 6.4. Then, $\widetilde{u}$ is the least upper bound of the environments $\widetilde{u}_k$, $k \in \omega$, which for every definition $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, and for all $d_0 \in D,\dots,d_{n-1} \in D$, are defined as follows:

$$\begin{array}{rcl}\widetilde{u}_0(\mathbf{f})(d_0,\dots,d_{n-1}) & = & \perp_D \\ \widetilde{u}_{k+1}(\mathbf{f})(d_0,\dots,d_{n-1}) & = & [\![\mathbf{B}]\!](\widetilde{u}_k[\mathbf{x}_0/d_0,\dots,\mathbf{x}_{n-1}/d_{n-1}]).\end{array}$$

Moreover, for every $k \in \omega$, $\widetilde{u}_k(\mathbf{f}) \sqsubseteq \widetilde{u}_{k+1}(\mathbf{f})$, and if $n \geq 1$ then $\widetilde{u}_k(\mathbf{f})$ is a continuous and therefore monotonic function.

The following lemma is a direct consequence of the above theorem:

Lemma 6.1
Let $\mathbf{P}$ and $\widetilde{u}$ be as in Definition 6.4. Then, for every $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, and for all $d_0 \in D,\dots,d_{n-1} \in D$

$$\widetilde{u}_k(\mathbf{f})(d_0,\dots,d_{n-1}) \quad \sqsubseteq \quad [\![\mathbf{B}]\!](\widetilde{u}_k[\mathbf{x}_0/d_0,\dots,\mathbf{x}_{n-1}/d_{n-1}]).$$

Notice that the semantics of programs of $FOFL$ have been defined with respect to an initial environment u. Recall now that the programs that we are considering do not contain occurrences of 'outside' variables (Definition 6.2, assumption 4). For this reason, in the following we will assume that the initial environment assigns the bottom value (of the appropriate type) to every variable in Var, and we will then talk directly about the least environment that satisfies the definitions in a given program.

7 The intensional language NVIL

In this section we define (with certain deviations from Yaghi (1984)) the syntax of a simple intensional language of nullary variables. As it will be demonstrated later in this paper, $NVIL$ can serve as the target language for transforming $FOFL$ programs. In the following definition notice that the syntax of $NVIL$ is defined with respect to a given set L of labels. In Yaghi (1984), L is taken to be the set ω of natural numbers, but in formalizing the algorithm a different choice for L proves more convenient, as we shall see.

Definition 7.1
Let L be a set of labels. The syntax of the intensional language $NVIL$ over L, is recursively defined by the following rules, in which $\mathbf{E}, \mathbf{E}_i$ denote expressions, *$\mathbf{F}, \mathbf{F}_i$ denote* definitions *and $\mathbf{P}$ denotes a* program*:*

$$\begin{array}{lcl}
\mathbf{E} & ::= & \mathbf{x} \in Var_0 \\
 & \mid & \mathbf{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}),\ \mathbf{c} \in \Sigma_n \\
 & \mid & \mathbf{call}_l(\mathbf{E}_0),\ l \in L \\
 & \mid & \mathbf{actuals}(\langle \mathbf{E}_l \rangle_{l \in I}),\ I \subseteq L \\
\mathbf{F} & ::= & (\mathbf{x} \doteq \mathbf{E}),\ \mathbf{x} \in Var_0 \\
\mathbf{P} & ::= & \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}
\end{array}$$

Notice that the **actuals** operator takes as argument a sequence of expressions indexed by a subset I of the set L of labels, and in this respect it is more general than the one defined in Yaghi (1984). The need for this extension will become apparent in the following sections. Similar restrictions as in $FOFL$ are adopted for the syntax of $NVIL$ programs.

Notice that the syntax of $NVIL$ only allows nullary variables to be defined and used in a program. On the other hand, both nullary and first-order constants can be used. Notice also the intensional operators that are adopted by the language; as $NVIL$ will be the target language for transforming $FOFL$ programs, the operators $\mathbf{call}_l$ and **actuals** will play a very important role in the elimination of function calls from the source programs.

As we have already explained, in intensional languages the meaning of an expression is a function from a set W of possible worlds to a set of data values. In the case of $NVIL$, we define the set of possible worlds as follows:

Definition 7.2
The set W of possible worlds of $NVIL$ is the set $List(L)$ of lists of elements of L.

Let D be a given domain. Then, the semantics of constant symbols of $NVIL$ with respect to D, are given by an interpretation function $\mathscr{C}'$, which assigns to every n-ary constant a function in $[(W \to D)^n \to (W \to D)]$. As $NVIL$ will be the target language for transforming programs of $FOFL$, the function $\mathscr{C}'$ is defined in terms of the interpretation function $\mathscr{C}$ for $FOFL$. More specifically:

Definition 7.3
For every $\mathbf{c} \in \Sigma_n$, *for every* $w \in W$, *and for all* $a_0, \ldots, a_{n-1} \in (W \to D)$,

$$\mathscr{C}'(\mathbf{c})(a_0, \ldots, a_{n-1})(w) = \mathscr{C}(\mathbf{c})(a_0(w), \ldots, a_{n-1}(w))$$

Let $IEnv$ be the set of *intensional environments* defined by $IEnv = Var_0 \to (W \to D)$. Then, the semantics of $NVIL$ are defined as follows:

Definition 7.4
The semantics of expressions of $NVIL$ *with respect to* $u \in IEnv$ *is recursively defined for every* $w \in W$, *as follows:*

$$\begin{array}{l}
[\![\mathbf{x}]\!](u)(w) = u(\mathbf{x})(w) \\
[\![\mathbf{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!](u)(w) = \mathscr{C}'(\mathbf{c})([\![\mathbf{E}_0]\!](u), \ldots, [\![\mathbf{E}_{n-1}]\!](u))(w) \\
[\![\mathbf{call}_l(\mathbf{E}_0)]\!](u)(w) = [\![\mathbf{E}_0]\!](u)(l : w) \\
[\![\mathbf{actuals}(\langle \mathbf{E}_l \rangle_{l \in I})]\!](u)(w) = [\![\mathbf{E}_{head(w)}]\!](u)(tail(w))
\end{array}$$

As in the case of $FOFL$, it can be easily shown that:

Theorem 7.1
For all expressions $\mathbf{E}$, $[\![\mathbf{E}]\!]$ is continuous and therefore monotonic.

Definition 7.5
The semantics of the program $\mathbf{P} = \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}$ *of* $NVIL$ *with respect to* $u \in IEnv$, *is defined as* $\widetilde{u}(\mathbf{result})$, *where* $\widetilde{u}$ *is the* least *intensional environment such that:*

1. *For every* $\mathbf{x} \in Var_0$ *with* $\mathbf{x} \notin func(\mathbf{P})$, $\widetilde{u}(\mathbf{x}) = u(\mathbf{x})$.
2. *For every definition* $(\mathbf{x} \doteq \mathbf{B})$ *in* $\mathbf{P}$, $\widetilde{u}(\mathbf{x}) = [\![\mathbf{B}]\!](\widetilde{u})$.

Notice that the semantics given above for $NVIL$ are standard, and their only difference from the ones given for $FOFL$ is that the former is defined on the richer domain $(W \to D)$ while the latter is defined on the domain D. Therefore, Theorem 6.2 and Lemma 7.1, transfer directly to the language $NVIL$ as well:

Theorem 7.2
Let $\mathbf{P}$ and $\widetilde{u}$ be as in Definition 7.5. Then, $\widetilde{u}$ is the least upper bound of the environments $\widetilde{u}_k$, $k \in \omega$, which for every definition $(\mathbf{x} \doteq \mathbf{B})$ in $\mathbf{P}$, are defined as follows:

$$\begin{array}{rcl}
\widetilde{u}_0(\mathbf{x}) & = & \bot_{W \to D} \\
\widetilde{u}_{k+1}(\mathbf{x}) & = & [\![\mathbf{B}]\!](\widetilde{u}_k)
\end{array}$$

Moreover, for every $k \in \omega$, $\widetilde{u}_k(\mathbf{x}) \sqsubseteq \widetilde{u}_{k+1}(\mathbf{x})$.

Lemma 7.1
Let $\mathbf{P}$ *and* $\widetilde{u}$ *be as in Definition 7.5. Then, for every* $(\mathbf{x} \doteq \mathbf{B})$ *in* $\mathbf{P}$,

$$\widetilde{u}_k(\mathbf{x}) \quad \sqsubseteq \quad [\![\mathbf{B}]\!](\widetilde{u}_k)$$

In the following, we let $call_l$ and $actuals$ be the functions that correspond to the object language operators $\mathbf{call}_l$ and $\mathbf{actuals}$.

$$\begin{array}{rcll} \mathscr{E}(\mathbf{x}) & = & \mathbf{x} & \\ \mathscr{E}(\mathbf{c}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1})) & = & \mathbf{c}(\mathscr{E}(\mathbf{E}_0),\ldots,\mathscr{E}(\mathbf{E}_{n-1})) & (n \geq 0) \\ \mathscr{E}(\mathbf{f}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1})) & = & \mathbf{call}_{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}(\mathbf{f}) & (n \geq 1) \end{array}$$

Fig. 5. Processing expressions of the source program.

8 Formal definition of the transformation algorithm

The main idea behind Yaghi's approach is that every function call in the source program will be translated into a *unique* intensional expression. This means that even if two function calls in a program are syntactically identical, they will be given different translations, as the following example illustrates:

Example 1
Consider the following first-order extensional program:

```
result  ≐  f(10)+f(10)
f(x)    ≐  x+1
```

The algorithm described in Yaghi (1984) would translate the above program as follows:

$$\begin{array}{lcl} \texttt{result} & \doteq & \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)} \\ \texttt{f} & \doteq & \texttt{x+1} \\ \texttt{x} & \doteq & \texttt{actuals(10,10)} \end{array}$$

However, there is no formal reason why the transformation should distinguish between identical function calls. Moreover, as we show below, Yaghi's algorithm can be formalized in a natural way if identical function calls receive the same translation. For simplicity in our presentation, the following assumption is adopted:

Assumption
Let **P** be a *FOFL* program. Then, the only nullary variable defined in **P** is the distinguished variable **result**.

The following conventions are also adopted. Let **P** be a first-order *FOFL* program and let *Sub*(**P**) be the set of subexpressions of **P**. Let **f** be a function defined in **P**. Then:

Definition 8.1
The set of labels *of calls to* **f** *in* **P** *is defined as:*

$$labels(\mathbf{f},\mathbf{P}) = \{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle \mid \mathbf{f}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}) \in Sub(\mathbf{P})\}$$

The transformation from extensional expressions to intensional ones is performed by the recursively defined function $\mathscr{E}$ given in Figure 5. Notice in particular the translation rule for function calls, and the fact that the subscript of the **call** operator is a sequence of expressions. In other words, the set L of labels over which the language *NVIL* is defined, is the set of sequences of expressions of the language *FOFL*.

$$\mathscr{A}(\mathbf{P}) = \bigcup_{\mathbf{F}\in\mathbf{P}} \mathscr{A}_{\mathbf{F}}(\mathbf{P})$$

$$\frac{\mathbf{F} = (\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B}),\ I = \mathit{labels}(\mathbf{f},\mathbf{P})}{\mathscr{A}_{\mathbf{F}}(\mathbf{P}) = \bigcup_{j=0}^{n-1}\{\mathbf{x}_j \doteq \mathbf{actuals}(\langle \mathscr{E}(l_j)\rangle_{l\in I})\}}$$

Fig. 6. Creating a new definition for each formal parameter in the source program.

$$\mathscr{D}(\mathbf{P}) = \bigcup_{\mathbf{F}\in\mathbf{P}} \overline{\mathscr{D}}(\mathbf{F})$$

$$\frac{\mathbf{F} = (\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B})}{\overline{\mathscr{D}}(\mathbf{F}) = (\mathbf{f} \doteq \mathscr{E}(\mathbf{B}))}$$

Fig. 7. Removing formal parameters from functions of the source program.

$$\mathit{Trans}(\mathbf{P}) = \mathscr{A}(\mathbf{P}) \cup \mathscr{D}(\mathbf{P})$$

Fig. 8. The overall translation.

Given a program **P**, the function $\mathscr{A}$ creates a new definition for each formal parameter of every function defined in **P**. Notice that $\mathscr{A}$ is defined in terms of $\mathscr{A}_{\mathbf{F}}$, which is a function that creates a set of new definitions, one for every formal parameter of the function defined by **F**. The definitions of $\mathscr{A}$ and $\mathscr{A}_{\mathbf{F}}$ are given in Figure 6.

The function $\mathscr{D}$ is used to remove the formal parameters from the function definitions in **P**, as shown in Figure 7.

The overall translation is performed by the function *Trans*, as shown in Figure 8.

This completes the presentation of the transformation algorithm. In the following section, example transformations that illustrate the above definitions, are given.

9 Example transformations

In this section we give two examples of the transformation algorithm. The first one is a simple non-recursive function, while the second one is a recursively defined factorial function.

Example 9.1
Consider the following simple first-order extensional program **P**:

```
result  ≐  f(f(10))
f(x)    ≐  x+1
```

There exist two function calls in **P**, namely `f(f(10))` and `f(10)`, and therefore the set *labels*(`f`, **P**) is equal to $\{\langle \texttt{f(10)}\rangle, \langle \texttt{10}\rangle\}$. In order to compute *Trans*(**P**) it suffices to

$$
\begin{array}{rl}
& [\![\mathtt{result}]\!](\widehat{u})([\,]) = \\
= & [\![\mathtt{call}_{l_0}\mathtt{(f)}]\!](\widehat{u})([\,]) \\
= & [\![\mathtt{f}]\!](\widehat{u})([l_0]) \\
= & [\![\mathtt{x+1}]\!](\widehat{u})([l_0]) \\
= & [\![\mathtt{x}]\!](\widehat{u})([l_0]) + 1 \\
= & [\![\mathtt{actuals}(\{\langle l_0, \mathtt{call}_{l_1}\mathtt{(f)}\rangle, \langle l_1, \mathtt{10}\rangle\})]\!](\widehat{u})([l_0]) + 1 \\
= & [\![\mathtt{call}_{l_1}\mathtt{(f)}]\!](\widehat{u})([\,]) + 1 \\
= & [\![\mathtt{f}]\!](\widehat{u})([l_1]) + 1 \\
= & [\![\mathtt{x+1}]\!](\widehat{u})([l_1]) + 1 \\
= & [\![\mathtt{x}]\!](\widehat{u})([l_1]) + 1 + 1 \\
= & [\![\mathtt{actuals}(\{\langle l_0, \mathtt{call}_{l_1}\mathtt{(f)}\rangle, \langle l_1, \mathtt{10}\rangle\})]\!](\widehat{u})([l_1]) + 1 + 1 \\
= & [\![\mathtt{10}]\!](\widehat{u})([\,]) + 1 + 1 \\
= & 12
\end{array}
$$

Fig. 9. Execution of the intensional program.

compute the sets $\mathscr{D}(\mathbf{F})$ and $\mathscr{A}(\mathbf{P})$. The first set can be computed using the definition of $\mathscr{E}$, and contains the following two definitions:

$$
\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{call}_{\langle \mathtt{f(10)}\rangle}\mathtt{(f)} \\
\mathtt{f} & \doteq & \mathtt{x+1}
\end{array}
$$

The set $\mathscr{A}(\mathbf{P})$ contains only one definition, corresponding to the formal parameter `x` of `f`.

$$
\begin{array}{rcl}
\mathscr{A}(\mathbf{P}) & = & \{\mathtt{x} \doteq \mathtt{actuals}(\{\langle\langle \mathtt{f(10)}\rangle, \mathscr{E}(\langle \mathtt{f(10)}\rangle_0)\rangle, \langle\langle \mathtt{10}\rangle, \mathscr{E}(\langle \mathtt{10}\rangle_0)\rangle\})\} \\
& = & \{\mathtt{x} \doteq \mathtt{actuals}(\{\langle\langle \mathtt{f(10)}\rangle, \mathscr{E}(\mathtt{f(10)})\rangle, \langle\langle \mathtt{10}\rangle, \mathscr{E}(\mathtt{10})\rangle\})\} \\
& = & \{\mathtt{x} \doteq \mathtt{actuals}(\{\langle\langle \mathtt{f(10)}\rangle, \mathtt{call}_{\langle \mathtt{10}\rangle}\mathtt{(f)}\rangle, \langle\langle \mathtt{10}\rangle, \mathtt{10}\rangle\})\}
\end{array}
$$

Therefore, the resulting intensional program of nullary variable definitions, is the following:

$$
\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{call}_{\langle \mathtt{f(10)}\rangle}\mathtt{(f)} \\
\mathtt{f} & \doteq & \mathtt{x+1} \\
\mathtt{x} & \doteq & \mathtt{actuals}(\{\langle\langle \mathtt{f(10)}\rangle, \mathtt{call}_{\langle \mathtt{10}\rangle}\mathtt{(f)}\rangle, \langle\langle \mathtt{10}\rangle, \mathtt{10}\rangle\})
\end{array}
$$

The notation used above is cumbersome; we can simplify it by letting l_0 and l_1 stand for the two labels $\langle \mathtt{f(10)}\rangle$ and $\langle \mathtt{10}\rangle$, respectively. Then, the program can be written as:

$$
\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{call}_{l_0}\mathtt{(f)} \\
\mathtt{f} & \doteq & \mathtt{x+1} \\
\mathtt{x} & \doteq & \mathtt{actuals}(\{\langle l_0, \mathtt{call}_{l_1}\mathtt{(f)}\rangle, \langle l_1, \mathtt{10}\rangle\})
\end{array}
$$

Let $\widehat{u}$ be the least environment that satisfies the definitions of the above *NVIL* program. Then, following the denotational semantics, we can compute the semantic value of the program, as shown in Figure 9.

Example 9.2
Consider the following recursive first-order extensional program **P**:

$$
\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{fact(2)} \\
\mathtt{fact(n)} & \doteq & \mathtt{if\ (n<=1)\ then\ 1\ else\ n*fact(n-1)}
\end{array}
$$

$$
\begin{array}{rl}
 & [\![\texttt{result}]\!](\widehat{u})([\,]) = \\
= & [\![\texttt{call}_{l_0}\texttt{(fact)}]\!](\widehat{u})([\,]) \\
= & [\![\texttt{fact}]\!](\widehat{u})([l_0]) \\
= & [\![\texttt{if (n<=1) then 1 else n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0]) \\
= & \mathit{if}\ ([\![\texttt{n<=1}]\!](\widehat{u})([l_0]))\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0])) \\
= & \mathit{if}\ (\widehat{u}(\texttt{n})([l_0]) <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0])) \\
= & \mathit{if}\ ([\![\texttt{actuals(}\{\langle l_0, 2\rangle, \langle l_1, \texttt{n-1}\rangle\}\texttt{)}]\!](\widehat{u})([l_0]) <= 1)\ \mathit{then}\ 1 \\
 & \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0])) \\
= & \mathit{if}\ (2 <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0])) \\
= & [\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0]) \\
= & \widehat{u}(\texttt{n})([l_0]) * [\![\texttt{call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0]) \\
= & 2 * [\![\texttt{call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_0]) \\
= & 2 * [\![\texttt{fact}]\!](\widehat{u})([l_1, l_0]) \\
= & 2 * [\![\texttt{if (n<=1) then 1 else n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_1, l_0]) \\
= & 2 * (\mathit{if}\ ([\![\texttt{n<=1}]\!](\widehat{u})([l_1, l_0]))\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_1, l_0]))) \\
= & 2 * (\mathit{if}\ ([\![\texttt{actuals(}\{\langle l_0, 2\rangle, \langle l_1, \texttt{n-1}\rangle\}\texttt{)}]\!](\widehat{u})([l_1, l_0]) <= 1)\ \mathit{then}\ 1 \\
 & \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_1, l_0]))) \\
= & 2 * (\mathit{if}\ ([\![\texttt{n-1}]\!](\widehat{u})([l_0]) <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_1, l_0]))) \\
= & 2 * (\mathit{if}\ (1 <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([l_1, l_0]))) \\
= & 2 * 1 \\
= & 2
\end{array}
$$

Fig. 10. Execution of the intensional program that results from `fact`.

Since there exist only two function calls in **P**, namely `fact(2)` and `fact(n-1)`, the set *labels*(`f`, **P**) is equal to $\{\langle 2\rangle, \langle \texttt{n-1}\rangle\}$. We let $l_0 = \langle 2\rangle$ and $l_1 = \langle \texttt{n-1}\rangle$. Then, the two definitions of the initial first-order program become after they are processed by $\mathscr{D}$:

$$
\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_{l_0}\texttt{(fact)} \\
\texttt{fact} & \doteq & \texttt{if (n<=1) then 1 else n*call}_{l_1}\texttt{(fact)}
\end{array}
$$

The set $\mathscr{A}(\mathbf{P})$ contains only one definition corresponding to the formal parameter `n` of `fact`:

$$\mathscr{A}(\mathbf{P}) \quad = \quad \{\texttt{n} \doteq \texttt{actuals(}\{\langle l_0, 2\rangle, \langle l_1, \texttt{n-1}\rangle\}\texttt{)}\}$$

Therefore, the final intensional program consists of the following set of definitions:

$$
\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_{l_0}\texttt{(fact)} \\
\texttt{fact} & \doteq & \texttt{if (n<=1) then 1 else n*call}_{l_1}\texttt{(fact)} \\
\texttt{n} & \doteq & \texttt{actuals(}\{\langle l_0, 2\rangle, \langle l_1, \texttt{n-1}\rangle\}\texttt{)}
\end{array}
$$

The above program can be executed following the same principles as in example 9.1 (see Figure 10). Notice that during the calculations the value of $\widehat{u}(\texttt{n})$ under the context $[l_0]$ is demanded three times. In practical terms, this means that if the value together with the context were appropriately saved when first encountered, then they could be reused when demanded again. This is a very important efficiency issue, and standard implementations of eduction use a 'warehouse' scheme for saving such intermediate results (see, for example, Faustini *et al.* (1991) and Du and Wadge (1990a), or the discussion in section 12).

10 Correctness proof

The correctness proof of the transformation algorithm is established by Theorems 10.1, 10.2 and 10.3 that follow. The main idea of the proof is to relate semantically a function call in the source first-order extensional program with the corresponding intensional expression that results from its translation. For example, given a first-order extensional program $\mathbf{P}$, we would like to give a semantic statement concerning a call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ in $\mathbf{P}$, and its translation $\mathbf{call}_{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}(\mathbf{f})$ in $Trans(\mathbf{P})$.

This equality appears, at first sight, to be obvious. It seems to relate two notations for the same function call. Appearances, however, are deceptive; the actual parameters $\mathbf{E}_i$ in the second expression are employed as syntactic objects only, as markers or labels; they are *mentioned* rather than *used*, in logical terminology. In our proof we show that, in the context in which they appear, the **call** operators work *as if* they were alternate notations for function application, a fact which is by no means obvious.

Our proof uses fixed point induction to establish a more complicated statement. Let u and $\widehat{u}$ be the least environments satisfying the definitions in $\mathbf{P}$ and $Trans(\mathbf{P})$, respectively, and let $w \in W$. We first prove the following statement:

$$(call_{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w) = u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)).$$

This looks like a weaker result than what we are actually looking for, because the right-hand side does not correspond exactly to the expression $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of the extensional program. However, a stronger result can be shown afterwards using an inductive argument, as we are going to see. It seems that the above statement can not itself be shown in one step. Instead, we need to show that the right-hand side approximates the left, and *vice versa*. The details of the proof are given below:

Theorem 10.1

Let $\mathbf{P}$ be a $FOFL$ program and let u be the least environment satisfying the definitions in $\mathbf{P}$. Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every function definition $(\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B})$ in $\mathbf{P}$, for every function call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$ and for every $w \in W$

$$(call_{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)).$$

Proof

It suffices to show that for all $k \in \omega$, for every function definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, for every call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$(call_{\langle \mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}\widehat{u}_k(\mathbf{f}))(w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u}_k)(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u}_k)(w)).$$

This can be established by induction on k. Notice that we only use the approximations of $\widehat{u}$ but not the approximations of u. Intuitively, this gives to the right-hand side of the above statement an ‘advantage’, which allows the $\sqsubseteq$ relation to be established.

Induction basis
It suffices to show that for every function definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, for every call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$(call_{\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle} \widehat{u}_0(\mathbf{f}))(w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u}_k)(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u}_k)(w)).$$

The above holds trivially because the left hand side is equal to the bottom value.

Induction hypothesis
We assume that for every function definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, for every call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$(call_{\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle} \widehat{u}_k(\mathbf{f}))(w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u}_k)(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u}_k)(w)).$$

Induction step
We show that for every $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, for every call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$(call_{\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle} \widehat{u}_{k+1}(\mathbf{f}))(w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u}_{k+1})(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u}_{k+1})(w)).$$

Using the semantics of *call*, the above statement can be equivalently written as follows:

$$\widehat{u}_{k+1}(\mathbf{f})(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) \sqsubseteq u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u}_{k+1})(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u}_{k+1})(w)).$$

Recalling that $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$ and $\mathbf{f} \doteq \mathscr{E}(\mathbf{B})$ in $Trans(\mathbf{P})$, and using Definition 6.4 and Theorem 7.2, the above is equivalent to the following:

$$[\![\mathscr{E}(\mathbf{B})]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) \sqsubseteq [\![\mathbf{B}]\!](u \oplus \rho_{k+1}),$$

where $\rho_{k+1}(\mathbf{x}_j) = [\![\mathscr{E}(\mathbf{E}_i)]\!](\widehat{u}_{k+1})(w)$, $j = 0, \ldots, n-1$. The above can be established by showing that for every subexpression $\mathbf{S}$ of $\mathbf{B}$, we have:

$$[\![\mathscr{E}(\mathbf{S})]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) \sqsubseteq [\![\mathbf{S}]\!](u \oplus \rho_{k+1}).$$

We therefore perform an inner structural induction on $\mathbf{S}$.

Structural induction basis
Case $\mathbf{S} = \mathbf{x}_j \in \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$. Then, according to the transformation algorithm, in the intensional program $Trans(\mathbf{P})$ a definition of the form $\mathbf{x}_j \doteq \mathbf{actuals}(\langle \mathscr{E}(l_j) \rangle_{l \in I})$ has been created, where $I = labels(\mathbf{f}, \mathbf{P})$. We have:

$$\begin{array}{ll} & [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) = \\ = & [\![\mathscr{E}(\mathbf{x}_j)]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) \\ & \text{(Because } \mathbf{S} = \mathbf{x}_j) \qquad\qquad III \\ = & [\![\mathbf{x}_j]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle : w) \\ & \text{(Definition of } \mathscr{E}) \end{array}$$

$$
\begin{aligned}
&= \widehat{u}_k(\mathbf{x}_j)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) \\
&\quad \text{(Semantics)} \\
&\sqsubseteq [\![\mathbf{actuals}(\langle \mathscr{E}(l_j)\rangle_{l\in I})]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) \\
&\quad \text{(Definition of } \mathbf{x}_j \text{ and Lemma 7.1)} \\
&= [\![\mathscr{E}(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle_j)]\!](\widehat{u}_k)(w) \\
&\quad \text{(Semantics of } \mathbf{actuals}) \\
&= [\![\mathscr{E}(\mathbf{E}_j)]\!](\widehat{u}_k)(w) \\
&\quad \text{(Selection of the } j\text{'th element of the sequence)} \\
&\sqsubseteq [\![\mathscr{E}(\mathbf{E}_j)]\!](\widehat{u}_{k+1})(w) \\
&\quad \text{(Because } \widehat{u}_k \sqsubseteq \widehat{u}_{k+1} \text{ by Theorem 7.2 and} \\
&\quad \;\text{because } [\![\mathscr{E}(\mathbf{E}_j)]\!] \text{ is monotonic by Theorem 7.1)} \\
&= \rho_{k+1}(\mathbf{x}_j) \\
&\quad \text{(Because } \rho_{k+1}(\mathbf{x}_j) = [\![\mathscr{E}(\mathbf{E}_j)]\!](\widehat{u}_{k+1})(w)) \\
&= [\![\mathbf{x}_j]\!](u \oplus \rho_{k+1}) \\
&\quad \text{(Definition of } \oplus) \\
&= [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
&\quad \text{(Because } \mathbf{S} = \mathbf{x}_j)
\end{aligned}
$$

Case $\mathbf{S} = \mathbf{c}$, where $\mathbf{c}$ is a nullary constant symbol. Then the proof is straightforward because for all $w \in W$, $\mathscr{C}'(\mathbf{c})(w) = \mathscr{C}(\mathbf{c})$, or in other words, $\mathscr{C}'(\mathbf{c})$ is a constant intension.

Structural induction step

Case $\mathbf{S} = \mathbf{c}(\mathbf{S}_0, \ldots, \mathbf{S}_{r-1})$. Recall that the semantics of constants in the intensional program are defined in a pointwise way in terms of the semantics of the constants in the extensional program. We have:

$$
\begin{aligned}
& [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) = \\
&= [\![\mathscr{E}(\mathbf{c}(\mathbf{S}_0, \ldots, \mathbf{S}_{r-1}))]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) \\
&\quad \text{(Assumption for } \mathbf{S}) \\
&= [\![\mathbf{c}(\mathscr{E}(\mathbf{S}_0), \ldots, \mathscr{E}(\mathbf{S}_{r-1}))]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) \\
&\quad \text{(Definition of } \mathscr{E}) \\
&= \mathscr{C}'(\mathbf{c})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u}_k), \ldots, [\![\mathscr{E}(\mathbf{S}_{r-1})]\!](\widehat{u}_k))(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w) \\
&\quad \text{(Semantics of constant symbols)} \\
&= \mathscr{C}(\mathbf{c})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w), \ldots, [\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u}_k)(\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\rangle : w)) \\
&\quad \text{(Definition 7.3)} \\
&\sqsubseteq \mathscr{C}(\mathbf{c})([\![\mathbf{S}_0]\!](u \oplus \rho_{k+1}), \ldots, [\![\mathbf{S}_{r-1}]\!](u \oplus \rho_{k+1})) \\
&\quad \text{(Structural induction hypothesis and} \\
&\quad \text{monotonicity of } \mathscr{C}(c)) \\
&= [\![\mathbf{c}(\mathbf{S}_0, \ldots, \mathbf{S}_{r-1})]\!](u \oplus \rho_{k+1}) \\
&\quad \text{(Semantics of constant symbols)} \\
&= [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
&\quad \text{(Assumption for } \mathbf{S})
\end{aligned}
$$

Case $\mathbf{S} = \mathbf{g}(\mathbf{S}_0, \ldots, \mathbf{S}_{r-1})$, where $\mathbf{g} \in func(\mathbf{P})$. Then, the left-hand side of the statement we want to establish can be written as follows:

$$
\begin{aligned}
& [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u}_k)(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w) = \\
= \; & [\![\mathscr{E}(\mathbf{g}(\mathbf{S}_0,\dots,\mathbf{S}_{r-1}))]\!](\widehat{u}_k)(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Assumption about } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{call}_{\langle \mathbf{S}_0,\dots,\mathbf{S}_{r-1}\rangle}(\mathbf{g})]\!](\widehat{u}_k)(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathscr{E}\text{)} \\
= \; & (call_{\langle \mathbf{S}_0,\dots,\mathbf{S}_{r-1}\rangle}\widehat{u}_k(\mathbf{g}))(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Semantics)} \\
\sqsubseteq \; & u(\mathbf{g})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u}_k)(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w),\dots,[\![\mathscr{E}(\mathbf{S}_{r-1})]\!](\widehat{u}_k)(\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle : w)) \\
& \text{(Outer induction hypothesis on } k\text{)} \\
\sqsubseteq \; & u(\mathbf{g})([\![\mathbf{S}_0]\!](u \oplus \rho_{k+1}),\dots,[\![\mathbf{S}_{r-1}]\!](u \oplus \rho_{k+1})) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } u(\mathbf{g}) \text{ from Theorem 6.2)} \\
= \; & [\![\mathbf{g}(\mathbf{S}_0,\dots,\mathbf{S}_{r-1})]\!](u \oplus \rho_{k+1}) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
& \text{(Because } \mathbf{S} = \mathbf{g}(\mathbf{S}_0,\dots,\mathbf{S}_{r-1})\text{)}
\end{aligned}
$$

This completes the proof of the theorem. □

Theorem 10.2

Let **P** be a *FOFL* program and let u be the least environment satisfying the definitions in **P**. Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program *Trans*(**P**). Then, for every function definition $(\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B})$ in **P**, for every function call $\mathbf{f}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})$ of **f** in **P**, and for every $w \in W$

$$u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\dots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq (call_{\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w).$$

Proof

It suffices to show that for all $k \in \omega$, for every function definition $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in **P**, for every call $\mathbf{f}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})$ of **f** in **P**, and for every $w \in W$:

$$u_k(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\dots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq (call_{\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w).$$

This can be established by induction on k. Notice that now we only use the approximations of u but not the approximations of $\widehat{u}$. Again, this gives to the right-hand side of the above statement an 'advantage', which allows the $\sqsubseteq$ relation to be established.

Induction basis

It suffices to show that for every function definition $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in **P**, for every call $\mathbf{f}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})$ of **f** in **P**, and for every $w \in W$:

$$u_0(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\dots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq (call_{\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w).$$

The above holds trivially because the left-hand side is equal to the bottom value.

Induction hypothesis

We assume that for every function definition $\mathbf{f}(\mathbf{x}_0,\dots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in **P**, for every call $\mathbf{f}(\mathbf{E}_0,\dots,\mathbf{E}_{n-1})$ of **f** in **P**, and for every $w \in W$:

$$u_k(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\dots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq (call_{\langle \mathbf{E}_0,\dots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w).$$

Induction step
We show that for every function definition $\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, for every call $\mathbf{f}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1})$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$u_{k+1}(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\ldots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq (call_{\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle}\widehat{u}(\mathbf{f}))(w).$$

Using the semantics of *call*, the above statement can be written as follows:

$$u_{k+1}(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w),\ldots,[\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w)) \sqsubseteq \widehat{u}(\mathbf{f})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w).$$

Recalling that $\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$ and that $\mathbf{f} \doteq \mathscr{E}(\mathbf{B})$ in $Trans(\mathbf{P})$, and using Theorem 6.2 and Definition 7.5, the above is equivalent to the following:

$$[\![\mathbf{B}]\!](u_k \oplus \rho) \sqsubseteq [\![\mathscr{E}(\mathbf{B})]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w).$$

where $\rho(\mathbf{x}_i) = [\![\mathscr{E}(\mathbf{E}_i)]\!](\widehat{u})(w)$, $i = 0,\ldots,n-1$. The above can be established by showing that for every subexpression $\mathbf{S}$ of $\mathbf{B}$, it is:

$$[\![\mathbf{S}]\!](u_k \oplus \rho) \sqsubseteq [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w).$$

We therefore perform a structural induction on $\mathbf{S}$.

Structural induction basis
Case $\mathbf{S} = \mathbf{x}_j \in \{\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\}$. In the transformed program $Trans(\mathbf{P})$, a definition of the form $\mathbf{x}_j \doteq \mathbf{actuals}(\langle\mathscr{E}(l_j)\rangle_{l\in I})$ has been created, where $I = labels(\mathbf{f},\mathbf{P})$. Consider the right-hand side of the statement we would like to establish:

$$\begin{aligned}
& [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w) = \\
= \; & [\![\mathscr{E}(\mathbf{x}_j)]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \; & [\![\mathbf{x}_j]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathscr{E}\text{)} \\
= \; & \widehat{u}(\mathbf{x}_j)(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Semantics)} \\
= \; & [\![\mathbf{actuals}(\langle\mathscr{E}(l_j)\rangle_{j\in I})]\!](\widehat{u})(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathbf{x}_j \text{ and Definition 7.5)} \\
= \; & [\![\mathscr{E}(\langle\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\rangle_j)]\!](\widehat{u})(w) \\
& \text{(Semantics of } \mathbf{actuals}\text{)} \\
= \; & [\![\mathscr{E}(\mathbf{E}_j)]\!](\widehat{u})(w) \\
& \text{(Selection of the } j\text{'th element of the sequence)} \\
= \; & \rho(\mathbf{x}_j) \\
& \text{(Because } \rho(\mathbf{x}_j) = [\![\mathscr{E}(\mathbf{E}_j)]\!](\widehat{u})(w)\text{)} \\
= \; & [\![\mathbf{x}_j]\!](u_k \oplus \rho) \\
& \text{(Definition of } \oplus\text{)} \\
= \; & [\![\mathbf{S}]\!](u_k \oplus \rho) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)}
\end{aligned}$$

Notice that in this case we have established the equality of the right- and left-hand sides of the statement under consideration.

Case $\mathbf{S} = \mathbf{c}$, where $\mathbf{c}$ is a nullary constant symbol. Then the proof is straightforward because for all $w \in W$, $\mathscr{C}'(\mathbf{c})(w) = \mathscr{C}(\mathbf{c})$, or in other words, $\mathscr{C}'(\mathbf{c})$ is a constant intension.

Structural induction step

Case $\mathbf{S} = \mathbf{c}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1})$. Recall that the semantics of constants in the intensional program are defined in a pointwise way in terms of the semantics of the constants in the extensional program. We have:

$$\begin{aligned}
& [\![\mathbf{S}]\!](u_k \oplus \rho) = \\
= \;& [\![\mathbf{c}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1})]\!](u_k \oplus \rho) \\
& \text{(Assumption for } \mathbf{S}) \\
= \;& \mathscr{C}(\mathbf{c})([\![\mathbf{S}_0]\!](u_k \oplus \rho), \dots, [\![\mathbf{S}_{r-1}]\!](u_k \oplus \rho)) \\
& \text{(Semantics of constant symbols)} \\
\sqsubseteq \;& \mathscr{C}(\mathbf{c})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w), \dots, [\![\mathscr{E}(\mathbf{S}_{r-1})]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w)) \\
& \text{(Structural induction hypothesis and monotonicity of } \mathscr{C}(c)) \\
= \;& \mathscr{C}'(\mathbf{c})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u}), \dots, [\![\mathscr{E}(\mathbf{S}_{r-1})]\!](\widehat{u}))(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition 7.3)} \\
= \;& [\![\mathbf{c}(\mathscr{E}(\mathbf{S}_0), \dots, \mathscr{E}(\mathbf{S}_{r-1}))]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Semantics of constants)} \\
= \;& [\![\mathscr{E}(\mathbf{c}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1}))]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathscr{E}) \\
= \;& [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Assumption for } \mathbf{S})
\end{aligned}$$

Case $\mathbf{S} = \mathbf{g}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1})$, where $\mathbf{g} \in func(\mathbf{P})$. Then, the left-hand side of the statement we want to establish can be written as follows:

$$\begin{aligned}
& [\![\mathbf{S}]\!](u_k \oplus \rho) = \\
= \;& [\![\mathbf{g}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1})]\!](u_k \oplus \rho) \\
& \text{(Because } \mathbf{S} = \mathbf{g}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1})) \\
= \;& u_k(\mathbf{g})([\![\mathbf{S}_0]\!](u_k \oplus \rho), \dots, [\![\mathbf{S}_{r-1}]\!](u_k \oplus \rho)) \\
& \text{(Semantics of application)} \\
\sqsubseteq \;& u_k(\mathbf{g})([\![\mathscr{E}(\mathbf{S}_0)]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w), \dots, [\![\mathscr{E}(\mathbf{S}_{r-1})]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w)) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } u_k(\mathbf{g}) \text{ from Theorem 6.2)} \\
\sqsubseteq \;& (call_{\langle \mathbf{S}_0, \dots, \mathbf{S}_{r-1}\rangle}(\widehat{u}(\mathbf{g})))(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Induction hypothesis on } k) \\
= \;& [\![\mathbf{call}_{\langle \mathbf{S}_0, \dots, \mathbf{S}_{r-1}\rangle}(\mathbf{g})]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Semantics of } \mathbf{call}) \\
= \;& [\![\mathscr{E}(\mathbf{g}(\mathbf{S}_0, \dots, \mathbf{S}_{r-1}))]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathscr{E}) \\
= \;& [\![\mathscr{E}(\mathbf{S})]\!](\widehat{u})(\langle \mathbf{E}_0, \dots, \mathbf{E}_{n-1}\rangle : w) \\
& \text{(Definition of } \mathbf{S})
\end{aligned}$$

This completes the proof of the theorem. □

Lemma 10.1
Let **P** *be a first-order extensional program and let u be the least environment satisfying the definitions in* **P**. *Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every function definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in* **P**, *for every function call $\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ of* **f** *in* **P**, *and for every $w \in W$*

$$(call_{\langle \mathbf{E}_0, \ldots, \mathbf{E}_{n-1} \rangle} \widehat{u}(\mathbf{f}))(w) = u(\mathbf{f})([\![\mathscr{E}(\mathbf{E}_0)]\!](\widehat{u})(w), \ldots, [\![\mathscr{E}(\mathbf{E}_{n-1})]\!](\widehat{u})(w))$$

Theorem 10.3
Let **P** be a first-order extensional program and let u be the least environment satisfying the definitions in **P**. Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every $w \in W$

$$u(\mathbf{result}) = \widehat{u}(\mathbf{result})(w)$$

Proof
By a straightforward structural induction on the defining expression of the variable **result** in **P** and using Lemma 10.1. □

11 Relationship with tagged dataflow

Now that we have established the theoretical foundation of Yaghi's (revised) transformation algorithm, we present a brief discussion of its practical implications for the eductive dataflow model.

The basic principle of the dataflow model of computation, is that data can be processed while they are in *motion*, flowing through a *dataflow network*. A dataflow network is a system of processing stations (or *nodes*), connected by a number of communication channels (or *arcs*). Each node may have one or more input and output arcs.

There exist two main paradigms of dataflow. In *pipeline* dataflow, data items flow along the arcs of a network in a first-in first-out way. Therefore, the edges can be thought as queues between the nodes. In *tagged* dataflow, data items are labelled with *tags*, and edges can now be thought as sets of such items. The purpose of the tags is to impose some *conceptual* ordering on the data items. A node can execute if it finds in its input arcs data items that have identical tags. The tagged approach eliminates the need to maintain first-in first-out queues on the arcs, and in this way it offers more parallelism than the pipeline model.

Tags have been extensively used to implement first-order functions on dataflow architectures (Kirkham *et al.*, 1985; Arvind and Nikhil, 1990). The main idea is that one would like to distinguish between data items that correspond to different function invocations. This can be achieved by letting each tag determine a particular invocation of a function. Intuitively, a tag can be thought of as a distinct *colour* (Arvind and Culler, 1987) that is assigned to the data items of a function invocation, so as that we can distinguish them from those data items that belong to other invocations of the same function.

Consider now Yaghi's transformation algorithm, and the intensional programs that result from it. We showed in section 9 how these programs can be easily interpreted with respect to a context. The method used to evaluate the intensional code is actually *demand-driven tagged dataflow*: demand-driven because the evaluator continuously asks for the value of identifiers under particular contexts, and tagged because the contexts used can be considered as tags that accompany data items during evaluation. Moreover, the **call** and **actuals** operators can be thought as operations that change the context part of the data that flow through the dataflow network. In other words, the notion of possible world of intensional logic is in direct correspondence to the notion of tag in the dataflow model. In particular, the lists of labels that are used as contexts by the intensional approach, correspond to the 'colour' idea of the dataflow community. Consequently, our work can also be viewed as a formalization through the use of intensional logic of the 'colouring' technique.

It should also be noted at this point that although the 'colouring' technique has been extensively used for implementing first-order programs on dataflow architectures, it does not immediately generalize to higher-order programs. Current treatments for higher-order functions on dataflow machines rely on introducing in the implementation data structures (representing closures). Our recent research (Rondogiannis and Wadge, 1994a, 1994b; Rondogiannis, 1994) has shown that a significant class of higher-order functions can be implemented based on an extended tagging scheme. It is our opinion that such an approach fits better to the tagged-dataflow ideas, and could for this reason result in dataflow implementations of higher-order languages which are simpler, more efficient, and more fault-tolerant. The theoretical and practical aspects of the extended tagging scheme will be discussed in a forthcoming paper (Rondogiannis and Wadge, 1996).

Although a further discussion of higher-order functions would be outside the scope of this paper, we should also cite at this point the work of Nelan on *firstification* (Nelan, 1991), a technique for transforming many higher-order programs into first-order ones. In general, we believe that higher-order functions is an intriguing area of research which deserves further study.

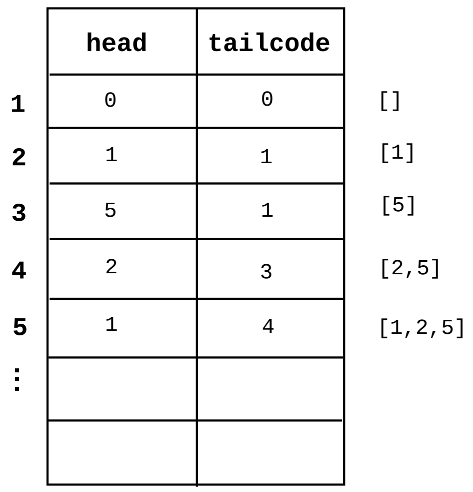
12 Implementation issues

Since its inception, Yaghi's transformation algorithm has been successfully used as the core implementation technique for a number of first-order languages. Such implementations include compilers for the Lucid language (Faustini *et al.*, 1991), the GLU language (Jagannathan and Dodd, 1994), as well as interpreters for sophisticated 3D spreadsheets (Du and Wadge, 1990a, 1990b). In the following we give a brief introduction to implementation issues regarding the technique. A detailed presentation and analysis of such issues is outside the scope of this paper and the interested reader is refered elsewhere (Rondogiannis and Wadge, 1994a; Rondogiannis, 1994).

An implementation of the transformation algorithm should focus on two important efficiency issues:

Efficient context operations

As we have already seen, the contexts (tags) required by the technique are lists of



	head	tailcode	
1	0	0	[]
2	1	1	[1]
3	5	1	[5]
4	2	3	[2,5]
5	1	4	[1,2,5]
⋮			

Fig. 11. The hash-consing technique.

natural numbers. In a real implementation however, it would be preferrable to have contexts that are natural numbers and not lists. There exists a simple encoding trick (known as 'hash-consing') for achieving this purpose. Hash-consing is a method of implementing lists which ensures that every list (considered as a sequence of items) has a unique representation. The main advantage of hash-consing is that two lists can be tested for equality with a single operation, rather than with a loop which scans the lists and compares corresponding elements. The only disadvantage is that with each cons operation we must consult a hash table to check that the list we are constructing does not already have a representation.

We store the list representatives in a table each row of which is a pair *(head, index of tail)*. The list is then represented (or encoded) by the index of the appropriate pair in the table (see Figure 11). The following primitive functions are used to implement hash-consing:

- *hashcons*(*head*, *tail_code*): uses a hash function to check if the pair (*head*, *tail_code*) already exists in the hash table. If it does not, then it inserts it. Finally, it returns the position of the pair in the table.
- *hashhead*(*list_code*): returns the first element of the pair found in the *list_code* position of the hash table.
- *hashtail*(*list_code*): returns the second element of the pair found in the *list_code* position of the hash table.

The above operations can be performed efficiently and the space occupied by the hash table is reasonable. Using hash-consing and the above primitives, an efficient implementation of the technique can be built that avoids expensive list management. Moreover, in cases where the value of a context needs to be saved during execution (this is further discussed below), the hash consing representation proves indispensable.

Hash consing gives us a method to represent, with small integers, points in a tree-shaped index space. The technique therefore has a broader practical significance:

it makes it possible to extend eduction to data structures whose components are not necessarily indexed by copies of the integers. For example, Tao (1994) presents a demand-driven attribute grammar system in which the attributes are intensions indexed by the nodes in a parse tree.

Avoiding recomputations

The eductive evaluation of an intensional program typically generates multiple demands for the same extension. In other words, the value v of an identifier **x** under a specific context w may be demanded many times during program execution (see Example 9.2). The 'Warehouse' is an associative store, usually implemented as a hash table, whose purpose is to keep this kind of information. If the value of the identifier **x** under the same context w is demanded again during program execution, then a lookup of the Warehouse can potentially save significant time. (Notice that contexts can now be easily saved in the Warehouse because they have been encoded as integers using hash-consing.)

It is important to note that the Warehouse is not a required component of the implementation. In fact, the only space that is actually essential for the technique is the table used for implementing hash-consing. However, experience shows that the use of the Warehouse gives an important benefit to the implementation by drastically reducing the number of steps that have to be performed in order to evaluate a program. The size of the Warehouse can be controlled either by using heuristics or by appropriate analysis-based techniques (Bagai, 1986), which guess or predict the length of time a given entry has to be retained.

In conclusion, it should be noted that the ideas discussed in this section have been incorporated in all the implementations of Lucid, GLU, and other related first-order languages and systems. In this paper, we present a formalization of, and correctness proof for, a practical technique which has been in use already for many years.

13 Conclusions

The goal of this paper is two-fold. First, it provides an intuitive introduction to the concepts of *intensional logic* and *eduction*, and describes their relationship to the dataflow model of computation. Second, it establishes in a rigorous way the connections between first-order functional languages and intensional languages of nullary variables. In this way it provides a formal basis for a technique that has successfully been used for implementing a variety of first-order functional systems.

More generally, we believe that many of the features of modern functional languages can be 'intensionalized', and that dataflow computation still holds further promise for the implementation of such languages. Our beliefs are strongly supported by our recent work that generalizes the technique described in this paper to apply to typed higher-order functional programs.

Acknowledgements

We would like to thank Arvind for usefull discussions during PACT94, and R. Kieburtz for the insightful feedback he provided as the external PhD examiner of the first author. This work has benefited from discussions with many members of the intensional programming group. The research was supported by the Natural Sciences and Engineering Research Council of Canada and a University of Victoria Graduate Fellowship.

References

Arvind and Culler, D. (1987) Dataflow architectures. In S. S. Thakkar (ed.), *Selected Reprints on Dataflow and Reduction Architectures*, pp. 79–101. IEEE Press.

Arvind and Nikhil, R. S. (1990) Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, **39**(3):300–318.

Bagai, R. (1986) Compilation of the Dataflow Language Lucid. Master's thesis, Department of Computer Science, University of Victoria.

Du, W. and Wadge, W. W. (1990a) The eductive implementation of a three-dimensional spreadsheet. *Software–Practice and Experience*, **20**(11):1097–1114, November.

Du, W. and Wadge, W. W. (1990b) A 3D spreadsheet based on intensional logic. *IEEE Software*: 78–89, July.

Dowty, D., Wall, R. and Peters, S. (1981) *Introduction to Montague Semantics.* Reidel.

Faustini, A. A., Ashcroft, E. A. and Jagannathan, R. (1991) An Intensional Language for Parallel Applications Programming. In B. K.Szymanski (ed.), *Parallel Functional Languages and Compilers*, pp. 11–49. ACM Press.

Faustini, A. A. and Wadge, W. W. (1987) An eductive interpreter for the language pLucid. *Proc. SIGPLAN 87 Conference on Interpreters and Interpretive techniques (SIGPLAN Notices* **22**(7)):86–91.

Gunter, C. (1992) *Semantics of Programming Languages.* MIT Press.

Jagannathan, R. and Dodd, C. (1994) GLU Programmer's Guide. *Technical Report SRI-CSL-94-06*, Computer Science Laboratory, SRI International, Menlo Park, CA, July.

Kirkham, C., Gurd, J. and Watson, I. (1985) The Manchester Prototype Dataflow Computer. *Commun. ACM*: 34–52, January.

Nelan, G. (1994) *Firstification.* PhD thesis, Department of Computer Science, Arizona State University.

Rondogiannis, P. (1994) *Higher-Order Functional Languages and Intensional Logic.* PhD thesis, Department of Computer Science, University of Victoria, Canada, December.

Rondogiannis, P. and Wadge, W. W. (1994a) Higher-Order Dataflow and its Implementation on Stock Hardware. *Proc. ACM Symposium on Applied Computing*, pp. 431–435. ACM Press.

Rondogiannis, P. and Wadge, W. W. (1994b) Compiling Higher-Order Functions for Tagged-Dataflow. In *Proc. IFIP International Conference on Parallel Architectures and Compilation Techniques.* North-Holland.

Rondogiannis, P. and Wadge, W. W. (1996) Higher-Order Functional Languages and Intensional Logic. (In preparation.)

Stoy, J. (1977) *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press.

Tao, S. (1994) *Indexical Attribute Grammars.* PhD thesis, Department of Computer Science, University of Victoria, Canada, December.

Tennent, R. (1991) *Semantics of Programming Languages.* Prentice Hall.

Thomason, R. (ed.) (1974) *Formal Philosophy, Selected Papers of R. Montague.* Yale University Press.

van Benthem, J. (1988) *A Manual of Intensional Logic*. CSLI Lecture Notes.

Wadge, W. W. and Ashcroft, E. A. (1985) *Lucid, the Dataflow Programming Language*. Academic Press.

Yaghi, A. A. (1984) *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, UK.