J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press 1

Higher-Order Functional Languages and Intensional Logic

P. RONDOGIANNIS

Department of Computer Science,
University of Ioannina,
P.O. Box 1186,
GR 45110 Ioannina, GREECE
(e-mail:prondo@cs.uoi.gr)

W. W. WADGE

Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, BC
CANADA V8W 3P6
(e-mail:wwadge@lucy.uvic.ca)

Abstract

In this paper we demonstrate that a broad class of higher-order functional programs can be transformed into semantically equivalent multidimensional intensional programs that contain only nullary variable definitions. The proposed algorithm systematically eliminates user-defined functions from the source program, by appropriately introducing context manipulation (i.e. intensional) operators. The transformation takes place in M steps, where M is the order of the initial functional program. During each step the order of the program is reduced by one, and the final outcome of the algorithm is an M-dimensional intensional program of order zero. As the resulting intensional code can be executed in a purely tagged-dataflow way, the proposed approach offers a promising new technique for the implementation of higher-order functional languages.

1 Introduction

This paper is the successor of (Rondogiannis & Wadge, 1997) in which we formally established the correctness of a transformation algorithm from first-order functional programs to intensional programs of nullary variables. In the present paper we extend our investigation to a broad class of higher-order functional programs. In particular, we define an algorithm which gradually transforms a given higher-order program into a semantically equivalent intensional program of nullary variables. As it was discussed in (Rondogiannis & Wadge, 1997), there exists a very close relationship between intensional languages and the tagged-dataflow model of computation: the notion of *context* (or *tag*, or *possible world*) plays a crucial role in both cases. In fact, tagged-dataflow machines provide the ideal hardware platform for executing intensional languages. Therefore, the immediate practical outcome of the algorithm developed in this paper is a technique for implementing higher-order functional languages in a purely dataflow way. It should be noted here that the proposed technique can also be implemented efficiently on conventional architectures, and is therefore an interesting alternative to the well-known graph reduction based techniques for implementing functional languages.

The paper is organized as follows: section 2 outlines the transformation algorithm for first-order programs. The material in this section is a brief presentation of the ideas in (Rondogiannis & Wadge, 1997) in order to make the present paper self-contained. For a complete and formal description of the first-order case, the interested reader should consult (Rondogiannis & Wadge, 1997). Section 3 presents an intuitive introduction to the transformation algorithm for higher-order programs. Section 4 introduces the mathematical notation that will be used throughout the paper. The simple higher-order functional language FL that will be the focus of our investigation, is presented in section 5 and its (classical) denotational semantics is given. The higher-order intensional language IL and the final zero-order intensional language $NVIL$ are developed in section 6 and their *synchronic* denotational semantics are presented. The transformation from FL to $NVIL$ is derived in section 7 and the correctness of the transformation is demonstrated in section 8. The paper concludes with discussion of the significance of the proposed technique, implementation issues, related work and future directions in the area of intensional transformations of functional programs.

The paper assumes a basic familiarity with intensional programming. The interested reader can consult the first three sections of (Rondogiannis & Wadge, 1997) for a quick introduction to the area.

2 The First-Order Case

Before considering higher-order programs, we outline the approach we adopt for the first-order case; this was initially developed in (Yaghi, 1984) and formalized in (Rondogiannis & Wadge, 1997). The algorithm transforms a first-order program into a set of zero-order definitions that contain context manipulation operations. As the semantics of the resulting code is based on Montague's Intensional Logic (Thomason, 1974), the resulting definitions are also referred as *intensional* definitions.

The transformation algorithm can be outlined as follows (see (Rondogiannis & Wadge, 1997) for a more detailed and formal exposition). For each function **f** defined in the source functional program,

1. 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 ith call of **f** in the program by $\texttt{call}_i\texttt{(f)}$. 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
```

The translation algorithm produces the following intensional program:

$$
\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)} \\
\texttt{f} & \doteq & \texttt{call}_0\texttt{(g)} \\
\texttt{g} & \doteq & \texttt{y} \\
\texttt{x} & \doteq & \texttt{actuals(4,5)} \\
\texttt{y} & \doteq & \texttt{actuals(x+1)}
\end{array}
$$

An execution model is established by defining the $\texttt{call}_i$ and `actuals` in terms of operations on finite lists of natural numbers (referred from now on as *tags* or *contexts*). Execution of the program starts by demanding the value of the variable `result` of the intensional program, under the empty tag [ ]. The operator $\texttt{call}_i$ corresponds to the operation of prefixing a tag w with i. On the other hand, `actuals` corresponds to taking the head i of a tag, and using it to select its ith argument. More formally, given intensions $a, a_0, \ldots, a_{n-1}$, and letting ":" denote the consing operation on lists, the semantic equations as introduced in (Yaghi, 1984) are[†]:

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

Following the above semantic rules, the intensional program obtained above can be interpreted as follows:

[†] Notice that we use different fonts when representing elements of the metalanguage (eg. functions, sets, etc.) and elements of the object language (eg. programs, expressions, etc.). The exact typographic conventions we adopt are presented in section 4.

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

The technique just described has been extensively used in the implementations of the Lucid functional-dataflow language (Wadge & Ashcroft, 1985) as well as in other functional languages and systems (Du & Wadge, 1990a; Du & Wadge, 1990b).

In the following we will use a slight modification of the above technique, which will make easier the treatment of higher-order functions. For example consider the definition

$$\texttt{x} \;\doteq\; \texttt{actuals(4,5)}$$

in the above translated program. We will rewrite the definition as:

$$\texttt{x} \;\doteq\; \texttt{case(actuals}_0\texttt{(4),actuals}_1\texttt{(5))}$$

The new `case` operator we have introduced allows the use of a new family of `actuals` operators which correspond more closely to the `call` operator. The semantics of the new operators are as follows:

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

The above formalization is equivalent to the previous one, but has an additional benefit: each $\texttt{actuals}_i$ operator is now unary, as is the case with each $\texttt{call}_i$ operator. This will help us formulate in a more elegant way certain of the properties of the transformation algorithm for higher-order programs.

3 The Higher-Order Case

3.1 Introduction

The basic idea for the generalization of the technique to higher-order programs was first presented in (Wadge, 1991) and has since been extended and formalized in (Rondogiannis, 1994; Rondogiannis & Wadge, 1994a). Intuitively, the technique can handle higher-order programs in which:

1. Function names can be passed as parameters but not returned as results.
2. Operation symbols are first-order.

The above conditions certainly impose some restrictions on the use of higher-order functions. However, the acceptable class of programs still remains significant. Useful functional systems that make such use of higher-order functions can easily be implemented based on the algorithm proposed in this paper.

The first of the conditions imposed above is met by many widely used higher-order functions such as `map` and `foldr`. For example, the following is an acceptable program that defines and uses `map`:

```
result        ≐  map(inc,[1,2,3])
map(f,x:xs)   ≐  if (x eq []) then [] else f(x):map(f,xs)
inc(y)        ≐  y+1
```

Moreover, many functions that do not obey the first condition can be rewritten, after a preprocessing stage, in such a way that the condition is satisfied.

Notice that the language we adopt, though higher order, still uses conventional mathematical notation in which arguments of a function call appear as a comma-separated parenthesized list directly after the function name/expression. Therefore, we write `map(f,xs)` rather than the "Curried" form `(map f xs)` which is more usual in functional languages. We use the former notation because the latter is normally interpreted as denoting `((map f) xs)`, and this implies that `map` returns a function, which our language does not permit.

The second restriction is less important as the most common operation symbols (such as `+`, `if-then-else`, `eq`, etc.) are usually first-order.

In the rest of this section we give an intuitive introduction to the proposed transformation technique. The main idea of the generalized transformation is that an M-order functional program can first be transformed into an $(M-1)$-order intensional program, using a similar technique as the one for the first-order case. The same procedure can then be repeated for the new program, until we finally get a zero-order intensional program.

The idea of tags is now more general: for a program of order M, a tag is an M-sequence (a sequence of length M) of lists, where each list corresponds to a different order of the program. The operators are also more general as they have to manipulate the new, more complicated tags. As the transformation for the higher-order case consists of a number of stages, we use a different set of operators for each stage. For the first step we use the operators $\mathtt{case}^{M-1}$, $\mathtt{actuals}_i^{M-1}$ and $\mathtt{call}_i^{M-1}$, where i ranges as in the first-order case. For the second step we use $\mathtt{case}^{M-2}$, $\mathtt{actuals}_i^{M-2}$ and $\mathtt{call}_i^{M-2}$, and so on.

The code that results from the transformation can be executed following the same basic principles as in the first-order case. In the rest of this section we present at an intuitive level the transformation algorithm and describe the semantics of the generalized operators.

3.2 An Example Transformation

Consider the following simple second-order program:

$$\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{apply(inc,8)+apply(dec,5)} \\
\mathtt{apply(f,x)} & \doteq & \mathtt{f(x)} \\
\mathtt{inc(y)} & \doteq & \mathtt{y+1} \\
\mathtt{dec(a)} & \doteq & \mathtt{a-1}
\end{array}$$

The function `apply` is second-order because its first argument is first-order. The generalized transformation in its first stage eliminates the first argument of `apply`:

$$\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{call}_0^1\mathtt{(apply)(8)+call}_1^1\mathtt{(apply)(5)} \\
\mathtt{apply(x)} & \doteq & \mathtt{f(x)} \\
\mathtt{inc(y)} & \doteq & \mathtt{y+1} \\
\mathtt{dec(a)} & \doteq & \mathtt{a-1} \\
\mathtt{f} & \doteq & \mathtt{case}^1\mathtt{(actuals}_0^1\mathtt{(inc),actuals}_1^1\mathtt{(dec))}
\end{array}$$

A comment on the notation in the above program: $\mathtt{call}_0^1\mathtt{(apply)(8)}$ is equivalent to the more usual notation $\mathtt{call}_0^1\ \mathtt{apply}\ \mathtt{8}$ which stands for $\mathtt{((call}_0^1\ \mathtt{apply)}\ \mathtt{8)}$.

We see that the resulting program contains only first-order user-defined functions. The only exception is the definition of `f`, which is an equation between function expressions. We can easily change this by introducing an auxiliary variable `z`:

$$\begin{array}{lcl}
\mathtt{result} & \doteq & \mathtt{call}_0^1\mathtt{(apply)(8)+call}_1^1\mathtt{(apply)(5)} \\
\mathtt{apply(x)} & \doteq & \mathtt{f(x)} \\
\mathtt{inc(y)} & \doteq & \mathtt{y+1} \\
\mathtt{dec(a)} & \doteq & \mathtt{a-1} \\
\mathtt{f(z)} & \doteq & \mathtt{case}^1\mathtt{(actuals}_0^1\mathtt{(inc)(z),actuals}_1^1\mathtt{(dec)(z))}
\end{array}$$

Notice now that in the above program the functions are all first-order (they all have only zero-order arguments). A non-standard aspect of this new program is the existence of certain function calls of the form $\mathbf{q}\mathtt{(}\mathbf{f}\mathtt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\mathtt{)}$[‡], where $\mathbf{q}$ is an intensional operator (such as the calls $\mathtt{call}_0^1\mathtt{(apply)(8)}$, $\mathtt{call}_1^1\mathtt{(apply)(5)}$, $\mathtt{actuals}_0^1\mathtt{(inc)(z)}$ and $\mathtt{actuals}_1^1\mathtt{(dec)(z)}$). Such calls will receive a special treatment in the next step of the transformation.

We can now perform the final step of the transformation that will result in a zero-order intensional program. We proceed as before, the main difference being that we use a new dimension and corresponding new operators. Notice also below the use of the "·" syntactic composition operator, which is introduced for notational convenience (that will become clearer in later sections of the paper). In particular, an expression of the form $\mathbf{q}_1 \cdot \mathbf{q}_2\mathtt{(}\mathbf{f}\mathtt{)}$ will be considered equivalent to $\mathbf{q}_1\mathtt{(}\mathbf{q}_2\mathtt{(}\mathbf{f}\mathtt{))}$

‡ Again, this is notationally equivalent to $\mathtt{(}\mathbf{q}\ \mathbf{f}\mathtt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\mathtt{)}$, which corresponds, in the more usual notation for functional languages, to $\mathtt{(}\mathbf{q}\ \mathbf{f}\ \mathbf{E}_0\ \cdots\ \mathbf{E}_{n-1}\mathtt{)}$.

(notice that the composition operator binds stronger than function application).

$$
\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}^1_0 \cdot \texttt{call}^0_0\texttt{(apply)+call}^1_1 \cdot \texttt{call}^0_1\texttt{(apply)} \\
\texttt{apply} & \doteq & \texttt{call}^0_0\texttt{(f)} \\
\texttt{inc} & \doteq & \texttt{y+1} \\
\texttt{dec} & \doteq & \texttt{a-1} \\
\texttt{f} & \doteq & \texttt{case}^1\texttt{(actuals}^1_0 \cdot \texttt{call}^0_0\texttt{(inc),actuals}^1_1 \cdot \texttt{call}^0_0\texttt{(dec))} \\
\texttt{z} & \doteq & \texttt{case}^0\texttt{(actuals}^0_0\texttt{(x))} \\
\texttt{y} & \doteq & \texttt{case}^0\texttt{(actuals}^0_0 \cdot \texttt{call}^1_0\texttt{(z))} \\
\texttt{a} & \doteq & \texttt{case}^0\texttt{(actuals}^0_0 \cdot \texttt{call}^1_1\texttt{(z))} \\
\texttt{x} & \doteq & \texttt{case}^0\texttt{(actuals}^0_0 \cdot \texttt{actuals}^1_0\texttt{(8),actuals}^0_1 \cdot \texttt{actuals}^1_1\texttt{(5))}
\end{array}
$$

The transformation is similar to the one for the first-order case, the main difference being the treatment of calls of the form $\mathbf{q}\texttt{(}\mathbf{f}\texttt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\texttt{)}$. Consider for example the call $\texttt{call}^1_0\texttt{(apply)(8)}$ and notice the expression $\texttt{actuals}^0_0 \cdot \texttt{actuals}^1_0\texttt{(8)}$ that appears in the final program corresponding to the actual parameter `8`. The new aspect here is the appearance of the operator $\texttt{actuals}^1_0$, which we will call the *inverse* of the operator $\texttt{call}^1_0$ that existed in the initial call. In general, the inverse of $\texttt{call}^m_i$ is $\texttt{actuals}^m_i$ and vice-versa. As a second example, consider the call $\texttt{actuals}^1_0\texttt{(inc)(z)}$. The expression that results for the actual parameter `z` is $\texttt{actuals}^0_0 \cdot \texttt{call}^1_0\texttt{(z)}$, because $\texttt{call}^1_0$ is the inverse of $\texttt{actuals}^1_0$. The above notions will be formalized in subsequent sections (and will be generalized for the case of function calls of the form $\mathbf{Q}\texttt{(}\mathbf{f}\texttt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\texttt{)}$, where $\mathbf{Q}$ is the syntactic composition of a number of intensional operators).

The (informal) algorithm for the higher-order case consists of repeating the following steps until the program becomes zero-order. For each function **f** of the current highest order m:

1. Number the textual occurrences of calls to **f** in the program, starting at 0.
2. Remove from the ith call to **f** all the actual parameters of order $(m-1)$. Prefix the call to **f** with $\texttt{call}^{m-1}_i$.
3. Remove from the definition of **f** the formal parameters of order $(m-1)$.
4. For every formal parameter **x** of **f** that was eliminated, introduce a $\texttt{case}^{m-1}$ definition. The $\texttt{case}^{m-1}$ operator takes as many arguments as are the calls to **f** in the program. More specifically, the i'th argument of $\texttt{case}^{m-1}$ corresponds to the i'th call to **f** in the program, and is an expression starting with $\texttt{actuals}^{m-1}_i$. Moreover, if the particular call to **f** is of the form $\mathbf{Q}\texttt{(}\mathbf{f}\texttt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\texttt{)}$, where $\mathbf{Q}$ is the syntactic composition of a number of intensional operators, the inverse of $\mathbf{Q}$ must be taken into consideration when creating the subexpressions of $\texttt{case}^{m-1}$ (more details on this will be given in section 7).

Given a source functional program of order M, the execution model for the final zero-order program that results from the transformation requires tags to be M-sequences of lists of natural numbers, where each list corresponds to a different order of the initial program (or equivalently, a different stage in the transformation). We will use the notation $\langle w_0, ..., w_{M-1}\rangle$ to denote a tag. The operators $\texttt{call}^m_i$ and

$\texttt{actuals}_i^m$ can now be thought of as operations on these more complicated tags. The semantics of $\texttt{call}_i^m$ can be described as follows: given a tag, m is used in order to select the corresponding list from the tag. The list is then prefixed with i and returned to the tag. On the other hand, $\texttt{actuals}_i^m$ takes from the tag the list corresponding to m, checks whether the head of the list is equal to i and returns the tail of the list to the tag. The new semantic equations regarding the operators that appear in the final (zero-order) programs that result from the transformation, are[§]:

$$\begin{array}{lcl}
(call_i^m(a))\langle w_0,\ldots,w_m,\ldots,w_{M-1}\rangle & = & a\langle w_0,\ldots,(i:w_m),\ldots,w_{M-1}\rangle \\
(actuals_i^m(a))\langle w_0,\ldots,(i:w_m),\ldots,w_{M-1}\rangle & = & a\langle w_0,\ldots,w_m,\ldots,w_{M-1}\rangle \\
(case^m(a_0,\ldots,a_{n-1}))\langle w_0,\ldots,(i:w_m),\ldots,w_{M-1}\rangle & = & a_i\langle w_0,\ldots,(i:w_m),\ldots,w_{M-1}\rangle
\end{array}$$

Notice that in the case of the $actuals_i^m$ operator, the semantic equation does not specify what happens if the check made by the operator fails. The result in this case is undefined. However, the test performed by $actuals_i^m$ never fails in the case of programs generated by the transformation.

The evaluation of a program starts with an M-sequence of empty lists, one for each order. Execution proceeds as in the first-order case, the only difference being that the appropriate list within the sequence is accessed every time.

The final zero-order programs that result from the transformation can be easily executed using an *EVAL* function (as this was done in the first order case). The *EVAL* function is not defined formally in the paper because it is used in order to introduce and illustrate ideas that will be formalized in other parts of the paper. One can think of *EVAL* as a simple interpreter that works by following the semantic equations of the intensional operators given above. Moreover, *EVAL* also performs a simple form of substitution: every time it needs to evaluate a nullary variable of the program under a specific context, it simply replaces the variable with its defining expression, and continues the evaluation. It is important to note again that *EVAL* can not be used to compute the value of a program that results in an intermediate step of the transformation, because such programs still contain user-defined functions. The final programs are much simpler, and the semantics of the operators described earlier in this section suffice in order to compute the program output.

Execution starts by demanding the value of `result` at the empty tag:

$$\begin{array}{ll}
 & EVAL(\texttt{result},\ \langle[\,],[\,]\rangle) \\
 & EVAL(\texttt{call}_0^1\cdot\texttt{call}_0^0\texttt{(apply)+call}_1^1\cdot\texttt{call}_1^0\texttt{(apply)},\ \langle[\,],[\,]\rangle) \\
= & EVAL(\texttt{call}_0^1\texttt{(call}_0^0\texttt{(apply))+call}_1^1\texttt{(call}_1^0\texttt{(apply))},\ \langle[\,],[\,]\rangle)
\end{array}$$

This can be calculated by computing independently (and then adding) the following two results:

$$EVAL(\texttt{call}_0^1\texttt{(call}_0^0\texttt{(apply))},\ \langle[\,],[\,]\rangle)$$

[§] The intensional operators that appear in intermediate steps of the transformation are generally higher-order. The semantics of such operators is described in Section 6.

and

$$EVAL(\texttt{call}^1_1\texttt{(call}^0_1\texttt{(apply))},\ \langle[\,],[\,]\rangle)$$

The two calculations are very similar in nature. We show the full procedure only for the second one (the first one can be easily shown to yield the value 9):

$$\begin{aligned}
& EVAL(\texttt{call}^1_1\texttt{(call}^0_1\texttt{(apply))},\ \langle[\,],[\,]\rangle)\\
= \;& EVAL(\texttt{call}^0_1\texttt{(apply)},\ \langle[\,],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{apply},\ \langle[\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{call}^0_0\texttt{(f)},\ \langle[\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{f},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{case}^1\texttt{(actuals}^1_0\cdot\texttt{call}^0_0\texttt{(inc),actuals}^1_1\cdot\texttt{call}^0_0\texttt{(dec))},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{actuals}^1_1\cdot\texttt{call}^0_0\texttt{(dec)},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{actuals}^1_1\texttt{(call}^0_0\texttt{(dec))},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)\\
= \;& EVAL(\texttt{call}^0_0\texttt{(dec)},\ \langle[\mathit{0},\mathit{1}],[\,]\rangle)\\
= \;& EVAL(\texttt{dec},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)\\
= \;& EVAL(\texttt{a-1},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)\\
= \;& EVAL(\texttt{a},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)-EVAL(\texttt{1},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)\\
= \;& EVAL(\texttt{a},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{case}^0\texttt{(actuals}^0_0\cdot\texttt{call}^1_1\texttt{(z))},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^0_0\cdot\texttt{call}^1_1\texttt{(z)},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^0_0\texttt{(call}^1_1\texttt{(z))},\ \langle[\mathit{0},\mathit{0},\mathit{1}],[\,]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{call}^1_1\texttt{(z)},\ \langle[\mathit{0},\mathit{1}],[\,]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{z},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{case}^0\texttt{(actuals}^0_0\texttt{(x))},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^0_0\texttt{(x)},\ \langle[\mathit{0},\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{x},\ \langle[\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{case}^0\texttt{(actuals}^0_0\cdot\texttt{actuals}^1_0\texttt{(8),actuals}^0_1\cdot\texttt{actuals}^1_1\texttt{(5))},\ \langle[\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^0_1\cdot\texttt{actuals}^1_1\texttt{(5)},\ \langle[\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^0_1\texttt{(actuals}^1_1\texttt{(5))},\ \langle[\mathit{1}],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{actuals}^1_1\texttt{(5)},\ \langle[\,],[\mathit{1}]\rangle)-\mathit{1}\\
= \;& EVAL(\texttt{5},\ \langle[\,],[\,]\rangle)-\mathit{1}\\
= \;& \texttt{5}-\texttt{1}\\
= \;& \texttt{4}
\end{aligned}$$

Therefore, the final result of the calculation will be the sum of the results of the two subcomputations, which is $9+4=13$. Notice that although the above calculation seems relatively lengthy, each operation that takes place at each step is primitive and can be performed very efficiently. Moreover, it should be noted that one could easily devise certain simple intensional transformations-optimizations that would enhance the performance of the produced code.

3.3 An Example Involving Recursion

Consider the following recursive second-order program which calculates a *function factorial*:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{ffac(sq,2)} \\
\texttt{ffac(h,n)} & \doteq & \texttt{if (n<=1) then 1 else h(n)*ffac(h,n-1)} \\
\texttt{sq(a)} & \doteq & \texttt{a} * \texttt{a}
\end{array}$$

We perform the first step of the algorithm as before, getting:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_0^1\texttt{(ffac)(2)} \\
\texttt{ffac(n)} & \doteq & \texttt{if (n<=1) then 1 else h(n)*call}_1^1\texttt{(ffac)(n-1)} \\
\texttt{h} & \doteq & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(sq),actuals}_1^1\texttt{(h))} \\
\texttt{sq(a)} & \doteq & \texttt{a} * \texttt{a}
\end{array}$$

Adding a variable `z` to both sides of the definition of `h`, we get the following first-order intensional program:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_0^1\texttt{(ffac)(2)} \\
\texttt{ffac(n)} & \doteq & \texttt{if (n<=1) then 1 else h(n)*call}_1^1\texttt{(ffac)(n-1)} \\
\texttt{h(z)} & \doteq & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(sq)(z),actuals}_1^1\texttt{(h)(z))} \\
\texttt{sq(a)} & \doteq & \texttt{a} * \texttt{a}
\end{array}$$

We can now continue the transformation, getting the following zero-order intensional program:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_0^1 \cdot \texttt{call}_0^0\texttt{(ffac)} \\
\texttt{ffac} & \doteq & \texttt{if (n<=1) then 1 else call}_0^0\texttt{(h)*call}_1^1 \cdot \texttt{call}_1^0\texttt{(ffac)} \\
\texttt{h} & \doteq & \texttt{case}^1\texttt{(actuals}_0^1 \cdot \texttt{call}_0^0\texttt{(sq),actuals}_1^1 \cdot \texttt{call}_1^0\texttt{(h))} \\
\texttt{sq} & \doteq & \texttt{a} * \texttt{a} \\
\texttt{n} & \doteq & \texttt{case}^0\texttt{(actuals}_0^0 \cdot \texttt{actuals}_0^1\texttt{(2),actuals}_1^0 \cdot \texttt{actuals}_1^1\texttt{(n-1))} \\
\texttt{z} & \doteq & \texttt{case}^0\texttt{(actuals}_0^0\texttt{(n),actuals}_1^0 \cdot \texttt{call}_1^1\texttt{(z))} \\
\texttt{a} & \doteq & \texttt{case}^0\texttt{(actuals}_0^0 \cdot \texttt{call}_0^1\texttt{(z))}
\end{array}$$

The output value of the above program can be easily computed as before (using the semantic rules of the intensional operators).

4 Mathematical Notation

The set of natural numbers is denoted by N. The set of functions from A to B is denoted by $A \to B$. The result of applying function f to argument a will be written $f(a)$ or, in special cases f_a. Our metalanguage is therefore consistent with our object language in that it uses conventional mathematical notation for function application; even in the metalanguage we will strictly avoid "Currying".

We will use a metalinguistics composition operator "$\cdot$" analogous to that found in the object language.

For notational simplicity, we usually denote a tuple $\langle z_0, z_1, \ldots, z_{n-1}\rangle$ by $\vec{z}$. The following generalization of set products is adopted: 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 4.1
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 $g = \{\langle x_0, b_0 \rangle, \ldots, \langle x_{n-1}, b_{n-1} \rangle\}$, we will often use the alternative notation $f[x_0/b_0, \ldots, x_{n-1}/b_{n-1}]$ 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., $\mathtt{f}, \mathtt{x}, \ldots$). 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., $f, x, \mathcal{E}, \mathcal{A}, \ldots$), and those that are used in order to talk about the syntax of the object language, for which we adopt the boldface font (e.g., $\mathbf{f}, \mathbf{x}, \mathbf{P}, \mathbf{E}, \ldots$).

In recent years, a significant progress has been made in enriching programming languages with a wide range of data types. Types impose *a priori* syntactic constraints on what constructs of a language can be combined, helping in this way the programmer to avoid writing meaningless or erroneous code. In this section, we define the syntax and semantics of the types that are adopted for the purposes of this paper.

Definition 4.2
The set $STyp$ of simple types *and the set Typ of types, are ranged over by σ and τ respectively and are recursively defined as follows:*

$$\begin{array}{rcl} \sigma & ::= & \iota \\ & \mid & (\sigma_0, \ldots, \sigma_{n-1}) \to \iota \\ \tau & ::= & \sigma \\ & \mid & \sigma \to \sigma \end{array}$$

The type ι is a single ground type. Notice that the result component of a member of $STyp$ is always ground, that is ι. As it will be described shortly, the languages considered in this paper are subject to this restriction, in the sense that all functions defined in them should have a type that belongs to *STyp*. On the other hand, the languages we are considering will have intensional operators (the operators `call` and `actuals`) with types of the form $\sigma \rightarrow \sigma$.

Definition 4.3
The order *of a simple type is recursively defined as follows:*

$$\begin{array}{rcl} order(\iota) & = & 0 \\ order((\sigma_0,\ldots,\sigma_{n-1}) \rightarrow \iota) & = & 1 + max(\{order(\sigma_i) \mid 0 \leq i \leq n-1\}) \end{array}$$

Definition 4.4
The denotation of a type with respect to a given domain D is recursively defined by the function $[\![\cdot]\!]_D$ (where the subscript D will often be omitted) as follows:

- $[\![\iota]\!]_D = D$
- $[\![(\sigma_0,\ldots,\sigma_{n-1}) \rightarrow \iota]\!]_D = [([\![\sigma_0]\!]_D,\ldots,[\![\sigma_{n-1}]\!]_D) \rightarrow [\![\iota]\!]_D]$
- $[\![\sigma_1 \rightarrow \sigma_2]\!]_D = [[\![\sigma_1]\!]_D \rightarrow [\![\sigma_2]\!]_D]$

We assume the existence of a set Σ of constant symbols of various types over *STyp*. Elements of Σ are assigned types by a type assignment function $\theta : \Sigma \rightarrow$ *STyp*. Constants are denoted by $\mathbf{c}$. We also assume the existence of a set *Var* of variable symbols of various types over *STyp*. Elements of *Var* are assigned types by a type assignment function π : *Var* $\rightarrow$ *STyp*. Variables are denoted by $\mathbf{f}, \mathbf{g}, \mathbf{x}, \ldots$. In particular, we use Var_0 to denote the variable symbols of type ι. Variable (constant) symbols of type ι are also called *nullary* or *individual* variables (constants). Non-nullary variables are also termed *function* variables.

5 The Higher-Order Functional Language *FL*

In this section, we define the syntax and denotational semantics of the typed, higher-order functional language *FL*.

Definition 5.1
The syntax of the functional language FL over Σ 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*:*

$$\begin{array}{rcl} \mathbf{E} & ::= & \mathbf{f} \in Var \\ & \mid & \mathbf{c}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}),\ \mathbf{c} \in \Sigma,\ n \geq 0 \\ & \mid & \mathbf{f}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}),\ \mathbf{f} \in Var,\ n \geq 0 \\ \mathbf{F} & ::= & \mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{E},\ \mathbf{f}, \mathbf{x}_i \in Var,\ n \geq 0 \\ \mathbf{P} & ::= & \{\mathbf{F}_0,\ldots,\mathbf{F}_{n-1}\} \end{array}$$

Notice that the syntax allows for nullary constants (when $n = 0$), which in this case will be written as $\mathbf{c}()$ or just $\mathbf{c}$. The syntax also allows for nullary variables as well as for nullary variable definitions. In both of these cases the empty pair of parentheses that follow the variable symbol will often be omitted. Notice also that the syntax of FL presented above will be further restricted by the well-typing rules that are introduced later on in this section.

Given a definition $\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{E}$, the variables $\mathbf{x}_i$ are the *formal parameters* or *formals* of $\mathbf{f}$, and $\mathbf{E}$ is the *defining expression* or the *body* of $\mathbf{f}$.

Definition 5.2
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 individual 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, exactly 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.*

The set of variables defined in a program $\mathbf{P}$ is denoted by $func(\mathbf{P})$. The type-checking rules for the language are given as natural deduction rules with sequents of the form $\mathbf{E} : \sigma$. The sequent $\mathbf{E} : \sigma$ asserts that $\mathbf{E}$ is a well-formed expression of type σ and that the identifiers and constants that are used in $\mathbf{E}$ have the types assigned to them by π and θ respectively. Notice that the following definition reflects the two restrictions mentioned in the beginning of subsection 3.1 (ie., that function names can be passed as parameters but not returned as results, and that operation symbols are first-order).

Definition 5.3
The set of well-typed expressions is recursively defined as follows:

$$\frac{\pi(\mathbf{f}) = \sigma}{\mathbf{f} : \sigma}$$

$$\frac{(\theta(\mathbf{c}) = (\iota,\ldots,\iota) \to \iota) \wedge (\forall i \in \{0,\ldots,n-1\}\ (\mathbf{E}_i : \iota))}{\mathbf{c}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}) : \iota}$$

$$\frac{(\mathbf{f} : (\sigma_0,\ldots,\sigma_{n-1}) \to \iota) \wedge (\forall i \in \{0,\ldots,n-1\}\ (\mathbf{E}_i : \sigma_i))}{\mathbf{f}(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}) : \iota}$$

Definition 5.4
A definition $\mathbf{f}(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{E}$ with $\mathbf{f} : (\sigma_0,\ldots,\sigma_{n-1}) \to \iota$ is well-typed if $\mathbf{E} : \iota$ and for all $i \in \{0,\ldots,n-1\}$, $\mathbf{x}_i : \sigma_i$.

Definition 5.5
A program $\{\mathbf{F}_0,\ldots,\mathbf{F}_{n-1}\}$ is well-typed if $\mathbf{F}_0,\ldots,\mathbf{F}_{n-1}$ are well-typed definitions.

In the following, we will often talk about zero-order programs, first-order programs, and so on. The following definition formalizes the above notions:

Definition 5.6

Let $\mathbf{P}$ *be an FL program. The order of* $\mathbf{P}$ *is defined as:*

$$Order(\mathbf{P}) = max(\{order(\pi(\mathbf{f})) \mid \mathbf{f} \in func(\mathbf{P})\})$$

Let D be a given domain. The semantics of constant symbols of FL with respect to D are specified by a given interpretation function $\mathcal{C}$, which assigns to every constant of type σ, a value in $[\![\sigma]\!]_D$. Let Exp_σ be the set of all expressions $\mathbf{E}$ of FL such that $\mathbf{E} : \sigma$. Let Env_π be the set of π-compatible environments, ie., those environments which for all $\mathbf{f} \in Var$, $u(\mathbf{f}) \in [\![\pi(\mathbf{f})]\!]_D$. This set is defined by $Env_\pi = \prod_{\mathbf{f} \in Var} [\![\pi(\mathbf{f})]\!]_D$. Then, the semantics of FL are defined using valuation functions $[\![\cdot]\!]_D^\sigma : Exp_\sigma \to [Env_\pi \to [\![\sigma]\!]_D]$, (where the subscripts and superscripts D, σ and π will be omitted when they are obvious from context).

Definition 5.7
The semantics of expressions of FL with respect to $u \in Env$*, are recursively defined as follows:*

$$\begin{array}{l}
[\![\mathbf{f}]\!](u) = u(\mathbf{f}) \\
[\![\mathbf{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!](u) = \mathcal{C}(\mathbf{c})([\![\mathbf{E}_0]\!](u), \ldots, [\![\mathbf{E}_{n-1}]\!](u)) \\
[\![\mathbf{f}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!](u) = u(\mathbf{f})([\![\mathbf{E}_0]\!](u), \ldots, [\![\mathbf{E}_{n-1}]\!](u))
\end{array}$$

It should be noted that $[\![\cdot]\!]$ as defined above is actually overloaded. A more accurate definition would involve the use of the $[\![\cdot]\!]_D^\sigma$'s, used for different expression types. For clarity reasons we have avoided making the notation more complicated.

Theorem 5.1
(Tennent, 1991, page 97) For all expressions $\mathbf{E}$, $[\![\mathbf{E}]\!]$ *is continuous and therefore monotonic.*

Definition 5.8
The semantics of the program $\mathbf{P} = \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}$ *of FL is defined as* $\widetilde{u}(\mathbf{result})$*, where* $\widetilde{u}$ *is the* least *environment such that for every* $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{E}$ *in* $\mathbf{P}$ *with* $\mathbf{f} : (\sigma_0, \ldots, \sigma_{n-1}) \to \iota$*, and for all* $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$*,*
$\widetilde{u}(\mathbf{f})(d_0, \ldots, d_{n-1}) = [\![\mathbf{E}]\!](\widetilde{u}[\mathbf{x}_0/d_0, \ldots, \mathbf{x}_{n-1}/d_{n-1}])$.

The above definition does not specify how the least environment $\widetilde{u}$ can be constructed. The following theorem states that $\widetilde{u}$ is the least upper bound of a chain of environments, which can be thought of as successive approximations to $\widetilde{u}$.

Theorem 5.2
(Tennent, 1991, page 96) Let $\mathbf{P}$ *and* $\widetilde{u}$ *be as in Definition 5.8. Then,* $\widetilde{u}$ *is the least upper bound of the environments* $\widetilde{u}_k$ *which are defined as follows:*

1. *For every* $\mathbf{f} \in Var$ *with* $\mathbf{f} \notin func(\mathbf{P})$, $\widetilde{u}_k(\mathbf{f}) = \perp$*, for all* $k \in N$*.*
2. *For every* $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{E}$ *in* $\mathbf{P}$*, with* $\mathbf{f} : (\sigma_0, \ldots, \sigma_{n-1}) \to \iota$*, and for all* $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$*,*

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

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

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

Lemma 5.1
Let $\mathbf{P}$ *and* $\widetilde{u}$ *be as in Definition 5.8. Then, for every* $\mathbf{f}\texttt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\texttt{)} \doteq \mathbf{E}$ *in* $\mathbf{P}$ *with* $\mathbf{f} : (\sigma_0,\ldots,\sigma_{n-1}) \to \iota$,

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

6 The Higher-Order Intensional Languages IL and $NVIL$

In this section we define the syntax of the intensional languages IL and $NVIL$ that are used in the transformation algorithm (IL stands for *intensional language* and $NVIL$ for *nullary variable intensional language*). The language IL is a higher-order intensional one; programs that appear in intermediate steps of the transformation belong to IL. On the other hand, the final zero-order programs that result from the transformation belong to the intensional language $NVIL$, which is simpler in structure than IL (and is introduced independently).

The difference between IL and FL is the presence of intensional operators. Due to the nature of the transformation, intensional operators in programs of IL appear in a specific way. Consider for example the following program obtained after the first step in the transformation algorithm (section 3):

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_0^1\texttt{(apply)(8)+call}_1^1\texttt{(apply)(5)} \\
\texttt{apply(x)} & \doteq & \texttt{f(x)} \\
\texttt{inc(y)} & \doteq & \texttt{y+1} \\
\texttt{dec(a)} & \doteq & \texttt{a-1} \\
\texttt{f(z)} & \doteq & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(inc)(z),actuals}_1^1\texttt{(dec)(z))}
\end{array}$$

If we examine the function calls in the program, we realize that some of them are of the form $\mathbf{q}\texttt{(}\mathbf{f}\texttt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\texttt{)}$. In general, function calls that appear in intermediate programs of the transformation will have the form $\mathbf{Q}\texttt{(}\mathbf{f}\texttt{)(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\texttt{)}$, where $\mathbf{Q}$ is a (possibly empty) sequence of the syntactic representations of intensional operators composed with each other.

The final (zero-order) programs that constitute the output of the transformation, have a simpler syntax than the programs that appear in the intermediate steps of the algorithm. Moreover, in these output programs, intensional operators can be applied not only to function variables but also to other expressions, something that does not happen in the intermediate programs of the transformation (as an example, consider the expression $\texttt{actuals}_0^0 \cdot \texttt{actuals}_0^1\texttt{(8)}$ in the final program of the first example in section 3). For these reasons, we independently define below the syntax of the two languages IL and $NVIL$.

The following definition formalizes the syntax of sequences of the syntactic representations of intensional operators. Notice that in the following, ϵ denotes the empty sequence.

Definition 6.1
The set $ISeq$ *of sequences of the syntactic representations of intensional operators*

is ranged over by $\mathbf{Q}$ *and is recursively defined as follows:*

$$
\begin{array}{lll}
\mathbf{Q} & ::= & \epsilon \\
 & | & \mathtt{call}_i^m,\ i, m \geq 0 \\
 & | & \mathtt{actuals}_i^m,\ i, m \geq 0 \\
 & | & \mathtt{call}_i^m \cdot \mathbf{Q},\ i, m \geq 0, \mathbf{Q} \neq \epsilon \\
 & | & \mathtt{actuals}_i^m \cdot \mathbf{Q},\ i, m \geq 0, \mathbf{Q} \neq \epsilon
\end{array}
$$

Notice that the elements of the set $ISeq$ are syntactic objects: they are sequences of the textual representations of intensional operators (i.e., sequences of elements of the object language) syntactically composed with each other. To every $\mathbf{Q} \in ISeq$ there corresponds a function Q which is the composition of the meanings of the elements of the sequence (this will be further discussed in subsection 6.1).

Taking into consideration the above remarks, we have the following definition concerning the syntax of the intensional language IL:

Definition 6.2

The syntax of the intensional language IL *is recursively defined by the following rules, in which* $\mathbf{E}, \mathbf{E}_i$ *denote* expressions*,* $\mathbf{B}$ *denotes a* body expression *of a definition,* $\mathbf{F}, \mathbf{F}_i$ *denote* definitions *and* $\mathbf{P}$ *denotes a* program*:*

$$
\begin{array}{lll}
\mathbf{E} & ::= & \mathbf{f} \in \mathit{Var} \\
 & | & \mathbf{c}\,\mathtt{(}\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\mathtt{)},\ \mathbf{c} \in \Sigma,\ n \geq 0 \\
 & | & \mathbf{Q}\,\mathtt{(}\mathbf{f}\mathtt{)}\,\mathtt{(}\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\mathtt{)},\ \mathbf{f} \in \mathit{Var},\ \mathbf{Q} \in \mathit{ISeq},\ n \geq \mathit{0} \\
\mathbf{B} & ::= & \mathbf{E} \\
 & | & \mathtt{case}^m\,\mathtt{(}\mathbf{E}_0, \ldots, \mathbf{E}_{r-1}\mathtt{)},\ r, m \geq 0 \\
\mathbf{F} & ::= & \mathbf{f}\,\mathtt{(}\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B},\ \mathbf{f}, \mathbf{x}_i \in \mathit{Var},\ n \geq \mathit{0} \\
\mathbf{P} & ::= & \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}
\end{array}
$$

Notice that ordinary function calls are allowed by the above syntax when $\mathbf{Q}$ is equal to the empty sequence. In such a case, the parentheses around $\mathbf{f}$ will be omitted.

The notions of well-typed definitions, well-typed programs and order of a program, are identical to the ones introduced in Definitions 5.4, 5.5 and 5.6. Moreover, the same assumptions as in Definition 5.2 are adopted for IL programs.

The final zero-order programs that result from the transformation, belong to the language $NVIL$. The syntax of $NVIL$ is defined below:

Definition 6.3

The syntax of the intensional language $NVIL$ *is recursively defined by the following rules, in which* $\mathbf{E}, \mathbf{E}_i$ *denote* expressions*,* $\mathbf{B}$ *denotes a* body expression*,* $\mathbf{F}, \mathbf{F}_i$ *denote* definitions *and* $\mathbf{P}$ *denotes a* program*:*

$$\begin{array}{lcl}
\mathbf{E} & ::= & \mathbf{f} \in \mathit{Var_0} \\
 & \mid & \mathtt{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}), \ \mathbf{c} \in \Sigma, \ n \geq 0 \\
 & \mid & \mathbf{Q}(\mathbf{E}), \ \mathbf{Q} \in \mathit{ISeq} \\
\mathbf{B} & ::= & \mathbf{E} \\
 & \mid & \mathtt{case}^m(\mathbf{E}_0, \ldots, \mathbf{E}_{r-1}), \ r, m \geq 0 \\
\mathbf{F} & ::= & \mathbf{f} \doteq \mathbf{B}, \ \mathbf{f} \in \mathit{Var_0} \\
\mathbf{P} & ::= & \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}
\end{array}$$

Notice that the language $NVIL$ is similar to the one defined in (Rondogiannis & Wadge, 1997), the only difference being that the operators are now multidimensional.

6.1 The Intensional Languages IL and NVIL: Synchronic Semantics

In this section we define the denotational semantics of the intensional languages IL and $NVIL$. The set of possible worlds in both languages is the set of infinite sequences of lists of natural numbers, that is $N \to \mathit{List}(N)$. Notice that, as we discussed in Section 3, for the transformation of an M-order functional program, contexts need only be M-sequences of lists of natural numbers. However, we would like the semantics to be defined in the most general way and be applicable to all programs no matter what their order is. Moreover, there is nothing to be lost by assuming that contexts are infinite sequences of lists, because in any particular translation only a finite number of the lists will be used. Therefore:

Definition 6.4
The set W of possible worlds of IL and $NVIL$ is the set $N \to \mathit{List}(N)$.

Given the above set W of possible worlds, we can define the set of possible denotations of a type σ, as follows:

Definition 6.5
Let D be a domain. The set of possible denotations of $\sigma \in STyp$ with respect to W and D is defined as

$$[\![\sigma]\!]^*_D = W \to [\![\sigma]\!]_D$$

In other words, in IL the elements of type σ are W-indexed families of "conventional" type σ functions over D; they are *not* conventional type σ functions over $W \to D$ (a much more complex domain). In defining the semantics of IL and $NVIL$, we follow the approach that has been used by Montague for giving semantics to higher-order intensional logic (Dowty *et al.*, 1981; Gallin, 1975). As this approach differs from the standard techniques used for assigning denotational semantics to functional languages, we will refer to it as the *synchronic* interpretation for reasons to be given shortly.

Let D be a given domain. Then, the semantics of constant symbols of IL (and $NVIL$) with respect to D, are given by an interpretation function $\mathcal{C}^*$, which assigns to every constant of type σ a function in $[\![\sigma]\!]^*_D$. As the languages IL and $NVIL$ will be used in the transformation process of FL programs, the function $\mathcal{C}^*$ is defined in terms of the interpretation function $\mathcal{C}$ for FL. More specifically:

Definition 6.6
For every $\mathbf{c} \in \Sigma$ *and for every* $w \in W$, $\mathcal{C}^*(\mathbf{c})(w) = \mathcal{C}(\mathbf{c})$.

The semantics of the intensional operators of the languages IL and $NVIL$ are given by the following definition:

Definition 6.7
Let $w \in (N \to List(N))$ *and* $a, a_0, \ldots, a_{n-1} \in [\![\sigma]\!]^*$. *The semantics of the intensional operators* call, actuals *and* case *are defined as follows:*

$$\begin{array}{lcl} call_i^m(a)(w) & = & a(w[m/(i : w_m)]) \\ actuals_i^m(a)(w) & = & \left\{ \begin{array}{ll} a(w[m/tail(w_m)]) & \text{if } head(w_m) = i \\ undefined & otherwise \end{array} \right. \\ case^m(a_0, \ldots, a_{n-1})(w) & = & a_{head(w_m)}(w) \end{array}$$

Given a sequence $\mathbf{Q} \in ISeq$, we can define the meaning of $\mathbf{Q}$ as the composition of the meanings of the intensional operators that constitute $\mathbf{Q}$. We will denote by Q the meaning of the sequence $\mathbf{Q}$. Similarly, we will often write $\mathbf{q}$ when referring to the syntactic representation of a single intensional operator, and q for the meaning of $\mathbf{q}$.

We can now proceed to define the semantics of expressions of IL. Let Exp_τ be the set of all expressions $\mathbf{B}$ of IL such that $\mathbf{B} : \sigma$. Let Env_π^* be the set of π-compatible synchronic environments defined by $Env_\pi^* = \prod_{\mathbf{f} \in Var} [\![\pi(\mathbf{f})]\!]_D^*$. Then, the synchronic semantics of the language IL is defined using valuation functions $[\![\cdot]\!]^* : Exp_\sigma \to [Env_\pi^* \to [\![\sigma]\!]_D^*]$, as follows:

Definition 6.8
The synchronic *interpretation of expressions of* IL *with respect to* $u \in Env_\pi^*$, *is recursively defined for every* $w \in W$, *as follows:*

$$\begin{array}{l} [\![\mathbf{f}]\!]^*(u)(w) = u(\mathbf{f})(w) \\ [\![\mathtt{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!]^*(u)(w) = \mathcal{C}^*(\mathbf{c})(w)([\![\mathbf{E}_0]\!]^*(u)(w), \ldots, [\![\mathbf{E}_{n-1}]\!]^*(u)(w)) \\ [\![\mathbf{Q}(\mathbf{f})(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!]^*(u)(w) = Q(u(\mathbf{f}))(w)([\![\mathbf{E}_0]\!]^*(u)(w), \ldots, [\![\mathbf{E}_{n-1}]\!]^*(u)(w)) \\ [\![\mathtt{case}^m(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})]\!]^*(u)(w) = case^m([\![\mathbf{E}_0]\!]^*(u), \ldots, [\![\mathbf{E}_{n-1}]\!]^*(u))(w) \end{array}$$

It can be seen from the above definition that the semantic equation for application is non-standard; it involves an individual "sampling" of the meanings of the subexpressions under the current context w.

The basic principle is that the value of (say) $\mathbf{f}(\mathbf{E})$ at world w is the value of $\mathbf{f}$ at world w applied to the value of $\mathbf{E}$ at the same world w - application is defined in a pointwise way. If we think of w as some (very) general "timepoint", we see that the value of $\mathbf{f}(\mathbf{E})$ at "time" w depends on the value of $\mathbf{E}$ at the same time w. Hence the name "synchronic" adopted for this interpretation.

In the case of $NVIL$, we also have the following semantic equation:

$$[\![\mathbf{Q}(\mathbf{E})]\!]^*(u)(w) = Q([\![\mathbf{E}]\!]^*(u))(w)$$

Before we introduce the semantics of programs, the following definition is necessary:

Definition 6.9
Let $d \in [\![\sigma]\!]_D$. Then, d^∞ is that function on W whose value at every $w \in W$ is equal to d.

We can now introduce the semantics of IL. Notice that the following definitions and theorems also apply to $NVIL$ programs (the difference being that $NVIL$ programs allow only nullary variable definitions).

Definition 6.10
The synchronic semantics of a program $\mathbf{P} = \{\mathbf{F}_0, \ldots, \mathbf{F}_{n-1}\}$ of IL (or $NVIL$) is defined as $\widetilde{u}(\mathbf{result})$, where $\widetilde{u}$ is the least *environment such that for every definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$ with $\mathbf{f} : (\sigma_0, \ldots, \sigma_{n-1}) \to \iota$, for all $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$, and all $w \in W$,*
$\widetilde{u}(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) = [\![\mathbf{B}]\!]^*(\widetilde{u}[\mathbf{x}_0/d_0^\infty, \ldots, \mathbf{x}_{n-1}/d_{n-1}^\infty])(w)$.

The above definition does not specify how the least environment $\widetilde{u}$ can be constructed. The following theorem states that $\widetilde{u}$ is the least upper bound of a chain of environments which can be thought as successive approximations to $\widetilde{u}$.

Theorem 6.1
Let $\mathbf{P}$ and $\widetilde{u}$ be as in Definition 6.10. Then, $\widetilde{u}$ is the least upper bound of the environments $\widetilde{u}_k$, $k \in N$, which are defined as follows:

1. *For every $\mathbf{f} \in Var$ with $\mathbf{f} \notin func(\mathbf{P})$, for all $w \in W$, $\widetilde{u}_k(\mathbf{f})(w) = \bot$, for all $k \in N$.*
2. *For every $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$, with $\mathbf{f} : (\sigma_0, \ldots, \sigma_{n-1}) \to \iota$, for all $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$, and all $w \in W$,*

$$\begin{aligned}
\widetilde{u}_0(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) &= \bot \\
\widetilde{u}_{k+1}(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) &= [\![\mathbf{B}]\!]^*(\widetilde{u}_k[\mathbf{x}_0/d_0^\infty, \ldots, \mathbf{x}_{n-1}/d_{n-1}^\infty])(w)
\end{aligned}$$

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

Proof
Analogous to the proof of Theorem 5.2. □

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

Lemma 6.1
Let $\mathbf{P}$ and $\widetilde{u}$ be as in Definition 6.10. Then, for every $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B}$ in $\mathbf{P}$ with $\mathbf{f} : (\sigma_0, \ldots, \sigma_{n-1}) \to \iota$, for all $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$ and for all $w \in W$,

$$\widetilde{u}_k(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) \sqsubseteq [\![\mathbf{B}]\!]^*(\widetilde{u}_k[\mathbf{x}_0/d_0^\infty, \ldots, \mathbf{x}_{n-1}/d_{n-1}^\infty])(w)$$

The following theorem will also be used in subsequent sections:

Theorem 6.2
For all expressions $\mathbf{B} \in Exp_\sigma$, $[\![\mathbf{B}]\!]^*$ is monotonic and continuous. Moreover, when $\sigma \neq \iota$, $[\![\mathbf{B}]\!]^*(u)(w)$ is monotonic and continuous, for all $u \in Env_\pi^*$ and $w \in W$.

6.2 Properties of the Synchronic Interpretation

In this subsection we investigate certain of the properties of the synchronic interpretation. Initially, we consider those programs of IL that do not contain any of the operators `call`, `actuals` and `case`. Notice that programs of this subset are actually FL programs, for which we have already defined a standard denotational interpretation (see Definition 5.8). The following theorem establishes the relationship between the standard and the synchronic semantics for programs of the above subset:

Theorem 6.3
Let $\mathbf{P}$ *be an* IL *program that does not contain the operators* `call`*,* `actuals` *and* `case`*, and let* u *and* $\widehat{u}$ *be the least environments that satisfy the definitions in* $\mathbf{P}$ *under the standard and the synchronic interpretations respectively. Then, for every* $w \in W$, $[\![\mathbf{P}]\!]^*(\widehat{u})(w) = [\![\mathbf{P}]\!](u)$.

Proof
(Outline) It suffices to show that for every definition $\mathbf{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B_f}$ in $\mathbf{P}$, with $\mathbf{x}_0 : \sigma_0, \ldots, \mathbf{x}_{n-1} : \sigma_{n-1}$,

$$\widehat{u}(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) = u(\mathbf{f})(d_0, \ldots, d_{n-1})$$

for all $d_0 \in [\![\sigma_0]\!]_D, \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]_D$, and all $w \in W$. This can be shown by a double induction: an outer computational induction on the approximations of $\widehat{u}$ and u, and an inner structural one on the body of $\mathbf{f}$. $\square$

Consider now the programs of $NVIL$. For these programs, a standard denotational interpretation $[\![\cdot]\!]_{(W \to D)}$ can easily be defined, as this was done in (Rondogiannis & Wadge, 1997)[page 85]. The following theorem shows that the standard and the synchronic interpretations coincide in the case of $NVIL$.

Theorem 6.4
Let $\mathbf{P}$ *be an* $NVIL$ *program and let* u *and* $\widehat{u}$ *be the least environments that satisfy the definitions in* $\mathbf{P}$ *under the standard and the synchronic interpretations respectively. Then,* $[\![\mathbf{P}]\!]^*(\widehat{u}) = [\![\mathbf{P}]\!]_{(W \to D)}(u)$.

Proof
(Outline) It suffices to show that for every (nullary) function symbol $\mathbf{f}$ that has a definition in $\mathbf{P}$, $\widehat{u}(\mathbf{f}) = u(\mathbf{f})$. This follows directly with a proof procedure similar to the one outlined for Theorem 6.3. $\square$

The above two theorems suggest that in order to show the correctness of the transformation algorithm, one can simply rely on the synchronic interpretation (see also Theorem 8.5 later on).

7 Formal Definition of the Transformation Algorithm

The purpose of this section is to formally define the transformation algorithm from higher-order functional programs to intensional programs of nullary variables. The algorithm consists of a number of steps; at each step, the order of the input program is reduced by one. The transformation ends when a zero-order intensional program is obtained. More specifically, the input to the algorithm is an M-order FL program ($M > 0$). After the first step of the algorithm, an $(M-1)$-order IL program is obtained. After M steps of the algorithm have taken place, a zero-order $NVIL$ program has resulted. This is the output of the transformation.

Therefore, it suffices to just describe a single step of the algorithm, that is, the procedure required to transform an m-order intensional program ($1 \le m \le M$) into an $(m-1)$-order one. Notice that this procedure also applies for the first step in the transformation, because we can consider the source FL program as an IL program that happens not to contain any intensional operators.

A step of the algorithm can be intuitively described as follows: given an m-order input program, we start by considering the m-order functions that are defined in it. The goal is to lower the order of these functions by eliminating their $(m-1)$-order formal parameters, appropriately processing at the same time all the calls to each such **f** in the program.

For each formal removed from the formal parameter list of **f**, a new definition is created and added to the program. Each such definition gathers together all the actual parameters that correspond to the particular formal and that appear in calls to **f** in the program. This "gathering" is performed with the use of the operators `case` and `actuals`.

In this way, the input m-order program has been transformed into an $(m-1)$-order one. The procedure that we described above can be used repeatedly, until all formals have been eliminated from all functions in the program. The final result will be a program that consists of a set of intensional nullary-variable definitions.

7.1 Preliminary Definitions

In this subsection we provide certain preliminary definitions that are helpful in formally defining the transformation algorithm.

Definition 7.1
Let $\mathbf{q}$ *be the syntactic representation of an intensional operator. The inverse of* $\mathbf{q}$ *is denoted by* $\mathbf{q}^{-1}$ *and is defined as follows:*

$$\mathbf{q}^{-1} \quad = \quad \begin{cases} \mathtt{call}_i^m & \textit{if } \mathbf{q} = \mathtt{actuals}_i^m \\ \mathtt{actuals}_i^m & \textit{if } \mathbf{q} = \mathtt{call}_i^m \end{cases}$$

Given $\mathbf{q}$, we will often write q for the meaning of $\mathbf{q}$ and q^{-1} for the meaning of $\mathbf{q}^{-1}$. It can easily be shown that whenever the composition of q and q^{-1} is defined (in either order), then it is equal to the identity function.

Definition 7.2
Let $\mathbf{Q} = \mathbf{q}_0 \cdot \mathbf{q}_1 \cdot \ldots \cdot \mathbf{q}_{r-1} \in ISeq$. *Then, the* inverse sequence *of* $\mathbf{Q}$ *is* $\mathbf{Q}^{-1} = \mathbf{q}_{r-1}^{-1} \cdot \ldots \cdot \mathbf{q}_1^{-1} \cdot \mathbf{q}_0^{-1}$.

Let $\mathbf{P}$ be a m-order program. In the following, we assume an ordering of the definitions in $\mathbf{P}$ (for example, a lexicographic one). This will allow us to talk about "order of textual appearance" of function calls. Let $Sub(\mathbf{P})$ be the set of subexpressions of $\mathbf{P}$. We adopt the following conventions:

- Let $\mathbf{f}$ be an m-order function defined in $\mathbf{P}$. The set of calls to the function $\mathbf{f}$ in $\mathbf{P}$ is defined as:

$$calls(\mathbf{P}, \mathbf{f}) = \{\mathbf{Q}(\mathbf{f})(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}) \in Sub(\mathbf{P})\}$$

- Let $\mathbf{f}$ be an m-order function defined in $\mathbf{P}$. Let $\mathbf{C}_0, \ldots, \mathbf{C}_{r-1}$ be the calls to $\mathbf{f}$ listed in the order of their textual appearance in $\mathbf{P}$. The function *label* assigns natural number labels to the calls of $\mathbf{f}$ in $\mathbf{P}$ in such a way that different calls receive different labels:

$$label(\mathbf{P}, \mathbf{f}, \mathbf{C}_i) = \min\{j \mid \mathbf{C}_i = \mathbf{C}_j\}$$

  In this way, function calls are numbered in their order of textual appearance in $\mathbf{P}$, except for function calls that have more than one occurrences in $\mathbf{P}$ which receive the label of their first appearance[¶].
- Let $\mathbf{f}$ be a function defined in $\mathbf{P}$. The list of positions in the formal parameter list of $\mathbf{f}$ of those formals that have order less than $(m-1)$, is denoted by $low(\mathbf{f}, m)$. The list is sorted in ascending order. For example, if only the zeroth and third argument of $\mathbf{f}$ are less than $(m-1)$-order, then $low(\mathbf{f}, m) = [0, 3]$.
- Let $\mathbf{f}$ be a function defined in $\mathbf{P}$. Let $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ be the formal parameters of $\mathbf{f}$. Then, the set of the formal parameters of $\mathbf{f}$ that have order equal to $(m-1)$, is represented by $high(\mathbf{f}, m)$. For example, if only the first and fourth arguments of $\mathbf{f}$ are $(m-1)$-order, then $high(\mathbf{f}, m) = \{\mathbf{x}_1, \mathbf{x}_4\}$. Notice that *high* has a different result type than the function *low* defined before.
- Let $\mathbf{x} \in Vars(\mathbf{P})$ with $\mathbf{x} : (\sigma_0, \ldots, \sigma_{k-1}) \to \iota$. Then, $Form(\mathbf{P}, \mathbf{x})$ is a list of k variable symbols, which satisfies the following:

  — No variable in the list appears in program $\mathbf{P}$.

  — Given $\mathbf{y} \neq \mathbf{x}$, $Form(\mathbf{P}, \mathbf{x})$ and $Form(\mathbf{P}, \mathbf{y})$ are disjoint.

  Intuitively, these are "fresh" variables that will be attached to both sides of new definitions that result during the execution of the algorithm.

Based on the above definitions, we can now present the transformation algorithm, on a step-by-step form.

7.2 Processing Expressions

We start by defining the function that processes the expressions of the source program. More specifically, the elimination of the $(m-1)$-order arguments from function calls in program $\mathbf{P}$, is accomplished by the $\mathcal{E}_{\mathbf{P},m}$ function defined below:

$$\frac{\mathbf{E} = \mathbf{f}}{\mathcal{E}_{\mathbf{P},m}(\mathbf{E}) = \mathbf{f}}$$

[¶] This is a slightly different numbering scheme than the one given in the informal presentation of the algorithm in Section 3. However, both schemes work equally well in practice.

$$\frac{\mathbf{E} = \mathtt{c}(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})}{\mathcal{E}_{\mathbf{P},m}(\mathbf{E}) = \mathtt{c}(\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0), \ldots, \mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{n-1}))}$$

$$\frac{\mathbf{E} = \mathbf{Q}(\mathtt{f})(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}),\ \mathit{order}(\mathbf{f}) = m,\ \mathit{low}(\mathbf{f}, m) = [i_0, \ldots, i_{k-1}],\ \mathit{label}(\mathbf{P}, \mathbf{f}, \mathbf{E}) = i}{\mathcal{E}_{\mathbf{P},m}(\mathbf{E}) = \mathbf{Q} \cdot \mathtt{call}_i^{m-1}(\mathtt{f})(\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{i_0}), \ldots, \mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{i_{k-1}}))}$$

$$\frac{\mathbf{E} = \mathbf{Q}(\mathtt{f})(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}),\ \mathit{order}(\mathbf{f}) < m}{\mathcal{E}_{\mathbf{P},m}(\mathbf{E}) = \mathbf{Q}(\mathtt{f})(\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0), \ldots, \mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{n-1}))}$$

The first rule is for the case of (possibly higher-order) variables that are encountered during the transformation. In this case, the expression is not affected by the transformation algorithm. The second rule applies in the case of constant symbols; then, the transformation proceeds with the arguments of the constant. The third rule is for the case where a function call is encountered, and the corresponding function is m-order. The arguments that cause the function to be m-order (that is the $(m-1)$-order ones) are removed, and the call is prefixed by the appropriate intensional operator. The fourth rule applies when the function under consideration is not m-order. In this case, the translation proceeds with the actual parameters of the function call without eliminating any of them.

7.3 Eliminating the Highest-Order Formals

The function $\mathcal{D}_m$ is used to process the definitions in $\mathbf{P}$, removing their $(m-1)$-order formal parameters. Notice that at the same time, the body of each definition is processed using the function $\mathcal{E}_{\mathbf{P},m}$. The definition of $\mathcal{D}_m$ is given below:

$$\mathcal{D}_m(\mathbf{P}) = \bigcup_{\mathbf{F} \in \mathbf{P}} \mathcal{D}_m^*(\mathbf{F})$$

$$\frac{\mathbf{F} = \mathtt{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{E},\ \mathit{low}(\mathbf{f}, m) = [i_0, \ldots, i_{k-1}]}{\mathcal{D}_m^*(\mathbf{F}) = \{\mathtt{f}(\mathbf{x}_{i_0}, \ldots, \mathbf{x}_{i_{k-1}}) \doteq \mathcal{E}_{\mathbf{P},m}(\mathbf{E})\}}$$

$$\frac{\mathbf{F} = \mathtt{f}(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathtt{case}^m(\mathbf{E}_0, \ldots, \mathbf{E}_{r-1}),\ \mathit{low}(\mathbf{f}, m) = [i_0, \ldots, i_{k-1}]}{\mathcal{D}_m^*(\mathbf{F}) = \{\mathtt{f}(\mathbf{x}_{i_0}, \ldots, \mathbf{x}_{i_{k-1}}) \doteq \mathtt{case}^m(\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0), \ldots, \mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{r-1}))\}}$$

Notice that we have supplied two rules, one for the case where the body starts with a `case` operator and one for the case where the body is an ordinary expression.

7.4 Creating New Definitions

In this part of the transformation algorithm, a new definition is created for each $(m-1)$-order formal parameter that existed in program $\mathbf{P}$. Before formally defining the function $\mathcal{A}_m$ that performs exactly this task, we need to define certain auxiliary functions. Let $\mathbf{f}$ be an m-order function defined in $\mathbf{P}$, and let $\mathbf{C}_0, \ldots, \mathbf{C}_{r-1}$ be the different function calls to $\mathbf{f}$ in $\mathbf{P}$, listed according to their labels (that is, $\mathit{label}(\mathbf{P}, \mathbf{f}, \mathbf{C}_0) = 0, \ldots, \mathit{label}(\mathbf{P}, \mathbf{f}, \mathbf{C}_{r-1}) = r-1$). Let $\mathbf{x}$ be the j'th formal parameter of $\mathbf{f}$.

- Let $\mathbf{C}_i = \mathbf{Q}\texttt{(}\mathbf{f}\texttt{)}\texttt{(}\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\texttt{)}$, $0 \le i \le r-1$, be one of the calls to $\mathbf{f}$. The expression $inv(\mathbf{C}_i, \mathbf{x})$ is defined as follows (recall that $\mathbf{x}$ is the j'th formal parameter of $\mathbf{f}$, $0 \le j \le n-1$):

$$inv(\mathbf{C}_i, \mathbf{x}) = \texttt{actuals}_i^{m-1} \cdot \mathbf{Q}^{-1}\texttt{(}\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)\texttt{)}$$

  The above expression will be used by $\mathcal{A}_m$ for creating the body of the new definitions.
- The function $params(\mathbf{P}, \mathbf{f}, \mathbf{x})$ is defined as follows:

$$params(\mathbf{P}, \mathbf{f}, \mathbf{x}) = [inv(\mathbf{C}_0, \mathbf{x}), \ldots, inv(\mathbf{C}_{r-1}, \mathbf{x})]$$

  In other words, the function *params* gathers together all the *inv* expressions that correspond to the formal parameter $\mathbf{x}$ of $\mathbf{f}$.

The function $\mathcal{A}_m$ creates a new definition for each $(m-1)$-order formal parameter in program $\mathbf{P}$. If the formal parameter $\mathbf{x}$ of $\mathbf{f}$ is $(m-1)$-order, then the function $\mathcal{A}_{\mathbf{f},\mathbf{x},m}$ returns a set that contains a new definition for this formal. The formal definition of $\mathcal{A}_m$ is given below:

$$\mathcal{A}_m(\mathbf{P}) = \bigcup_{\mathbf{f} \in func(\mathbf{P})} \bigcup_{\mathbf{x} \in high(\mathbf{f},m)} \mathcal{A}_{\mathbf{f},\mathbf{x},m}(\mathbf{P})$$

$$\frac{params(\mathbf{P}, \mathbf{f}, \mathbf{x}) = [\mathbf{A}_0, \ldots, \mathbf{A}_{r-1}],\ Form(\mathbf{P}, \mathbf{x}) = \vec{\mathbf{z}}}{\mathcal{A}_{\mathbf{f},\mathbf{x},m}(\mathbf{P}) = \{\texttt{x}\texttt{(}\vec{\mathbf{z}}\texttt{)} \doteq \texttt{case}^{m-1}\texttt{(}\mathbf{A}_0\texttt{(}\vec{\mathbf{z}}\texttt{)}, \ldots, \mathbf{A}_{r-1}\texttt{(}\vec{\mathbf{z}}\texttt{))}\}}$$

7.5 The Overall Transformation

The translation of an m-order IL program into an $(m-1)$-order one, is performed by the function $Step_m$ shown below:

$$Step_m(\mathbf{P}) = \mathcal{D}_m(\mathbf{P}) \cup \mathcal{A}_m(\mathbf{P})$$

Finally, given an M-order FL program $\mathbf{P}$, the overall transformation of $\mathbf{P}$ into an intensional program of nullary variables, is described by the function $Trans_M$, given below:

$$Trans_M(\mathbf{P}) = Step_1(\cdots(Step_M(\mathbf{P}))\cdots)$$

This completes the formal description of the transformation. It can easily be verified that the programs that result at each intermediate step of the algorithm are syntactically valid IL programs, while the final program is a valid $NVIL$ one.

8 Correctness Proof of the Transformation

In this section we present in a rigorous way the correctness proof of the intensionalization technique for higher-order programs. In the following, we first make an assumption that helps us simplify the notation in the subsequent presentation. Then, the correctness proof of the transformation algorithm is presented in detail.

As discussed in the previous sections, let $\mathbf{P}_M$, $M > 0$, be the M-order source functional program on which the transformation algorithm is applied. The programs that result at successive stages of the algorithm are $\mathbf{P}_{M-1}, \ldots, \mathbf{P}_0$.

To simplify the notation, in the following we assume that all the functions defined in $\mathbf{P}_M$, have their highest-order arguments first, i.e., if $\mathbf{f}\texttt{(}\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\texttt{)} \doteq \mathbf{B_f}$ is a function defined in $\mathbf{P}_M$, then $order(\mathbf{x}_0) \geq order(\mathbf{x}_1) \geq \cdots \geq order(\mathbf{x}_{n-1})$. This helps us avoid notational complexities that would arise if we assumed that the placement of the formals is arbitrary. Notice that this property is preserved by the transformation, that is if it holds for $\mathbf{P}_M$ it will also hold for all $\mathbf{P}_m$, $0 \leq m \leq M$.

It can easily be checked that the above assumption does not affect the generality of the proof. Lifting the assumption does not alter the logic of the proof, but simply adds a level of notational complexity.

The following definition is used in the following discussion:

Definition 8.1
Let $w \in W$. The function $\Downarrow$: $(W, ISeq) \to W$ is recursively defined as follows:

$$
\begin{array}{lcl}
(w \Downarrow \epsilon) & = & w \\
(w \Downarrow \texttt{call}_i^m) & = & w[m/(i : w_m)] \\
(w \Downarrow \texttt{actuals}_i^m) & = & \begin{cases} w[m/tail(w_m)] & \textit{if } head(w_m) = i \\ undefined & \textit{otherwise} \end{cases} \\
(w \Downarrow (\mathbf{q}_0 \cdot \mathbf{q}_1 \cdot \ldots \cdot \mathbf{q}_{n-1})) & = & (w \Downarrow \mathbf{q}_0) \Downarrow (\mathbf{q}_1 \cdot \ldots \cdot \mathbf{q}_{n-1})
\end{array}
$$

In other words, $\Downarrow$ performs the context switch corresponding to the composition of the meanings of the operators in the given sequence.

The following Lemma can be easily shown:

Lemma 8.1
Let $w \in W$, $\mathbf{Q} \in ISeq$ and $a \in [\![\sigma]\!]^*$. Then:

$$
\begin{array}{ccc}
(w \Downarrow \mathbf{Q}) \Downarrow \mathbf{Q^{-1}} & = & w \\
Q(a)w & = & a(w \Downarrow \mathbf{Q})
\end{array}
$$

whenever $(w \Downarrow \mathbf{Q})$ is defined.

Consider now the program $\mathbf{P}_m$ $(0 < m \leq M)$. The following theorem establishes a relationship between the meaning of functions defined in $\mathbf{P}_m$ and the meaning of functions in $\mathbf{P}_{m-1}$.

Theorem 8.1
Let u and $\widehat{u}$ be the least environments that satisfy under the synchronic interpretation the definitions in $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$ respectively. Then:

- *For every definition $(\mathbf{f}\texttt{(}\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\texttt{)} \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0, \ldots, \mathbf{x}_{n-1} : \sigma_{n-1}$ and there exists $0 \leq l \leq n-1$ such that $order(\sigma_0) = (m-1), \ldots, order(\sigma_{l-1}) = (m-1)$ and $order(\sigma_l) < (m-1), \ldots, order(\sigma_{n-1}) < (m-1)$, then for every function call $\mathbf{E} = \mathbf{Q}\texttt{(}\mathbf{f}\texttt{)}\texttt{(}\mathbf{E}_0, \ldots, \mathbf{E}_{n-1}\texttt{)}$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_l \in [\![\sigma_l]\!], \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]$ and for all $w \in W$,*

  $$
  \begin{array}{l}
  Q \cdot call_i^{m-1}(\widehat{u}(\mathbf{f}))(w)(d_l, \ldots, d_{n-1}) \sqsubseteq \\
  Q(u(\mathbf{f}))(w)([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\widehat{u})(w), \ldots, [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\widehat{u})(w), d_l, \ldots, d_{n-1})
  \end{array}
  $$

  where $i = label(\mathbf{P}, \mathbf{f}, \mathbf{E})$.

- *For every definition* $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ *in* $\mathbf{P}_m$*, if* $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ *and* $order(\sigma_0) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$*, then for all* $d_0 \in [\![\sigma_0]\!],\ldots,d_{n-1} \in [\![\sigma_{n-1}]\!]$ *and for all* $w \in W$*,*

$$\widehat{u}(\mathbf{f})(w)(d_0,\ldots,d_{n-1}) \sqsubseteq u(\mathbf{f})(w)(d_0,\ldots,d_{n-1})$$

Proof
The proof is by a lengthy but in general straightforward computational induction over the stages in constructing $\widehat{u}$ (but *not* u). More specifically, it suffices to show that the above statements hold for all approximations $\widehat{u}_k$, $k \in N$, of the environment $\widehat{u}$. In other words, it suffices to show the following two statements:

- For every definition $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ and there exists $0 \le l \le n-1$ such that $order(\sigma_0) = (m-1),\ldots,order(\sigma_{l-1}) = (m-1)$ and $order(\sigma_l) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$, then for every function call $\mathbf{E} = \mathbf{Q}\,(\mathbf{f})\,(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1})$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_l \in [\![\sigma_l]\!],\ldots,d_{n-1} \in [\![\sigma_{n-1}]\!]$ and for all $w \in W$,

  $$\begin{array}{l} Q\cdot call_i^{m-1}(\widehat{u}_k(\mathbf{f}))(w)(d_l,\ldots,d_{n-1}) \sqsubseteq \\ Q(u(\mathbf{f}))(w)([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\widehat{u}_k)(w),\ldots,[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\widehat{u}_k)(w),d_l,\ldots,d_{n-1}) \end{array}$$

  where $i = label(\mathbf{P},\mathbf{f},\mathbf{E})$.
- For every definition $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ and $order(\sigma_0) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$, then for all $d_0 \in [\![\sigma_0]\!],\ldots,d_{n-1} \in [\![\sigma_{n-1}]\!]$ and for all $w \in W$,

  $$\widehat{u}_k(\mathbf{f})(w)(d_0,\ldots,d_{n-1}) \sqsubseteq u(\mathbf{f})(w)(d_0,\ldots,d_{n-1})$$

We demonstrate the above using induction on k. For $k = 0$, that is for $\widehat{u}_0$, the above trivially hold because the left hand side of each statement is equal to the bottom value. Assume that the claim holds for $k \ge 0$. We show the claim for $k+1$. That is, we show that[‖]:

- For every definition $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ and there exists $0 \le l \le n-1$ such that $order(\sigma_0) = (m-1),\ldots,order(\sigma_{l-1}) = (m-1)$ and $order(\sigma_l) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$, then for every function call $\mathbf{E} = \mathbf{Q}\,(\mathbf{f})\,(\mathbf{E}_0,\ldots,\mathbf{E}_{n-1})$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_l \in [\![\sigma_l]\!],\ldots,d_{n-1} \in [\![\sigma_{n-1}]\!]$ and for all $w \in W$,

  $$\begin{array}{l} Q\cdot call_i^{m-1}(\underline{\widehat{u}_{k+1}}(\mathbf{f}))(w)(d_l,\ldots,d_{n-1}) \sqsubseteq \\ Q(u(\mathbf{f}))(w)([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\underline{\widehat{u}_{k+1}})(w),\ldots,[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\underline{\widehat{u}_{k+1}})(w),d_l,\ldots,d_{n-1}) \end{array}$$

  where $i = label(\mathbf{P},\mathbf{f},\mathbf{E})$.
- For every definition $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ and $order(\sigma_0) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$, then for all $d_0 \in [\![\sigma_0]\!],\ldots,d_{n-1} \in [\![\sigma_{n-1}]\!]$ and for all $w \in W$,

  $$\underline{\widehat{u}_{k+1}}(\mathbf{f})(w)(d_0,\ldots,d_{n-1}) \sqsubseteq u(\mathbf{f})(w)(d_0,\ldots,d_{n-1})$$

‖ As suggested by one of the reviewers, in the following steps of the proof we underline the differences that appear from one step to the next one. This is intended to save the reader from carefully checking to see exactly what has been changed.

Using the semantics of $call_i^{m-1}$ and Q, the above two statements can be written as follows:

- For every definition $(\mathtt{f}\,\mathtt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0:\sigma_0,\ldots,\mathbf{x}_{n-1}:\sigma_{n-1}$ and there exists $0 \le l \le n-1$ such that $order(\sigma_0)=(m-1),\ldots,order(\sigma_{l-1})=(m-1)$ and $order(\sigma_l)<(m-1),\ldots,order(\sigma_{n-1})<(m-1)$, then for every function call $\mathbf{E}=\mathbf{Q}\,\mathtt{(f)}\,\mathtt{(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\mathtt{)}$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_l\in[\![\sigma_l]\!],\ldots,d_{n-1}\in[\![\sigma_{n-1}]\!]$ and for all $w\in W$,

  $$\underline{\widehat{u}_{k+1}(\mathbf{f})(w\Downarrow(\mathbf{Q}\cdot\mathtt{call}_i^{m-1}))}(d_l,\ldots,d_{n-1}) \sqsubseteq$$
  $$\underline{u(\mathbf{f})(w\Downarrow\mathbf{Q})}([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\widehat{u}_{k+1})(w),\ldots,[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\widehat{u}_{k+1})(w),d_l,\ldots,d_{n-1})$$

  where $i=label(\mathbf{P},\mathbf{f},\mathbf{E})$.
- For every definition $(\mathtt{f}\,\mathtt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0:\sigma_0,\ldots,\mathbf{x}_{n-1}:\sigma_{n-1}$ and $order(\sigma_0)<(m-1),\ldots,order(\sigma_{n-1})<(m-1)$, then for all $d_0\in[\![\sigma_0]\!],\ldots,d_{n-1}\in[\![\sigma_{n-1}]\!]$ and for all $w\in W$,

  $$\widehat{u}_{k+1}(\mathbf{f})(w)(d_0,\ldots,d_{n-1}) \sqsubseteq u(\mathbf{f})(w)(d_0,\ldots,d_{n-1})$$

Recall now that $\mathtt{f}\,\mathtt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B_f}$ in $\mathbf{P}_m$ and also $\mathtt{f}\,\mathtt{(}\mathbf{x}_l,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathcal{E}_{\mathbf{P},m}(\mathbf{B_f})$ in $\mathbf{P}_{m-1}$. The idea is to get equivalent statements that involve the body of the function $\mathbf{f}$. More specifically, we use Definition 6.10 to replace the right hand sides of the statements, and Theorem 6.1 to replace the left hand sides of the statements (notice that in the statements that we obtain below, for brevity reasons we use the perturbation notation $\oplus$ instead of the $[\cdots]$ notation that is used in Definition 6.10 and Theorem 6.1).

Therefore, it suffices to show that:

- For every definition $(\mathtt{f}\,\mathtt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0:\sigma_0,\ldots,\mathbf{x}_{n-1}:\sigma_{n-1}$ and there exists $0 \le l \le n-1$ such that $order(\sigma_0)=(m-1),\ldots,order(\sigma_{l-1})=(m-1)$ and $order(\sigma_l)<(m-1),\ldots,order(\sigma_{n-1})<(m-1)$, then for every function call $\mathbf{E}=\mathbf{Q}\,\mathtt{(f)}\,\mathtt{(}\mathbf{E}_0,\ldots,\mathbf{E}_{n-1}\mathtt{)}$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_l\in[\![\sigma_l]\!],\ldots,d_{n-1}\in[\![\sigma_{n-1}]\!]$ and for all $w\in W$,

  $$\underline{[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{B_f})]\!]^*(\widehat{u}_k\oplus\sigma)(w\Downarrow(\mathbf{Q}\cdot\mathtt{call}_i^{m-1}))} \sqsubseteq \underline{[\![\mathbf{B_f}]\!]^*(u\oplus\sigma\oplus\widehat{\rho}_{k+1})(w\Downarrow\mathbf{Q})}$$

  where $\underline{\sigma(\mathbf{x}_j)=d_j^\infty}$, $\underline{l\le j\le n-1}$, and $\underline{\widehat{\rho}_{k+1}(\mathbf{x}_j)=([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)]\!]^*(\widehat{u}_{k+1})(w))^\infty}$, $\underline{0\le j\le l-1}$.
- For every definition $(\mathtt{f}\,\mathtt{(}\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\mathtt{)} \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0:\sigma_0,\ldots,\mathbf{x}_{n-1}:\sigma_{n-1}$ and $order(\sigma_0)<(m-1),\ldots,order(\sigma_{n-1})<(m-1)$, then for all $d_0\in[\![\sigma_0]\!],\ldots,d_{n-1}\in[\![\sigma_{n-1}]\!]$ and for all $w\in W$,

  $$\underline{[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{B_f})]\!]^*(\widehat{u}_k\oplus\sigma)(w)} \sqsubseteq \underline{[\![\mathbf{B_f}]\!]^*(u\oplus\sigma)(w)}$$

  $\underline{\text{where } \sigma(\mathbf{x}_j)=d_j^\infty,\ 0\le j\le n-1}$.

In the following, we give the proof for the first of the above statements. The proof for the second statement is simpler, and can be given in a similar way. Notice that the proof of each of the above statements, uses at some point the induction hypothesis of the other statement.

To prove the first of the statements, we consider any function $\mathbf{f}$ in the program that satisfies the requirements set by the statement. We proceed by distinguishing two cases, regarding whether the definition of $\mathbf{f}$ starts with a `case` operator or not. We will only show the proof for the latter case (the proof for the former one is similar).

The proof can be established by structural induction on the body of the function, that is by showing that for every subexpression $\mathbf{S}$ of $\mathbf{B_f}$:

$$[\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow (\mathbf{Q} \cdot \mathtt{call}_i^{m-1})) \sqsubseteq [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q})$$

In the following, for simplicity we denote the sequence $\mathbf{Q} \cdot \mathtt{call}_i^{m-1}$ by $\widehat{\mathbf{Q}}$.

Structural Induction Basis:
Case 1: $\mathbf{S}$ is equal to a variable $\mathbf{x}_j \in \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$, and $\mathbf{x}_j$ is $(m-1)$-order. Then, a definition of the form:

$$\mathbf{x}_j\mathtt{(}\vec{\mathbf{z}}\mathtt{)} \doteq \mathtt{case}^{m-1}\mathtt{(}\mathbf{A}_0\mathtt{(}\vec{\mathbf{z}}\mathtt{)}, \ldots, \mathbf{A}_{r-1}\mathtt{(}\vec{\mathbf{z}}\mathtt{))}$$

has been created in $\mathbf{P}_{m-1}$, where the $\mathbf{A}_0, \ldots, \mathbf{A}_{r-1}$ are derived as indicated by the function $params(\mathbf{P}, \mathbf{f}, \mathbf{x}_j)$. We first demonstrate the following:

$$\widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}}) \sqsubseteq \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q}) \tag{1}$$

Assume that $\vec{\mathbf{z}}$ is of the form $(\mathbf{z}_0, \ldots, \mathbf{z}_{k-1})$, and $\mathbf{z}_0 : \sigma'_0, \ldots, \mathbf{z}_{k-1} : \sigma'_{k-1}$. Let $\vec{e} = (e_0, \ldots, e_{k-1})$ be a k-tuple such that $e_0 \in [\![\sigma'_0]\!], \ldots, e_{k-1} \in [\![\sigma'_{k-1}]\!]$. Moreover, let $\phi(\mathbf{z}_i) = e_i^\infty$, $0 \le i \le k-1$. The proof of (1) is as follows:

$$\begin{array}{ll}
 & \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}})(\vec{e}) = \\
\sqsubseteq & [\![\mathtt{case}^{m-1}\mathtt{(}\mathbf{A}_0\mathtt{(}\vec{\mathbf{z}}\mathtt{)}, \ldots, \mathbf{A}_{r-1}\mathtt{(}\vec{\mathbf{z}}\mathtt{))}]\!]^*(\widehat{u}_k \oplus \phi)(w \Downarrow \widehat{\mathbf{Q}}) \\
 & \text{(Definition of } \mathbf{x}_j \text{ and Lemma 6.1)} \\
= & [\![\mathbf{A}_i\mathtt{(}\vec{\mathbf{z}}\mathtt{)}]\!]^*(\widehat{u}_k \oplus \phi)(w \Downarrow \widehat{\mathbf{Q}}) \\
 & \text{(Semantics of } \mathtt{case}^{m-1}\text{)} \\
= & [\![\mathtt{actuals}_i^{m-1} \cdot \mathbf{Q}^{-1}\mathtt{(}\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)\mathtt{)(}\vec{\mathbf{z}}\mathtt{)}]\!]^*(\widehat{u}_k \oplus \phi)(w \Downarrow \widehat{\mathbf{Q}}) \\
 & \text{(Definition of } \mathbf{A}_i \text{ from function } params(\mathbf{P}, \mathbf{f}, \mathbf{x}_j)\text{)} \\
= & [\![\mathbf{Q}^{-1}\mathtt{(}\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)\mathtt{)}]\!]^*(\widehat{u}_k \oplus \phi)(w \Downarrow \mathbf{Q})(\vec{e}) \\
 & \text{(Semantics of } \mathtt{actuals}_i^{m-1} \text{ and application)} \\
= & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)]\!]^*(\widehat{u}_k \oplus \phi)(w)(\vec{e}) \\
 & \text{(Lemma 8.1)} \\
= & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)]\!]^*(\widehat{u}_k)(w)(\vec{e}) \\
 & (\mathbf{E}_j \text{ does not contain any of the } \mathbf{z}_i) \\
\sqsubseteq & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)]\!]^*(\widehat{u}_{k+1})(w)(\vec{e}) \\
 & \text{(Monotonicity of } [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_j)]\!]^*\text{)} \\
= & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q})(\vec{e}) \\
 & \text{(Definition of } \widehat{\rho}_{k+1}\text{)}
\end{array}$$

The above result is used in the proof given below. The left hand side of the statement we want to establish can be written as:

$$
\begin{array}{rl}
& [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) = \\
= & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{x}_j)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Because } \mathbf{S} = \mathbf{x}_j) \\
= & [\![\mathbf{x}_j]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Definition of } \mathcal{E}_{\mathbf{P},m}) \\
= & \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Variable } \mathbf{x}_j \text{ is } (m-1)\text{-order}) \\
\sqsubseteq & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q}) \\
& (\text{Because of relation (1) above}) \\
= & [\![\mathbf{x}_j]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
& (\text{Variable } \mathbf{x}_j \text{ gets a value from } \widehat{\rho}_{k+1}) \\
= & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
& (\text{Because } \mathbf{S} = \mathbf{x}_j)
\end{array}
$$

Case 2: $\mathbf{S}$ is equal to a variable $\mathbf{x}_j \in \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$, and $\mathbf{x}_j$ is less than $(m-1)$-order. We should remind here that d^∞ is a constant intension, and therefore its value does not vary from context to context. The left hand side of the statement we want to establish can be written as follows:

$$
\begin{array}{rl}
& [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) = \\
= & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{x}_j)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Because } \mathbf{S} = \mathbf{x}_j) \\
= & [\![\mathbf{x}_j]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Definition of } \mathcal{E}_{\mathbf{P},m}) \\
= & \sigma(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}}) \\
& (\text{Because } \mathbf{x}_j \text{ is less than } (m-1)\text{-order}) \\
= & \sigma(\mathbf{x}_j)(w \Downarrow \mathbf{Q}) \\
& (\text{Because } \sigma(\mathbf{x}_j) = d_j^\infty) \\
= & [\![\mathbf{x}_j]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
& (\text{Variable } \mathbf{x}_j \text{ gets a value from } \sigma) \\
= & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
& (\text{Because } \mathbf{S} = \mathbf{x}_j)
\end{array}
$$

Case 3: $\mathbf{S}$ is equal to a nullary constant symbol $\mathbf{c}$. The proof in this case is straightforward, because the denotation of $\mathbf{c}$ is a constant intension, and therefore its value is independent of context.

Case 4: $\mathbf{S}$ is equal to $\mathbf{h}$, where $\mathbf{h}$ is not a formal of $\mathbf{f}$. In this case, the order of $\mathbf{h}$ is strictly less than m (because m-order functions only have full applications in $\mathbf{P}_m$). Recall now that the outer induction hypothesis for functions of order less than m, specifies that $\widehat{u}_k(\mathbf{h}) \sqsubseteq u(\mathbf{h})$. Using this, and the fact that $u(\mathbf{h})$ does not depend on the m-th dimension, we get the desired result.

Structural Induction Step.
Case 1: $\mathbf{S} = \mathbf{x}_j(\mathbf{S}_0, \ldots, \mathbf{S}_{r-1})$ where $\mathbf{x}_j \in \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$, and $\mathbf{x}_j$ is $(m-1)$-order.

The proof uses the following fact which was also used in the induction basis:

$$\widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}}) \sqsubseteq \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q}) \tag{2}$$

In the following, notice that none of the arguments of $\mathbf{x}_j$ is eliminated during the transformation because all of them are less than $(m-1)$-order. The proof is as follows:

$$\begin{array}{ll}
 & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) = \\
= & [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{x}_j\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
 & \text{(Because } \mathbf{S} = \mathbf{x}_j\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,)) \\
= & [\![\mathbf{x}_j\,(\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_0),\ldots,\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_{r-1}))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}) \\
 & \text{(Definition of } \mathcal{E}_{\mathbf{P},m}) \\
= & \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}})([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_0)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}),\ldots, \\
 & \quad [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_{r-1})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}})) \\
 & \text{(Variable } \mathbf{x}_j \text{ is } (m-1)\text{-order)} \\
\sqsubseteq & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q})([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_0)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}}),\ldots, \\
 & \quad [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{S}_{r-1})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{\mathbf{Q}})) \\
 & \text{(Because } \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{\mathbf{Q}}) \sqsubseteq \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q})) \\
\sqsubseteq & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow \mathbf{Q})([\![\mathbf{S}_0]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}),\ldots, \\
 & \quad [\![\mathbf{S}_{r-1}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q})) \\
 & \text{(Using structural induction hypothesis and monotonicity)} \\
= & [\![\mathbf{x}_j\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
 & \text{(Variable } \mathbf{x}_j \text{ gets a value from } \widehat{\rho}_{k+1}) \\
= & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow \mathbf{Q}) \\
 & \text{(Because } \mathbf{S} = \mathbf{x}_j\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,))
\end{array}$$

Case 2: $\mathbf{S} = \mathbf{x}_j\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,)$ where $\mathbf{x}_j \in \{\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\}$, and $\mathbf{x}_j$ is less than $(m-1)$-order. Then, $\mathbf{x}_j$ gets its value from σ in both sides of the statement we want to establish. Notice also that $\sigma(\mathbf{x}_j) = d_j^\infty$, i.e., it is a constant intension, and therefore its value is independent of context. The proof is then similar in structure to the one for the above case.

Case 3: $\mathbf{S} = \mathbf{c}\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,)$. The proof for this case is simple and uses the fact that $\mathcal{C}^*(\mathbf{c})$ is a constant intension.

Case 4: $\mathbf{S} = \mathbf{\Phi}\,(\mathbf{g})\,(\mathbf{S}_0,\ldots,\mathbf{S}_{r-1}\,)$ where $\mathbf{\Phi}$ is a sequence of intensional operators and $\mathbf{g}$ is a function defined in $\mathbf{P}_m$. The proof is similar as before (we need to consider two cases: one for $\mathbf{g}$ being m-order and one for less than m-order).

This completes the proof of the theorem. □

Theorem 8.2
Let u and $\widehat{u}$ be the least environments that satisfy under the synchronic interpretation the definitions in $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$ respectively. Then:

- *For every definition $(\mathbf{f}\,(\mathbf{x}_0,\ldots,\mathbf{x}_{n-1}\,) \doteq \mathbf{B_f})$ in $\mathbf{P}_m$, if $\mathbf{x}_0 : \sigma_0,\ldots,\mathbf{x}_{n-1} : \sigma_{n-1}$ and there exists $0 \leq l \leq n-1$ such that $order(\sigma_0) = (m-1),\ldots,order(\sigma_{l-1}) = (m-1)$ and $order(\sigma_l) < (m-1),\ldots,order(\sigma_{n-1}) < (m-1)$, then for every* *function call* $\mathbf{E} = \mathbf{Q}\,(\mathbf{f})\,(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ *to* $\mathbf{f}$ *in* $\mathbf{P}_m$*, for all* $d_l \in [\![\sigma_l]\!], \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]$ *and for all* $w \in W$*,*

$$Q \cdot call_i^{m-1}(\widehat{u}(\mathbf{f}))(w)(d_l, \ldots, d_{n-1}) \sqsupseteq$$
$$Q(u(\mathbf{f}))(w)([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\widehat{u})(w), \ldots, [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\widehat{u})(w), d_l, \ldots, d_{n-1})$$

where $i = label(\mathbf{P}, \mathbf{f}, \mathbf{E})$*.*

- *For every definition* $(\mathbf{f}\,(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ *in* $\mathbf{P}_m$*, if* $\mathbf{x}_0 : \sigma_0, \ldots, \mathbf{x}_{n-1} : \sigma_{n-1}$ *and* $order(\sigma_0) < (m-1), \ldots, order(\sigma_{n-1}) < (m-1)$*, then for all* $d_0 \in [\![\sigma_0]\!], \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]$ *and for all* $w \in W$*,*

$$\widehat{u}(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) \sqsupseteq u(\mathbf{f})(w)(d_0, \ldots, d_{n-1})$$

Proof
Following the same ideas as the proof for Theorem 8.1 (but now using computational induction on the approximations of u). □

Theorem 8.3
Let u *and* $\widehat{u}$ *be the least environments that satisfy under the synchronic interpretation the definitions in* $\mathbf{P}_m$ *and* $\mathbf{P}_{m-1}$ *respectively. Then:*

- *For every definition* $(\mathbf{f}\,(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ *in* $\mathbf{P}_m$*, if* $\mathbf{x}_0 : \sigma_0, \ldots, \mathbf{x}_{n-1} : \sigma_{n-1}$ *and there exists* $0 \leq l \leq n-1$ *such that* $order(\sigma_0) = (m-1), \ldots, order(\sigma_{l-1}) = (m-1)$ *and* $order(\sigma_l) < (m-1), \ldots, order(\sigma_{n-1}) < (m-1)$*, then for every function call* $\mathbf{E} = \mathbf{Q}\,(\mathbf{f})\,(\mathbf{E}_0, \ldots, \mathbf{E}_{n-1})$ *to* $\mathbf{f}$ *in* $\mathbf{P}_m$*, for all* $d_l \in [\![\sigma_l]\!], \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]$ *and for all* $w \in W$*,*

$$Q \cdot call_i^{m-1}(\widehat{u}(\mathbf{f}))(w)(d_l, \ldots, d_{n-1}) =$$
$$Q(u(\mathbf{f}))(w)([\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_0)]\!]^*(\widehat{u})(w), \ldots, [\![\mathcal{E}_{\mathbf{P},m}(\mathbf{E}_{l-1})]\!]^*(\widehat{u})(w), d_l, \ldots, d_{n-1})$$

where $i = label(\mathbf{P}, \mathbf{f}, \mathbf{E})$*.*
- *For every definition* $(\mathbf{f}\,(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \doteq \mathbf{B_f})$ *in* $\mathbf{P}_m$*, if* $\mathbf{x}_0 : \sigma_0, \ldots, \mathbf{x}_{n-1} : \sigma_{n-1}$ *and* $order(\sigma_0) < (m-1), \ldots, order(\sigma_{n-1}) < (m-1)$*, then for all* $d_0 \in [\![\sigma_0]\!], \ldots, d_{n-1} \in [\![\sigma_{n-1}]\!]$ *and for all* $w \in W$*,*

$$\widehat{u}(\mathbf{f})(w)(d_0, \ldots, d_{n-1}) = u(\mathbf{f})(w)(d_0, \ldots, d_{n-1})$$

Proof
A direct consequence of Theorems 8.1 and 8.2. □

Notice that although in the following we will only use the second of the statements of the above Theorem, the first one is also essential (the proof of the induction step of the second statement uses at some point the induction hypothesis of the first one).

The following theorem demonstrates that the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$ are semantically equivalent under the synchronic interpretation.

Theorem 8.4
Let u *and* $\widehat{u}$ *be the least environments that satisfy under the synchronic interpretation the definitions in* $\mathbf{P}_m$ *and* $\mathbf{P}_{m-1}$ *respectively. Then,* $[\![\mathbf{P}_m]\!]^*(u) = [\![\mathbf{P}_{m-1}]\!]^*(\widehat{u})$*.*

Proof

Straightforward, by applying the second statement of Theorem 8.3 on the variable **result** of the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$. $\square$

It remains to show that the initial functional program $\mathbf{P}_M$, has the same standard denotational semantics as the final zero-order intensional program $\mathbf{P}_0$. This is demonstrated by the following theorem:

Theorem 8.5
Let $\mathbf{P}_M$ be an M-order FL program and let $\mathbf{P}_{M-1}, \ldots, \mathbf{P}_0$ be the intensional programs that result at the successive stages of the transformation algorithm. Let u_M and u_0 be the least environments that satisfy the definitions of $\mathbf{P}_M$ and $\mathbf{P}_0$ under the standard interpretations. Then, for every $w \in W$,

$$[\![\mathbf{P}_M]\!]_D(u_M) = [\![\mathbf{P}_0]\!]_{(W \to D)}(u_0)(w)$$

Proof
Let $\widehat{u}_M, \ldots, \widehat{u}_0$ be the least environments that satisfy the definitions in the programs $\mathbf{P}_M, \ldots, \mathbf{P}_0$ under the synchronic interpretation. Then, for every $w \in W$:

$$\begin{array}{rl} & [\![\mathbf{P}_M]\!]_D(u_M) = \\ = & [\![\mathbf{P}_M]\!]^*_D(\widehat{u}_M)(w) \\ & \text{(Theorem 6.3)} \\ = & [\![\mathbf{P}_{M-1}]\!]^*_D(\widehat{u}_{M-1})(w) \\ & \text{(Theorem 8.4)} \\ & \ldots \\ = & [\![\mathbf{P}_0]\!]^*_D(\widehat{u}_0)(w) \\ & \text{(Theorem 8.4)} \\ = & [\![\mathbf{P}_0]\!]^*_{(W \to D)}(u_0)(w) \\ & \text{(Theorem 6.4)} \end{array}$$

$\square$

The correctness proof given above concludes the formal presentation of the transformation algorithm from higher-order functional programs to intensional programs of nullary variables. It should be mentioned here that the proof did not just serve the purpose of validating the correctness of the algorithm; it also suggested changes that had to be performed to the initially proposed algorithm (Wadge, 1991). Notice that the transformation for higher-order programs is much more sophisticated than the one for the first-order case (Rondogiannis & Wadge, 1997), and it is imperative that any informal intuitions one might have, be supported by formal reasoning.

9 Significance of the Results

The results presented here are first of all of real practical significance because they provide a genuine alternative to the conventional techniques for implementing a significant class of higher-order functional languages. The technique has proven to be practical both on conventional single processor architectures (see next section) as well as on a networked coarse-grain dataflow system (see (Ashcroft *et al.*, 1995) which describes the eduction-based implementation of the multidimensional language GLU). Furthermore, the technique could conceivably be used to enhance Web-based (and therefore tagged and demand-driven) applications like Intensional HTML (Wadge *et al.*, 1998). Finally, the proposed approach could be used to implement (some) higher-order programs on fine-grain dataflow machines, should these ever become practical (these issues are discussed in detail in the next section).

The results are also of theoretical significance, in that they shed new light on the relationship between (extensional) functional programming and intensional logic. Intensional logics of various types have already proven useful in different areas of computer science, but mainly in artificial intelligence and verification (temporal logics). We have now shown that a relatively simple intensional logic can give us a new and different perspective on the computations involved in evaluating higher-order expressions.

We can also view the results of this paper and its predecessor as shedding further light on the relationship between iteration and recursion, the yin and yang of computation. Indeed, we can view our results as generalizations of the well-known technique for implementing tail recursion as (conventional) iteration. In the first-order case, we can understand the translation as a technique for reducing general first-order recursion to *branching-time* iteration. The generalization presented in this paper can then be understood as a technique for reducing restricted typed higher-order recursion to multidimensional branching-time iteration.

Finally, the results point to a closer connection between the notions of *higher-order* and *multidimensional* than on the surface would appear to be the case. Multidimensionality is playing an increasingly important role in many branches of computing (for example, OLAP/multidimensional technologies in databases) and yet is apparently absent from functional languages. We have shown that, in effect, a higher-order language already has a form of multidimensionality built into its computation model. This also suggests that explicit mutlidimensionality can be added to functional languages (and vice-versa) without creating enormous interaction problems.

10 Implementation Issues

The transformation algorithm developed in this paper generalizes the one for first-order programs that was formalized in (Rondogiannis & Wadge, 1997). As we have seen in the previous sections, the algorithm transforms a significant class of higher-order programs into multidimensional zero-order intensional programs. The resulting intensional code can be executed on conventional architectures using the same basic principles as described in (Rondogiannis & Wadge, 1997)[section 12], the only difference being that the contexts are now multidimensional.

In (Rondogiannis & Wadge, 1997) we saw that the lists of natural numbers that are created during execution can be coded as small natural numbers, using the well-known *hash-consing* technique. The same technique applies here as well: given an M-order functional program, the contexts required for its execution are M-sequences of lists of natural numbers; using hash-consing, contexts become M-sequences of natural numbers, which are much more convenient to handle. The structure of the warehouse is similar to the one before, the only difference being that tags are now multidimensional.

In other words, the structure of an implementation of the proposed technique consists of three interacting components:

- The *execution engine* which implements the *EVAL* function used in Section 3.
- The *List Store* which is a hashing-based table for encoding lists by small natural numbers.
- The *Warehouse* which is used for keeping already computed results (identifiers together with their values under particular contexts).

The transformation algorithm proposed in this paper has been implemented and given promising efficiency results (Rondogiannis & Wadge, 1993; Rondogiannis, 1994; Rondogiannis & Wadge, 1994b).

In (Rondogiannis & Wadge, 1993), the technique is used in order to build a compiler for an ISWIM-like functional language (that uses higher-order functions in the way presented in this paper). The compiler has been implemented in *C* and produces *C* code as its output. In this implementation, the List Store is a *closed* hash table and collisions are resolved using *linear probing* (Knuth, 1975, Algorithm L). In other words, whenever a collision occurs, the next position of the table is examined. The Warehouse implementation uses an open scheme (Knuth, 1975, pages 513–514) in which collisions are resolved using chaining. Although the compiler does not use any special optimizations, it is shown to have a performance comparable to well-known implementations based on graph-reduction.

A different implementation of the technique is undertaken in (Rondogiannis, 1994; Rondogiannis & Wadge, 1994b). The main idea is that the List Store and the Warehouse can be appropriately merged into a single structure. This is achieved by incorporating context-related information into traditional activation records. This new technique appears to be more promising compared to the previous technique as it avoids certain of the overheads associated with hashing.

In the present paper we will not pursue implementation issues any further. As a final comment on these issues, it should also be noted that the intensional approach for implementing functional languages poses a new set of interesting problems that in our opinion deserve further investigation. One such problem is the characterization of the *dimensionality* of variables that appear in the target intensional code. More specifically, it is possible that a variable in the zero-order program that results from the transformation does not depend on all the dimensions but only on just a few of them. The knowledge of the dimensionality of particular variables is crucial. For example, if a variable does not depend on any dimension (i.e. it is constant in every context) we simply need to have one entry for it in the warehouse together with an indication that the variable does not depend on any dimension. In this way, both space and time savings are ensured, which in turn result in more efficient implementations. A promising approach for dimensionality analysis is outlined in

(Dodd, 1996) (which however applies to a different class of multidimensional languages).

11 Related Work

Our work is connected to the recent research on *higher-order removal* (Chin & Darlington, 1996) and *firstification* (Nelan, 1991), whose purpose is to reduce a given higher-order functional program into a first-order one. The practical outcome of both techniques is that the resulting first-order programs can be executed in a more efficient way than the source higher-order ones. Chin's and Darlington's transformation is formulated using unfold/fold rules while Nelan takes a more direct approach in his firstification algorithm.

Our work differs from both approaches in that the result of our transformation is a multidimensional intensional program of nullary variables. Moreover, our goal is to transform the source program into a form which can be educed (executed in a dataflow style using context manipulation), while the goal of both firstification and higher-order removal is to serve as a forms of optimization for the source higher-order programs.

Despite their differences, a further comparison between firstification, higher-order removal and the intensional technique would be beneficial as it would reveal the potential and the shortcomings of each approach. We believe that such a comparison should be based an two main criteria:

1. *Code Size.* In order for such a comparison to take place one must identify the parameters that play an important role in each of the techniques. For example, the order of a source program seems to be an important factor that affects the final code size in the case of the intensional approach. We believe that the intensional approach may have an advantage in terms of code size as it gathers together in a single definition all the actual parameters that correspond to a formal parameter of a function.
2. *Execution Speed and Memory Requirements.* This is a much harder to establish comparison due to the different philosophy behind the techniques: firstification and higher-order removal result in conventional functional programs while the intensional approach results in multidimensional zero-order programs which can be executed with eduction-based techniques. Therefore, such a comparison would require the availability of robust implementations of the above three techniques.

As a limitation of all three approaches, we should mention the fact that none of them has been extended to apply to a fully higher-order functional language. This suggests that higher-order functions are of a fundamental nature and their elimination from functional programs may require more sophisticated extensions of the above three techniques.

Reducing the order of the source program is also the goal of a technique originally proposed by Reynolds (Reynolds, 1972). However, in order for this to be achieved, data-structures have to be introduced in the program. Moreover, the resulting code actually simulates the runtime behavior of the source program. Therefore, although elegant, Reynolds technique does not serve the same goals as the technique we propose in this paper.

12 Conclusions and Future Work

In this paper we have presented and formalized a technique for transforming a significant class of higher-order functional programs into zero-order multidimensional intensional programs. The transformation we propose is of practical interest, since it can be used in order to implement functional languages in a tagged dataflow way.

The syntax of the functional languages considered in this paper imposes some restrictions on the use of higher-order functions. More specifically, the only partially applied objects that can appear in a program, are function names. Consider for example the following program:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{g(8)} \\
\texttt{g(x)} & \doteq & \texttt{twice(add(x),x)} \\
\texttt{twice(f,y)} & \doteq & \texttt{f(f(y))} \\
\texttt{add(a)(b)} & \doteq & \texttt{a+b}
\end{array}$$

This is clearly not a valid program of the language FL: the call to the function `twice` has as an actual parameter the partially applied call `add(x)`. In the following, we demonstrate the problems that we face when we attempt to apply the technique developed in this paper on programs such as the above. The highest order formal parameter in this program, is the formal `f` of the `twice` function. If we attempt to eliminate this parameter as usual, we get the following result:

$$\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{g(8)} \\
\texttt{g(x)} & \doteq & \texttt{call}_0^1\texttt{(twice)(x)} \\
\texttt{twice(y)} & \doteq & \texttt{f(f(y))} \\
\texttt{add(a)(b)} & \doteq & \texttt{a+b} \\
\texttt{f} & \doteq & \texttt{case}^1\texttt{(actuals}_0^1\texttt{(add(x)))}
\end{array}$$

Notice now that the variable `x` appears free in the definition of `f`, while it is bound in the definition of `g`. The resulting program cannot be semantically equivalent to the initial one. Therefore, the transformation has to be performed in a different way. We conjecture that the extended transformation will first have to take care of those variables that cause problems (like the formal parameter **x** of **g** above). The authors are currently investigating techniques for applying the transformation to general higher-order programs.

Another interesting problem for further research is to consider the target multidimensional intensional languages as programming languages (and not just as transformation-related ones) and investigate other potential applications they may have. An approach in this direction is reported in (Rondogiannis *et al.*, 1997) for the case of intensional logic programming languages. We believe that a similar potential exists in the area of intensional functional programming.

Acknowledgements: We would like to thank the reviewers for their useful comments and suggestions. In particular we would like to acknowledge one of the reviewers whose insightful remarks have helped us significantly improve the quality of our paper.

References

Ashcroft, E. A., Faustini, A. A., Jagannathan, R., & Wadge, W. W. (1995). *Multidimensional programming*. Oxford University Press.

Chin, W., & Darlington, J. (1996). A Higher-Order Removal Method. *Lisp and Symbolic Computation*, **9**, 287–322.

Dodd, C. (1996). Rank Analysis in the GLU Compiler. *Pages 76–82 of:* Orgun, M., & Ashcroft, E. (eds), *Intensional Programming I*. World Scientific.

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

Du, W., & Wadge, W. W. (1990a). A 3D Spreadsheet Based on Intensional Logic. *IEEE Software*, July, 78–89.

Du, W., & Wadge, W. W. (1990b). The Eductive Implementation of a Three-dimensional Spreadsheet. *Software-Practice and Experience*, **20**(11), 1097–1114.

Gallin, D. (1975). *Intensional and Higher-Order Modal Logic*. North-Holland.

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

Knuth, D. E. (1975). *The Art of Computer Programming (Sorting and Searching)*. Vol. 3. Addison-Wesley.

Nelan, G. (1991). *Firstification*. Ph.D. thesis, Department of Computer Science, Arizona State University, U.S.A.

Reynolds, J. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Pages 717–740 of: Proceedings of the 25th ACM National Conference*.

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

Rondogiannis, P., & Wadge, W. W. (1993). A Dataflow Implementation Technique for Lazy Typed Functional Languages. *Pages 23–42 of: Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*.

Rondogiannis, P., & Wadge, W. W. (1994a). Compiling Higher-Order Functions for Tagged-Dataflow. *Pages 269–278 of: Proceedings of the IFIP International Conference on Parallel Architectures and Compilation Techniques*. North-Holland.

Rondogiannis, P., & Wadge, W. W. (1994b). Higher-Order Dataflow and its Implementation on Stock Hardware. *Pages 431–435 of: Proceedings of the ACM Symposium on Applied Computing*. ACM Press.

Rondogiannis, P., & Wadge, W. W. (1997). First-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, **7**(1), 73–101.

Rondogiannis, P., Gergatsoulis, M., & Panayiotopoulos, T. (1997). Cactus: A Branching-Time Logic Programming Language. *Pages 511–524 of: Proceedings of the International Joint Conference on Qualitative and Quantitative Practical Reasoning*. Lecture Notes in Artificial Intelligence, vol. 1244. Springer Verlag.

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

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

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

Wadge, W. W. (1991). Higher-Order Lucid. *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming.*

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

Wadge, W.W., Brown, G.D., m.c. schraefel, & Yildirim, T. (1998). Intensional HTML. *Pages 128–139 of: Proceedings of the Fourth International Workshop on Principles of Digital Document Processing (PODDP '98).* Lecture Notes in Computer Science (LNCS) 1481. Springer-Verlag.

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