

# Transforming First-Order Functional Programs to Intensional Programs of Nullary Variables: Theoretical Foundations

P. Rondogiannis and W. W. Wadge  
Department of Computer Science  
University of Victoria  
P.O. Box 3055, Victoria, B.C.  
CANADA V8W 3P6.  
e-mail: {prondo,wwadge}@csr.uvic.ca

## Abstract

In this paper we present a revised formulation and a correctness proof of Yaghi's [Yag84] transformation algorithm from first-order extensional programs to intensional programs of nullary variables. The formal definition of the algorithm is a functional one, and its main difference from the one given in [Yag84] is that if two expressions in the source program are identical, then they are assigned identical intensional expressions during the translation. The correctness proof of the algorithm is established by showing that a function call in the extensional program has - informally speaking - the same meaning as the intensional expression that results from its translation.

## 1 Introduction

The first work to establish a transformation algorithm from extensional to intensional programs, was A. Yaghi's Ph.D. dissertation [Yag84]. Motivated by Montague's intensional logic [Mon74, DWP81], 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.

Yaghi's work, apart from its theoretical significance, had practical implications as well: the resulting intensional programs can be interpreted in a very simple way. In fact, the Lucid functional-dataflow language [AW76, AW77, WA85, EAAJ91], other Lucid-related systems [DW90b, DW90a], as well as standard functional languages [RW93, RW94b], have been successfully implemented based on this approach. However, there are two important aspects of the technique, that were not developed in [Yag84]:

1. The transformation algorithm from extensional to intensional programs presented in [Yag84], is semi-formal and non-functional.
2. A correctness proof of the transformation is not given in [Yag84], and has remained an open problem since then.

It is the purpose of this paper to resolve the above two issues, establishing in this way a semantics preserving transformation from extensional to intensional programs. The rest of the paper is organized as follows: Section 2 outlines Yaghi's transformation algorithm. In Sections 3 and 4, we present an alternative formulation of the algorithm, and illustrate it by examples. The correctness proof for the transformation is given in Section 5, and the paper concludes by discussing the main points of our work.

In the following sections, we assume a basic familiarity with the work described in [Yag84], as well as an understanding of domain theory and denotational semantics [Sto77, Ten91, Gun92].

## 2 Yaghi's Transformation Algorithm

This section presents an outline of Yaghi's transformation algorithm. The source extensional language under consideration, is a first-order subset of ISWIM [Lan66], referred as *Iwade* in [Yag84]. Intuitively, *Iwade* does not allow nested **where** clauses, and it requires that all variables in a program are distinct. The target intensional language is referred as *DE* in [Yag84], and it only allows nullary variable definitions. Moreover, *DE* is enriched with a set of *intensional* operators, which play an important role in the translation process. For precise formal definitions of *Iwade* and *DE*, see [Yag84, page 3-3] and [Yag84, pages 2-38, 3-23] respectively. The transformation algorithm, can be outlined as follows:

1. Let **f** be a function appearing in the source extensional program. Number the textual occurrences of calls to **f** in the program, starting at 1 (including calls in the body of the definition of **f**).
2. Replace the *i*th call of **f** in the program by **call<sub>i</sub>(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 extensional program:

```

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

```

The following intensional program is obtained, when the algorithm is applied:

```

result  ≐ call1(f) + call2(f)
f       ≐ call1(g)
g       ≐ y
x       ≐ actuals(4, 5)
y       ≐ actuals(x + 1)

```

In the following we give a brief informal description of the semantics of *DE* programs (for a precise definition see [Yag84]). In general, the semantics of *Iwade* and *DE* programs are defined using standard techniques. Their only difference is that the underlying domain of the latter is a much richer one than the domain of the former. More specifically, if *D* is the domain of the source *Iwade* programs, then the domain of the target *DE* programs is  $(W - D)$ , where *W* is the set *List(N)* of lists of natural numbers. Elements of  $(W - D)$  are called *intensions*. Therefore, **call** and **actuals** are operators that take as arguments intensions. The corresponding semantic equations associated with these two operations, are [Yag84]:

$$\begin{aligned}
 (call_i(a))(w) &= a(i : w) \\
 (actuals(a_1, \dots, a_n))(i : w) &= (a_i)(w)
 \end{aligned}$$

where  $a, a_1, \dots, a_n \in (W - D)$ ,  $w \in W$ , and  $:$  is the usual consing operation on lists. Moreover, the semantic interpretation of constant symbols appearing in *DE* programs, is defined in a pointwise way in terms of the interpretation of the corresponding constants that appear in *Iwade* programs [Yag84].

## 3 A Revised Formulation of Yaghi's 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 3.1** Consider the following first-order extensional program:

$$\begin{aligned}\text{result} &\doteq f(10) + f(10) \\ f(x) &\doteq x + 1\end{aligned}$$

The algorithm described in [Yag84] would translate the above program as follows:

$$\begin{aligned}\text{result} &\doteq \text{call}_1(f) + \text{call}_2(f) \\ f &\doteq x + 1 \\ x &\doteq \text{actuals}(10, 10)\end{aligned}$$

However, such a translation is not natural and proves quite difficult to formalize. Therefore, we revise Yaghi's algorithm so as to operate in a "referentially transparent" way: identical function calls should be assigned identical intensional expressions. For this purpose, we will use a *Gödel numbering function*. Let  $Exp$  be the set of expressions of programs of *Iwade*. Then:

**Theorem 3.1** [LP81, pages 242-243] There exists a one-to-one map  $[\cdot] : Exp \rightarrow N$ . For every  $E \in Exp$ ,  $[E]$  is called the *Gödel number* of  $E$ .

The *Gödel numbering* captures the situation described above: it assigns different numbers to syntactically different function calls, but assigns the same number to indistinguishable calls.

**Example 3.2** Consider again the extensional program given in example 3.1. Let  $[f(10)] = l \in N$ . Then, the translation of the expression  $f(10) + f(10)$  under the *Gödel numbering* scheme, will be  $\text{call}_l(f) + \text{call}_l(f)$ .

We are now in a position to formally define the revised transformation algorithm. For simplicity, we assume that the only nullary variable defined in  $P$  is the distinguished variable **result**. Moreover, we also assume that the only variables that can appear in  $P$  are the functional variables defined in  $P$  as well as their formal parameters.

The following definitions will be used in the rest of this paper:

**Definition 3.1** Let  $u : A \rightarrow B$  and  $\rho : S \rightarrow B$ , where  $S \subseteq A$ . Then, the *perturbation*  $u \oplus \rho$  of  $u$  with respect to  $\rho$  is defined as:

$$(u \oplus \rho)(x) = \begin{cases} \rho(x) & \text{if } x \in S \\ u(x) & \text{otherwise} \end{cases}$$

**Definition 3.2** Let  $I$  and  $S$  be sets. An  $I$ -indexed sequence is any function  $s : I \rightarrow S$ , and is denoted by  $\langle s_i \rangle_{i \in I}$ .

Let  $P$  be a first-order extensional program,  $Sub(P)$  be the set of subexpressions of  $P$  and  $func(P)$  be the set of functions defined in  $P$ . Let  $f \in func(P)$ . Then:

- The set of labels of calls to  $f$  in  $P$  is defined as:

$$\text{labels}(f, P) = \{[f(E_1, \dots, E_n)] \mid f(E_1, \dots, E_n) \in Sub(P)\}$$

- The selector function  $\odot$  on labels is defined as:

$$[f(E_1, \dots, E_n)] \odot j = E_j, \quad j \in \{1, \dots, n\}$$

The transformation from extensional expressions to intensional ones is performed by the following recursively defined function  $\mathcal{E}$ :

$$\frac{E = x}{\mathcal{E}(E) = x}$$

$$\frac{E = c(E_1, \dots, E_n)}{\mathcal{E}(E) = c(\mathcal{E}(E_1), \dots, \mathcal{E}(E_n))}$$

$$\frac{E = f(E_1, \dots, E_n)}{\mathcal{E}(E) = \text{call}_{|E|}(f)}$$

Given a program  $P$  and a function  $f(x_1, \dots, x_n) \doteq B_f$  defined in  $P$ , the function  $\mathcal{A}_f$  is used to create a set of new definitions, one for every formal parameter of  $f$ :

$$\frac{I = \text{labels}(f, P)}{\mathcal{A}_f(P) = \bigcup_{j=1}^n \{x_j \doteq \text{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})\}}$$

Notice that the **actuals** operator we adopt is more general than the one used in [Yag84]. In our case, **actuals** takes as argument a sequence of expressions indexed by a set  $I \subseteq N$ . In Yaghi's case, the set  $I$  is always equal to  $\{1, \dots, n\}$ , for some  $n \in N$ .

The function  $\mathcal{D}$  removes the formal parameters from a function definition and at the same time uses  $\mathcal{E}$  to process the body of the definition.

$$\frac{F = (f(x_1, \dots, x_n) \doteq B_f)}{\mathcal{D}(F) = (f \doteq \mathcal{E}(B_f))}$$

The overall transformation can be described as follows:

$$\text{Trans}(P) = \left( \bigcup_{f \in \text{func}(P)} \mathcal{A}_f(P) \right) \cup \left( \bigcup_{F \in P} \{\mathcal{D}(F)\} \right)$$

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

## 4 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 4.1** Consider the following simple first-order extensional program  $P$ :

$$\begin{aligned} \text{result} &\doteq f(f(10)) \\ f(x) &\doteq x + 1 \end{aligned}$$

Assume that  $[f(f(10))] = l_1$  and that  $[f(10)] = l_2$ , where  $l_1, l_2 \in N$ . Therefore,  $\text{labels}(P) = \{l_1, l_2\}$ . In order to compute  $\text{Trans}(P)$  it suffices to compute the sets  $(\bigcup_{F \in P} \{\mathcal{D}(F)\})$  and  $\mathcal{A}_f(P)$ . The first set can be computed using the definition of  $\mathcal{E}$ , and contains the following two definitions:

$$\begin{aligned} \text{result} &\doteq \text{call}_{l_1}(f) \\ f &\doteq x + 1 \end{aligned}$$

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

$$\begin{aligned} \mathcal{A}_f(P) &= \{x \doteq \text{actuals}(\{\langle l_1, \mathcal{E}(l_1 \odot 1) \rangle, \langle l_2, \mathcal{E}(l_2 \odot 1) \rangle\})\} \\ &= \{x \doteq \text{actuals}(\{\langle l_1, \mathcal{E}(f(10)) \rangle, \langle l_2, \mathcal{E}(10) \rangle\})\} \\ &= \{x \doteq \text{actuals}(\{\langle l_1, \text{call}_{l_2}(f) \rangle, \langle l_2, 10 \rangle\})\} \end{aligned}$$

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

$$\begin{aligned} \text{result} &\doteq \text{call}_{l_1}(f) \\ f &\doteq x + 1 \\ x &\doteq \text{actuals}(\{\langle l_1, \text{call}_{l_2}(f) \rangle, \langle l_2, 10 \rangle\}) \end{aligned}$$



**Example 4.2** Consider the following recursive first-order extensional program  $P$ :

```

result   $\doteq$  fact(3)
fact(n)  $\doteq$  if (n<=1) then 1 else n*fact(n-1)

```

Assume the  $\llbracket \text{fact}(3) \rrbracket = l_1$  and that  $\llbracket \text{fact}(n-1) \rrbracket = l_2$ . The two definitions of the initial first-order extensional program become after they are processed by  $\mathcal{D}$ :

```

result   $\doteq$  call $_{l_1}$ (fact)
fact     $\doteq$  if (n<=1) then 1 else n*call $_{l_2}$ (fact)

```

The set  $\mathcal{A}_{\text{fact}}(P)$  contains only one definition for the formal parameter  $n$ :

$$\mathcal{A}_{\text{fact}}(P) = \{n \doteq \text{actuals}(\{\langle l_1, 3 \rangle, \langle l_2, n-1 \rangle\})\}$$

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

```

result   $\doteq$  call $_{l_1}$ (fact)
fact     $\doteq$  if (n<=1) then 1 else n*call $_{l_2}$ (fact)
n        $\doteq$  actuals(\{\langle l_1, 3 \rangle, \langle l_2, n-1 \rangle\})

```

## 5 Correctness Proof

The correctness proof of the transformation algorithm is established by Theorems 5.1, 5.2 and 5.3 to follow. Recall that we have made the assumption that the source functional language does not allow individual variable definitions (except for the distinguished variable **result**). Moreover, the only individual variables that can appear in the right hand side of a definition are the formal parameters of the function being defined.

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  $P$ , we would like to give a semantic statement concerning a call  $E = f(E_1, \dots, E_n)$  in  $P$ , and its translation  $\text{call}_{\llbracket E \rrbracket}(f)$  in  $\text{Trans}(P)$ . Let  $u$  and  $\hat{u}$  be the least environments satisfying the definitions in  $P$  and  $\text{Trans}(P)$  respectively, and let  $w \in W$ . The idea is to first prove the following statement:

$$(\text{call}_{\llbracket E \rrbracket}(\hat{u}(f)))(w) = u(f)(\llbracket \mathcal{E}(E_1) \rrbracket(\hat{u})(w), \dots, \llbracket \mathcal{E}(E_n) \rrbracket(\hat{u})(w))$$

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  $f(E_1, \dots, E_n)$  of the extensional program. However, a stronger result can be shown afterwards using an inductive argument, as we are going to see. It turns out 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 5.1** Let  $P$  be a first-order extensional program and let  $u$  be the least environment satisfying the definitions in  $P$ . Let  $\hat{u}$  be the least environment satisfying the definitions in the translated program  $\text{Trans}(P)$ . Then, for every function definition  $(f(x_1, \dots, x_n) \doteq B_f)$  in  $P$ , for every function call  $E = f(E_1, \dots, E_n)$  of  $f$  in  $P$  and for every  $w \in W$

$$(\text{call}_{\llbracket E \rrbracket}(\hat{u}(f)))(w) \sqsubseteq u(f)(\llbracket \mathcal{E}(E_1) \rrbracket(\hat{u})(w), \dots, \llbracket \mathcal{E}(E_n) \rrbracket(\hat{u})(w))$$

**Proof:** The theorem is established by induction on the approximations  $\hat{u}_k$ ,  $k \in N$  of  $\hat{u}$ . In other words, we show that for every  $k \geq 0$ , for every function  $f$  defined in  $P$ , for every function call  $E = f(E_1, \dots, E_n)$  of  $f$  in  $P$ , and for every  $w \in W$ :

$$(\text{call}_{\llbracket E \rrbracket}(\hat{u}_k(f)))(w) \sqsubseteq u(f)(\llbracket \mathcal{E}(E_1) \rrbracket(\hat{u}_k)(w), \dots, \llbracket \mathcal{E}(E_n) \rrbracket(\hat{u}_k)(w))$$

Notice that we only use the approximations of  $\hat{u}$  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. The basis case is for  $k = 0$  and it holds trivially because the left hand side of the above statement is equal to the bottom value. We assume that the above statement holds for  $k$  and we show that it holds for  $k + 1$ , i.e.,

$$(call_{\lceil E \rceil}(\hat{u}_{k+1}(f)))(w) \sqsubseteq u(f)(\llbracket \mathcal{E}(\mathbf{E}_1) \rrbracket(\hat{u}_{k+1})(w), \dots, \llbracket \mathcal{E}(\mathbf{E}_n) \rrbracket(\hat{u}_{k+1})(w))$$

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

$$\hat{u}_{k+1}(f)(\lceil E \rceil : w) \sqsubseteq u(f)(\llbracket \mathcal{E}(\mathbf{E}_1) \rrbracket(\hat{u}_{k+1})(w), \dots, \llbracket \mathcal{E}(\mathbf{E}_n) \rrbracket(\hat{u}_{k+1})(w))$$

Recalling that  $f(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{B}_f$  in  $\mathbf{P}$  and  $f \doteq \mathcal{E}(\mathbf{B}_f)$  in  $Trans(\mathbf{P})$ , and using the definition of  $\hat{u}_{k+1}$  in terms of  $u_k$  (see for example [Ten91]), the above is equivalent to the following:

$$\llbracket \mathcal{E}(\mathbf{B}_f) \rrbracket(\hat{u}_k)(\lceil E \rceil : w) \sqsubseteq \llbracket \mathbf{B}_f \rrbracket(u \oplus \rho_{k+1})$$

where  $\rho_{k+1}(\mathbf{x}_j) = \llbracket \mathcal{E}(\mathbf{E}_j) \rrbracket(\hat{u}_{k+1})(w)$ ,  $j = 1, \dots, n$ . The above can be established by showing that for every subexpression  $\mathbf{S}$  of  $\mathbf{B}_f$ , we have:

$$\llbracket \mathcal{E}(\mathbf{S}) \rrbracket(\hat{u}_k)(\lceil E \rceil : w) \sqsubseteq \llbracket \mathbf{S} \rrbracket(u \oplus \rho_{k+1})$$

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

*Structural Induction Basis.*

Case  $\mathbf{S} = \mathbf{x}_j \in \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ . Then, in the intensional program  $Trans(\mathbf{P})$ , a definition of the form  $\mathbf{x}_j \doteq \text{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})$  has been created, where  $I = \text{labels}(f, \mathbf{P})$ . We have:

$$\begin{aligned} & \llbracket \mathcal{E}(\mathbf{S}) \rrbracket(\hat{u}_k)(\lceil E \rceil : w) = \\ &= \llbracket \mathcal{E}(\mathbf{x}_j) \rrbracket(\hat{u}_k)(\lceil E \rceil : w) \\ & \quad (\text{Because } \mathbf{S} = \mathbf{x}_j) \\ &= \llbracket \mathbf{x}_j \rrbracket(\hat{u}_k)(\lceil E \rceil : w) \\ & \quad (\text{Definition of } \mathcal{E}) \\ &= \hat{u}_k(\mathbf{x}_j)(\lceil E \rceil : w) \\ & \quad (\text{Semantics of variables}) \\ &\sqsubseteq \llbracket \text{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I}) \rrbracket(\hat{u}_k)(\lceil E \rceil : w) \\ & \quad (\text{Definition of } \mathbf{x}_j \text{ and } \hat{u}_k) \\ &= \llbracket \mathcal{E}(\lceil E \rceil \odot j) \rrbracket(\hat{u}_k)(w) \\ & \quad (\text{Semantics of actuals}) \\ &= \llbracket \mathcal{E}(\mathbf{E}_j) \rrbracket(\hat{u}_k)(w) \\ & \quad (\text{Definition of } \odot) \\ &\sqsubseteq \llbracket \mathcal{E}(\mathbf{E}_j) \rrbracket(\hat{u}_{k+1})(w) \\ & \quad (\text{Because } \hat{u}_k \sqsubseteq \hat{u}_{k+1} \text{ and using} \\ & \quad \text{the monotonicity of } \llbracket \mathcal{E}(\mathbf{E}_j) \rrbracket) \\ &= \llbracket \mathbf{x}_j \rrbracket(u \oplus \rho_{k+1}) \\ & \quad (\text{Because } \rho_{k+1}(\mathbf{x}_j) = \llbracket \mathcal{E}(\mathbf{E}_j) \rrbracket(\hat{u}_{k+1})(w)) \\ &= \llbracket \mathbf{S} \rrbracket(u \oplus \rho_{k+1}) \\ & \quad (\text{Because } \mathbf{S} = \mathbf{x}_j) \end{aligned}$$

*Structural Induction Step.*

Case  $\mathbf{S} = \mathbf{c}(\mathbf{S}_1, \dots, \mathbf{S}_n)$ . The interpretation function  $\mathcal{C}'$  for constants in the intensional program, is defined in a pointwise way in terms of the interpretation function  $\mathcal{C}$  for constants in the extensional



program (see [Yag84, page 3-4]). We have:

$$\begin{aligned}
& \llbracket \mathcal{E}(\mathbf{S}) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) = \\
&= \llbracket \mathcal{E}(\mathbf{c}(\mathbf{S}_1, \dots, \mathbf{S}_n)) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Assumption for } \mathbf{S}) \\
&= \llbracket \mathbf{c}(\mathcal{E}(\mathbf{S}_1), \dots, \mathcal{E}(\mathbf{S}_n)) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Definition of } \mathcal{E}) \\
&= \mathcal{C}'(\mathbf{c})(\llbracket \mathcal{E}(\mathbf{S}_1) \rrbracket(\hat{u}_k), \dots, \llbracket \mathcal{E}(\mathbf{S}_n) \rrbracket(\hat{u}_k))(\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Semantics of constant symbols}) \\
&= \mathcal{C}(\mathbf{c})(\llbracket \mathcal{E}(\mathbf{S}_1) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w), \dots, \llbracket \mathcal{E}(\mathbf{S}_n) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w)) \\
&\quad (\text{Definition of } \mathcal{C}' \text{ in terms of } \mathcal{C}) \\
&\sqsubseteq \mathcal{C}(\mathbf{c})(\llbracket \mathbf{S}_1 \rrbracket(u \oplus \rho_{k+1}), \dots, \llbracket \mathbf{S}_n \rrbracket(u \oplus \rho_{k+1})) \\
&\quad (\text{Structural induction hypothesis and} \\
&\quad \text{monotonicity of } \mathcal{C}(\mathbf{c})) \\
&= \llbracket \mathbf{c}(\mathbf{S}_1, \dots, \mathbf{S}_n) \rrbracket(u \oplus \rho_{k+1}) \\
&\quad (\text{Semantics of constant symbols}) \\
&= \llbracket \mathbf{S} \rrbracket(u \oplus \rho_{k+1}) \\
&\quad (\text{Assumption for } \mathbf{S})
\end{aligned}$$

Case  $\mathbf{S} = \mathbf{g}(\mathbf{S}_1, \dots, \mathbf{S}_r)$ , where  $\mathbf{g} \in \text{func}(\mathbf{P})$ . Assume that the definition of  $\mathbf{g}$  in  $\mathbf{P}$  is  $\mathbf{g}(\mathbf{y}_1, \dots, \mathbf{y}_r) \doteq \mathbf{B}_\mathbf{g}$ . Then, by the transformation algorithm, the definition for  $\mathbf{g}$  in  $\text{Trans}(\mathbf{P})$  is  $\mathbf{g} \doteq \mathcal{E}(\mathbf{B}_\mathbf{g})$ . We have:

$$\begin{aligned}
& \llbracket \mathcal{E}(\mathbf{S}) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) = \\
&= \llbracket \mathcal{E}(\mathbf{g}(\mathbf{S}_1, \dots, \mathbf{S}_r)) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Assumption about } \mathbf{S}) \\
&= \llbracket \text{call}_{\lceil \mathbf{S}_1 \rceil}(\mathbf{g}) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Definition of } \mathcal{E}) \\
&= (\text{call}_{\lceil \mathbf{S}_1 \rceil}(\hat{u}_k(\mathbf{g}))) (\lceil \mathbf{E} \rceil : w) \\
&\quad (\text{Semantics}) \\
&\sqsubseteq u(\mathbf{g})(\llbracket \mathcal{E}(\mathbf{S}_1) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w), \dots, \llbracket \mathcal{E}(\mathbf{S}_r) \rrbracket(\hat{u}_k)(\lceil \mathbf{E} \rceil : w)) \\
&\quad (\text{Outer induction hypothesis on } k) \\
&\sqsubseteq u(\mathbf{g})(\llbracket \mathbf{S}_1 \rrbracket(u \oplus \rho_{k+1}), \dots, \llbracket \mathbf{S}_r \rrbracket(u \oplus \rho_{k+1})) \\
&\quad (\text{Structural induction hypothesis and} \\
&\quad \text{monotonicity of } u(\mathbf{g})) \\
&= \llbracket \mathbf{g}(\mathbf{S}_1, \dots, \mathbf{S}_r) \rrbracket(u \oplus \rho_{k+1}) \\
&\quad (\text{Semantics of application}) \\
&= \llbracket \mathbf{S} \rrbracket(u \oplus \rho_{k+1}) \\
&\quad (\text{Because } \mathbf{S} = \mathbf{g}(\mathbf{S}_1, \dots, \mathbf{S}_r))
\end{aligned}$$

This completes the proof of the theorem. ■

**Theorem 5.2** Let  $\mathbf{P}$  be a first-order extensional program and let  $u$  be the least environment satisfying the definitions in  $\mathbf{P}$ . Let  $\hat{u}$  be the least environment satisfying the definitions in the translated program  $\text{Trans}(\mathbf{P})$ . Then, for every function definition  $(\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$  of  $\mathbf{f}$  in  $\mathbf{P}$ , for every function call  $\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \dots, \mathbf{E}_n)$  in  $\mathbf{P}$ , and for every  $w \in W$

$$u(\mathbf{f})(\llbracket \mathcal{E}(\mathbf{E}_1) \rrbracket(\hat{u})(w), \dots, \llbracket \mathcal{E}(\mathbf{E}_n) \rrbracket(\hat{u})(w)) \sqsubseteq (\text{call}_{\lceil \mathbf{E}_1 \rceil}(\hat{u}(\mathbf{f}))(w))$$

**Proof:** The theorem is established by induction on the approximations  $u_k$ ,  $k \in N$  of  $u$ . The proof is similar to the one given for Theorem 5.1. ■

**Lemma 5.1** Let  $\mathbf{P}$  be a first-order extensional program and let  $u$  be the least environment satisfying the definitions in  $\mathbf{P}$ . Let  $\hat{u}$  be the least environment satisfying the definitions in the translated program  $\text{Trans}(\mathbf{P})$ . Then, for every function definition  $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f}$  in  $\mathbf{P}$ , for every function call  $\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \dots, \mathbf{E}_n)$  in  $\mathbf{P}$ , and for every  $w \in W$

$$(\text{call}_{\lceil \mathbf{E}_1 \rceil}(\hat{u}(\mathbf{f}))(w) = u(\mathbf{f})(\llbracket \mathcal{E}(\mathbf{E}_1) \rrbracket(\hat{u})(w), \dots, \llbracket \mathcal{E}(\mathbf{E}_n) \rrbracket(\hat{u})(w))$$

$k$	$call_{l_1}(\widehat{u}_k(\mathbf{f}))(w)$	$u(\mathbf{f})(\llbracket call_{l_2}(\mathbf{f}) \rrbracket(\widehat{u}_k)(w))$
0	$\perp$	$\perp$
1	$\perp$	$\perp$
2	$\perp$	$\perp$
3	$\perp$	12
4	12	12

Table 1: An illustration of the first part of the proof

**Theorem 5.3** Let  $\mathbf{P}$  be a first-order extensional 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  $w \in W$

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

**Proof:** By a structural induction on the defining expression of the variable  $\mathbf{result}$  in  $\mathbf{P}$  and using Lemma 5.1. ■

## 6 An Illustration of the Proof

To illustrate the technique used for the proof, consider the program that was given in Example 4.1. We show how the theorem can be applied to the outer call to function  $\mathbf{f}$  in that example. Similar arguments apply for the inner call to  $\mathbf{f}$ . It suffices to show that for every  $k \geq 0$ , we have:

$$\begin{aligned} call_{l_1}(\widehat{u}_k(\mathbf{f}))(w) &\sqsubseteq u(\mathbf{f})(\llbracket call_{l_2}(\mathbf{f}) \rrbracket(\widehat{u}_k)(w)) \\ u_k(\mathbf{f})(\llbracket call_{l_2}(\mathbf{f}) \rrbracket(\widehat{u})(w)) &\sqsubseteq call_{l_1}(\widehat{u}(\mathbf{f}))(w) \end{aligned}$$

Using the technique for computing the least environment from its approximations (see for example [Ten91]), one can compute the values of  $\widehat{u}_k$  and  $u_k$  for various values of  $k \in N$ . For example, the validity of the first of the above statements for  $k = 0$ , can be shown by first evaluating the left hand side of the statement:

$$\begin{aligned} call_{l_1}(\widehat{u}_0(\mathbf{f}))(w) &= \\ &= (\widehat{u}_0(\mathbf{f}))(l_1 : w) \\ &= \perp \end{aligned}$$

and then evaluating the right hand side, which also yields the  $\perp$  value:

$$\begin{aligned} u(\mathbf{f})(\llbracket call_{l_2}(\mathbf{f}) \rrbracket(\widehat{u}_0)(w)) &= \\ &= u(\mathbf{f})(\widehat{u}_0(\mathbf{f})(l_2 : w)) \\ &= u(\mathbf{f})(\perp) \\ &= \perp \end{aligned}$$

Tables 1 and 2 have been constructed in this way, and they illustrate Theorems 5.1 and 5.2 respectively. Notice that every entry in the second column of the two tables, approximates the corresponding entry in the third column.

## 7 Discussion

The main difficulty in giving a correctness proof for the transformation algorithm, lies in the fact that it is not straightforward to relate the source functional program (and its semantics) to the resulting intensional program (and its semantics). Some of the complications are outlined below:



$k$	$u_k(\mathbf{f})(\llbracket \text{call}_{l_2}(\mathbf{f}) \rrbracket(\hat{u})(w))$	$\text{call}_{l_1}(\hat{u}(\mathbf{f}))(w)$
0	$\perp$	12
1	12	12

Table 2: An illustration of the second part of the proof

1. The intensional program that results from the transformation has significant syntactic differences from the source extensional one. Note in particular that the formal parameters in the latter have become individual definitions in the former. Therefore, a syntax-based correctness proof may face considerable difficulties. The author has undertaken one such approach, attempting to identify a sequence of intensional transformations that correspond to the notion of beta-reduction. Although some interesting results were obtained, this approach proved to be quite harder than the one presented in this paper.
2. The precise formal definition of Yaghi's transformation algorithm we gave in Section 3, helped us formulate the exact result that we had to demonstrate. It should be noted here that the author tried at first to formalize Yaghi's non-referentially transparent scheme, using the notion of an *occurrence* of a function call in the program. However, such an approach proved to be quite inflexible and did not easily lead to the right intuitions.
3. The proof requires a double induction: an outer computational one and an inner structural one. Moreover, notice that the statement in Lemma 5.1, is not symmetric: the intensional environment appears in both sides of the statement, while the extensional one appears in only one of them.
4. It would be expected that Lemma 5.1 can be demonstrated directly, i.e., without first showing that the right hand side of the statement approximates the left, and vice-versa. However, such a proof does not seem to be possible.

Finally, we should mention that the transformation algorithm and the proof can readily be extended for a language that allows "outside" variables as well as nullary variable definitions. Moreover, the techniques illustrated in this paper can be extended to apply to more "demanding" intensionalization procedures, such as for example the ones that have been suggested for higher-order functional programs [Wad91, RW94a].

## References

- [AW76] E. Ashcroft and W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM J. on Computing*, 5(3):336–354, September 1976.
- [AW77] E. Ashcroft and W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.
- [DW90a] W. Du and W. W. Wadge. The Eductive Implementation of a Three-dimensional Spreadsheet. *Software-Practice and Experience*, 20(11):1097–1114, November 1990.
- [DW90b] W. Du and W.W.Wadge. A 3D Spreadsheet Based on Intensional Logic. *IEEE Software*, pages 78–89, July 1990.
- [DWP81] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics*. Reidel Publishing Company, 1981.
- [EAAJ91] A. A. Faustini E. A. Ashcroft and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

- [Gun92] C. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.
- [Lan66] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966.
- [LP81] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.
- [Mon74] R. Montague. *Formal Philosophy, Selected Papers of R. Montague*. Yale University Press, 1974.
- [RW93] P. Rondogiannis and W. Wadge. A Dataflow Implementation Technique for Lazy Typed Functional Languages. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 23–42, 1993.
- [RW94a] P. Rondogiannis and W. Wadge. Compiling Higher-Order Functions for Tagged-Dataflow. In *Proceedings of the IFIP/ACM International Conference on Parallel Architectures and Compilation Techniques*. North-Holland, August 1994.
- [RW94b] P. Rondogiannis and W. Wadge. Higher-Order Dataflow and its Implementation on Stock Hardware. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, 1994.
- [Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Ten91] R. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.
- [WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.
- [Wad91] W. W. Wadge. Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, 1991.
- [Yag84] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.