Higher-Order Functional Languages and Intensional Logic

by

Panagiotis Rondogiannis
Ptihion, University of Patras, 1989
M.Sc., University of Victoria, 1991

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
in the
Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)

Dr. M. Levy, Departmental Member (Dept. of Computer Science)

Dr. G. Shoja, Departmental Member (Dept. of Computer Science)

Dr. J. Phillips, Outside Member (Dept. of Mathematics and Statistics)

Dr. R. Kieburtz, External Examiner (Dept. of Comp. Sc., Oregon Graduate Institute)

Supervisor: Dr. W. W. Wadge

Abstract

The purpose of this dissertation is to demonstrate that higher-order functional programs can be transformed into zero-order intensional ones in a semantics preserving way. As there exists a straightforward execution model for the resulting intensional programs, the practical outcome of our research is a promising, well-defined implementation technique for functional languages. On the foundational side, the goal of our study is to bring new insights and a better understanding of the nature of functional programming.

The starting point of our research is the work of A. Yaghi [Yag84] and W. Wadge [Wad91], who were the first to define transformation algorithms from functional to intensional languages. More specifically, Yaghi studied the first-order subset of functional languages, while Wadge extended Yaghi's technique to apply to a significant class of higher-order functional programs. The main shortcoming of both these works is that the transformations they provide are semi-formal and consequently they lack a correctness proof. In particular, although the algorithm in [Yag84] is relatively easy to understand intuitively, the one in [Wad91] is much more complex, making in this way imperative the need for a precise formulation.

We start by revising, formalizing and giving a correctness proof of Yaghi's transformation algorithm for first-order functional programs. The formal definition we give is based on the idea that if two expressions in the source program are identical, then they are assigned identical intensional expressions during the translation. The correctness proof of the algorithm is established by showing that a function call in the extensional program has the same meaning as the intensional expression that results from its translation.

We then consider the translation of higher-order functional programs into zero-order intensional ones. We demonstrate that although Wadge's algorithm is in the right direction, it does not always preserve the semantics of the source programs. To overcome this deficiency, we define a richer target intensional language and an extended algorithm which

compiles the source functional programs into zero-order programs of this new language. We develop the *synchronic* denotational semantics of the intensional language, based on which we give the correctness proof of the extended transformation algorithm.

The transformation algorithm developed in this dissertation can be used as the basis for new implementation strategies for functional languages. We propose two such strategies, one hashing-based and the other stack-based, and discuss their relative merits. We conclude by demonstrating that the transformation algorithm we propose offers a solution to the problem of implementing higher-order functions on dataflow machines.

Examiners:

Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)

Dr. M. Levy, Departmental Member (Dept. of Computer Science)

Dr. G. Shoja, Departmental Member (Dept. of Computer Science)

Dr. J. Phillips, Outside Member (Dept. of Mathematics and Statistics)

Dr. R. Kieburtz, External Examiner (Dept. of Comp. Sc., Oregon Graduate Institute)

Table of Contents

List of Tables

List of Figures

Acknowledgements

There are many people that have contributed in their own way in the completion of this dissertation. First and most importantly, my supervisor Bill Wadge, whom I came to know when I registered as a Master's student in his "Dataflow Computation" course. I never expected at that point that the *branching time* concept that Bill was illustrating in class would become the topic of my Ph.D. dissertation. I am grateful to Bill for all his guidance, patience and support throughout my studies, and I will always feel proud and priviledged to have been one of his students.

The members of my Ph.D. committee R. Kieburtz, M. Levy, J. Phillips and A. Shoja have been very helpful and supportive. In particular I would like to thank Ali Shoja for all the interesting discussions we occasionally had during my Ph.D. studies.

I had a great time living and studying in Victoria and this is largely due to all the friends that I made here. I hope that we will often have the chance to meet again in the future.

Lia Kontopidi is always for me a source of encouragement and advice. Her smile has helped me cope with many of the difficulties and disappointments that are there when you start a Ph.D.

This dissertation is dedicated to my parents as a recognition of all their tireless efforts, their love and understanding. One of the many things that I owe to them is the love for knowledge, which motivated me to undertake graduate work. My brother Thanasis is always for me the best person to share ideas with (scientific or not). I wish him good luck in his own Ph.D. studies. My sister Marianna, although much younger than me, has been a motivating example of erudition.

Victoria is one of the most beautiful places I have ever been. However, my island Lefkada and the warm Ionian See have constantly been in my mind for the last few years, making my farewell a less difficult one.

Στους γονείς μου

Αριστοφάνη και Αγγελική

Chapter 1

Introduction

The purpose of this dissertation is to demonstrate that higher-order functional programs can be transformed into zero-order intensional ones, in a semantics preserving way. As there exists a straightforward execution model for the resulting intensional programs, the practical outcome of our research is a promising implementation technique for functional languages. On the foundational side, the goal of our study is to bring new insights and a better understanding of the nature of functional programming.

The rest of this chapter is devoted to an intuitive introduction of the main underlying concepts and the most important contributions of this dissertation. We first outline the basic notions of functional programming, and give an introduction to intensional logic and the associated paradigm of intensional programming. The computational model of *eduction* that has been used for implementing intensional languages is then presented. The transformation of functional programs into intensional ones is discussed, and the main problems in giving such a transformation are presented. We conclude by highlighting the main contributions of our work and by giving a chapterwise summary of the dissertation.

1.1 Functional Languages

One of the major challenges of computer science is the design of programming language paradigms that would free the programmers from low-level, machine-related tasks. Such paradigms are usually based on sound mathematical foundations and they allow for a cleaner and more declarative way of programming. The class of *functional* or *applicative* programming languages [Hud89, FH88, Jon87], in which computation is carried out entirely through the evaluation of expressions, is one such approach. As an example, consider the following recursively defined functional program, which computes the fourth Fibonacci number:

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

Functions like `fib` above whose arguments are simple data values (integers, reals, and so on), are called *first-order* functions. One of the most important characteristics of functional programming is the use of *higher-order* functions, that is functions which take as parameters other functions or return functions as results. For example:

```
result      ≐  twice(sq,2)
twice(f,x)  ≐  f(f(x))
sq(y)       ≐  y*y
```

In this example, `twice` is a function of two arguments, the first one of which is another function, and the second one is an integer. The function `twice` is *second-order* because its first argument is a first-order function. Along the same lines, we can have *third-order* functions, or in general *m-order* ones, for every natural number m. These notions will be precisely defined in Chapter 2.

One problem with functional programming languages is that they can not easily express *iteration* in a natural way. For example, the Fibonacci function we gave in this section is a recursively defined one, although there exists a simple iterative algorithm for solving the

same problem. In the next section we describe intensional languages and how they can be used to express iterative algorithms in a problem-oriented manner.

1.2 Intensional Logic and Intensional Languages

Intensional logic [Tho74, DWP81, vB88] is a mathematical formal system for describing entities whose value depends on implicit contexts. The need for such a logic became apparent when the study of natural languages was undertaken by linguists and logicians. Consider for example the following natural language expression:

Iceland is covered with a glacier

The truth value of this expression varies according to an implicit time context: at the present time the above expression is false; however, there existed some time in the past, when the expression was true. Therefore, the semantic value of the expression is really a function from time-points to truth values. One can easily think of other expressions whose truth value depends on more than one coordinates, such as for example *space*, *speaker*, *audience*, and so on. In general, the semantic value of an expression is a function from contexts (also called *possible worlds*) to a set of values. This function is called the *intension* of the expression. The value of the intension at a particular context, is called the *extension* of the expression at that particular context. Consider now the following two expressions:

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

The intension of the first expression is a function which given a date returns the exchange rate on that day. This intension can be visualized as shown in Table 1.1. On the other hand, the intension of the second expression is a function which given a date returns the exchange rate on the previous day. This intension is represented in Table 1.2.

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

Table 1.1: The intension of the first expression

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

Table 1.2: The intension of the second expression

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

The above examples indicate that the meaning of many natural language expressions can be captured using intensions as well as context switching operators (like "Yesterday's"). In intensional logic the concept of intension is prevalent and a change of context occurs by the use of appropriate operators and not by explicit context manipulation. This intuitively justifies why intensional logic has been proven to be an effective tool in the study of the semantics of natural languages.

Intensional Programming is a programming paradigm that is based on intensional logic. The main characteristic of intensional languages is that they are equipped with context switching operators, which allow values from different contexts to be combined without explicit context manipulation. One such intensional language is Lucid [WA85], in which the value of an expression depends on a hidden time parameter. Therefore, the value of a Lucid expression is a *stream* of ordinary data values. Moreover, the usual operations (like **+**, **if-then-else**, and so on), take streams as arguments and apply to them in a pointwise

way. For example, consider the following simple Lucid program:

```
result  ≐  2+3
```

The meanings of 2 and 3 above, are the infinite streams $\langle 2,2,\ldots\rangle$ and $\langle 3,3,\ldots\rangle$ respectively. The meaning of the variable **result** is the stream $\langle 5,5,\ldots\rangle$, which results from adding in a pointwise way the two other streams. Lucid provides a way of creating more "interesting" streams, using the intensional binary operator **fby**. Given two streams $x = \langle x_0, x_1, \ldots\rangle$ and $y = \langle y_0, y_1, \ldots\rangle$, the stream $(x\ fby\ y)$ is defined at every time t as follows:

$$(x\ fby\ y)_t = \begin{cases} x_0 & \text{if } t = 0 \\ y_{t-1} & \text{if } t > 0 \end{cases}$$

For example, the meaning of the Lucid program

```
result  ≐  a
a       ≐  1 fby a+1
```

is the stream $\langle 1,2,3,\ldots\rangle$. Using the **fby** operator, one can easily express the iterative version of the Fibonacci function as follows:

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

Notice that the above program computes the stream of *all* Fibonacci numbers, that is the stream $\langle 1,1,2,3,5,\ldots\rangle$. Notice also that the Lucid `fib` program does not have any function definitions, and in this respect it is simpler than the `fib` recursive function of Section 1.1. Moreover, as the next section illustrates, there exists a very simple technique for implementing such Lucid programs.

1.3 The Computational Model of Eduction

The traditional implementation of Lucid programs like the ones given in the last section, is based on a computational model known as eduction [WA85]. We illustrate the main idea of eduction using an example. Suppose we want to calculate the second Fibonacci number. In order to do so, we demand the value of `result` at time 2. This generates a demand for `fib` at time 2, which creates a demand for `(1 fby (fib+g))` at time 2. But now, according to the semantics of **fby**, this will generate a demand for `(fib+g)` at time 1. The overall execution by an eductive evaluator *EVAL*, is given in Figure 1.1. Therefore, eduction is based on demand propagation, and the way this is achieved is by simply following the semantics of the program under consideration.

figure 1.1 Execution of intensional code

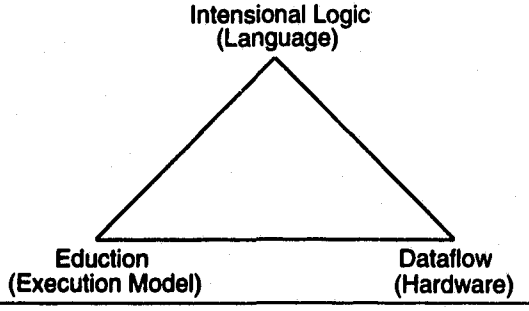
$$
\begin{aligned}
& EVAL(\texttt{result}, 2) = \\
= \;& EVAL(\texttt{fib}, 2) \\
= \;& EVAL(\texttt{(1 fby (fib+g))}, 2) \\
= \;& EVAL(\texttt{(fib+g)}, 1) \\
= \;& EVAL(\texttt{fib}, 1) + EVAL(\texttt{g}, 1) \\
= \;& EVAL(\texttt{(1 fby (fib+g))}, 1) + EVAL(\texttt{(0 fby fib)}, 1) \\
= \;& EVAL(\texttt{(fib+g)},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

Notice that the main characteristic of eduction is that it computes the value of expressions with respect to contexts. There exists a class of hardware architectures (namely the *dataflow* one [JGW85, AN90]), that efficiently supports such execution with respect to context. In other words, dataflow machines are ideal candidates on which eduction can be implemented. This suggests the triangle given in Figure 1.2, in which:

- Intensional Logic provides the language paradigm on which programs are written or compiled to.

- Eduction provides the conceptual execution model for implementing the intensional programs.
- Dataflow architectures provide the appropriate hardware on which eduction can be executed in an efficient way.

figure 1.2 Intensional Logic, Eduction and Dataflow



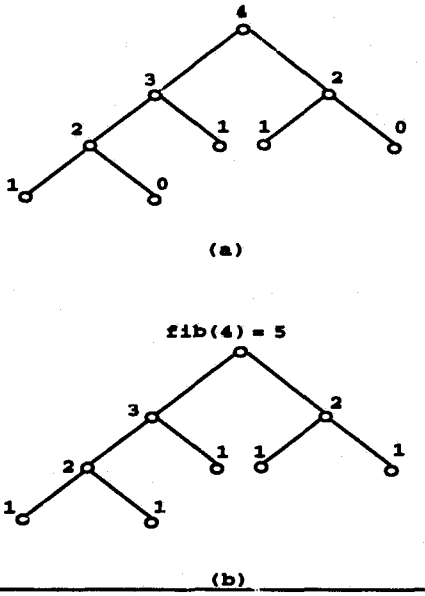
The above description suggests that in order to implement a programming language on a dataflow architecture, we would first have to devise a way of compiling programs of this language into (semantically equivalent) intensional ones. The next section discusses how this can be done for the case of functional languages, or in other words how functional programs can be transformed into Lucid-like programs on which eduction can be easily performed.

1.4 Intensional Logic and Functional Languages

The first work to establish a relationship between intensional logic and functional programming was A. Yaghi's Ph.D. dissertation [Yag84]. In his work, Yaghi used intensional logic to formalize an implementation technique for first-order functional languages that was invented by C. Ostrum at the University of Waterloo. Yaghi first defined a simple intensional programming language that only supported nullary variable definitions. He then showed that the main idea in Ostrum's implementation could be understood as a translation of the source functional program into a program of this intensional language.

The main idea behind Yaghi's work is that functions are really intensions. In the following, we demonstrate his ideas with an example. Consider the Fibonacci program that was presented in Section 1.1. In order to compute `fib(4)`, we need to know `fib(3)` and `fib(2)`. Similarly, `fib(3)` requires `fib(2)` and `fib(1)`, and so on. Therefore, one can actually think of the formal parameter `n` as being a labeled tree of the form shown in Figure 1.3(a). Similarly, the function `fib` can be thought of as a labeled tree that has been created

figure 1.3 (a) Tree for the parameter `n` (b) Tree for the function `fib`



by "consulting" the tree for `n`. Figure 1.3(b) illustrates the corresponding tree. The bottom labels of the tree for `fib` are all equal to 1, because this is the value that `fib` takes when the corresponding value of `n` is less than 2. As we move up the tree for `fib`, the label on each node is formed by adding the values of the right and left children of the node. The initial program can be transformed into a new one that reflects the above ideas:

```
result  ≐  call1(fib)
fib     ≐  if (n<2) then 1 else call2(fib)+call3(fib)
n       ≐  actuals(4,n-1,n-2)
```

Notice that the above program is a Lucid-like one, the only difference being that it is manipulating tree intensions and not just stream ones. The definition of **n** in terms of **actuals** expresses the fact that **n** is a tree with root labeled 4; the root of the left subtree is equal to the current root minus one, and the root of the right subtree is the current root minus two. Clearly, one can proceed in this way and create the whole tree for **n** as give in Figure 1.3(a). The operators $\mathbf{call}_i$ are used in order to create the tree for `fib`. The definition for `fib` can be read as follows:

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

In other words, $\mathbf{call}_2$ selects the root of the left subtree of the current node of `fib`, while $\mathbf{call}_3$ the root of the right subtree. The operator $\mathbf{call}_1$ returns the root of the tree for `fib`.

The above description presents at an intuitive level the relationship between first-order functional programs and intensional ones. The algorithm for performing the transformation in a systematic way is given in [Yag84]. Yaghi was motivated mainly by practical considerations, and therefore his work, although ground-breaking, is incomplete in two respects:

- The transformation algorithm from first-order programs to intensional programs is semi-formal.
- A correctness proof of the algorithm, although attempted by Yaghi and subsequently by others, was not obtained.

Moreover, Yaghi only considered first-order functional languages, a fact that restricted the usefulness of his proposal. A generalization of the technique to higher-order functional programs would be a very significant step, because until today the implementation of such programs on dataflow architectures has always been problematic: the approach usually followed is to adopt some hybrid non-dataflow implementation scheme. The following quotes are relevant:

> "The general **apply** schema [for implementing higher-order functions on dataflow machines] is of course not inexpensive" [AN90]

> "... [the language] Id has adopted much of the flavor of modern functional languages, including higher-order functions (which, incidentally, are not easily implemented on a dataflow machine)" [Hud91]

In 1991, W. Wadge suggested [Wad91] that it might be possible to use a variation of Yaghi's technique to gradually transform higher-order functional programs into intensional programs of nullary variables. The main idea in [Wad91] is that the translation proceeds in stages; at each stage the highest order formal parameters are eliminated from function definitions, and a new definition is created for each such formal. Moreover, different intensional operators are used for each stage of the translation process. As an example, consider the program:

```
result     ≐ apply(inc,8)
apply(f,x) ≐ f(x)
inc(y)     ≐ y+1
```

This second-order program is initially transformed into the following first-order one (for the moment, we do not give any further details on how the transformation is performed or what the semantics of the intensional operators are):

```
result     ≐ call⟨2,1⟩apply(8)
apply(x)   ≐ f(x)
inc(y)     ≐ y+1
f(z)       ≐ actuals2(inc(z))
```

Then, using an identical procedure, the above first-order intensional program can be reduced to the following zero-order intensional program, which is the output of the trans-

formation:

$$\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{\langle 1,1\rangle}(\texttt{call}_{\langle 2,1\rangle}(\texttt{apply})) \\
\texttt{apply} &\doteq \texttt{call}_{\langle 1,1\rangle}(\texttt{f}) \\
\texttt{inc} &\doteq \texttt{y+1} \\
\texttt{f} &\doteq \texttt{actuals}_2(\texttt{call}_{\langle 1,1\rangle}(\texttt{inc})) \\
\texttt{z} &\doteq \texttt{actuals}_1(\texttt{x}) \\
\texttt{y} &\doteq \texttt{actuals}_1(\texttt{z}) \\
\texttt{x} &\doteq \texttt{actuals}_1(8)
\end{aligned}$$

In general, the material in [Wad91] is presented at an informal level, and can only be considered as a general suggestion of how higher-order programs should be treated. Moreover, although the underlying ideas in [Wad91] are in the right direction, the overall technique is inadequate as we have demonstrated in [Ron92] (see also Section 2.9). Since the work in [Yag84] and [Wad91] forms the starting point of our investigations, we will describe it in more detail in Chapter 2.

1.5 The Purpose of this Dissertation

The purpose of this dissertation is to establish in a precise way the relationships between functional languages and intensional logic. The main contributions of our work, can be summarized as follows:

1. We give a formal definition and a correctness proof of Yaghi's transformation algorithm. It should be emphasized at this point that both problems are non-trivial, and remained open for almost one decade.

2. We define a higher-order intensional language and present its denotational semantics. This language will serve as the target one for transforming higher-order functional programs.

3. We give a precise transformation algorithm from a significant class of higher-order functional programs to the target intensional language that we defined.

4. We demonstrate the correctness proof of the transformation algorithm we propose. In this way, we establish for the first time, a semantics preserving transformation from higher-order functional programs into intensional programs.

5. We show that the transformation algorithm can be used as the basis for new implementation strategies for higher-order functional languages, that are based on the eduction model.

1.6 Summary of the Dissertation

In this section we present a chapterwise summary of the contents of this dissertation:

Chapter 2 introduces the basic mathematical notation that we adopt. The syntax and semantics of a simple higher-order functional language are presented. An intensional language of nullary variables is introduced, and Yaghi's algorithm for transforming first-order functional programs into programs of this language, is outlined. Wadge's proposal for extending Yaghi's approach to apply to a class of higher-order functional programs is presented, and its deficiencies are identified and discussed. The chapter concludes with discussion of other related work.

In Chapter 3, we give for the first time a rigorous formal definition and a correctness proof of a revised version of Yaghi's transformation algorithm. The main points of the proof are highlighted, discussed and illustrated by examples.

In Chapter 4, we introduce the higher-order intensional language IL. The purpose of IL is to serve as the target language for transforming higher-order functional programs. For this reason, IL is equipped with powerful intensional operators that can capture the complexities of the source functional language. We define the *synchronic* denotational semantics of IL, and prove certain of its properties.

In Chapter 5, we formally define the transformation algorithm from the class of higher-order functional programs we consider, to intensional programs of nullary variables. The algorithm is motivated by examples and some of its properties are identified and proved.

In Chapter 6 we present a correctness proof for the algorithm introduced in Chapter 5. The main points of the proof are highlighted and the insights gained from it are discussed.

Chapter 7 introduces certain practical implications of our work. In particular, having as a starting point the transformation algorithm introduced in Chapter 5, we propose two eduction-based strategies for implementing higher-order functional languages. Moreover, we demonstrate that the transformation technique developed in this dissertation, offers a solution to the long-lasting problem of implementing higher-order functions on dataflow machines.

Chapter 8 concludes the dissertation by discussing open problems as well as possible extensions of our work.

Chapter 2

Background and Related Work

This chapter introduces the background material that is used throughout the dissertation. We assume familiarity with the main notions of set theory and logic [Sto79, Bar77] as well as a basic understanding of domain theory and denotational semantics [Man74, Sto77, Ten76, EW82, Ten91, Gun92]. In the following, we initially present the mathematical notation we adopt. Then, a simple typed functional language is introduced and its denotational semantics are defined. An intensional language of nullary variables is presented, and Yaghi's algorithm [Yag84] for transforming first-order functional programs into programs of this language, is outlined. Wadge's suggestion [Wad91] for extending Yaghi's approach to a significant class of higher-order programs, is then presented. The chapter concludes with a discussion of related research.

2.1 Mathematical Notation

The set of natural numbers is denoted by N. The domain and range of a function f are represented by $dom(f)$ and $range(f)$ respectively. For simplicity, we write in certain cases f_a instead of $f(a)$. Moreover, when convenient, we use the λ-notation to represent functions

[Bar84, HS86]. The set of functions from A to B is denoted by $A \rightarrow B$ or B^A. Given two sets I and S, an I-indexed sequence is any function $s : I \rightarrow S$, and is denoted by $\langle s_i \rangle_{i \in I}$. The set I is called the index set of s. The composition $\lambda x.f(g(x))$ of two functions f and g is denoted by $f \circ g$. 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

$$\Pi_{i \in I} A_i = \{f : I \rightarrow \bigcup_{i \in I} A_i \mid \forall i \in I, f(i) \in A_i\}$$

The functions f can be thought of as sets of tuples with one component from A_i for every $i \in I$. The perturbation of a function with respect to another function, is defined as follows:

Definition 2.1 Let $f : A \rightarrow B$ and $g : S \rightarrow B$, where $S \subseteq A$. Then, the *perturbation* $f \oplus g$ of f with respect to g is defined as:

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

Given a function $g = \{\langle x_1, b_1 \rangle, \ldots, \langle x_n, b_n \rangle\}$, we will often write $f[x_1/b_1, \ldots, x_n/b_n]$ instead of $f \oplus g$.

We write $List(N)$ for the set of lists of natural numbers. The usual list operations *head*, *tail* and *cons* are adopted. The infix notation ":" will often be used instead of *cons*.

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, $[A \rightarrow B]$ 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., $\mathbf{f}, \mathbf{x}, \mathbf{P}, \mathbf{E}, \ldots$).

2.2 Types

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 dissertation.

Definition 2.2 The set Typ of types is recursively defined as follows:

- $\iota \in Typ$.
- If $\tau_1, \ldots, \tau_n \in Typ$ then $(\tau_1, \ldots, \tau_n) \rightarrow \iota \in Typ$.

Notice that the result component of a member of Typ is always ground, that is, equal to ι. As it will be described shortly, the languages that are considered in this dissertation, are subject to this restriction in the sense that all functions defined in them, should have a type that belongs to Typ. The various objects that are used by the functional language that we will be adopting can be classified according to their *type level* or *order*. Intuitively, the simplest kind of objects allowed by the language are ordinary data values (such as for example integers or reals). These are classified as being type-0 (or zero-order). The language also allows functions whose arguments are type-0 objects and whose results are type-0 objects; these are the type-1 objects. In general, we classify as type-$(n+1)$ (or $(n+1)$-order) all those functions whose arguments are type-n or less, and their result is type-0. Formally, the order and the denotation of a type are defined as follows:

Definition 2.3 The *order* of a type $\tau \in Typ$ is defined as follows:

$$\begin{aligned} order(\iota) &= 0 \\ order((\tau_1, \ldots, \tau_n) \rightarrow \iota) &= 1 + max(\{order(\tau_i) \mid 1 \leq i \leq n\}) \end{aligned}$$

Definition 2.4 The denotation of $\tau \in Typ$ 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:

- $[\![\iota]\!]_D = D$
- $[\![(\tau_1, \ldots, \tau_n) \rightarrow \iota]\!]_D = [([\![\tau_1]\!]_D, \ldots, [\![\tau_n]\!]_D) \rightarrow [\![\iota]\!]_D]$

A *signature* Σ is a set of constant symbols of various types over Typ. Elements of Σ are assigned types by a *type assignment* function $\theta : \Sigma \rightarrow Typ$. Constants are denoted by $\mathbf{c}$. The set Σ_n, $n \in N$, is the subset of Σ whose elements have order less than or equal to n:

$$\Sigma_n = \{\mathbf{c} \in \Sigma \mid order(\theta(\mathbf{c})) \leq n\}$$

We also assume the existence of a set Var of variable symbols, whose elements are assigned types by $\pi : Var \rightarrow Typ$. Variables are denoted by $\mathbf{f}, \mathbf{g}, \mathbf{x}, \ldots$. As before,

$$Var_n = \{\mathbf{f} \in Var \mid order(\pi(\mathbf{f})) \leq n\}$$

Variable (constant) symbols of type ι are also called *nullary* or *individual* variables (constants). Non-nullary variables are also termed *function* variables.

2.3 The Functional Language FL

In this section, we define the syntax and denotational semantics of the typed, higher-order functional language FL. In the following, FL will also be referred as an *extensional*

language, to distinguish it from *intensional* languages, that will be defined later on in this dissertation.

Definition 2.5 The syntax of the functional language FL 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*:

$$\begin{aligned}
\mathbf{E} &::= \mathbf{f} \in Var \\
&\mid \mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ \mathbf{c} \in \Sigma_1 \\
&\mid \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ \mathbf{f} \in Var \\
\mathbf{F} &::= (\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}),\ \mathbf{f}, \mathbf{x}_1, \ldots, \mathbf{x}_n \in Var \\
\mathbf{P} &::= \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}
\end{aligned}$$

Given a definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}$, the variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the *formal parameters* or *formals* of $\mathbf{f}$, and $\mathbf{E}$ is the *defining expression* or the *body* of $\mathbf{f}$.

Definition 2.6 Let $\mathbf{P} = \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}$ be a program. Then the following assumptions are adopted:

1. Exactly one of the $\mathbf{F}_1, \ldots, \mathbf{F}_n$ 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, at most once in the whole program.
3. The formal parameters of a function definition in $\mathbf{P}$ can only appear in the body of that definition.
4. The only variables that can appear in $\mathbf{P}$ are the ones defined in $\mathbf{P}$ and their formal parameters.

The set of variables defined in a program $\mathbf{P}$ is denoted by $func(\mathbf{P})$, while the set of variables that are defined or appear as formal parameters in $\mathbf{P}$ is denoted by $Vars(\mathbf{P})$. The

type-checking rules for the language are given as natural deduction rules with sequents of the form $\mathbf{E} : \tau$. The sequent $\mathbf{E} : \tau$ asserts that $\mathbf{E}$ is a well-formed expression of type τ provided that the identifiers and constants that are used in $\mathbf{E}$, have the types assigned to them by π and θ respectively.

Definition 2.7 The set of well-typed expressions is recursively defined as follows:

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

$$\frac{(\theta(\mathbf{c}) = (\tau_1, \ldots, \tau_n) \rightarrow \iota) \wedge (\mathbf{E}_1 : \tau_1, \ldots, \mathbf{E}_n : \tau_n)}{\mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n) : \iota}$$

$$\frac{(\pi(\mathbf{f}) = (\tau_1, \ldots, \tau_n) \rightarrow \iota) \wedge (\mathbf{E}_1 : \tau_1, \ldots, \mathbf{E}_n : \tau_n)}{\mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n) : \iota}$$

Definition 2.8 A definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}$ with $\mathbf{f} : (\tau_1, \ldots, \tau_n) \rightarrow \iota$ is well-typed if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and $\mathbf{E} : \iota$.

Definition 2.9 A program $\{\mathbf{F}_1, \ldots, \mathbf{F}_n\}$ is well-typed if $\mathbf{F}_1, \ldots, \mathbf{F}_n$ are well-typed definitions.

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

Definition 2.10 Let $\mathbf{P}$ be an *FL* program. The order of $\mathbf{P}$ is defined as:

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

Definition 2.11 The language *FOFL* is the subset of *FL* in which all programs are first-order.

The purpose of this dissertation is to investigate how extensional languages like $FOFL$ and FL, can be transformed in a sound way, into intensional languages (to be defined shortly). For this purpose, we will start our investigations from the simpler first-order language $FOFL$, and in later chapters we will extend our results to the more powerful and expressive language FL.

2.4 The Semantics of FL

Let D be a domain. Then, the semantics of constant symbols of FL with respect to D, are obtained by a given interpretation function $\mathcal{C}$, which assigns to every constant of type τ, a function in $[\![\tau]\!]_D$. Let Exp_τ be the set of all expressions $\mathbf{E}$ of FL such that $\mathbf{E} : \tau$. Let Env_π be the set of π-compatible environments defined by $Env_\pi = \Pi_{\mathbf{f} \in Var}[\![\pi(\mathbf{f})]\!]_D$. Then, the semantics of FL is defined using valuation functions $[\![\cdot]\!]_D : Exp_\tau \rightarrow [Env_\pi \rightarrow [\![\tau]\!]_D]$, (where the subscripts D and π will be omited when they are obvious from context).

Definition 2.12 The semantics of expressions of FL with respect to $u \in Env$, are recursively defined as follows:

$$[\![\mathbf{f}]\!](u) = u(\mathbf{f})$$
$$[\![\mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n)]\!](u) = \mathcal{C}(\mathbf{c})([\![\mathbf{E}_1]\!](u), \ldots, [\![\mathbf{E}_n]\!](u))$$
$$[\![\mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)]\!](u) = u(\mathbf{f})([\![\mathbf{E}_1]\!](u), \ldots, [\![\mathbf{E}_n]\!](u))$$

Definition 2.13 The semantics of the program $\mathbf{P} :\!- \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}$ of FL with respect to $u \in Env_\pi$, is defined as $\tilde{u}(\mathbf{result})$, where $\tilde{u}$ is the *least* environment such that:

1. For every $\mathbf{f} \in Var$ with $\mathbf{f} \notin func(\mathbf{P})$, $\tilde{u}(\mathbf{f}) = u(\mathbf{f})$.
2. For every $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$ such that $\mathbf{f} : (\tau_1, \ldots, \tau_n) \rightarrow \iota$, and for all $d_1 \in [\![\tau_1]\!]_D, \ldots, d_n \in [\![\tau_n]\!]_D$, $\tilde{u}(\mathbf{f})(d_1, \ldots, d_n) = [\![\mathbf{E}]\!](\tilde{u}[\mathbf{x}_1/d_1, \ldots, \mathbf{x}_n/d_n])$.

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

Theorem 2.1 [Ten91, page 96] Let $\mathbf{P}$ and $\tilde{u}$ be as in Definition 2.13. Then, $\tilde{u}$ is the least upper bound of the environments $\tilde{u}_k$, $k \in N$, which for every definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$, with $\mathbf{f} : (\tau_1, \ldots, \tau_n) \to \iota$, and for all $d_1 \in [\![\tau_1]\!]_D, \ldots, d_n \in [\![\tau_n]\!]_D$, are defined as follows:

$$\begin{aligned} \tilde{u}_0(\mathbf{f})(d_1, \ldots, d_n) &= \bot_D \\ \tilde{u}_{k+1}(\mathbf{f})(d_1, \ldots, d_n) &= [\![\mathbf{E}]\!](\tilde{u}_k[\mathbf{x}_1/d_1, \ldots, \mathbf{x}_n/d_n]) \end{aligned}$$

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

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

Lemma 2.1 Let $\mathbf{P}$ and $\tilde{u}$ be as in Definition 2.13. Then, for every definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$ with $\mathbf{f} : (\tau_1, \ldots, \tau_n) \to \iota$,

$$\tilde{u}_k(\mathbf{f})(d_1, \ldots, d_n) \sqsubseteq [\![\mathbf{E}]\!](\tilde{u}_k[\mathbf{x}_1/d_1, \ldots, \mathbf{x}_n/d_n])$$

for all $d_1 \in [\![\tau_1]\!]_D, \ldots, d_n \in [\![\tau_n]\!]_D$.

The following theorem will also be used in subsequent chapters:

Theorem 2.2 [Ten91, page 97] For all expressions $\mathbf{E} \in Exp_\tau$, $[\![\mathbf{E}]\!]$ is monotonic and continuous. Moreover, when $\tau \neq \iota$, $[\![\mathbf{E}]\!](u)$ is monotonic and continuous, for all $u \in Env$.

Notice that the semantics of programs of FL have been defined with respect to an initial environment u. Recall now that the programs that we are considering do not contain occurrences of "outside" variables (Definition 2.6). For this reason, we can assume that the initial environment assigns the bottom value (of the appropriate type) to every variable in Var, and we can then talk directly about the least environment that satisfies the definitions in a given program.

2.5 The Intensional Language NVIL

In this section we define (following [Yag84]) the syntax of a simple intensional language of nullary variables. As it will be demonstrated later in this chapter, $NVIL$ can serve as the target language for transforming $FOFL$ programs.

Definition 2.14 The syntax of the intensional language $NVIL$ 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*:

$$
\begin{array}{rcl}
\mathbf{E} & ::= & \mathbf{f} \in Var_0 \\
& | & \mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ \mathbf{c} \in \Sigma_1 \\
& | & \mathbf{call}_i(\mathbf{E}_0),\ i \in N \\
& | & \mathbf{actuals}(\langle \mathbf{E}_i \rangle_{i \in I}),\ I \subseteq N \\
\mathbf{F} & ::= & (\mathbf{f} \doteq \mathbf{E}),\ \mathbf{f} \in Var_0 \\
\mathbf{P} & ::= & \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}
\end{array}
$$

Similar restrictions as in FL are adopted for the syntax of $NVIL$ programs. The typing rules for $NVIL$ are given along the same lines as those of FL. There exist two new rules that have to do with the intensional operators $\mathbf{call}_i$ and **actuals**:

$$\frac{\mathbf{E}_0 : \iota}{\mathbf{call}_i(\mathbf{E}_0) : \iota}$$

$$\frac{\forall i \in I (\mathbf{E}_i : \iota)}{\mathbf{actuals}(\langle \mathbf{E}_i \rangle_{i \in I}) : \iota}$$

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

2.6 The Semantics of NVIL

As discussed in Chapter 1, in intensional languages the meaning of an expression is a function from a set of *possible worlds* to a set of data values. Depending on the target application, a possible world may be a moment in time, a position in space, a node in a tree structure, and so on. In the language $NVIL$ under consideration, variables denote trees of data values (as pointed out in Section 1.4). A node in such a tree can be identified by a list of natural numbers. Therefore:

Definition 2.15 The set W of possible worlds of $NVIL$ is the set $List(N)$ of lists of natural numbers.

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

Definition 2.16 For every n-ary constant $\mathbf{c} \in \Sigma_1$, for every $w \in W$, and for all $a_1, \ldots, a_n \in (W \to D)$, $\mathcal{C}'(\mathbf{c})(a_1, \ldots, a_n)(w) = \mathcal{C}(\mathbf{c})(a_1(w), \ldots, a_n(w))$.

Let Exp be the set of all expressions of $NVIL$. Let Env_π be the set of π-compatible environments defined by $Env_\pi = \Pi_{\mathbf{f} \in Var}[\![\pi(\mathbf{f})]\!]_{(W \to D)}$. Then, the semantics of $NVIL$ is defined using valuation functions $[\![\cdot]\!] : Exp \to [Env_\pi \to (W \to D)]$, as follows:

Definition 2.17 The interpretation of expressions of $NVIL$ with respect to $u \in Env$, is recursively defined for every $w \in W$, as follows:

$$[\![\mathbf{f}]\!](u)(w) = u(\mathbf{f})(w)$$
$$[\![\mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n)]\!](u)(w) = \mathcal{C}'(\mathbf{c})([\![\mathbf{E}_1]\!](u), \ldots, [\![\mathbf{E}_n]\!](u))(w)$$
$$[\![\mathbf{call}_i(\mathbf{E})]\!](u)(w) = [\![\mathbf{E}]\!](u)(i : w)$$
$$[\![\mathbf{actuals}(\langle \mathbf{E}_i \rangle_{i \in I})]\!](u)(w) = [\![\mathbf{E}_{head(w)}]\!](u)(tail(w))$$

Definition 2.18 The semantics of the program $\mathbf{P} = \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}$ of *NVIL* with respect to $u \in Env_\pi$, is defined as $\tilde{u}(\mathbf{result})$, where $\tilde{u}$ is the *least* environment such that:

1. For every $\mathbf{f} \in Var$ with $\mathbf{f} \notin func(\mathbf{P})$, $\tilde{u}(\mathbf{f}) = u(\mathbf{f})$.
2. For every definition $(\mathbf{f} \doteq \mathbf{E})$ in $\mathbf{P}$, $\tilde{u}(\mathbf{f}) = [\![\mathbf{E}]\!](\tilde{u})$.

Notice that the semantics given above for *NVIL* are standard, and their only difference from the ones given for *FL* is that the former is defined on the richer domain $(W \to D)$, while the latter is defined on the domain D. Therefore, the Theorems 2.1, 2.2, and Lemma 2.1, transfer directly to the language *NVIL* as well.

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

2.7 Yaghi's Transformation

The first work to establish a relationship between extensional and intensional functional languages was Ali Yaghi's Ph.D. dissertation [Yag84]. More specifically, Yaghi discovered and described an algorithm for transforming a *FOFL* program into an *NVIL* one.

Yaghi's work, apart from its theoretical significance, had practical implications as well: the resulting intensional programs can be interpreted in a very simple way based on the eduction model. In fact, the Lucid functional-dataflow language [AW76, AW77, WA85], as well as other Lucid-related systems [DW90b, DW90a], are nowadays traditionally implemented based on Yaghi's approach. However, there are two important aspects of the technique, that were not developed in [Yag84] (and which are resolved in this dissertation, Chapter 3):

1. The transformation algorithm from *FOFL* to *NVIL* programs given in [Yag84], is semi-formal and not functional.

2. A correctness proof of the transformation is not given in [Yag84], and has remained an open problem since then.

In this section, we describe Yaghi's technique and outline how it can be used as the basis of an interpreter for first-order functional languages. The algorithm is shown below:

1. Let **f** be a function defined in the source extensional program. Number the textual occurrences of calls to **f** in the program, starting at 1 (including calls in the body of the definition of **f**).

2. Replace the ith call of **f** in the program by $\mathbf{call}_i(\mathbf{f})$. Remove the formal parameters from the definition of **f**, so that **f** is defined as an ordinary individual variable.

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

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

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

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

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

Execution of the program is achieved by actually following the denotational semantics of the intensional program, and using the semantic rules for **call** and **actuals** presented in the previous section. The interpreter starts evaluating the variable **result** of the intensional program under the empty context, i.e., the list [ ]. Every time a variable is encountered during evaluation, the interpreter replaces it by its defining expression. In the following, we use $EVAL$ to represent the function of the evaluator (interpreter). Execution proceeds as shown in Figure 2.1. In his dissertation, Yaghi conjectured that higher-order programs can

figure 2.1 Execution of intensional code

$$
\begin{aligned}
& EVAL(\texttt{call}_1\texttt{(f)+call}_2\texttt{(f)},[\,]) \\
= \;& EVAL(\texttt{call}_1\texttt{(f)},[\,]) + EVAL(\texttt{call}_2\texttt{(f)},[\,]) \\
= \;& EVAL(\texttt{f},[1]) + EVAL(\texttt{f},[2]) \\
= \;& EVAL(\texttt{call}_1\texttt{(g)},[1]) + EVAL(\texttt{call}_1\texttt{(g)},[2]) \\
= \;& EVAL(\texttt{g},[1,1]) + EVAL(\texttt{g},[1,2]) \\
= \;& EVAL(\texttt{y},[1,1]) + EVAL(\texttt{y},[1,2]) \\
= \;& EVAL(\texttt{actuals(x+1)},[1,1]) + EVAL(\texttt{actuals(x+1)},[1,2]) \\
= \;& EVAL(\texttt{x+1},[1]) + EVAL(\texttt{x+1},[2]) \\
= \;& EVAL(\texttt{x},[1]) + EVAL(\texttt{1},[1]) + EVAL(\texttt{x},[2]) + EVAL(\texttt{1},[2]) \\
= \;& EVAL(\texttt{x},[1]) + 1 + EVAL(\texttt{x},[2]) + 1 \\
= \;& EVAL(\texttt{actuals(4,5)},[1]) + 1 + EVAL(\texttt{actuals(4,5)},[2]) + 1 \\
= \;& EVAL(\texttt{4},[\,]) + 1 + EVAL(\texttt{5},[\,]) + 1 \\
= \;& 4 + 1 + 5 + 1 \\
= \;& 11
\end{aligned}
$$

be intensionalized in a similar way, but that probably a richer set of possible worlds would be required.

2.8 Higher-Order Programs: Wadge's Suggestion

The first attempt for generalizing Yaghi's technique to apply to higher-order programs, is described in [Wad91]. In that paper, W. Wadge outlines a technique that could potentially extend Yaghi's intensionalization algorithm. The reader should be cautioned at this point that the following discussion is given at an informal level, following the description in

[Wad91], and that certain of the notions introduced in this section will be corrected and extended in subsequent chapters.

The main idea of Wadge's proposal is that given an m-order program, one can appropriately transform it into an $(m-1)$-order intensional program. This can be performed by eliminating the $(m-1)$-order formals from function definitions, in a similar way as in Yaghi's technique. The same procedure can then be repeated for the new program, until an intensional program of nullary variables is obtained.

Every stage in the transformation corresponds to a different order that is eliminated from the program. Therefore, we use a different set of operators at each step. Let m be the order of the initial program. Then, for the first step we use the operators $\mathbf{actuals}_m$ and $\mathbf{call}_{\langle m,i\rangle}$, where i ranges as in the first-order case. For the second step, we use $\mathbf{actuals}_{m-1}$ and $\mathbf{call}_{\langle m-1,i\rangle}$, and so on.

Consequently, contexts are now multidimensional: for the translation of an m-order program, a context is an m-tuple of lists, where each list corresponds to a different order of the program. The code that results from the transformation can be executed following the same basic principles as in the first-order case. The above ideas are illustrated with the following simple second-order extensional program:

```
result     ≐ apply(inc,8)
apply(f,x) ≐ f(x)
inc(y)     ≐ y+1
```

The function **apply** is second-order because of its first argument. The generalized transformation, in its first stage eliminates this argument:

```
result     ≐ call⟨2,1⟩apply(8)
apply(x)   ≐ f(x)
inc(y)     ≐ y+1
f          ≐ actuals2(inc)
```

We see that the program that resulted above is first-order: all the functions have zero-order arguments. The only exception is the definition of **f** which is an equation between function expressions. In [Wad91], the suggestion is made that this can be changed by introducing a formal parameter **z** for **f**:

$$\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{\langle 2,1\rangle}\texttt{apply(8)} \\
\texttt{apply(x)} &\doteq \texttt{f(x)} \\
\texttt{inc(y)} &\doteq \texttt{y+1} \\
\texttt{f(z)} &\doteq \texttt{actuals}_2\texttt{(inc(z))}
\end{aligned}$$

This completes the first stage of the transformation. Now, we have a first-order intensional program, and we can apply the technique for the first-order case, which gives the final program:

$$\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{\langle 1,1\rangle}\texttt{(call}_{\langle 2,1\rangle}\texttt{(apply))} \\
\texttt{apply} &\doteq \texttt{call}_{\langle 1,1\rangle}\texttt{(f)} \\
\texttt{inc} &\doteq \texttt{y+1} \\
\texttt{f} &\doteq \texttt{actuals}_2\texttt{(call}_{\langle 1,1\rangle}\texttt{(inc))} \\
\texttt{z} &\doteq \texttt{actuals}_1\texttt{(x)} \\
\texttt{y} &\doteq \texttt{actuals}_1\texttt{(z)} \\
\texttt{x} &\doteq \texttt{actuals}_1\texttt{(8)}
\end{aligned}$$

In the execution model for a program of order m, contexts are m-tuples of lists of natural numbers, and each list corresponds to a different order of the initial program (or equivalently, a different stage in the transformation). We will use the notation $\langle w_1, ..., w_m \rangle$ to denote a context. The operators **call** and **actuals** can now be thought of as operations on these more complicated contexts.

Let $s \in \{1, \ldots, m\}$. Consider the operator $\textbf{call}_{\langle s,i\rangle}$. Given a context, s is used in order to select the corresponding list from the context. The list is then prefixed with i and returned to the context. On the other hand, $\textbf{actuals}_s$ takes from the context the list corresponding to s, uses its head i to select its ith argument, and returns the tail of the list to the context.

Let $a, a_1, \ldots, a_n$ be intensions. Then, the new semantic equations, which are implicit in [Wad91], are:

$$(call_{(s,i)}(a))(\langle w_1, \ldots, w_s, \ldots, w_m \rangle) = (a)(\langle w_1, \ldots, (i : w_s), \ldots, w_m \rangle)$$
$$(actuals_s(a_1, \ldots, a_n))(\langle w_1, \ldots, w_s, \ldots, w_m \rangle) = (a_{head(w_s)})(\langle w_1, \ldots, tail(w_s), \ldots, w_m \rangle)$$

The new operators can therefore be viewed as a generalization of the operators for the first-order case. The evaluation of a program starts with an m-tuple that contains m empty lists, one for each order. Execution proceeds as in the first-order case, the only difference being that the appropriate list within the tuple is accessed every time. The execution of the **apply** program is given in Figure 2.2. The technique described above works for this example, but this is not the case in general. The following section indicates why.

figure 2.2 Execution of the intensional code that results from **apply**

$$\begin{aligned}
& EVAL(\texttt{call}_{(1,1)}(\texttt{call}_{(2,1)}(\texttt{apply})), \langle [\,], [\,] \rangle) \\
= \; & EVAL(\texttt{call}_{(2,1)}(\texttt{apply}), \langle [1], [\,] \rangle) \\
= \; & EVAL(\texttt{apply}, \langle [1], [1] \rangle) \\
= \; & EVAL(\texttt{call}_{(1,1)}(\texttt{f}), \langle [1], [1] \rangle) \\
= \; & EVAL(\texttt{f}, \langle [1,1], [1] \rangle) \\
= \; & EVAL(\texttt{actuals}_2(\texttt{call}_{(1,1)}(\texttt{inc})), \langle [1,1], [1] \rangle) \\
= \; & EVAL(\texttt{call}_{(1,1)}(\texttt{inc}), \langle [1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{inc}, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{y+1}, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{y}, \langle [1,1,1], [\,] \rangle) + EVAL(1, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{actuals}_1(\texttt{z}), \langle [1,1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{z}, \langle [1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{actuals}_1(\texttt{x}), \langle [1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{x}, \langle [1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{actuals}_1(\texttt{8}), \langle [1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{8}, \langle [\,], [\,] \rangle) + 1 \\
= \; & 8 + 1 \\
= \; & 9
\end{aligned}$$

2.9 Limitations of Wadge's Algorithm

The following example illustrates that the algorithm described in the previous section does not always preserve the semantics of the source functional programs. Consider the following second-order program:

```
result      ≐ twice(inc,8)
twice(f,x)  ≐ f(f(x))
inc(y)      ≐ y+1
```

The function **twice** is second-order because of its first argument. According to the algorithm, this argument must be eliminated in the first step. Moreover, a new formal **z** is added to the newly created definition for **f**:

```
result      ≐ call⟨2,1⟩twice(8)
twice(x)    ≐ f(f(x))
inc(y)      ≐ y+1
f(z)        ≐ actuals2(inc(z))
```

Now we have a first-order program, and we can apply the second step of the transformation:

```
result  ≐ call⟨1,1⟩(call⟨2,1⟩(twice))
twice   ≐ call⟨1,1⟩(f)
inc     ≐ y+1
f       ≐ actuals2(call⟨1,1⟩(inc))
z       ≐ actuals1(call⟨1,2⟩(f), x)
y       ≐ actuals1(z)
x       ≐ actuals1(8)
```

The execution of the resulting program is illustrated in Figure 2.3. It can be easily realized that the execution fails, without producing any final result: at the last step shown in Figure 2.3, the semantic equation for the **actuals** operator can not be applied, because the second

component of the context is empty. In other words, the source functional program and the resulting intensional program, are not semantically equivalent.

figure 2.3 Execution of the intensional code that results `twice`

$$
\begin{aligned}
& EVAL(\texttt{call}_{(1,1)}(\texttt{call}_{(2,1)}(\texttt{twice})), \langle [\,], [\,] \rangle) \\
= \; & EVAL(\texttt{call}_{(2,1)}(\texttt{twice}), \langle [1], [\,] \rangle) \\
= \; & EVAL(\texttt{twice}, \langle [1], [1] \rangle) \\
= \; & EVAL(\texttt{call}_{(1,1)}(\texttt{f}), \langle [1], [1] \rangle) \\
= \; & EVAL(\texttt{f}, \langle [1,1], [1] \rangle) \\
= \; & EVAL(\texttt{actuals}_2(\texttt{call}_{(1,1)}(\texttt{inc})), \langle [1,1], [1] \rangle) \\
= \; & EVAL(\texttt{call}_{(1,1)}(\texttt{inc}), \langle [1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{inc}, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{y+1}, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{y}, \langle [1,1,1], [\,] \rangle) + EVAL(1, \langle [1,1,1], [\,] \rangle) \\
= \; & EVAL(\texttt{actuals}_1(\texttt{z}), \langle [1,1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{z}, \langle [1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{actuals}_1(\texttt{call}_{(1,2)}(\texttt{f}), \texttt{x}), \langle [1,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{call}_{(1,2)}(\texttt{f}), \langle [1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{f}, \langle [2,1], [\,] \rangle) + 1 \\
= \; & EVAL(\texttt{actuals}_2(\texttt{call}_{(1,1)}(\texttt{inc})), \langle [2,1], [\,] \rangle) + 1
\end{aligned}
$$

The following remarks are in order, concerning the transformation for higher-order programs and its application on the above example:

1. After the first step in the transformation was performed, a variable `z` was introduced and attached to the function variable `inc`. This decision ignores the effect that the **actuals** operator has on contexts, and is therefore semantically incorrect. Under this translation scheme, the evaluation of many programs is terminated abnormally.

2. After the first step in the transformation, the defining expression for the variable `result` is $\texttt{call}_{(2,1)}\texttt{twice(8)}$. Should this be treated as $\texttt{call}_{(2,1)}\texttt{(twice(8))}$ or as $(\texttt{call}_{(2,1)}\texttt{twice})\texttt{(8)}$? The parenthesization proves to be extremely important when considering the correctness proof of the transformation.

3. The **actuals** operator appears to be working correctly, but this is not the case: it

forces variables to be evaluated in different contexts than they should. A stronger operator is required in order to ensure the correctness of the transformation.

4. What are the semantics of the **call** and **actuals** operators that appear in intermediate steps of the transformation? Notice that these operators may have as arguments higher-order objects and not intensions, in which case a different approach to their semantics should be adopted.

5. How can the translation be formally defined in a functional way? This problem exists for Yaghi's algorithm as well, but becomes much more difficult for the case of higher-order programs.

6. How can the correctness of the translation be established? This is possibly the most demanding open question, since it is not apparent how one can relate the semantics of the source extensional program to the semantics of the resulting intensional one.

The main purpose of this dissertation is to settle the above issues, establishing in this way a semantics preserving transformation from higher-order extensional to intensional programs.

2.10 Discussion of Related Work

In the last sections we have outlined the work described in [Yag84] and [Wad91] on transforming extensional programs into intensional ones. These two references are the starting point of our research, and in this respect they are closely related to our work. To our knowledge, there have not been any other attempts to relate functional programming and intensional logic in the sense described in this dissertation.

Our work is connected to the recent research on *firstification* [Nel91], whose purpose is to reduce a given higher-order functional program into a first-order one. The practical outcome of firstification is that the resulting first-order programs can be executed in a more efficient way than the source higher-order ones. Our work differs from firstification in that

the result of our transformation is an intensional program of nullary variables. Moreover, our goal is to transform the source program into a form which can be educed, while the goal of firstification is to serve as a form of optimization for the source higher-order programs.

Reducing the order of the source program is also the goal of a technique originally proposed by Reynolds [Rey72]. However, in order for this to be achieved, data-structures have to be introduced in the program. Moreover, the resulting program actually simulates the runtime behavior of the source one. Therefore, although elegant, Reynolds technique does not serve the same goals as the technique we propose in this dissertation.

Our work is also connected and contributes to the field of *dataflow computation*. More specifically, the implementation of higher-order functions on *tagged-dataflow* machines, has always been a problematic issue: the solutions usually adopted are inelegant and prove computationally expensive in practice, mainly because they do not take advantage of the tagging capabilities of the underlying machine. The technique we develop offers a solution to this long-lasting problem in the implementation of dynamic dataflow. We will return to this point and discuss it in detail, in Chapter 7.

There exists a growing body of research which examines other important applications of intensional logic in computer science. Most of this research has been taking place in the area of programming languages and its main trend is to examine how existing programming language paradigms can be enriched with ideas from intensional logic.

Functional programming was possibly the first programming paradigm to receive the influence of intensional logic. The functional-intensional language Lucid [WA85, AW76, AW77], is the first example of this influence. Lucid is a language based on the concept of *streams*, which are actually intensions over a set of time-points. Moreover, the language supports a rich set of operations on streams (i.e., intensional operations). At the time that Lucid was introduced, it was widely believed that applicative languages are inherently incapable of describing dynamic activity. This assumption was challenged by Lucid, as it soon became obvious that the language could describe dynamic computations in a natural and problem-oriented way. Since its inception, Lucid has been extended in several ways. Its

variants have been used to specify 3D spreadsheets [DW90b, DW90a], parallel computation models such as systolic arrays [Du91], attribute grammars [Tao94, Tao93], real-time systems [FL89, JPL93], database systems [PP94], and so on.

Logic programming [SE86, Llo87, Apt90, vK76] is another programming paradigm which has benefited from its interaction with intensional logic. Many intensional logic programming languages have been proposed [Org94, Wad85, Wad88], and significant results have been obtained regarding their semantics [Org91, OW92]. A work in this area that is related to our research, is D. Rolston's Ph.D. dissertation [Rol92], which examines the relationship between logic programming and intensional logic programming. The philosophy of Rolston's approach was also motivated by Yaghi's work. However, the techniques adopted in [Rol92] have significant differences from the methods used in [Yag84], possibly reflecting in this way the differences between logic and functional programming.

Another area in which intensional logic seems to be playing an increasingly important role, is that of *concurrency.* In this context, intensional logic is used to specify properties which a concurrent program should satisfy, such as for example *deadlock freedom.* For an introduction to such application, the reader is referred to [vB88, Chapter 5].

Chapter 3

Intensionalizing First-Order Programs

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

3.1 A Formal Definition of Yaghi's Algorithm

In Chapter 2, we outlined the algorithm proposed by Yaghi for translating first-order extensional programs into intensional programs of nullary variables. There are two main problems with Yaghi's approach:

1. The definition of the algorithm given in [Yag84] is semi-formal.

2. A correctness proof of the algorithm is not provided.

In this section we give a solution to the first of the above problems. The second problem is settled in Section 3.3.

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

Example 3.1 Consider the following first-order extensional program:

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

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

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

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

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

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

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

Before presenting the formal definition of the algorithm, we state the following assumption, which simplifies the subsequent discussion without affecting the expressive power of the *FOFL* language:

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

We are now in a position to formally describe the revised algorithm. The formal definition consists of three components. The first one describes how to translate extensional expressions into appropriate intensional ones. The second component processes the definitions of the source program and also creates a set of new definitions, one for every formal parameter that existed in the source program. Finally, the third component combines and coordinates the actions of the first two.

The following conventions are adopted. Let **P** be a first-order extensional program and let $Sub(\mathbf{P})$ be the set of subexpressions of **P**. Let **f** be a function defined in **P**. Then:

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

$$labels(\mathbf{f}, \mathbf{P}) = \{\lceil \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n) \rceil \mid \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n) \in Sub(\mathbf{P})\}$$

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

$$\lceil \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n) \rceil \odot j = \mathbf{E}_j,\ j \in \{1, \ldots, n\}$$

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

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

$$\frac{\mathbf{E} = \mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n)}{\mathcal{E}(\mathbf{E}) = \mathbf{c}(\mathcal{E}(\mathbf{E}_1), \ldots, \mathcal{E}(\mathbf{E}_n))}$$

$$\frac{\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)}{\mathcal{E}(\mathbf{E}) = \mathbf{call}_{|\mathbf{E}|}(\mathbf{f})}$$

Given a program $\mathbf{P}$ and a function definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B_f}$ in $\mathbf{P}$, the function $\mathcal{A}_\mathbf{f}$ is used to create a set of new definitions, one for every formal parameter of $\mathbf{f}$:

$$\frac{I = \mathit{labels}(\mathbf{f}, \mathbf{P})}{\mathcal{A}_\mathbf{f}(\mathbf{P}) = \bigcup_{j=1}^{n} \{\mathbf{x}_j \doteq \mathbf{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})\}}$$

The last step of the transformation, removes the formal parameters from the function definitions and appropriately uses $\mathcal{E}$ and $\mathcal{A}$ to create the target intensional program:

$$\frac{\mathbf{F} = (\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B_f})}{\mathcal{D}(\mathbf{F}) = (\mathbf{f} \doteq \mathcal{E}(\mathbf{B_f}))}$$

$$\mathit{Trans}(\mathbf{P}) = \Big(\bigcup_{\mathbf{f} \in \mathit{func}(\mathbf{P})} \mathcal{A}_\mathbf{f}(\mathbf{P}) \Big) \cup \Big(\bigcup_{\mathbf{F} \in \mathbf{P}} \{\mathcal{D}(\mathbf{F})\} \Big)$$

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

3.2 Example Transformations

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

Example 3.3 Consider the following simple first-order extensional program **P**:

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

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

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

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

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

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

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

Let $\widehat{u}$ be the least environment that satisfies the definitions of the above *NVIL* program. Then, following the denotational semantics, we can compute the semantic value of the program, as shown in Figure 3.1.

figure 3.1 Execution of the intensional program

$$
\begin{aligned}
& [\![\texttt{result}]\!](\widehat{u})([\,]) = \\
= \; & [\![\texttt{call}_{l_1}\texttt{(f)}]\!](\widehat{u})([\,]) \\
= \; & [\![\texttt{f}]\!](\widehat{u})([l_1]) \\
= \; & [\![\texttt{x+1}]\!](\widehat{u})([l_1]) \\
= \; & [\![\texttt{x}]\!](\widehat{u})([l_1]) + 1 \\
= \; & [\![\texttt{actuals}(\{\langle l_1, \texttt{call}_{l_2}\texttt{(f)}\rangle, \langle l_2, 10\rangle\})]\!](\widehat{u})([l_1]) + 1 \\
= \; & [\![\texttt{call}_{l_2}\texttt{(f)}]\!](\widehat{u})([\,]) + 1 \\
= \; & [\![\texttt{f}]\!](\widehat{u})([l_2]) + 1 \\
= \; & [\![\texttt{x+1}]\!](\widehat{u})([l_2]) + 1 \\
= \; & [\![\texttt{x}]\!](\widehat{u})([l_2]) + 1 + 1 \\
= \; & [\![\texttt{actuals}(\{\langle l_1, \texttt{call}_{l_2}\texttt{(f)}\rangle, \langle l_2, 10\rangle\})]\!](\widehat{u})([l_2]) + 1 + 1 \\
= \; & [\![10]\!](\widehat{u})([\,]) + 1 + 1 \\
= \; & 12
\end{aligned}
$$

Example 3.4 Consider the following recursive first-order extensional program **P**:

$$
\begin{aligned}
\texttt{result} \;&\doteq\; \texttt{fact(2)} \\
\texttt{fact(n)} \;&\doteq\; \texttt{if (n<=1) then 1 else n*fact(n-1)}
\end{aligned}
$$

Assume the $\lceil\texttt{fact(2)}\rceil = l_1$ and that $\lceil\texttt{fact(n-1)}\rceil = l_2$. The two definitions of the initial first-order extensional program become after they are processed by $\mathcal{D}$:

$$
\begin{aligned}
\texttt{result} \;&\doteq\; \texttt{call}_{l_1}\texttt{(fact)} \\
\texttt{fact} \;&\doteq\; \texttt{if (n<=1) then 1 else n*call}_{l_2}\texttt{(fact)}
\end{aligned}
$$

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

$$
\mathcal{A}_{\texttt{fact}}(\mathbf{P}) \;=\; \{\texttt{n} \doteq \texttt{actuals}(\{\langle l_1, 2\rangle, \langle l_2, \texttt{n-1}\rangle\})\}
$$

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

$$\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{l_1}\texttt{(fact)}\\
\texttt{fact} &\doteq \texttt{if (n<=1) then 1 else n*call}_{l_2}\texttt{(fact)}\\
\texttt{n} &\doteq \texttt{actuals}(\{\langle l_1, 2\rangle, \langle l_2, \texttt{n-1}\rangle\})
\end{aligned}$$

The above program can be executed following the same principles as in example 3.3 (see Figure 3.2). Notice that during the calculations the value of $u(\texttt{n})$ under the context $[l_1]$ is demanded three times. This means that if the value together with the context were appropriately saved when first encountered, then they could be reused when demanded again.

figure 3.2 Execution of the intensional program that results from `fact`

$$\begin{aligned}
& [\![\texttt{result}]\!](\widehat{u})([\,]) =\\
=\ & [\![\texttt{call}_{l_1}\texttt{(fact)}]\!](\widehat{u})([\,])\\
=\ & [\![\texttt{fact}]\!](\widehat{u})([l_1])\\
=\ & [\![\texttt{if (n<=1) then 1 else n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1])\\
=\ & \mathit{if}\ ([\![\texttt{n<=1}]\!](\widehat{u})([l_1]))\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1]))\\
=\ & \mathit{if}\ (\widehat{u}(\texttt{n})([l_1]) <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1]))\\
=\ & \mathit{if}\ ([\![\texttt{actuals}(\{\langle l_1, 2\rangle, \langle l_2, \texttt{n-1}\rangle\})]\!](\widehat{u})([l_1]) <= 1)\ \mathit{then}\ 1\\
& \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1]))\\
=\ & \mathit{if}\ (2 <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1]))\\
=\ & [\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1])\\
=\ & \widehat{u}(\texttt{n})([l_1]) * [\![\texttt{call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1])\\
=\ & 2 * [\![\texttt{call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_1])\\
=\ & 2 * [\![\texttt{fact}]\!](\widehat{u})([l_2, l_1])\\
=\ & 2 * [\![\texttt{if (n<=1) then 1 else n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_2, l_1])\\
=\ & 2 * (\mathit{if}\ ([\![\texttt{n<=1}]\!](\widehat{u})([l_2, l_1]))\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_2, l_1])))\\
=\ & 2 * (\mathit{if}\ ([\![\texttt{actuals}(\{\langle l_1, 2\rangle, \langle l_2, \texttt{n-1}\rangle\})]\!](\widehat{u})([l_2, l_1]) <= 1)\ \mathit{then}\ 1\\
& \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_2, l_1]))\\
=\ & 2 * (\mathit{if}\ ([\![\texttt{n-1}]\!](\widehat{u})([l_1]) <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_2, l_1])))\\
=\ & 2 * (\mathit{if}\ (1 <= 1)\ \mathit{then}\ 1\ \mathit{else}\ ([\![\texttt{n*call}_{l_2}\texttt{(fact)}]\!](\widehat{u})([l_2, l_1])))\\
=\ & 2 * 1\\
=\ & 2
\end{aligned}$$

3.3 Correctness Proof

The correctness proof of the transformation algorithm is established by Theorems 3.2, 3.3 and 3.4 to follow. The main idea of the proof is to relate semantically a function call in the source first-order extensional program with the corresponding intensional expression that results from its translation. For example, given a first-order extensional program $\mathbf{P}$, we would like to give a semantic statement concerning a call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ in $\mathbf{P}$, and its translation $\mathbf{call}_{\lceil \mathbf{E} \rceil}(\mathbf{f})$ in $Trans(\mathbf{P})$. Let u and $\widehat{u}$ be the least environments satisfying the definitions in $\mathbf{P}$ and $Trans(\mathbf{P})$ respectively, and let $w \in W$. The idea is to first prove the following statement:

$$(call_{\lceil \mathbf{E} \rceil}(\widehat{u}(\mathbf{f})))(w) = u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w), \ldots, [\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w))$$

This looks like a weaker result than what we are actually looking for, because the right hand side does not correspond exactly to the expression $\mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ of the extensional program. However, a stronger result can be shown afterwards using an inductive argument, as we are going to see. It turns out that the above statement can not itself be shown in one step. Instead, we need to show that the right hand side approximates the left, and vice-versa. The details of the proof are given below:

Theorem 3.2 Let $\mathbf{P}$ be a first-order extensional program and let u be the least environment satisfying the definitions in $\mathbf{P}$. Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every function definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}$, for every function call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ of $\mathbf{f}$ in $\mathbf{P}$ and for every $w \in W$

$$(call_{\lceil \mathbf{E} \rceil}(\widehat{u}(\mathbf{f})))(w) \sqsubseteq u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w), \ldots, [\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w))$$

Proof: The theorem is established by induction on the approximations $\widehat{u}_k$, $k \in N$, of $\widehat{u}$. In other words, we show that for every $k \geq 0$, for every function $\mathbf{f}$ defined in $\mathbf{P}$, for every

function call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$(call_{\lceil \mathbf{E} \rceil}(\widehat{u}_k(\mathbf{f})))(w) \sqsubseteq u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u}_k)(w), \ldots, [\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u}_k)(w))$$

Notice that we only use the approximations of $\widehat{u}$ but not the approximations of u. Intuitively, this gives to the right hand side of the above statement an "advantage", which allows the $\sqsubseteq$ relation to be established. The basis case is for $k = 0$ and it holds trivially because the left hand side of the above statement is equal to the bottom value. We assume that the above statement holds for k and we show that it holds for $k + 1$, i.e.,

$$(call_{\lceil \mathbf{E} \rceil}(\widehat{u}_{k+1}(\mathbf{f})))(w) \sqsubseteq u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u}_{k+1})(w), \ldots, [\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u}_{k+1})(w))$$

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

$$\widehat{u}_{k+1}(\mathbf{f})(\lceil \mathbf{E} \rceil : w) \sqsubseteq u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u}_{k+1})(w), \ldots, [\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u}_{k+1})(w))$$

Recalling that $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B_f}$ in $\mathbf{P}$ and $\mathbf{f} \doteq \mathcal{E}(\mathbf{B_f})$ in $Trans(\mathbf{P})$, and using Definition 2.13 and Theorem 2.1, the above is equivalent to the following:

$$[\![\mathcal{E}(\mathbf{B_f})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \sqsubseteq [\![\mathbf{B_f}]\!](u \oplus \rho_{k+1})$$

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

$$[\![\mathcal{E}(\mathbf{S})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \sqsubseteq [\![\mathbf{S}]\!](u \oplus \rho_{k+1})$$

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

Structural Induction Basis.

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

$$
\begin{aligned}
& [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) = \\
= \; & [\![\mathcal{E}(\mathbf{x}_j)]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \; & [\![\mathbf{x}_j]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & \widehat{u}_k(\mathbf{x}_j)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics)} \\
\sqsubseteq \; & [\![\mathbf{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathbf{x}_j \text{ and Lemma 2.1)} \\
= \; & [\![\mathcal{E}(\lceil \mathbf{E} \rceil \odot j)]\!](\widehat{u}_k)(w) \\
& \text{(Semantics of } \mathbf{actuals}\text{)} \\
= \; & [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u}_k)(w) \\
& \text{(Definition of } \odot\text{)} \\
\sqsubseteq \; & [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u}_{k+1})(w) \\
& \text{(Theorem 2.1 and monotonicity of } [\![\mathcal{E}(\mathbf{E}_j)]\!]\text{)} \\
= \; & [\![\mathbf{x}_j]\!](u \oplus \rho_{k+1}) \\
& \text{(Because } \rho_{k+1}(\mathbf{x}_j) = [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u}_{k+1})(w)\text{)} \\
= \; & [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)}
\end{aligned}
$$

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

Structural Induction Step.

Case $\mathbf{S} = \mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)$. Recall that the semantics of constants in the intensional program are defined in a pointwise way in terms of the semantics of the constants in the extensional program. We have:

$$
\begin{aligned}
& [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) = \\
= \; & [\![\mathcal{E}(\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Assumption for } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{c}(\mathcal{E}(\mathbf{S}_1), \ldots, \mathcal{E}(\mathbf{S}_r))]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & \mathcal{C}'(\mathbf{c})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u}_k), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u}_k))(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics of constant symbols)} \\
= \; & \mathcal{C}(\mathbf{c})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w)) \\
& \text{(Definition of } \mathcal{C}' \text{ in terms of } \mathcal{C}\text{)} \\
\sqsubseteq \; & \mathcal{C}(\mathbf{c})([\![\mathbf{S}_1]\!](u \oplus \rho_{k+1}), \ldots, [\![\mathbf{S}_r]\!](u \oplus \rho_{k+1})) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } \mathcal{C}(c)\text{)} \\
= \; & [\![\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!](u \oplus \rho_{k+1}) \\
& \text{(Semantics of constant symbols)} \\
= \; & [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}
$$

Case $\mathbf{S} = \mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)$, where $\mathbf{g} \in func(\mathbf{P})$. Then, the left hand side of the statement we want to establish, can be written as follows:

$$
\begin{aligned}
& [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) = \\
= \; & [\![\mathcal{E}(\mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Assumption about } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{call}_{\lceil \mathbf{S} \rceil}(\mathbf{g})]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & (call_{\lceil \mathbf{S} \rceil}(\widehat{u}_k(\mathbf{g})))(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics)} \\
\sqsubseteq \; & u(\mathbf{g})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u}_k)(\lceil \mathbf{E} \rceil : w)) \\
& \text{(Outer induction hypothesis on } k\text{)} \\
\sqsubseteq \; & u(\mathbf{g})([\![\mathbf{S}_1]\!](u \oplus \rho_{k+1}), \ldots, [\![\mathbf{S}_r]\!](u \oplus \rho_{k+1})) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } u(\mathbf{g}) \text{ from Theorem 2.2)} \\
= \; & [\![\mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!](u \oplus \rho_{k+1}) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}]\!](u \oplus \rho_{k+1}) \\
& \text{(Because } \mathbf{S} = \mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)\text{)}
\end{aligned}
$$

This completes the proof of the theorem. ■

Theorem 3.3 Let $\mathbf{P}$ be a first-order extensional program and let u be the least environment satisfying the definitions in $\mathbf{P}$. Let $\widehat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every function definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f})$ in $\mathbf{P}$, for every function call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$

$$u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w),\ldots,[\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w)) \sqsubseteq (call_{\lceil \mathbf{E} \rceil}(\widehat{u}(\mathbf{f})))(w)$$

Proof: The theorem is established by induction on the approximations u_k, $k \in N$, of u. In other words, we show that for every $k \geq 0$, for every function $\mathbf{f}$ defined in $\mathbf{P}$, for every function call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ of $\mathbf{f}$ in $\mathbf{P}$, and for every $w \in W$:

$$u_k(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w),\ldots,[\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w)) \sqsubseteq (call_{\lceil \mathbf{E} \rceil}(\widehat{u}(\mathbf{f})))(w)$$

Notice that now we only use the approximations of u but not the approximations of $\widehat{u}$. Again, this gives to the right hand side of the above statement an "advantage", which allows the $\sqsubseteq$ relation to be established. The basis case is for $k = 0$ and it holds trivially because the left hand side of the above statement is equal to the bottom value. We assume that the above statement holds for k and we show that it holds for $k + 1$, i.e.,

$$u_{k+1}(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w),\ldots,[\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w)) \sqsubseteq (call_{\lceil \mathbf{E} \rceil}(\widehat{u}(\mathbf{f})))(w)$$

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

$$u_{k+1}(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\widehat{u})(w),\ldots,[\![\mathcal{E}(\mathbf{E}_n)]\!](\widehat{u})(w)) \sqsubseteq \widehat{u}(\mathbf{f})(\lceil \mathbf{E} \rceil : w)$$

Recalling that $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$ in $\mathbf{P}$ and that $\mathbf{f} \doteq \mathcal{E}(\mathbf{B_f})$ in $Trans(\mathbf{P})$, and using Definition 2.13 and Theorem 2.1, the above is equivalent to the following:

$$[\![\mathbf{B_f}]\!](u_k \oplus \rho) \sqsubseteq [\![\mathcal{E}(\mathbf{B_f})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w)$$

where $\rho(\mathbf{x}_j) = [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u})(w)$, $j = 1, \ldots, n$. The above can be established by showing that for every subexpression $\mathbf{S}$ of $\mathbf{B_f}$, it is:

$$[\![\mathbf{S}]\!](u_k \oplus \rho) \sqsubseteq [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w)$$

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

Structural Induction Basis.

Case $\mathbf{S} = \mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. In the transformed program $Trans(\mathbf{P})$, a definition of the form $\mathbf{x}_j \doteq \mathbf{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})$ has been created, where $I = labels(\mathbf{f}, \mathbf{P})$. Consider the right hand side of the statement we would like to establish:

$$\begin{aligned}
& [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) = \\
= \; & [\![\mathcal{E}(\mathbf{x}_j)]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \; & [\![\mathbf{x}_j]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & \widehat{u}(\mathbf{x}_j)(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics)} \\
= \; & [\![\mathbf{actuals}(\langle \mathcal{E}(i \odot j) \rangle_{i \in I})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathbf{x}_j \text{ and Definition 2.18)} \\
= \; & [\![\mathcal{E}(\lceil \mathbf{E} \rceil \odot j)]\!](\widehat{u})(w) \\
& \text{(Semantics of } \mathbf{actuals}\text{)} \\
= \; & [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u})(w) \\
& \text{(Definition of } \odot\text{)} \\
= \; & [\![\mathbf{x}_j]\!](u_k \oplus \rho) \\
& \text{(Because } \rho(\mathbf{x}_j) = [\![\mathcal{E}(\mathbf{E}_j)]\!](\widehat{u})(w)\text{)} \\
= \; & [\![\mathbf{S}]\!](u_k \oplus \rho) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)}
\end{aligned}$$

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

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

Structural Induction Step

Case $\mathbf{S} = \mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)$. Recall that the semantics of constants in the intensional program are defined in a pointwise way in terms of the semantics of the constants in the extensional program. We have:

$$
\begin{aligned}
& [\![\mathbf{S}]\!](u_k \oplus \rho) = \\
= \; & [\![\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!](u_k \oplus \rho) \\
& \text{(Assumption for } \mathbf{S}\text{)} \\
= \; & \mathcal{C}(\mathbf{c})([\![\mathbf{S}_1]\!](u_k \oplus \rho), \ldots, [\![\mathbf{S}_r]\!](u_k \oplus \rho)) \\
& \text{(Semantics of constant symbols)} \\
\sqsubseteq \; & \mathcal{C}(\mathbf{c})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w)) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } \mathcal{C}(c)\text{)} \\
= \; & \mathcal{C}'(\mathbf{c})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u}), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u}))(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{C}' \text{ in terms of } \mathcal{C}\text{)} \\
= \; & [\![\mathbf{c}(\mathcal{E}(\mathbf{S}_1), \ldots, \mathcal{E}(\mathbf{S}_r))]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics of constants)} \\
= \; & [\![\mathcal{E}(\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}
$$

Case $\mathbf{S} = \mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)$, where $\mathbf{g} \in func(\mathbf{P})$. Then, the left hand side of the statement we want to establish can be written as follows:

$$
\begin{aligned}
& [\![\mathbf{S}]\!](u_k \oplus \rho) = \\
= \; & [\![\mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!](u_k \oplus \rho) \\
& \text{(Because } \mathbf{S} = \mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r)\text{)} \\
= \; & u_k(\mathbf{g})([\![\mathbf{S}_1]\!](u_k \oplus \rho), \ldots, [\![\mathbf{S}_r]\!](u_k \oplus \rho)) \\
& \text{(Semantics of application)} \\
\sqsubseteq \; & u_k(\mathbf{g})([\![\mathcal{E}(\mathbf{S}_1)]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w), \ldots, [\![\mathcal{E}(\mathbf{S}_r)]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w)) \\
& \text{(Structural induction hypothesis and} \\
& \text{monotonicity of } u_k(\mathbf{g}) \text{ from Theorem 2.2)} \\
\sqsubseteq \; & (call_{\lceil \mathbf{S} \rceil}(\widehat{u}(\mathbf{g})))(\lceil \mathbf{E} \rceil : w) \\
& \text{(Induction hypothesis on } k\text{)} \\
= \; & [\![\mathbf{call}_{\lceil \mathbf{S} \rceil}(\mathbf{g})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Semantics of } \mathbf{call}\text{)} \\
= \; & [\![\mathcal{E}(\mathbf{g}(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathcal{E}\text{)} \\
= \; & [\![\mathcal{E}(\mathbf{S})]\!](\widehat{u})(\lceil \mathbf{E} \rceil : w) \\
& \text{(Definition of } \mathbf{S}\text{)}
\end{aligned}
$$

This completes the proof of the theorem. ■

Lemma 3.1 Let **P** be a first-order extensional program and let u be the least environment satisfying the definitions in **P**. Let $\hat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every function definition $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$ in **P**, for every function call $\mathbf{E} = \mathbf{f}(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ of **f** in **P**, and for every $w \in W$

$$(call_{\lceil \mathbf{E} \rceil}(\hat{u}(\mathbf{f})))(w) = u(\mathbf{f})([\![\mathcal{E}(\mathbf{E}_1)]\!](\hat{u})(w),\ldots,[\![\mathcal{E}(\mathbf{E}_n)]\!](\hat{u})(w))$$

Theorem 3.4 Let **P** be a first-order extensional program and let u be the least environment satisfying the definitions in **P**. Let $\hat{u}$ be the least environment satisfying the definitions in the translated program $Trans(\mathbf{P})$. Then, for every $w \in W$

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

Proof: By a straightforward structural induction on the defining expression of the variable **result** in **P** and using Lemma 3.1. ∎

3.4 An Illustration of the Proof

To illustrate the technique used for the proof, consider the program that was given in Example 3.3. We show how the theorem can be applied to the outer call to function `f` in that example. Similar arguments apply for the inner call to `f`. It suffices to show that for every $k \geq 0$ and $w \in W$, we have:

$$\begin{aligned} call_{l_1}(\hat{u}_k(\mathtt{f}))(w) &\sqsubseteq u(\mathtt{f})([\![\mathtt{call}_{l_2}(\mathtt{f})]\!](\hat{u}_k)(w)) \\ u_k(\mathtt{f})([\![\mathtt{call}_{l_2}(\mathtt{f})]\!](\hat{u})(w)) &\sqsubseteq call_{l_1}(\hat{u}(\mathtt{f}))(w) \end{aligned}$$

Using the technique for computing the least environment, that was presented in Definition 2.13, one can compute the values of $\hat{u}_k$ and u_k for various values of $k \in N$. For example, the validity of the first of the above statements for $k = 0$, can be shown by first evaluating

k	$call_{l_1}(\widehat{u}_k(\mathtt{f}))(w)$	$u(\mathtt{f})([\![\mathtt{call}_{l_2}\,\mathtt{f}]\!](\widehat{u}_k)(w))$
0	$\bot$	$\bot$
1	$\bot$	$\bot$
2	$\bot$	$\bot$
3	$\bot$	12
4	12	12

Table 3.1: An illustration of the first part of the proof

k	$u_k(\mathtt{f})([\![\mathtt{call}_{l_2}\,\mathtt{f}]\!](\widehat{u})(w))$	$call_{l_1}(\widehat{u}(\mathtt{f}))(w)$
0	$\bot$	12
1	12	12

Table 3.2: An illustration of the second part of the proof

the left hand side of the statement:

$$\begin{aligned} & call_{l_1}(\widehat{u}_0(\mathtt{f}))(w) = \\ = \;& (\widehat{u}_0(\mathtt{f}))(l_1 : w) \\ = \;& \bot \end{aligned}$$

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

$$\begin{aligned} & u(\mathtt{f})([\![\mathtt{call}_{l_2}\,\mathtt{f}]\!](\widehat{u}_0)(w)) = \\ = \;& u(\mathtt{f})(\widehat{u}_0(\mathtt{f})(l_2 : w)) \\ = \;& u(\mathtt{f})(\bot) \\ = \;& \bot \end{aligned}$$

Tables 3.1 and 3.2 have been constructed in this way, and they illustrate Theorems 3.2 and 3.3 respectively. Notice that every entry in the second column of the two tables, approximates the corresponding entry in the third column.

3.5 Discussion

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

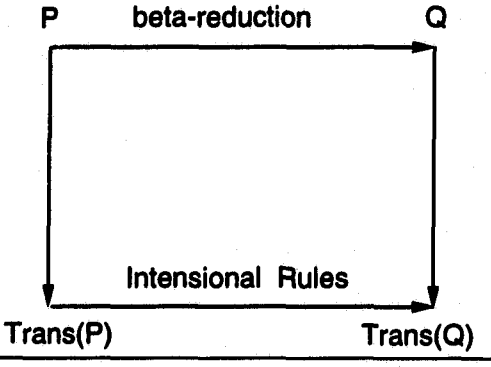
1. The intensional program that results from the transformation has significant syntactic differences from the source extensional one. Note in particular that formal parameters in the latter have become individual variables in the former. Therefore, a syntax-based correctness proof may face considerable difficulties. The author has undertaken one such approach (see Figure 3.3), attempting to identify a sequence of intensional transformations that correspond to the notion of beta-reduction. Although some interesting results were obtained, this approach proved to be much harder than the one presented in this chapter.

2. The precise formal definition of Yaghi's transformation algorithm given in Section 3.1 helped us formulate the exact result that we had to demonstrate. It should be noted here that the author tried at first to formalize Yaghi's non-referentially transparent scheme, using the notion of an *occurrence* of a function call in the program. However, such an approach proved to be quite inflexible and did not easily lead to the right intuitions.

3. The proof requires a double induction: an outer computational one and an inner structural one. Moreover, notice that the statement in Lemma 3.1, is not symmetric: the intensional environment appears in both sides of the statement, while the extensional one appears in only one of them.

4. It might be expected that Lemma 3.1 can be demonstrated directly, i.e., without first showing that the right hand side of the statement approximates the left, and vice-versa. However, such a proof does not seem to be possible.

figure 3.3 An alternative proof technique



Finally, we should mention that the transformation algorithm and the proof can readily be extended for a language that allows "outside" variables as well as nullary variable definitions. However, as these extensions would not add to the expressive power of the language under consideration, we refrain from such an attempt. Instead, in the following chapters we illustrate how the above ideas can be extended and enhanced in order to apply to the higher-order language FL.

Chapter 4

A Higher-Order Intensional Language

This chapter initiates the study of the relationship between higher-order functional languages and intensional logic. To motivate the subsequent discussion, we first present an example of the algorithm that we propose for transforming higher-order extensional programs into intensional programs of nullary variables (the precise definition of the algorithm is deferred until Chapter 5). We then give a formal definition of the syntax of the target intensional language that is used in the transformation, and present its *synchronic* denotational semantics. The chapter concludes with an investigation of the properties of the synchronic interpretation with respect to the standard denotational ones.

4.1 An Example Transformation

In this section, we describe at an intuitive level the algorithm we propose for transforming higher-order functional programs into intensional programs of nullary variables. Consider

the following second-order functional program:

$$\begin{aligned} \texttt{result} &\doteq \texttt{apply(inc,8)} \\ \texttt{apply(f,x)} &\doteq \texttt{f(x)} \\ \texttt{inc(y)} &\doteq \texttt{y+1} \end{aligned}$$

As in [Wad91], we start by eliminating the highest-order formal parameters first. In this case, the parameter **f** of **apply** is eliminated, and a new definition is introduced for it. Notice that the intensional operators that are used below, are different from the ones adopted in [Wad91]: we are now using operators of the form $\mathbf{call}_L$ where $L \subseteq N \times N$, and $\mathbf{actuals}_{m,R}$ where $m \in N$ and $R \subseteq N \times (N \times N)$. Intuitively, we now have a single **call** which has a set as a subscript. This notation will prove more convenient in the following, as it helps us avoid the long chains of different **call** operators. The change in the **actuals** operator is of a more fundamental nature, and will be further explained later on in the dissertation. We should mention at this point that the **actuals** operator used in [Wad91] is insufficient for the purposes of the transformation, and therefore may lead to programs that are not semantically equivalent to the source functional ones.

$$\begin{aligned} \texttt{result} &\doteq (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)} \\ \texttt{apply(x)} &\doteq \texttt{f(x)} \\ \texttt{f} &\doteq \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1, \texttt{inc}\rangle\} \\ \texttt{inc(y)} &\doteq \texttt{y+1} \end{aligned}$$

In the above program, the number 2 appearing in the subscripts of the intensional operators indicates the fact that these operators are used in order to transform a *second*-order program into a first-order one. Moreover, l_1 is a natural number that uniquely characterizes the function call **apply(inc,8)** (i.e., the Gödel number of this call). We can now add a new variable **z** in both sides of the definition of **f**, getting:

$$
\begin{array}{lcl}
\texttt{result} & \doteq & (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)} \\
\texttt{apply(x)} & \doteq & \texttt{f(x)} \\
\texttt{f(z)} & \doteq & (\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{inc}\rangle\})\texttt{(z)} \\
\texttt{inc(y)} & \doteq & \texttt{y+1}
\end{array}
$$

The program that results is a first-order intensional one. Notice the parenthesization of the expression $(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}$, which is different than the one given in [Wad91], and which ensures that the resulting program is semantically equivalent to the source one.

We must now enter **z** inside the scope of the intensional operator **actuals**. This is done by prefixing it with $\texttt{call}_{\{\langle 2,l_1\rangle\}}$; that is, with the same intensional operator that appears in the corresponding call to **apply**. In this way, we (intuitively speaking) cancel the effect that the intensional operator **actuals** has on **z**. This step is not performed and is the main limitation of the algorithm given in [Wad91].

$$
\begin{array}{lcl}
\texttt{result} & \doteq & (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)} \\
\texttt{apply(x)} & \doteq & \texttt{f(x)} \\
\texttt{f(z)} & \doteq & \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{inc}(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)})\rangle\} \\
\texttt{inc(y)} & \doteq & \texttt{y+1}
\end{array}
$$

This completes the first step of the transformation. We can now perform the second (and last for this case) step of the algorithm, which will result in an intensional program of nullary variables. The algorithm eliminates the (zero-order) variables **x**, **z**, **y**, adding at the same time a new definition for each one of them in the program. The final program

that results from the transformation is the following:

$$
\begin{array}{lcl}
\texttt{result} & \doteq & \texttt{call}_{\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}}\texttt{(apply)} \\
\texttt{apply} & \doteq & \texttt{call}_{\{\langle 1,l_3\rangle\}}\texttt{(f)} \\
\texttt{f} & \doteq & \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1, \texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}\rangle\} \\
\texttt{inc} & \doteq & \texttt{y+1} \\
\texttt{x} & \doteq & \texttt{actuals}_{1,\{\langle l_2,\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}\rangle\}}(\{\langle l_2, 8\rangle\}) \\
\texttt{z} & \doteq & \texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle\}}(\{\langle l_3, \texttt{x}\rangle\}) \\
\texttt{y} & \doteq & \texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4, \texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}\rangle\})
\end{array}
$$

Now the number 1 that appears in the subscripts of certain intensional operators, signifies the fact that a translation of a *first*-order program into a program of nullary variable definitions has taken place. Notice also that above we have used the natural numbers l_2, l_3 and l_4 to uniquely represent the function calls $\texttt{(call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}$, $\texttt{f(x)}$, and $\texttt{inc(call}_{\{\langle 2,l_1\rangle\}}\texttt{(z))}$ respectively.

The execution of the resulting zero-order intensional program is performed in an analogous way as in the first-order case. The difference is that now the contexts and the operators are richer. Given an m-order program, contexts are m-tuples of lists of natural numbers. The operators **c_ll** and **actuals** perform context changes in the following way: $\textbf{call}_L$ performs a multiple consing operation on the current context, as dictated by the list L. On the other hand, the operator $\textbf{actuals}_{k,R}$ performs a multiple tail operation on contexts. The precise definition of the semantics of the two operators will be given later in this chapter.

The main objective of the rest of this chapter is to precisely define the semantics of the programs that result at the various stages of the transformation algorithm.

4.2 The Intensional Language IL: Syntax

In this section we define the syntax of the intensional language IL which serves as the target language for transforming higher-order extensional programs.

Definition 4.1 The syntax of the intensional language IL 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*:

$$
\begin{array}{rcl}
\mathbf{E} & ::= & \mathbf{f} \in Var \\
& | & \mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ \mathbf{c} \in \Sigma_1 \\
& | & \mathbf{call}_L(\mathbf{E}_0),\ L \subseteq N \times N \\
& | & \mathbf{actuals}_{m,R}(\langle \mathbf{E}_i \rangle_{i \in I}),\ m \in N,\ I \subseteq N,\ R \subseteq N \times (N \times N) \\
& | & \mathbf{E}_0(\mathbf{E}_1, \ldots, \mathbf{E}_n) \\
\mathbf{F} & ::= & (\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E}),\ \mathbf{f}, \mathbf{x}_1, \ldots, \mathbf{x}_n \in Var \\
\mathbf{P} & ::= & \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}
\end{array}
$$

Definition 4.2 The set of well-typed expressions of the language IL is recursively defined as follows:

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

$$\frac{(\theta(\mathbf{c}) = (\tau_1, \ldots, \tau_n) \rightarrow \iota) \wedge (\mathbf{E}_1 : \tau_1, \ldots, \mathbf{E}_n : \tau_n)}{\mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n) : \iota}$$

$$\frac{\mathbf{E}_0 : \tau}{\mathbf{call}_L(\mathbf{E}_0) : \tau}$$

$$\frac{\forall i \in I (\mathbf{E}_i : \tau)}{\mathbf{actuals}_{m,R}(\langle \mathbf{E}_i \rangle_{i \in I}) : \tau}$$

$$\frac{(\mathbf{E}_0 : (\tau_1, \ldots, \tau_n) \rightarrow \iota) \wedge (\forall i \in \{1, \ldots, n\}, \mathbf{E}_i : \tau_i)}{\mathbf{E}_0(\mathbf{E}_1, \ldots, \mathbf{E}_n) : \iota}$$

The notions of well-typed definitions and well-typed programs are identical to the ones introduced in Definitions 2.8 and 2.9. Moreover, the same assumptions as in Definition 2.6 are adopted for IL programs.

4.3 The Intensional Language IL: Synchronic Semantics

In this section we define the denotational semantics of the intensional language IL. The set of possible worlds of IL is the set of infinite sequences of lists of natural numbers, that is $N \rightarrow List(N)$. Notice that as we discussed in Section 4.1, for the transformation of an m-order extensional program contexts need only be m-tuples 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, because in any particular translation, only a finite number of them will be used. Therefore:

Definition 4.3 The set W of possible worlds of IL is the set $N \rightarrow List(N)$.

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

Definition 4.4 Let D be a domain. The set of possible denotations of $\tau \in Typ$ with respect to W and D is defined as

$$[\![\tau]\!]^*_D = W \rightarrow [\![\tau]\!]_D$$

In defining the semantics of IL, we follow the approach that has been used by Montague for giving semantics to higher-order intensional logic [DWP81, Gal75]. 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 that will soon become obvious.

We first define the following two context manipulation operations: a multiple consing operation on infinite sequences of lists of natural numbers, and a multiple tail operation on such sequences. These two operations will be used in order to define the semantics of the **call** and **actuals** operators.

Definition 4.5 Let $L = \{\langle i_1, j_1\rangle, \ldots, \langle i_k, j_k\rangle\}$ where $i_1, \ldots, i_k, j_1, \ldots, j_k \in N$ and $i_1 \neq i_2 \neq \cdots \neq i_k$. Let $w \in (N \rightarrow \mathit{List}(N))$. The *multiple consing* on w with respect to L is the function $\Downarrow$ defined as follows:

$$w \Downarrow L \;=\; w[i_1/(j_1 : w_{i_1}), \ldots, i_k/(j_k : w_{i_k})]$$

Definition 4.6 Let $L = \{\langle i_1, j_1\rangle, \ldots, \langle i_k, j_k\rangle\}$ where $i_1, \ldots, i_k, j_1, \ldots, j_k \in N$ and $i_1 \neq i_2 \neq \cdots \neq i_k$. Let $w \in (N \rightarrow \mathit{List}(N))$. The *multiple tail* operation on w with respect to L is the function $\Uparrow$ defined as follows:

$$w \Uparrow L = \begin{cases} w[i_1/\mathit{tail}(w_{i_1}), \ldots, i_k/\mathit{tail}(w_{i_k})] & \text{if } (\mathit{head}(w_{i_1}) = j_1) \wedge \cdots \wedge (\mathit{head}(w_{i_k}) = j_k) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Let D be a given domain. Then, the semantics of constant symbols of IL with respect to D, are given by an interpretation function $\mathcal{C}^*$, which assigns to every constant of type τ, a function in $[\![\tau]\!]^*_D$. As the language IL will be used as the target for transforming programs of FL, the function $\mathcal{C}^*$ is defined in terms of the interpretation function $\mathcal{C}$ for FL. More specifically:

Definition 4.7 For every $\mathbf{c} \in \Sigma$ and for every $w \in W$, $\mathcal{C}^*(\mathbf{c})(w) = \mathcal{C}(\mathbf{c})$.

Let Exp_τ be the set of all expressions $\mathbf{E}$ of IL such that $\mathbf{E} : \tau$. Let Env^*_π be the set of π-compatible synchronic environments defined by $\mathit{Env}^*_\pi = \Pi_{\mathbf{f} \in Var}[\![\pi(\mathbf{f})]\!]^*_D$. Then, the

synchronic semantics of the language IL is defined using valuation functions $[\![\cdot]\!]^* : Exp_\tau \rightarrow [Env^*_\pi \rightarrow [\![\tau]\!]^*_D]$, as follows:

Definition 4.8 The *synchronic* interpretation of expressions of IL with respect to $u \in Env^*_\pi$, is recursively defined for every $w \in W$, as follows:

$$[\![\mathbf{f}]\!]^*(u)(w) = u(\mathbf{f})(w)$$
$$[\![\mathbf{c}(\mathbf{E}_1,\ldots,\mathbf{E}_n)]\!]^*(u)(w) = \mathcal{C}^*(\mathbf{c})(w)([\![\mathbf{E}_1]\!]^*(u)(w),\ldots,[\![\mathbf{E}_n]\!]^*(u)(w))$$
$$[\![\mathbf{call}_L(\mathbf{E})]\!]^*(u)(w) = [\![\mathbf{E}]\!]^*(u)(w \Downarrow L)$$
$$[\![\mathbf{actuals}_{m,R}(\langle \mathbf{E}_i\rangle_{i\in I})]\!]^*(u)(w) = [\![\mathbf{E}_{head(w_m)}]\!]^*(u)(w \Uparrow R_{head(w_m)})$$
$$[\![\mathbf{E}_0(\mathbf{E}_1,\ldots,\mathbf{E}_n)]\!]^*(u)(w) = [\![\mathbf{E}_0]\!]^*(u)(w)([\![\mathbf{E}_1]\!]^*(u)(w),\ldots,[\![\mathbf{E}_n]\!]^*(u)(w))$$

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. This justifies the name "synchronic" adopted for this interpretation.

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

Definition 4.9 Let $d \in [\![\tau]\!]_D$. Then, d^∞ is that function on W whose value at every $w \in W$ is equal to d.

Definition 4.10 The synchronic semantics of a program $\mathbf{P} = \{\mathbf{F}_1,\ldots,\mathbf{F}_n\}$ of IL with respect to $u \in Env^*$, is defined as $\tilde{u}(\mathbf{result})$, where $\tilde{u}$ is the *least* environment such that:

1. For every $\mathbf{f} \in Var$ with $\mathbf{f} \notin func(\mathbf{P})$, $\tilde{u}(\mathbf{f}) = u(\mathbf{f})$.
2. For every definition $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$ with $\mathbf{f} : (\tau_1,\ldots,\tau_n) \rightarrow \iota$, for all $d_1 \in [\![\tau_1]\!]_D,\ldots,d_n \in [\![\tau_n]\!]_D$, and all $w \in W$,
   $\tilde{u}(\mathbf{f})(w)(d_1,\ldots,d_n) = [\![\mathbf{E}]\!]^*(\tilde{u}[\mathbf{x}_1/d_1^\infty,\ldots,\mathbf{x}_n/d_n^\infty])(w)$.

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

Theorem 4.1 Let $\mathbf{P}$ and $\tilde{u}$ be as in Definition 4.10. Then, $\tilde{u}$ is the least upper bound of the environments $\tilde{u}_k$, $k \in N$, which for every definition $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$, with $\mathbf{f} : (\tau_1,\ldots,\tau_n) \rightarrow \iota$, for all $d_1 \in [\![\tau_1]\!]_D,\ldots,d_n \in [\![\tau_n]\!]_D$, and all $w \in W$, are defined as follows:

$$\begin{aligned} \tilde{u}_0(\mathbf{f})(w)(d_1,\ldots,d_n) &= \perp_D \\ \tilde{u}_{k+1}(\mathbf{f})(w)(d_1,\ldots,d_n) &= [\![\mathbf{E}]\!]^*(\tilde{u}_k[\mathbf{x}_1/d_1^\infty,\ldots,\mathbf{x}_n/d_n^\infty])(w) \end{aligned}$$

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

Proof: Analogous to the proof of Theorem 2.1. ∎

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

Lemma 4.1 Let $\mathbf{P}$ and $\tilde{u}$ be as in Definition 4.10. Then, for every definition $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{E}$ in $\mathbf{P}$ with $\mathbf{f} : (\tau_1,\ldots,\tau_n) \rightarrow \iota$, for all $d_1 \in [\![\tau_1]\!]_D,\ldots,d_n \in [\![\tau_n]\!]_D$ and for all $w \in W$,

$$\tilde{u}_k(\mathbf{f})(w)(d_1,\ldots,d_n) \sqsubseteq [\![\mathbf{E}]\!]^*(\tilde{u}_k[\mathbf{x}_1/d_1^\infty,\ldots,\mathbf{x}_n/d_n^\infty])(w)$$

The following theorem will also be used in subsequent chapters:

Theorem 4.2 For all expressions $\mathbf{E} \in Exp_\tau$, $[\![\mathbf{E}]\!]^*$ is monotonic and continuous. Moreover, when $\tau \neq \iota$, $[\![\mathbf{E}]\!]^*(u)(w)$ is monotonic and continuous, for all $u \in Env_\pi^*$ and $w \in W$.

4.4 Properties of the Synchronic Interpretation

In this section we investigate certain of the properties of the synchronic interpretation that we have defined in this chapter. More specifically, we consider two subsets of the language IL, and we examine the connections between the synchronic semantics and the standard denotational semantics that can be defined for these subsets. The two subsets are:

- Those programs of IL that do not contain any intensional operators **call** and **actuals**. Notice that programs of this subset are actually FL programs, for which we have already defined a standard denotational interpretation (see Definition 2.13).
- Those programs of IL that only contain nullary variable definitions. In other words, this is the restriction of IL on Var_0, and we denote it by IL_0. For these programs, a standard denotational interpretation $[\![\cdot]\!]_{(W \to D)}$ can be defined, as this was done for the language $NVIL$ (see Definition 2.18).

The following two theorems establish the relationship between the standard and the synchronic semantics for programs of the two subsets of IL we have just defined:

Theorem 4.3 Let **P** be an IL program that does not contain any intensional operators, and let u and u^* be the least environments that satisfy the definitions in **P** under the standard and the synchronic interpretations respectively. Then, for every $w \in W$, $[\![\mathbf{P}]\!]^*(u^*)(w) = [\![\mathbf{P}]\!](u)$.

Proof: It suffices to show that for every definition $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathrm{E}_\mathbf{f}$ in **P**, with $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$, it is:

$$u^*(\mathbf{f})(w)(d_1, \ldots, d_n) = u(\mathbf{f})(d_1, \ldots, d_n)$$

for all $d_1 \in [\tau_1]_D, \ldots, d_n \in [\tau_n]_D$, and all $w \in W$. This can be shown by a double induction: an outer computational induction on the approximations of u^* and u, and an inner structural one on the body of **f**. ∎

Theorem 4.4 Let **P** be an IL_0 program and let u and u^* be the least environments that satisfy the definitions in **P** under the standard and the synchronic interpretations respectively. Then, $[\![\mathbf{P}]\!]^*(u^*) = [\![\mathbf{P}]\!]_{(W \to D)}(u)$.

Proof: It suffices to show that for every function **f** that has a definition in **P**, $u^*(\mathbf{f}) = u(\mathbf{f})$. This follows directly with a proof similar to the one given for Theorem 4.3. ■

Chapter 5

Intensionalizing Higher-Order Programs

In this chapter, we describe and formally define the intensionalization procedure for higher-order extensional programs. Our presentation proceeds as follows: we first describe the transformation at an intuitive level. We then present a formal definition of the algorithm, and show that it is well-defined. The chapter concludes with examples of translation of higher-order programs under the proposed scheme.

5.1 The Transformation: an Overview

The purpose of this section is to define at an intuitive level the transformation algorithm from higher-order extensional 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, where $M > 0$. 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 IL 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 ($1 \leq m \leq M$), into an $(m-1)$-order one. Notice that this procedure also applies for the first step in the transformation, because we can assume that the source FL program is an IL program that does not contain any intensional operators.

A step of the algorithm can be 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. Let $\mathbf{f}$ be such a function. A typical call to $\mathbf{f}$ in the program will be of the form $\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$, where $\mathbf{call}_L$ is an intensional operator that has been introduced in the program during the previous stages of the transformation. The subscript L is a set that reflects the processing that has been performed on this particular call to $\mathbf{f}$ until now. We adopt the convention that "usual" calls to $\mathbf{f}$ of the form $\mathbf{f}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ can equivalently be written as $\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$, with $L = \emptyset$. In this case, $L = \emptyset$ signifies that no formal parameters of $\mathbf{f}$ have been removed so far in the previous steps of the algorithm, because the order of $\mathbf{f}$ has been lower than the order of other functions in the program.

Recall now that $\mathbf{f}$ is m-order and lets assume for simplicity that only its first argument is $(m-1)$-order. Lets denote by $\mathbf{E}$ the expression $\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$. Then, the algorithm will transform this call of $\mathbf{f}$ into $\mathbf{call}_{L \cup \{\langle m, \lceil \mathbf{E} \rceil \rangle\}}(\mathbf{f})(\mathbf{E}'_2, \ldots, \mathbf{E}'_n)$, where $\lceil \mathbf{E} \rceil$ is the Gödel number of $\mathbf{E}$ and $\mathbf{E}'_2, \ldots, \mathbf{E}'_n$ are the expressions $\mathbf{E}_1, \ldots, \mathbf{E}_n$ after they have been appropriately processed (the details will be given in Section 5.4). Notice how the set L is being built: every pair that is added to the set, reflects the order that is being currently eliminated, as well as the specific function call under consideration. In other words, the subscript of the **call** operator indicates the history of the orders of the actual parameters that have been removed from a call to $\mathbf{f}$, as well as the different forms that this call has taken during different steps of the transformation.

At the same time that the calls to **f** in the program are processed, the definition of **f** has to be altered as well. The main idea is to remove the $(m-1)$-order formals from the formal list of **f**, creating for each one of them a new definition and adding it to the program. Let **x** be an $(m-1)$-order formal of **f**. Then, the new definition created for **x**, will gather together all the actual parameters that correspond to **x** and that appear in calls to **f** in the program. This "gathering" is performed with the use of the intensional operator $\mathbf{actuals}_{m,R}$. The purpose of m and R is to guide in the selection of the appropriate argument of the operator.

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.

5.2 The Target Language

The programs that result at each step of the transformation algorithm, are programs of the language IL. However, as we would like to precisely define the algorithm and later reason about its correctness, we have to identify the exact subset of IL to which these target programs belong. This subset is characterized by the following definition:

Definition 5.1 The syntax of the intensional language SIL is recursively defined by the following rules, in which $\mathbf{S}, \mathbf{S}_i$ denote *simple expressions*, $\mathbf{E}, \mathbf{E}_i$ denote *expressions*, $\mathbf{F}, \mathbf{F}_i$ denote *definitions* and **P** denotes a *program*:

$$
\begin{aligned}
\mathbf{S} &::= \mathbf{call}_L(\mathbf{f}) \\
&\mid \mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_n) \\
&\mid \mathbf{call}_L(\mathbf{f})(\mathbf{S}_1, \ldots, \mathbf{S}_n) \\
\mathbf{E} &::= \mathbf{S} \\
&\mid \mathbf{actuals}_{m,R}(\langle \mathbf{S}_i \rangle_{i \in I})
\end{aligned}
$$

$$\mathbf{F} ::= (\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{E})$$
$$\mathbf{P} ::= \{\mathbf{F}_1, \ldots, \mathbf{F}_n\}$$

The typing rules and the semantics of IL transfer directly to SIL. Notice that the syntax of SIL imposes certain restrictions on the use of intensional operators. In particular:

- The argument of a **call** operator can only be a variable and not an arbitrary expression.
- The argument of the **actuals** operator can only be a sequence of *simple* expressions.

5.3 Preliminary Definitions

In this section, we describe the transformation of m-order SIL programs into $(m-1)$-order ones. Let $\mathbf{P}$ be an m-order SIL program, and let $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B_f}$ be an m-order function defined in $\mathbf{P}$. As before, we assume the existence of a Gödel function $\lceil \cdot \rceil$ which assigns unique natural numbers to expressions. Let $Sub(\mathbf{P})$ be the set of subexpressions of $\mathbf{P}$. We adopt the following conventions:

- The set of calls to the function $\mathbf{f}$ in $\mathbf{P}$ is defined as:

$$calls(\mathbf{f}, \mathbf{P}) = \{\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n) \in Sub(\mathbf{P})\}$$

- The set of labels of calls to $\mathbf{f}$ in $\mathbf{P}$ is defined as:

$$labels(\mathbf{f}, \mathbf{P}) = \{\lceil \mathbf{C} \rceil \mid \mathbf{C} \in calls(\mathbf{f}, \mathbf{P})\}$$

- Given a member of the set of labels, the function *set* extracts the subscript of the **call** operator of the expression that corresponds to this label:

$$set(\lceil \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n) \rceil) = L$$

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

$$\lceil \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n) \rceil \odot k = \mathbf{E}_k,\ k \in \{1, \ldots, n\}$$

- The function *options* is defined as follows:

$$options(\mathbf{f}, \mathbf{P}, m) = \langle set(i) \cup \{\langle m, i \rangle\} \rangle_{i \in labels(\mathbf{f}, \mathbf{P})}$$

  Intuitively, $options(\mathbf{f}, \mathbf{P}, m)$ is a sequence of the subscripts that calls to $\mathbf{f}$ will have after this step of the transformation is complete.

- Let $\mathbf{g}$ be a function in $\mathbf{P}$. Then, $low(\mathbf{g}, m)$ is the list of positions in the formal parameter list of $\mathbf{g}$, of those formals of $\mathbf{g}$ that have order less than $(m-1)$. For example, if only the second and third argument of $\mathbf{g}$ are less than $(m-1)$-order, then, $low(\mathbf{g}, m) = [2, 3]$.

- Let $\mathbf{F}$ be a definition for the function $\mathbf{g}$ in $\mathbf{P}$. The set of positions in the formal parameter list of $\mathbf{g}$ of those formals that have order equal to $(m-1)$, is represented by $high(\mathbf{F}, m)$. For example, if only the first and fourth arguments of $\mathbf{g}$ are $(m-1)$-order, then $high(\mathbf{F}, m) = \{1, 4\}$.

- Let $\mathbf{x} \in Vars(\mathbf{P})$ with $\mathbf{x} : (\tau_1, \ldots, \tau_k) \rightarrow \iota$. Then, a function $Form$ can be easily defined such that $Form(\mathbf{x}, \mathbf{P})$ is a list of k variable symbols, which satisfies the following:

  - No variable in the list appears in the program $\mathbf{P}$.
  - Given $\mathbf{y} \neq \mathbf{x}$, $Form(\mathbf{x}, \mathbf{P})$ and $Form(\mathbf{y}, \mathbf{P})$ do not have any elements in common.

5.4 A Formal Definition of the Transformation

We are now in a position to give the formal definition of the transformation algorithm. The elimination of the $(m-1)$-order arguments from function calls, is accomplished by the $\mathcal{E}_m$ function defined in Figure 5.1. The first rule is for the case of (possibly higher-order)

figure 5.1 Processing expressions of the program.

$$\frac{\mathbf{E} = \mathbf{call}_L(\mathbf{f})}{\mathcal{E}_m(\mathbf{E}) = \mathbf{call}_L(\mathbf{f})}$$

$$\frac{\mathbf{E} = \mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n)}{\mathcal{E}_m(\mathbf{E}) = \mathbf{c}(\mathcal{E}_m(\mathbf{E}_1), \ldots, \mathcal{E}_m(\mathbf{E}_n))}$$

$$\frac{\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ order(\mathbf{f}) = m,\ low(\mathbf{f}, m) = [i_1, \ldots, i_k]}{\mathcal{E}_m(\mathbf{E}) = (\mathbf{call}_{L \cup \{(m, \lceil \mathbf{E} \rceil)\}}(\mathbf{f}))(\mathcal{E}_m(\mathbf{E}_{i_1}), \ldots, \mathcal{E}_m(\mathbf{E}_{i_k}))}$$

$$\frac{\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n),\ order(\mathbf{f}) < m}{\mathcal{E}_m(\mathbf{E}) = \mathbf{call}_L(\mathbf{f})(\mathcal{E}_m(\mathbf{E}_1), \ldots, \mathcal{E}_m(\mathbf{E}_n))}$$

$$\frac{\mathbf{E} = \mathbf{actuals}_{k,R}\langle \mathbf{E}_i \rangle_{i \in I}}{\mathcal{E}_m(\mathbf{E}) = \mathbf{actuals}_{k,R}\langle \mathcal{E}_m(\mathbf{E}_i) \rangle_{i \in I}}$$

variables that are encountered during the transformation, and which are prefixed by a $\mathbf{call}_L$ operator (notice that if $L = \emptyset$, then the expression corresponds to just a variable). 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. Notice, that if two function calls in the program are the same, then their translations according to $\mathcal{E}_m$ are identical. 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. Finally, the last rule says that in order to translate an **actuals** expression, it suffices to translate every expression in the argument sequence of the operator.

The function $\mathcal{D}_m$ is used to process the definitions in **P**, removing their $(m-1)$-order formal parameters. Notice that at the same time, the body of each definition is processed using the function $\mathcal{E}_m$. The definition of $\mathcal{D}_m$ is given in Figure 5.2.

The function $\mathcal{A}_m$ creates a new definition for each $(m-1)$-order formal parameter in the program **P**. Let **F** be the definition for function **f** in **P**. If the j'th argument of **f** is $(m-1)$-order, then the function $\mathcal{A}_{\mathbf{F},j,m}$ returns a set that contains a new definition for this formal. The body of this definition consists of the operator **actuals** applied to a set of arguments; intuitively, these arguments are the processed actual parameters that appear in calls to **f** in the program, and that correspond to the j'th formal parameter of **f**. The formal definition of $\mathcal{A}_m$ is given in Figure 5.3.

figure 5.2 Eliminating the $(m-1)$-order formals from definitions

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

$$\frac{\mathbf{F} = (\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}),\ low(\mathbf{f},m) = [i_1,\ldots,i_k]}{\mathcal{D}'_m(\mathbf{F}) = \{\mathbf{f}(\mathbf{x}_{i_1},\ldots,\mathbf{x}_{i_k}) \doteq \mathcal{E}_m(\mathbf{B_f})\}}$$

Notice that the definitions that result from the function $\mathcal{A}_m$, are valid definitions of the language *IL* but not of the language *SIL*. In order to transform them to *SIL* definitions, we need to pass the formal parameters inside the scope of the **actuals** operator. In order for this to be done in a semantics-preserving way, we first need to prefix the formals with appropriate **call** operators that cancel the effect that **actuals** has on them. The way this is performed is described in Figure 5.4, by the function In.

The translation of an m-order *SIL* program into an $(m-1)$-order one, is performed by

figure 5.3 Creating a new definition for each $(m-1)$-order formal

$$\mathcal{A}_m(\mathbf{P}) = \bigcup_{\mathbf{F}\in\mathbf{P}} \bigcup_{j\in high(\mathbf{F},m)} \mathcal{A}_{\mathbf{F},j,m}(\mathbf{P})$$

$$\frac{\mathbf{F} = (\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f}),\ R = options(\mathbf{f},\mathbf{P},m),\ Form(\mathbf{x}_j,\mathbf{P}) = [\mathbf{z}_1,\ldots,\mathbf{z}_k]}{\mathcal{A}_{\mathbf{F},j,m}(\mathbf{P}) = \{\mathbf{x}_j(\mathbf{z}_1,\ldots,\mathbf{z}_k) \doteq (\mathbf{actuals}_{m,R}\langle\mathcal{E}_m(i\odot j)\rangle_{i\in dom(R)})(\mathbf{z}_1,\ldots,\mathbf{z}_k)\}}$$

figure 5.4 Passing formal parameters inside **actuals**

$$In(\mathbf{P}) = \bigcup_{\mathbf{F}\in\mathbf{P}} In'(F)$$

$$\frac{\mathbf{F} = (\mathbf{x}(\mathbf{z}_1,\ldots,\mathbf{z}_k) \doteq (\mathbf{actuals}_{m,R}\langle\mathbf{E}_i\rangle_{i\in dom(R)})(\mathbf{z}_1,\ldots,\mathbf{z}_k))}{In'(\mathbf{F}) = \{\mathbf{x}(\mathbf{z}_1,\ldots,\mathbf{z}_k) \doteq \mathbf{actuals}_{m,R}\langle\mathbf{E}_i(\mathbf{call}_{R_i}(\mathbf{z}_1),\ldots,\mathbf{call}_{R_i}(\mathbf{z}_k))\rangle_{i\in dom(R)}\}}$$

the function $Step_m$ shown in Figure 5.5.

figure 5.5 Transforming an m-order SIL program into an $(m-1)$-order one

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

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 in Figure 5.6.

5.5 Properties of the Algorithm

The well-definedness of the algorithm is ensured by the following theorem:

Theorem 5.1 Let $\mathbf{P}$ be an m-order SIL program. Then, $Step_m(\mathbf{P})$ is a SIL program.

figure 5.6 The overall translation of an M-order program

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

Proof: It suffices to show that $\mathcal{D}_m$ and $In \circ \mathcal{A}_m$ return SIL definitions. First, notice that the function $\mathcal{E}_m$ when applied to SIL expressions it returns SIL expressions, and when it is applied to simple SIL expressions it returns simple SIL expressions.

Consider now the function $\mathcal{D}_m$. Obviously, $\mathcal{D}_m$ simply removes certain formals from a definition while processing the definition's body with the function $\mathcal{E}_m$. Therefore, every definition that it produces is a SIL one.

Consider on the other hand the function $In \circ \mathcal{A}_m$. The function $\mathcal{A}_m$ produces definitions of the form:

$$\mathbf{x}_j(\mathbf{z}_1, \ldots, \mathbf{z}_k) \doteq (\mathbf{actuals}_{m,R}\langle \mathcal{E}_m(i \odot j)\rangle_{i \in dom(R)})(\mathbf{z}_1, \ldots, \mathbf{z}_k)$$

The expressions that appear in the argument of the **actuals** are of the form $\mathcal{E}_m(i \odot j)$, where i is the Gödel number of a simple SIL expression of the form $\mathbf{call}_L(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r)$. By the definition of $\mathcal{E}_m$, $\mathcal{E}_m(i \odot j)$ is equal to $\mathcal{E}_m(\mathbf{S}_j)$, which is a simple SIL expression. We distinguish two cases:

1. $\mathbf{x}_j : \iota$. Then, $k = 0$ (i.e., no new formals are introduced for $\mathbf{x}_j$) and the definition for $\mathbf{x}_j$ that will be returned from $In \circ \mathcal{A}_m$ is a SIL one.

2. $\mathbf{x}_j : \tau$ and $\tau \neq \iota$. Then, $k > 0$ (i.e., new formal parameters are introduced for $\mathbf{x}_j$). Also, as $\mathcal{E}_m(\mathbf{S}_j)$ has higher-order type and at the same time it is a simple SIL expression, it can only be of the form $\mathbf{call}_L(\mathbf{h})$, for some $L \subseteq N \times N$ and $\mathbf{h} \in Vars(\mathbf{P})$. When the formals $\mathbf{z}_1, \ldots, \mathbf{z}_k$ are pushed inside the scope of **actuals** by the function In, the expression that we are considering becomes $\mathbf{call}_L(\mathbf{h})(\mathbf{call}_{R_i}(\mathbf{z}_1), \ldots, \mathbf{call}_{R_i}(\mathbf{z}_k))$. But this is a simple SIL expression. Therefore, the definition for $\mathbf{x}_j$ that will be

returned by $In \circ \mathcal{A}_m$ is a *SIL* one.

Therefore, all definitions returned by $In \circ \mathcal{A}_m$ are definitions of the language *SIL*. This concludes the proof of the theorem. ■

Lemma 5.1 Let **P** be an m-order *SIL* program, where $m > 1$. Then, $In(\mathcal{A}_m(\mathbf{P}))$ is a set of definitions of the form

$$\mathbf{x}(\mathbf{z}_1, \ldots, \mathbf{z}_k) \doteq \mathbf{actuals}_{m,R}(\langle \mathbf{S}_i \rangle_{i \in dom(R)})$$

where for every $i \in dom(R)$, there exists $L \subseteq N \times N$ and $\mathbf{h} \in Vars(\mathbf{P})$ such that $\mathbf{S}_i = \mathbf{call}_L(\mathbf{h})(\mathbf{call}_{R_i}(\mathbf{z}_1), \ldots, \mathbf{call}_{R_i}(\mathbf{z}_k))$.

Proof: The fact that $m > 1$ implies that **x** is a variable of order greater than 1. Then, the lemma follows directly from the proof of Theorem 5.1. ■

Finally, the following theorem guarantees the correctness of the In function. In other words, it shows that the placement of the new formal parameters inside the scope of **actuals** by prefixing them with appropriate **call** operators, is a semantics preserving transformation.

Theorem 5.2 Let **P** be an m-order *SIL* program. Then, the synchronic semantics of $\mathcal{D}_m(\mathbf{P}) \cup \mathcal{A}_m(\mathbf{P})$ and $\mathcal{D}_m(\mathbf{P}) \cup In(\mathcal{A}_m(\mathbf{P}))$ coincide.

Proof: (Outline) It suffices to show that the least environment that satisfies the definitions in program $\mathcal{D}_m(\mathbf{P}) \cup \mathcal{A}_m(\mathbf{P})$ under the synchronic interpretation, also satisfies the definitions in $\mathcal{D}_m(\mathbf{P}) \cup In(\mathcal{A}_m(\mathbf{P}))$, and vice-versa. This follows in a straightforward way using the semantics of the **call** and **actuals** operators. ■

5.6 Transformation of the apply Program

In this section, we give an example use of the transformation algorithm. The program we consider is second-order due to the function **apply**, whose first argument is a first-order function.

$$\begin{aligned}
\texttt{result} &\doteq \texttt{apply(inc,8)}\\
\texttt{apply(f,x)} &\doteq \texttt{f(x)}\\
\texttt{inc(y)} &\doteq \texttt{y+1}
\end{aligned}$$

In the first step of the transformation the formal parameter **f** of **apply** is eliminated. Let $l_1 = \lceil \texttt{apply(inc,8)} \rceil$. Then, according to the definition of the function $\mathcal{E}_m$, the call **apply(inc,8)** will become $(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}$ (notice that $m = 2$, because the program is second-order). The argument **f** is eliminated from **apply**, and a new definition is created for it using the $\mathcal{A}_m$ function. One can easily verify that the set $options(\mathbf{f}, \mathbf{P}, m)$ is equal to $\{\langle l_1, \{\langle 2, l_1\rangle\}\rangle\}$. Let **z** be the variable returned by the function *Form*. Then, the program that results after the first step of the transformation is complete, is the following:

$$\begin{aligned}
\texttt{result} &\doteq (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}\\
\texttt{apply(x)} &\doteq \texttt{f(x)}\\
\texttt{f(z)} &\doteq (\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1, \texttt{inc}\rangle\})\texttt{(z)}\\
\texttt{inc(y)} &\doteq \texttt{y+1}
\end{aligned}$$

Now we must advance **z** before we enter it inside the scope of the intensional operation. This is done by prefixing it with $\texttt{call}_{\{\langle 2,l_1\rangle\}}$, i.e., with the same intensional operator that appears in the corresponding call to **apply**:

$$\begin{aligned}
\texttt{result} &\doteq (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}\\
\texttt{apply(x)} &\doteq \texttt{f(x)}\\
\texttt{f(z)} &\doteq \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1, \texttt{inc}(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)})\rangle\}\\
\texttt{inc(y)} &\doteq \texttt{y+1}
\end{aligned}$$

The next step in the transformation can now be performed. This consists of eliminating the (zero-order) variables `x`, `z`, `y`, and adding a new definition for each one of them in the program. Let $l_2 = \lceil(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(apply))(8)}\rceil$, $l_3 = \lceil\texttt{f(x)}\rceil$, and $l_4 = \lceil\texttt{inc(call}_{\{\langle 2,l_1\rangle\}}\texttt{(z))}\rceil$. The final program that results from the transformation is the following:

$$
\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}}\texttt{(apply)}\\
\texttt{apply} &\doteq \texttt{call}_{\{\langle 1,l_3\rangle\}}\texttt{(f)}\\
\texttt{f} &\doteq \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1, \texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}\rangle\}\\
\texttt{inc} &\doteq \texttt{y+1}\\
\texttt{x} &\doteq \texttt{actuals}_{1,\{\langle l_2,\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}\rangle\}}(\{\langle l_2, 8\rangle\})\\
\texttt{z} &\doteq \texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle\}}(\{\langle l_3, \texttt{x}\rangle\})\\
\texttt{y} &\doteq \texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4, \texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}\rangle\})
\end{aligned}
$$

Let $\widehat{u}$ be the least environment that satisfies the definitions of the resulting zero-order intensional program under the synchronic interpretation. Then, the meaning of the program can be computed as shown in Figure 5.7.

5.7 Transformation of the `twice` Program

In this section we present the correct transformation of the `twice` program introduced in Section 2.9. Recall that the source program is the following:

```
result      ≐ twice(inc,8)
twice(f,x)  ≐ f(f(x))
inc(y)      ≐ y+1
```

Let $l_1 = \lceil\texttt{twice(inc,8)}\rceil$. As before, we eliminate the highest-order formal, which in

figure 5.7 The meaning of the intensional program that results from **apply**

$$
\begin{aligned}
& [\![\texttt{call}_{\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}}\texttt{(apply)}]\!]^*(\widehat{u})(\langle[\,],[\,],\ldots\rangle) = \\
= \;& [\![\texttt{apply}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{call}_{\{\langle 1,l_3\rangle\}}\texttt{(f)}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{f}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}\rangle\}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}]\!]^*(\widehat{u})(\langle[l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{inc}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y+1}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + [\![\texttt{1}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4,\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}\rangle\})]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}]\!]^*(\widehat{u})(\langle[l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{z}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle\}}(\{\langle l_3,\texttt{x}\rangle\})]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{x}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_2,\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}\rangle\}}(\{\langle l_2,8\rangle\})]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{8}]\!]^*(\widehat{u})(\langle[\,],[\,],\ldots\rangle) + 1 \\
= \;& 8 + 1 \\
= \;& 9
\end{aligned}
$$

this case is the variable `f`, creating at the same time a new definition for `f`:

$$\begin{aligned}
&\texttt{result} &&\doteq\ (\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{twice}))(8)\\
&\texttt{twice(x)} &&\doteq\ \texttt{f(f(x))}\\
&\texttt{f(z)} &&\doteq\ (\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{inc}\rangle\})(\texttt{z})\\
&\texttt{inc(y)} &&\doteq\ \texttt{y+1}
\end{aligned}$$

As usual, we must advance `z` before we enter it inside **actuals**. This is done by prefixing it with $\texttt{call}_{\{\langle 2,l_1\rangle\}}$, i.e., with the same intensional operator that appears in the corresponding call to `twice`:

$$\begin{aligned}
&\texttt{result} &&\doteq\ (\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{twice}))(8)\\
&\texttt{twice(x)} &&\doteq\ \texttt{f(f(x))}\\
&\texttt{f(z)} &&\doteq\ \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{inc}(\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{z}))\rangle\}\\
&\texttt{inc(y)} &&\doteq\ \texttt{y+1}
\end{aligned}$$

We are now in the position to perform the last step of the transformation. Let $l_2 = \lceil(\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{twice}))(8)\rceil$, $l_3 = \lceil\texttt{f(f(x))}\rceil$, $l_4 = \lceil\texttt{inc}(\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{z}))\rceil$ and $l_5 = \lceil\texttt{f(x)}\rceil$. The final step of the transformation gives the following program:

$$\begin{aligned}
&\texttt{result} &&\doteq\ \texttt{call}_{\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}}(\texttt{twice})\\
&\texttt{twice} &&\doteq\ \texttt{call}_{\{\langle 1,l_3\rangle\}}(\texttt{f})\\
&\texttt{f} &&\doteq\ \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{call}_{\{\langle 1,l_4\rangle\}}(\texttt{inc})\rangle\}\\
&\texttt{inc} &&\doteq\ \texttt{y+1}\\
&\texttt{x} &&\doteq\ \texttt{actuals}_{1,\{\langle l_2,\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}\rangle\}}(\{\langle l_2,8\rangle\})\\
&\texttt{z} &&\doteq\ \texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle,\langle l_5,\{\langle 1,l_5\rangle\}\rangle\}}(\{\langle l_3,\texttt{call}_{\{\langle 1,l_5\rangle\}}(\texttt{f})\rangle,\langle l_5,\texttt{x}\rangle\})\\
&\texttt{y} &&\doteq\ \texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4,\texttt{call}_{\{\langle 2,l_1\rangle\}}(\texttt{z})\rangle\})
\end{aligned}$$

The meaning of the intensional program that results from the translation of the `twice` extensional program, is computed in Figure 5.8.

figure 5.8 The meaning of the intensional program that results from `twice`

$$
\begin{aligned}
& [\![\texttt{call}_{\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}}\texttt{(twice)}]\!]^*(\widehat{u})(\langle[\,],[\,],\ldots\rangle) = \\
= \;& [\![\texttt{twice}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{call}_{\{\langle 1,l_3\rangle\}}\texttt{(f)}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{f}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}\rangle\}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) \\
= \;& [\![\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}]\!]^*(\widehat{u})(\langle[l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{inc}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y+1}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + [\![\texttt{1}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4,\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}\rangle\})]\!]^*(\widehat{u})(\langle[l_4,l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}]\!]^*(\widehat{u})(\langle[l_3,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{z}]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle,\langle l_5,\{\langle 1,l_5\rangle\}\rangle\}}(\{\langle l_3,\texttt{call}_{\{\langle 1,l_5\rangle\}}\texttt{(f)}\rangle,\langle l_5,\texttt{x}\rangle\})]\!]^*(\widehat{u})(\langle[l_3,l_2],[l_1],\ldots\rangle) \\
& +1 \\
= \;& [\![\texttt{call}_{\{\langle 1,l_5\rangle\}}\texttt{(f)}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{f}]\!]^*(\widehat{u})(\langle[l_5,l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle\}}\{\langle l_1,\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}\rangle\}]\!]^*(\widehat{u})(\langle[l_5,l_2],[l_1],\ldots\rangle) + 1 \\
= \;& [\![\texttt{call}_{\{\langle 1,l_4\rangle\}}\texttt{(inc)}]\!]^*(\widehat{u})(\langle[l_5,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{inc}]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{y+1}]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle) + [\![\texttt{1}]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle) + 1 \\
= \;& [\![\texttt{y}]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle)1 + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_4,\{\langle 1,l_4\rangle\}\rangle\}}(\{\langle l_4,\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}\rangle\})]\!]^*(\widehat{u})(\langle[l_4,l_5,l_2],[\,],\ldots\rangle) + 1 + 1 \\
= \;& [\![\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{(z)}]\!]^*(\widehat{u})(\langle[l_5,l_2],[\,],\ldots\rangle) + 1 + 1 \\
= \;& [\![\texttt{z}]\!]^*(\widehat{u})(\langle[l_5,l_2],[l_1],\ldots\rangle) + 1 + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_3,\{\langle 1,l_3\rangle\}\rangle,\langle l_5,\{\langle 1,l_5\rangle\}\rangle\}}(\{\langle l_3,\texttt{call}_{\{\langle 1,l_5\rangle\}}\texttt{(f)}\rangle,\langle l_5,\texttt{x}\rangle\})]\!]^*(\widehat{u})(\langle[l_5,l_2],[l_1],\ldots\rangle) \\
& +1 + 1 \\
= \;& [\![\texttt{x}]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) + 1 + 1 \\
= \;& [\![\texttt{actuals}_{1,\{\langle l_2,\{\langle 2,l_1\rangle,\langle 1,l_2\rangle\}\rangle\}}(\{\langle l_2,8\rangle\})]\!]^*(\widehat{u})(\langle[l_2],[l_1],\ldots\rangle) + 1 + 1 \\
= \;& [\![\texttt{8}]\!]^*(\widehat{u})(\langle[\,],[\,],\ldots\rangle) + 1 + 1 \\
= \;& 8 + 1 + 1 \\
= \;& 10
\end{aligned}
$$

5.8 An Example Involving Recursion

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

$$\begin{aligned}
&\texttt{result} &&\doteq \texttt{ffac(sq,4)}\\
&\texttt{ffac(h,n)} &&\doteq \texttt{if (n<1) then 1 else h(n)*ffac(h,n-1)}\\
&\texttt{sq(a)} &&\doteq \texttt{a*a}
\end{aligned}$$

Let $l_1 = \lceil\texttt{ffac(sq,4)}\rceil$ and $l_2 = \lceil\texttt{ffac(h,n-1)}\rceil$. The first step of the transformation gives the following output (before introducing the new formal parameters for h):

$$\begin{aligned}
&\texttt{result} &&\doteq (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{ffac})\texttt{(4)}\\
&\texttt{ffac(n)} &&\doteq \texttt{if (n<1) then 1 else h(n)*(call}_{\{\langle 2,l_2\rangle\}}\texttt{ ffac)(n-1)}\\
&\texttt{h} &&\doteq \texttt{actuals}_{2,\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle,\langle l_2,\{\langle 2,l_2\rangle\}\rangle\}}(\{\langle l_1,\texttt{sq}\rangle,\langle l_2,\texttt{h}\rangle\})\\
&\texttt{sq(a)} &&\doteq \texttt{a*a}
\end{aligned}$$

We now introduce a new variable z and pass it inside the actuals operator. For simplicity, we write R_1 instead of $\{\langle l_1,\{\langle 2,l_1\rangle\}\rangle,\langle l_2,\{\langle 2,l_2\rangle\}\rangle\}$.

$$\begin{aligned}
&\texttt{result} &&\doteq (\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{ffac})\texttt{(4)}\\
&\texttt{ffac(n)} &&\doteq \texttt{if (n<1) then 1 else h(n)*(call}_{\{\langle 2,l_2\rangle\}}\texttt{ ffac)(n-1)}\\
&\texttt{h(z)} &&\doteq \texttt{actuals}_{2,R_1}(\{\langle l_1,\texttt{sq(call}_{\{\langle 2,l_1\rangle\}}\texttt{(z))}\rangle,\langle l_2,\texttt{h(call}_{\{\langle 2,l_2\rangle\}}\texttt{(z))}\rangle\})\\
&\texttt{sq(a)} &&\doteq \texttt{a*a}
\end{aligned}$$

Let $l_3 = \lceil(\texttt{call}_{\{\langle 2,l_1\rangle\}}\texttt{ffac})\texttt{(4)}\rceil$, $l_4 = \lceil\texttt{h(n)}\rceil$, $l_5 = \lceil(\texttt{call}_{\{\langle 2,l_2\rangle\}}\texttt{ffac})\texttt{(n-1)}\rceil$, $l_6 = \lceil\texttt{sq(call}_{\{\langle 2,l_1\rangle\}}\texttt{(z))}\rceil$, and $l_7 = \lceil\texttt{h(call}_{\{\langle 2,l_2\rangle\}}\texttt{(z))}\rceil$. In the last step of the transformation, all first-order functions are processed, and a new definition is created for every zero-order formal parameter in the program. It can be easily verified that the last step of the trans-

formation gives the following output:

$$
\begin{aligned}
\texttt{result} &\doteq \texttt{call}_{\{\langle 2,l_1\rangle\},\{\langle 1,l_3\rangle\}}\texttt{ffac} \\
\texttt{ffac} &\doteq \texttt{if (n<1) then 1 else (call}_{\{\langle 1,l_4\rangle\}}\texttt{ h)*(call}_{\{\langle 2,l_2\rangle\},\{\langle 1,l_5\rangle\}}\texttt{ ffac)} \\
\texttt{h} &\doteq \texttt{actuals}_{2,R_1}(\{\langle l_1, \texttt{call}_{\{\langle 1,l_6\rangle\}}(\texttt{sq})\rangle, \langle l_2, \texttt{call}_{\{\langle 1,l_7\rangle\}}(\texttt{h})\rangle\}) \\
\texttt{sq} &\doteq \texttt{a * a} \\
\texttt{n} &\doteq \texttt{actuals}_{1,R_2}(\{\langle l_3, 4\rangle\}) \\
\texttt{z} &\doteq \texttt{actuals}_{1,R_3}(\{\langle l_4, \texttt{n}\rangle, \langle l_7, \texttt{call}_{\{\langle 2,l_2\rangle\}}(\texttt{z})\rangle\}) \\
\texttt{a} &\doteq \texttt{actuals}_{1,R_4}(\{\langle l_6, \texttt{call}_{\{\langle 1,l_6\rangle\}}(\texttt{sq})\rangle\})
\end{aligned}
$$

The meaning of the program can be computed using the same ideas as in the previous examples.

Chapter 6

Theoretical Foundations

In this chapter we present in a rigorous way the correctness proof of the intensionalization technique for higher-order programs. In the following, we first make some assumptions that help us simplify the subsequent presentation. We then introduce certain notational conventions, and give an informal outline of the proof. Finally, the correctness proof of the transformation algorithm is presented in detail, and its main points are identified and discussed.

6.1 Assumptions

To simplify the presentation of the correctness proof, certain assumptions are adopted, which we outline below. As before, let $\mathbf{P}_M$, $M > 0$, be the M-order source extensional program on which the transformation algorithm is to be applied. Let $\mathbf{P}_m$, $0 \leq m \leq M$ be the *SIL* programs that result at each step of the transformation. Then:

1. We can view $\mathbf{P}_M$ as an intensional program (that is, a *SIL* program) that does not contain any intensional operators. In this way we achieve a homogeneous view of the transformation algorithm: each step takes as input a *SIL* program and produces as

output a *SIL* program of lower order. Moreover, in this way we avoid having to give a separate proof for the first step in the transformation.

2. We assume that all the functions defined in $\mathbf{P}_M$, have their highest-order arguments first, i.e., if $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$ is a function defined in $\mathbf{P}_M$, then $order(\mathbf{x}_1) \geq order(\mathbf{x}_2) \geq \cdots \geq order(\mathbf{x}_n)$. 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 realized that the above assumptions do not impose any loss of generality, and that they help reduce the heavy notation that is required for the proof that will follow.

We now introduce a theorem that will be used in certain parts of the proof. We first define the notion of the *restriction* of a function:

Definition 6.1 Let $f : A \rightarrow B$ and $K \subseteq A$. The *restriction* $f|K$ of f on K is the function $g : K \rightarrow B$ such that for all $k \in K$, $g(k) = f(k)$.

The following theorem suggests that the semantics of a program $\mathbf{P}_m$ that results in an intermediate step of the transformation, only depends on those dimensions of the context that are greater than m.

Theorem 6.1 Let $\mathbf{P}_M$, $M > 0$ be an M-order extensional program and let $\mathbf{P}_m$, $0 \leq m \leq M$ be a program that results during the transformation of $\mathbf{P}_M$. Let u be the least environment that satisfies the definitions of $\mathbf{P}_m$ and let $K = \{M, M-1,\ldots,m+1\}$. Then, for every $\mathbf{f} \in func(\mathbf{P}_m)$, and for all $w_1, w_2 \in W$ with $w_1|K = w_2|K$, $u(\mathbf{f})(w_1) = u(\mathbf{f})(w_2)$.

Proof: (Outline) By the definition of the transformation algorithm it is easy to conclude that the operators **call** and **actuals** that appear in $\mathbf{P}_m$ manipulate at most the dimensions $M, M-1,\ldots,m+1$ of the contexts. Therefore the semantics of function variables in the program are independent of the rest of the dimensions. ∎

Program	$\mathbf{P}_m$	$\mathbf{P}_{m-1}$
Function	$\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$	$\mathbf{f}(\mathbf{x}_{l+1},\ldots,\mathbf{x}_n) \doteq \mathcal{E}_m(\mathbf{B_f})$
Full Call	$\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$	$\mathbf{call}_{L\cup\{\langle m,\lceil\mathbf{E}\rceil\rangle\}}(\mathbf{f})(\mathcal{E}_m(\mathbf{E}_{l+1}),\ldots,\mathcal{E}_m(\mathbf{E}_n))$

Table 6.1: Notation for functions with order equal to m.

6.2 Notation

The basic idea of the correctness proof is to show that the transformation from the program $\mathbf{P}_m$ to $\mathbf{P}_{m-1}$, is performed in a semantics preserving way. Let $\mathbf{f}$ be a function defined in $\mathbf{P}_m$ as $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$. In the proof that will follow, there exist two cases that have to be considered separately, depending on the order of $\mathbf{f}$ in $\mathbf{P}_m$:

Case 1: The order of $\mathbf{f}$ is equal to m. Then, every call to $\mathbf{f}$ in $\mathbf{P}_m$ is of the form $\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$. Assume that $\mathbf{x}_1,\ldots,\mathbf{x}_l$, $1 \leq l \leq n$ are the $(m-1)$-order arguments of $\mathbf{f}$. Consider now the effect that the transformation algorithm has on $\mathbf{P}_m$ and $\mathbf{f}$. The transformed program will be $\mathbf{P}_{m-1}$, and the new definition for $\mathbf{f}$ in $\mathbf{P}_{m-1}$ will be $\mathbf{f}(\mathbf{x}_{l+1},\ldots,\mathbf{x}_n) \doteq \mathcal{E}_m(\mathbf{B_f})$. Lets denote by $\mathbf{E}$ the expression $\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$. Then, according to the definition of the transformation algorithm, in $\mathbf{P}_{m-1}$ this call will have the form $\mathbf{call}_{L\cup\{\langle m,\lceil\mathbf{E}\rceil\rangle\}}(\mathbf{f})(\mathcal{E}_m(\mathbf{E}_{l+1}),\ldots,\mathcal{E}_m(\mathbf{E}_n))$. The above facts are summarized in Table 6.1

Case 2: The order of $\mathbf{f}$ is less than m. Consider the transformed program $\mathbf{P}_{m-1}$. Then, according to the transformation algorithm, the definition for $\mathbf{f}$ in $\mathbf{P}_{m-1}$ will be $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathcal{E}_m(\mathbf{B_f})$. We do not need to introduce any notation regarding the calls of $\mathbf{f}$ in the program, because as we are going to see, the theorem for this case is going to be much easier to state than in Case 1.

Moreover, for every one of the above cases, two subcases have to be considered, as the following definition suggests:

Definition 6.2 Let $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$ be a function definition in $\mathbf{P}_m$. The definition of

f is an *ordinary* one if the operator **actuals** does not appear in $\mathbf{B_f}$. Otherwise, it is an *actuals* definition.

Notice that the main difference between *actuals* and *ordinary* definitions, lies in their structure. As demonstrated by Lemma 5.1, an *actuals* definition has a known structure, while an *ordinary* one has a structure which is not known in advance. As a result, when an *ordinary* definition is considered, a structural induction proof will often be required, while a direct proof will often be possible for an *actuals* definition.

6.3 An Outline of the Proof

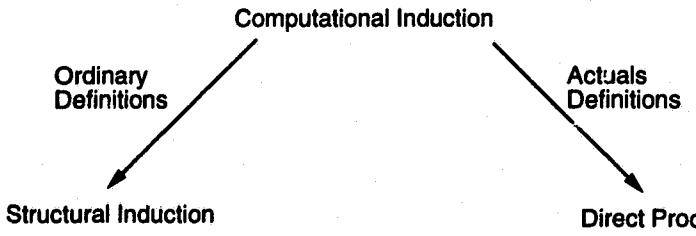
Let u and $\widehat{u}$ be the least environments that satisfy the definitions in the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$respectively. The basic idea of the proof is to establish a relationship between the values of **f** in $\mathbf{P}_m$ and in $\mathbf{P}_{m-1}$, i.e., between $u(\mathbf{f})$ and $\widehat{u}(\mathbf{f})$. Intuitively, we expect these two to be the same for functions that have order less than m in $\mathbf{P}_m$. However, if **f** is an m-order function in $\mathbf{P}_m$, then its $(m-1)$-order arguments will be eliminated by the transformation algorithm, and therefore $u(\mathbf{f})$ and $\widehat{u}(\mathbf{f})$ will be different (they will denote functions of different arities). However, if we expect the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$ to be semantically equivalent, it is reasonable to believe that there exists a strong relationship between $\widehat{u}(\mathbf{f})$ and $u(\mathbf{f})$. More specifically, we expect to establish that $\widehat{u}(\mathbf{f})$ and $u(\mathbf{f})$ are two functions that produce identical results under certain conditions. These conditions are present in the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$, or in other words the function **f** is used in the two programs in such a way that semantic equivalence is ensured.

It turns out that in order to establish such a result, we must proceed in two steps: first we need to show a $\sqsubseteq$ relation and then a $\sqsupseteq$ one, regarding the values of $\widehat{u}(\mathbf{f})$ and $u(\mathbf{f})$. In order to demonstrate the first relation, we perform an induction on the approximations $\widehat{u}_k$ of u (computational induction), while we keep u unchanged. To demonstrate the second relation, we perform an induction on the approximations u_k of u, while we keep $\widehat{u}$ unchanged.

Moreover, each one of the above steps requires two cases to be considered, depending on the kind of definition that **f** has in program $\mathbf{P}_m$. The first case is when **f** has an *ordinary* definition in $\mathbf{P}_m$, and the second is when **f** has an *actuals* definition in $\mathbf{P}_m$. In the former case, a structural induction is performed on the body of **f**, while on the later a direct proof is obtained.

The structure of the proof is illustrated in Figure 6.1. In the following section we present the overall proof in detail.

figure 6.1 Demonstrating each of the $\sqsubseteq$ and $\sqsupseteq$ relations.



6.4 Correctness Proof of the Transformation

As discussed in the previous sections, let $\mathbf{P}_M$, $M > 0$, be the M-order source extensional program on which the transformation algorithm is applied. The programs that result at successive stages of the algorithm are $\mathbf{P}_M, \mathbf{P}_{M-1}, \ldots, \mathbf{P}_0$. Consider $\mathbf{P}_m$, $0 \leq m < M$. 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 6.2 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 $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1), \ldots, order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$call_{L \cup \{(m, \lceil \mathbf{E} \rceil)\}}(\widehat{u}(\mathbf{f}))(w)(d_{l+1}, \ldots, d_n) \sqsubseteq$$
$$call_L(u(\mathbf{f}))(w)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u})(w), \ldots, [\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u})(w), d_{l+1}, \ldots, d_n)$$

- For every definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1), \ldots, \ldots, order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}(\mathbf{f})(w)(d_1, \ldots, d_n) \sqsubseteq u(\mathbf{f})(w)(d_1, \ldots, d_n)$$

Proof: 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 $k \in N$, for every definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1), \ldots, order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for every call $\mathbf{E} =$

$\mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$call_{L\cup\{\langle m,\lceil \mathbf{E}\rceil\rangle\}}(\widehat{u}_k(\mathbf{f}))(w)(d_{l+1},\ldots,d_n) \sqsubseteq$$
$$call_L(u(\mathbf{f}))(w)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u}_k)(w),\ldots,[\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u}_k)(w),d_{l+1},\ldots,d_n)$$

- For every $k \in N$, for every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1),\ldots,order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}_k(\mathbf{f})(w)(d_1,\ldots,d_n) \sqsubseteq u(\mathbf{f})(w)(d_1,\ldots,d_n)$$

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 $k \geq 0$. We show the claim for $k+1$. That is, we show that:

- For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1),\ldots,order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1),\ldots,order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$call_{L\cup\{\langle m,\lceil \mathbf{E}\rceil\rangle\}}(\widehat{u}_{k+1}(\mathbf{f}))(w)(d_{l+1},\ldots,d_n) \sqsubseteq$$
$$call_L(u(\mathbf{f}))(w)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u}_{k+1})(w),\ldots,[\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u}_{k+1})(w),d_{l+1},\ldots,d_n)$$

- For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1),\ldots,order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}_{k+1}(\mathbf{f})(w)(d_1,\ldots,d_n) \sqsubseteq u(\mathbf{f})(w)(d_1,\ldots,d_n)$$

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

- Let $\widehat{L} = L \cup \{\langle m,\lceil \mathbf{E}\rceil\rangle\}$. For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1),\ldots,order(\tau_l) =$

$(m-1)$ and $order(\tau_{l+1}) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}_{k+1}(\mathbf{f})(w \Downarrow \widehat{L})(d_{l+1}, \ldots, d_n) \sqsubseteq$$
$$u(\mathbf{f})(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u}_{k+1})(w), \ldots, [\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u}_{k+1})(w), d_{l+1}, \ldots, d_n)$$

- For every definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}_{k+1}(\mathbf{f})(w)(d_1, \ldots, d_n) \sqsubseteq u(\mathbf{f})(w)(d_1, \ldots, d_n)$$

Recall now that $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f}$ in $\mathbf{P}_m$ and also $\mathbf{f}(\mathbf{x}_{l+1}, \ldots, \mathbf{x}_n) \doteq \mathcal{E}_m(\mathbf{B}_\mathbf{f})$ in $\mathbf{P}_{m-1}$. The idea is to use Definition 4.10 and Theorem 4.1 in order to get equivalent statements that involve the body of the function $\mathbf{f}$. Therefore, it suffices to show that:

- For every definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1), \ldots, order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$[\![\mathcal{E}_m(\mathbf{B}_\mathbf{f})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \sqsubseteq [\![\mathbf{B}_\mathbf{f}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)$$

where $\sigma(\mathbf{x}_j) = d_j^\infty$, $l+1 \leq j \leq n$, and $\widehat{\rho}_{k+1}(\mathbf{x}_j) = ([\![\mathcal{E}_m(\mathbf{E}_j)]\!]^*(\widehat{u}_{k+1})(w))^\infty$, $1 \leq j \leq l$.

- For every definition $(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1, \ldots, \mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1), \ldots, order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!], \ldots, d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$[\![\mathcal{E}_m(\mathbf{B}_\mathbf{f})]\!]^*(\widehat{u}_k \oplus \sigma)(w) \sqsubseteq [\![\mathbf{B}_\mathbf{f}]\!]^*(u \oplus \sigma)(w)$$

where $\sigma(\mathbf{x}_j) = d_j^\infty$, $1 \leq j \leq n$.

In the following, we give a full proof for the first of the above statements. The proof for the second statement is simpler, and can be given in a similar way. 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 the following two cases:

1. The function $\mathbf{f}$ has an *ordinary* definition in program $\mathbf{P}_m$.
2. The function $\mathbf{f}$ has an *actuals* definition in program $\mathbf{P}_m$.

We prove each case separately, for the reasons we described in the introductory sections of this chapter.

Proof for Ordinary Definitions

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

$$[\![\mathcal{E}_m(\mathbf{S})]\!]^*(\hat{u}_k \oplus \sigma)(w \Downarrow \hat{L}) \sqsubseteq [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \hat{\rho}_{k+1})(w \Downarrow L)$$

Structural Induction Basis:

Case 1: $\mathbf{S}$ is equal to a variable $\mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, and $\mathbf{x}_j$ is $(m-1)$-order. Then, a definition of the form $\mathbf{x}_j(\mathbf{z}_1, \ldots, \mathbf{z}_t) \doteq (\mathbf{actuals}_{m,R}\langle\mathcal{E}_m(i \odot j)\rangle_{i \in dom(R)})(\mathbf{z}_1, \ldots, \mathbf{z}_t)$ is created in $\mathbf{P}_{m-1}$, where $R = \mathit{options}(\mathbf{f}, \mathbf{P}_m, m)$. Using the semantics of application and Lemma 4.1, it can easily be shown that

$$\hat{u}_k(\mathbf{x}_j) \sqsubseteq [\![\mathbf{actuals}_{m,R}\langle\mathcal{E}_m(i \odot j)\rangle_{i \in dom(R)}]\!]^*(\hat{u}_k)$$

This fact is used in the proof given below. The following facts are also used: given the context $(w \Downarrow \hat{L})$, $\mathit{head}((w \Downarrow \hat{L})_m) = \lceil \mathbf{E} \rceil$ because $\hat{L} = L \cup \{\langle m, \lceil \mathbf{E} \rceil \rangle\}$. Moreover, it can be easily shown from the definition of the set $R = \mathit{options}(\mathbf{f}, \mathbf{P}_m, m)$ that $R_{\lceil \mathbf{E} \rceil} = \hat{L}$. The left

hand side of the statement we want to establish can be written as:

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \;& [\![\mathcal{E}_m(\mathbf{x}_j)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \;& [\![\mathbf{x}_j]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of the transformation algorithm)} \\
= \;& \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{L}) \\
& \text{(Variable } \mathbf{x}_j \text{ is } (m-1)\text{-order)} \\
\sqsubseteq \;& [\![\mathbf{actuals}_{m,R}\langle \mathcal{E}_m(i \odot j)\rangle_{i \in dom(R)}]\!]^*(\widehat{u}_k)(w \Downarrow \widehat{L}) \\
& \text{(Replacing } \mathbf{x}_j \text{ by its defining expression)} \\
= \;& [\![\mathcal{E}_m(\lceil \mathbf{E} \rceil \odot j)]\!]^*(\widehat{u}_k)((w \Downarrow \widehat{L}) \Uparrow R_{\lceil \mathbf{E} \rceil}) \\
& \text{(Semantics of } \mathbf{actuals}\text{)} \\
= \;& [\![\mathcal{E}_m(\mathbf{E}_j)]\!]^*(\widehat{u}_k)((w \Downarrow \widehat{L}) \Uparrow \widehat{L}) \\
& \text{(Definition of } \odot \text{ and } R_{\lceil \mathbf{E} \rceil} = \widehat{L}\text{)} \\
= \;& [\![\mathcal{E}_m(\mathbf{E}_j)]\!]^*(\widehat{u}_k)(w) \\
& \text{(Because } (w \Downarrow \widehat{L}) \Uparrow \widehat{L} = w\text{)} \\
\sqsubseteq \;& [\![\mathcal{E}_m(\mathbf{E}_j)]\!]^*(\widehat{u}_{k+1})(w) \\
& \text{(Monotonicity of } [\![\mathcal{E}_m(\mathbf{E}_j)]\!]^* \text{)} \\
= \;& \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow L) \\
& \text{(Definition of } \widehat{\rho}_{k+1}\text{)} \\
= \;& [\![\mathbf{x}_j]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Variable } \mathbf{x}_j \text{ gets a value from } \widehat{\rho}_{k+1}\text{)} \\
= \;& [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)}
\end{aligned}
$$

Case 2: **S** is equal to a variable $\mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, 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:

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{x}_j)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \; & [\![\mathbf{x}_j]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & \sigma(\mathbf{x}_j)(w \Downarrow \widehat{L}) \\
& \text{(Because } \mathbf{x}_j \text{ is less than } (m-1)\text{-order)} \\
= \; & \sigma(\mathbf{x}_j)(w \Downarrow L) \\
& \text{(Because } \sigma(\mathbf{x}_j) = d_j^\infty\text{)} \\
= \; & [\![\mathbf{x}_j]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Because } \mathbf{x}_j \text{ is less than } (m-1)\text{-order)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)}
\end{aligned}
$$

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

Case 4: **S** is equal to $\mathbf{call}_K(\mathbf{h})$, where **h** is not a formal of **f**. In this case, the order of **h** is strictly less than m (because m-order only have full applications in $\mathbf{P}_m$). Recall now that the outer induction hypothesis for functions of order less than m, specifies that:

$$\widehat{u}_k(\mathbf{h}) \sqsubseteq u(\mathbf{h})$$

The left hand side of the statement we want to establish can then be written as follows:

$$\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{call}_K(\mathbf{h}))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Because } \mathbf{S} = \mathbf{x}_j\text{)} \\
= \; & [\![\mathbf{call}_K(\mathbf{h})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & \mathit{call}_K(\widehat{u}_k(\mathbf{h}))(w \Downarrow \widehat{L}) \\
& \text{(Semantics of } \mathbf{call}\text{)} \\
\sqsubseteq \; & \mathit{call}_K(u(\mathbf{h}))(w \Downarrow \widehat{L}) \\
& \text{(Outer induction hypothesis)} \\
= \; & \mathit{call}_K(u(\mathbf{h}))(w \Downarrow L) \\
& \text{(Because of Theorem 6.1)} \\
= \; & [\![\mathbf{call}_K(\mathbf{h})]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of } \mathbf{call}\text{)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Because } \mathbf{S} = \mathbf{call}_K(\mathbf{h})\text{)}
\end{aligned}$$

Structural Induction Step.

Case 1: $\mathbf{S} = \mathbf{x}_j(\mathbf{S}_1, \ldots, \mathbf{S}_r)$ where $\mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, and $\mathbf{x}_j$ is $(m-1)$-order. The proof uses the following fact which was shown in the induction basis:

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

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 has as follows:

$$\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{x}_j(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Assumption for } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{x}_j(\mathcal{E}_m(\mathbf{S}_1), \ldots, \mathcal{E}_m(\mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Semantics of application)} \\
\sqsubseteq \; & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Because } \widehat{u}_k(\mathbf{x}_j)(w \Downarrow \widehat{L}) \sqsubseteq \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow L)\text{)} \\
\sqsubseteq \; & \widehat{\rho}_{k+1}(\mathbf{x}_j)(w \Downarrow L)([\![\mathbf{S}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L), \ldots, [\![\mathbf{S}_r]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)) \\
& \text{(Using structural induction hypothesis and monotonicity)} \\
= \; & [\![\mathbf{x}_j(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}$$

Case 2: $\mathbf{S} = \mathbf{x}_j(\mathbf{S}_1, \ldots, \mathbf{S}_r)$ where $\mathbf{x}_j \in \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, 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. Then:

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \;& [\![\mathbf{x}_j(\mathcal{E}_m(\mathbf{S}_1), \ldots, \mathcal{E}_m(\mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \;& \sigma(\mathbf{x}_j)(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Semantics of application)} \\
= \;& \sigma(\mathbf{x}_j)(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Because } \sigma(\mathbf{x}_j) \text{ is a constant intension)} \\
\sqsubseteq \;& \sigma(\mathbf{x}_j)(w \Downarrow L))([\![\mathbf{S}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L), \ldots, [\![\mathbf{S}_r]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)) \\
& \text{(Using the induction hypothesis and monotonicity)} \\
= \;& [\![\mathbf{x}_j(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of application)} \\
= \;& [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}
$$

Case 3: $\mathbf{S} = \mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)$. The proof in this case uses the fact that the denotation $\mathcal{C}^*(\mathbf{c})$ is a constant intension. The proof has as follows:

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Assumption for } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{c}(\mathcal{E}_m(\mathbf{S}_1), \ldots, \mathcal{E}_m(\mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & \mathcal{C}^*(\mathbf{c})(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Semantics of constant symbols)} \\
= \; & \mathcal{C}^*(\mathbf{c})(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Because } \mathcal{C}^*(\mathbf{c}) \text{ is a constant intension)} \\
\sqsubseteq \; & \mathcal{C}^*(\mathbf{c})(w \Downarrow L)([\![\mathbf{S}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L), \ldots, [\![\mathbf{S}_r]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)) \\
& \text{(Using structural induction hypothesis and monotonicity)} \\
= \; & [\![\mathbf{c}(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of constant symbols)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}
$$

Case 4: $\mathbf{S} = \mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r)$ where $\mathbf{g}$ is an m-order function defined in $\mathbf{P}_m$. Assume that the corresponding definition for $\mathbf{g}$ in $\mathbf{P}_m$ is $\mathbf{g}(\mathbf{y}_1, \ldots, \mathbf{y}_r) \doteq \mathbf{B_g}$ and that $\mathbf{y}_1, \ldots, \mathbf{y}_t$, $0 \leq t \leq r$ are the $(m-1)$-order parameters of $\mathbf{g}$. Then, in $\mathbf{P}_{m-1}$, the corresponding call is $\mathbf{call}_{\widehat{L}_1}(\mathbf{g})(\mathcal{E}_m(\mathbf{S}_{t+1}), \ldots, \mathcal{E}_m(\mathbf{S}_r))$, where $\widehat{L}_1 = L_1 \cup \{\langle m, \lceil \mathbf{S} \rceil \rangle\}$. Also, the definition for $\mathbf{g}$ in $\mathbf{P}_{m-1}$ becomes $\mathbf{g}(\mathbf{y}_{t+1}, \ldots, \mathbf{y}_r) \doteq \mathcal{E}_m(\mathbf{B_g})$.

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Assumption for S)} \\
= \; & [\![\mathbf{call}_{\widehat{L}_1}(\mathbf{g})(\mathcal{E}_m(\mathbf{S}_{t+1}), \ldots, \mathcal{E}_m(\mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & call_{\widehat{L}_1}(\widehat{u}_k(\mathbf{g}))(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_{t+1})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Semantics of call)} \\
= \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Outer induction hypothesis)} \\
= \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Because of Theorem 6.1)} \\
\sqsubseteq \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow L)([\![\mathbf{S}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L), \ldots, [\![\mathbf{S}_n]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)) \\
& \text{(Structural induction hypothesis and monotonicity)} \\
= \; & [\![\mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Assumption for S)}
\end{aligned}
$$

Case 5: $\mathbf{S} = \mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r)$ where $\mathbf{g}$ is a function defined in $\mathbf{P}_m$, of order less than m. Recall that by the outer induction hypothesis it is:

$$\widehat{u}_k(\mathbf{g}) \sqsubseteq u(\mathbf{g})$$

Then, the left hand side of the statement we want to establish can be written as follows:

$$\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) = \\
= \; & [\![\mathcal{E}_m(\mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Assumption for } \mathbf{S}\text{)} \\
= \; & [\![\mathbf{call}_{L_1}(\mathbf{g})(\mathcal{E}_m(\mathbf{S}_1), \ldots, \mathcal{E}_m(\mathbf{S}_r))]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & call_{L_1}(\widehat{u}_k(\mathbf{g}))(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Semantics of } \mathbf{call}\text{)} \\
\sqsubseteq \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow \widehat{L})([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Outer induction hypothesis)} \\
= \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow L)([\![\mathcal{E}_m(\mathbf{S}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}), \ldots, [\![\mathcal{E}_m(\mathbf{S}_r)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L})) \\
& \text{(Because of Theorem 6.1)} \\
\sqsubseteq \; & call_{L_1}(u(\mathbf{g}))(w \Downarrow L)([\![\mathbf{S}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L), \ldots, [\![\mathbf{S}_n]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)) \\
& \text{(Structural induction hypothesis and monotonicity)} \\
= \; & [\![\mathbf{call}_{L_1}(\mathbf{g})(\mathbf{S}_1, \ldots, \mathbf{S}_r)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}$$

This completes the proof of the theorem for the case of *ordinary* definitions.

Proof for Actuals Definitions

In this case, the definition for the function $\mathbf{f}$ in program $\mathbf{P}_m$ is of the form $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B_f}$ where $\mathbf{B_f} = \mathbf{actuals}_{m',R}(\langle \mathbf{S}_i \rangle_{i \in dom(R)})$. Notice that $m' > m$, because this definition has been created in a previous step of the transformation algorithm. We want to establish the following result:

$$[\![\mathcal{E}_m(\mathbf{B_f})]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L}) \sqsubseteq [\![\mathbf{B_f}]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L)$$

Notice now that the two contexts $(w \Downarrow \widehat{L})$ and $(w \Downarrow L)$ agree on the m' dimension, because their only difference is in the m dimension and $m < m'$. Therefore, using the semantics of **actuals**, it suffices to show that for every $i \in dom(R)$:

$$[\![\mathcal{E}_m(\mathbf{S}_i)]\!]^*(\widehat{u}_k \oplus \sigma)(w \Downarrow \widehat{L} \Uparrow R_i) \sqsubseteq [\![\mathbf{S}_i]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w \Downarrow L \Uparrow R_i)$$

By Lemma 5.1, we know that there exists a function $\mathbf{g}$ and $K \subseteq N \times N$ such that $\mathbf{S}_i = \mathbf{call}_K(\mathbf{g})(\mathbf{Q}_1,\ldots,\mathbf{Q}_n)$, where $\mathbf{Q}_j = \mathbf{call}_{R_i}(\mathbf{x}_j)$, $j = 1,\ldots,n$. In the translated program $\mathbf{P}_{m-1}$ the corresponding call is of the form $\mathcal{E}_m(\mathbf{S}_i) = \mathbf{call}_{\widehat{K}}(\mathbf{g})(\mathcal{E}_m(\mathbf{Q}_{l+1}),\ldots,\mathcal{E}_m(\mathbf{Q}_n))$, where $\widehat{K} = K \cup \{\langle m, \lceil \mathbf{S}_i \rceil \rangle\}$. For brevity, we let $w_1 = w \Downarrow \widehat{L} \Uparrow R_i$ and $w_2 = w \Downarrow L \Uparrow R_i$. It can be shown that:

$$[\![\mathcal{E}_m(\mathbf{Q}_j)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1) \sqsubseteq [\![\mathbf{Q}_j]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w_2) \tag{6.1}$$

This follows easily, using the definitions of σ and $\widehat{\rho}_{k+1}$. Consider now the left hand side of the statement to be established. This can be written as follows:

$$
\begin{aligned}
& [\![\mathcal{E}_m(\mathbf{S}_i)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1) = \\
= \; & [\![\mathcal{E}_m(\mathbf{call}_K(\mathbf{g})(\mathbf{Q}_1, \ldots, \mathbf{Q}_n))]\!]^*(\widehat{u}_k \oplus \sigma)(w_1) \\
& \text{(Assumption for } \mathbf{S}_i\text{)} \\
= \; & [\![\mathbf{call}_{\widehat{K}}(\mathbf{g})(\mathcal{E}_m(\mathbf{Q}_{l+1}), \ldots, \mathcal{E}_m(\mathbf{Q}_n))]\!]^*(\widehat{u}_k \oplus \sigma)(w_1) \\
& \text{(Definition of } \mathcal{E}_m\text{)} \\
= \; & \mathit{call}_{\widehat{K}}(\widehat{u}_k(\mathbf{g}))(w_1)([\![\mathcal{E}_m(\mathbf{Q}_{l+1})]\!]^*(\widehat{u}_k \oplus \sigma)(w_1), \ldots, [\![\mathcal{E}_m(\mathbf{Q}_n)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1)) \\
& \text{(Semantics of } \mathbf{call}\text{)} \\
= \; & \mathit{call}_K(u(\mathbf{g}))(w_1)([\![\mathcal{E}_m(\mathbf{Q}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1), \ldots, [\![\mathcal{E}_m(\mathbf{Q}_n)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1)) \\
& \text{(Outer induction hypothesis)} \\
= \; & \mathit{call}_K(u(\mathbf{g}))(w_2)([\![\mathcal{E}_m(\mathbf{Q}_1)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1), \ldots, [\![\mathcal{E}_m(\mathbf{Q}_n)]\!]^*(\widehat{u}_k \oplus \sigma)(w_1)) \\
& \text{(Because of Theorem 6.1)} \\
\sqsubseteq \; & \mathit{call}_K(u(\mathbf{g}))(w_2)([\![\mathbf{Q}_1]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w_2), \ldots, [\![\mathbf{Q}_n]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w_2)) \\
& \text{(Using equation 6.1 and monotonicity)} \\
= \; & [\![\mathbf{call}_K(\mathbf{g})(\mathbf{Q}_1, \ldots, \mathbf{Q}_n)]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w_2) \\
& \text{(Semantics of application)} \\
= \; & [\![\mathbf{S}_i]\!]^*(u \oplus \sigma \oplus \widehat{\rho}_{k+1})(w_2) \\
& \text{(Assumption for } \mathbf{S}\text{)}
\end{aligned}
$$

This completes the proof for *actuals* definition and the proof for the whole theorem. ■

Theorem 6.3 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 $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1),\ldots,order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1),\ldots,order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$call_{L\cup\{(m,\lceil\mathbf{E}\rceil)\}}(\widehat{u}(\mathbf{f}))(w)(d_{l+1},\ldots,d_n) \sqsupseteq$$
$$call_L(u(\mathbf{f}))(w)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u})(w),\ldots,[\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u})(w),d_{l+1},\ldots,d_n)$$

- For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1),\ldots,\ldots,order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$\widehat{u}(\mathbf{f})(w)(d_1,\ldots,d_n) \sqsupseteq u(\mathbf{f})(w)(d_1,\ldots,d_n)$$

Proof: Following the same ideas as the proof for Theorem 6.2. ∎

Theorem 6.4 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:

- For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and there exists $1 \leq l \leq n$ such that $order(\tau_1) = (m-1),\ldots,order(\tau_l) = (m-1)$ and $order(\tau_{l+1}) < (m-1),\ldots,order(\tau_n) < (m-1)$, then for every call $\mathbf{E} = \mathbf{call}_L(\mathbf{f})(\mathbf{E}_1,\ldots,\mathbf{E}_n)$ to $\mathbf{f}$ in $\mathbf{P}_m$, for all $d_{l+1} \in [\![\tau_{l+1}]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all $w \in W$,

$$call_{L\cup\{(m,\lceil\mathbf{E}\rceil)\}}(\widehat{u}(\mathbf{f}))(w)(d_{l+1},\ldots,d_n) =$$
$$call_L(u(\mathbf{f}))(w)([\![\mathcal{E}_m(\mathbf{E}_1)]\!]^*(\widehat{u})(w),\ldots,[\![\mathcal{E}_m(\mathbf{E}_l)]\!]^*(\widehat{u})(w),d_{l+1},\ldots,d_n)$$

- For every definition $(\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) \doteq \mathbf{B}_\mathbf{f})$ in $\mathbf{P}_m$, if $\mathbf{x}_1 : \tau_1,\ldots,\mathbf{x}_n : \tau_n$ and $order(\tau_1) < (m-1),\ldots,\ldots,order(\tau_n) < (m-1)$, then for all $d_1 \in [\![\tau_1]\!],\ldots,d_n \in [\![\tau_n]\!]$ and for all

$w \in W$,

$$\widehat{u}(\mathbf{f})(w)(d_1, \ldots, d_n) = u(\mathbf{f})(w)(d_1, \ldots, d_n)$$

Proof: A direct consequence of Theorems 6.2 and 6.3. ■

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

Theorem 6.5 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 6.4 on the variable **result** of the programs $\mathbf{P}_m$ and $\mathbf{P}_{m-1}$. ■

It remains to show that the initial extensional 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 6.6 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 $u^*_M, \ldots, u^*_0$ 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 $w \in W$:

$$
\begin{aligned}
& [\![\mathbf{P}_M]\!]_D(u_M) = \\
= \; & [\![\mathbf{P}_M]\!]^*_D(u^*_M)(w) \\
& \text{(Theorem 4.3)} \\
= \; & [\![\mathbf{P}_{M-1}]\!]^*_D(u^*_{M-1})(w) \\
& \text{(Theorem 6.5)} \\
& \ldots \\
= \; & [\![\mathbf{P}_0]\!]^*_D(u^*_0)(w) \\
& \text{(Theorem 6.5)} \\
= \; & [\![\mathbf{P}_0]\!]^*_{(W \to D)}(u_0)(w) \\
& \text{(Theorem 4.4)}
\end{aligned}
$$

■

6.5 Discussion

The correctness proof given in the previous section, concludes the formal presentation of the transformation algorithm from higher-order extensional 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. Notice that the transformation for higher-order programs is much more sophisticated than the one for the first-order case, and it is imperative that the informal intuitions one may have, be supported by formal reasoning.

The most crucial change that the proof suggested was the change in the **actuals** operator. One can easily realize by closely examining Case 1 of the structural induction basis, that the **actuals** operator suggested in [Wad91] would not work. The problem would be that the operator would not perform all the necessary "cleaning" of the context that is

required, leaving in this way unnecessary information "floating around". From our experience, this resulted in unacceptable performance in the implementation of many programs. It should be noted that these performance problems could not be intuitively explained by the author, before the proof was undertaken.

Another important aspect of the algorithm that is reflected in the proof is the fact that M different dimensions are used in a controlled way, in the transformation of an M-order program. Theorem 6.1 indicates that a program that appears in an intermediate stage of the transformation, only depends on those dimensions that have appeared until now in the translation. This theorem is used in many different parts of the proof. However, we should note at this point that it is not at all obvious that *exactly* M (and not less) dimensions are necessary (although such a choice is a mathematically appealing one as it associates one dimension for every order of the program).

Chapter 7

Implementation Strategies

As it was discussed in Chapter 1, the conceptual basis for implementing intensional languages is the eduction model of execution. However, each intensional language has its own idiosyncrasies, and the concrete way in which eduction can be implemented for a specific language, is not always straightforward. In this chapter we examine how the zero-order subset of the *IL* language can be educed, providing in this way an implementation technique for the source functional language *FL*. More specifically, we propose two strategies for implementing eduction on von Neumann machines. The first approach heavily relies on hashing techniques [Knu75], while the second one is an enhancement of the traditional activation-record ideas [ASU86] to include context information. Both of the strategies we propose, focus on resolving the following two issues:

1. **Efficient context operations.** Recall that the contexts are tuples of lists of natural numbers. During computation, the lists can get arbitrarily long, especially if the source functional program contains recursively defined functions. Therefore, it is imperative that lists be represented in such a way so as that they can be efficiently manipulated.

2. **Avoidance of recomputations**. The value of a variable under a specific context may be demanded many times during program execution. A scheme that avoids duplication of computational effort is necessary.

The last section of the chapter investigates the relationship between eduction and the dataflow model of computation. More specifically, we demonstrate that the technique developed in this dissertation offers a solution to the implementation of higher-order functions on dataflow architectures.

7.1 A Hashing-Based Implementation

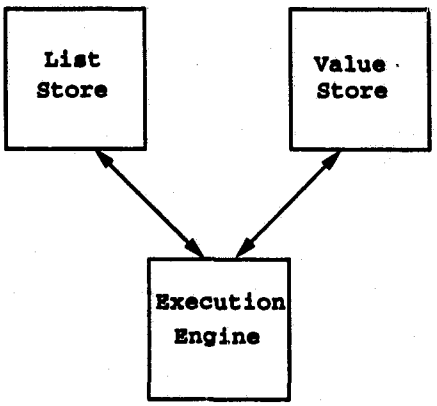
Traditionally, the Lucid language and all the Lucid-related systems, have been implemented based on a technique known as *eduction* [WA85, DW90b, DW90a]. However, until now, only first-order languages have been implemented based on the eduction model. In this section we describe an enchancement of the eduction engine that can handle the implementation of higher-order functional programs.

The architecture consists of three interacting components (Figure 7.1): the *List Store*, the *Value Store*, and the *Execution Engine*. In the following we describe in detail the purpose of each component.

7.1.1 The List Store

The purpose of this component is to ensure a compact representation of contexts as well as efficient operations on them. Recall that for an m-order functional program, a context w is an m-tuple $\langle w_1, \ldots, w_m \rangle$, where $w_1, \ldots, w_m$ are lists of natural numbers. The solution we adopt is to convert each list into a natural number, using a technique known in computer science as "hash-consing" (described below). In this way, contexts appear as m-tuples $\langle n_1, \ldots, n_m \rangle$, where $n_1, \ldots, n_m \in N$, and their manipulation is much more convenient.

figure 7.1 Architecture of the implementation



The List Store component in Figure 7.1, takes care of the details of hash-consing and also supports functions that can be used to implement the **call** and **actuals** operators.

The idea of hash-consing is to store a list in a hash table [Knu75], as a pair *(head, position of tail)*. The list is then represented by the position of the tuple in the table (Figure 7.2).

figure 7.2 The hash-consing technique

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

The following primitive functions are supported by the List Store:

- *hashcons(head, tail_code)* : Uses a hash function to check if the pair (*head, tail_code*)

already exists in the hash table. If it does not, then it inserts it. Finally, it returns the position of the pair in the table.

- *hashhead(list_code)* : It returns the first element of the pair found in the *list_code* position of the hash table.
- *hashtail(list_code)* : It returns the second element of the pair found in the *list_code* position of the hash table.

The above operations can in general be performed in an efficient way. They are used by the Execution Engine to implement the semantic equations of the **call** and **actuals** operators that appear in the intensional code that results from the transformation. Moreover, the space occupied by the hash table is reasonable. Encoding lists as natural numbers using the above primitives, allows a greater flexibility on context manipulation.

7.1.2 The Value Store

During the execution of a program, many identical computations take place. The technique we propose has an inherent potential for avoiding unnecessary recalculations. More specifically, consider an identifier **x** whose value v has already been computed under a specific context w. The Value Store is a hash table whose purpose is to keep this kind of information. A schematic description is given in Figure 7.3. If the value of the identifier **x** under the same context w is demanded again during program execution, then a lookup of the Value Store can potentially save significant time.

It is important to note that the Value Store is not a *required* component of the architecture. In fact, the only space that is actually essential for the technique, is the table used for implementing hash-consing. However, our experience shows that the use of the Value Store gives an important benefit to the technique, by drastically reducing the number of steps that have to be performed in order to evaluate a program. One problem that the implementation faces with respect to the Value Store, is how to control its size: as new entries

figure 7.3 The Value Store

Idntf	Tag	Value
x	<6,39>	20
y	<10,8>	100

are added, the Value Store gets bigger and bigger, and the insertion and search procedures start to become less efficient. The size of the Value Store can be controlled in two ways:

- **Heuristics:** A particularly successful one is the *retirement age scheme* which has been used in Lucid implementations. It follows similar ideas as the "Least Recently Used" technique for performing page replacement in operating systems.
- Analysis **Based Techniques:** Compile-time analysis of the source program can be used to predict how long a specific entry needs to stay in the Value Store [Bag86].

The efficiency of the above scheme can be significantly improved by performing *dimensionality analysis* at compile-time. The main idea is that the value of an identifier in the target program under a given context, may only depend on a subset of the fields of the context. Therefore, space and time savings can result from determining the dependence between identifiers and context fields. Then, one need only save for each identifier, those components of the context that are absolutely necessary. A promising technique for dimensionality analysis of multidimensional intensional languages is presented in [Du91], and can be adapted for the language *IL* that we propose in this dissertation.

7.1.3 The Execution Engine

The Execution Engine coordinates the actions of the List and Value components of the architecture. Its main purpose is to evaluate the program of nullary intensional definitions that results at the end of the transformation algorithm. This code has the form:

$$\begin{aligned} \mathbf{result} &= \mathbf{B}_0 \\ \mathbf{f}_1 &= \mathbf{B}_1 \\ &\vdots \\ \mathbf{f}_n &= \mathbf{B}_n \end{aligned}$$

In the following, we describe a simple interpretative Execution Engine. Given a nullary variable intensional program **P**, the Execution Engine starts by demanding the value of the variable **result** of **P** under a context consisting of empty lists. To be more accurate, the initial context is of the form $\langle \epsilon, \ldots, \epsilon \rangle$, where ϵ is the encoding under hash-consing of the empty list. The Execution Engine performs different actions, depending on the kind of expressions that it evaluates. We present each case separately:

Variables: When a variable **f** is encountered during the execution of the program, the following actions are performed. The variable is first looked up in the Value Store under the current context w. If it is found, then the corresponding value is returned. Otherwise, the body **B** of **f** is evaluated in order to compute the value of **f** in the current context w. The result of the evaluation is then inserted in the Value Store. It should be noted that compile time analysis of the program may indicate that some identifiers should never be stored in the Value Store because they are never going to be demanded. Such an optimization can help reduce the space occupied by the Value Store as well as the search and insert access times.

Call Operator: When the expression $\mathbf{call}_L(\mathbf{E})$ is encountered during execution, the evaluator performs the following. The current context w is transformed according to the semantic

definition of the **call** operator. Then, the expression **E** is evaluated under the new context. We should note at this point that the transformation that is performed on the current context, is a multi-consing operation as dictated by the subscript L of the **call** operator. This operation is implemented using the *hashcons* primitive provided by the List Store.

Actuals Operator: To evaluate $\mathbf{actuals}_{m,R}(E)$ the following actions are required. The m-th element of the current context w is selected, and its head is extracted and used to select the corresponding argument of **actuals**. The context w is transformed according to the semantic definition of the **actuals** operator. The argument that was selected is then evaluated under the new context. The above actions are implemented using the *hashhead* and *hashtail* primitives of the List Store.

Constants: To evaluate $\mathbf{c}(\mathbf{E}_1, \ldots, \mathbf{E}_n)$ under a context w, the expressions $\mathbf{E}_1, \ldots, \mathbf{E}_n$ are evaluated under w, and the meaning of **c** is applied to the results. However, we should emphasize here that is it is not always necessary (or desirable) to evaluate all the arguments of the constant **c** at once. For example, the **if-then-else** operation first evaluates its first argument, and according to the result produced, either evaluates its second or its third argument.

7.2 An Activation-Record Based Implementation

In this section, we present an alternative implementation strategy which attempts to reduce the overhead associated with the hashing-based approach. The main idea of the technique is that the List Store and the Value Store can be appropriately merged into a single structure. This is achieved by incorporating context-related information into traditional activation records.

7.2.1 Incorporating Contexts into Activation Records

To illustrate the proposed technique, consider the program given in section 5.6. During execution of the program (Figure 5.7), the formal parameter `x` of `apply` is demanded at some point under the context $\langle [l_2], [l_1] \rangle$. This demand for `x` is directly related to the demand for `apply` under the same context $\langle [l_2], [l_1] \rangle$. Similarly, notice the demands for both `z` and `f` under the context $\langle [l_3, l_2], [l_1] \rangle$. In general, whenever a zero-order formal parameter in the program that results before the last step in the transformation (like `x` and `z` above) is demanded under a context, there has existed a previous demand, under the same context, for the function variable (like `apply` and `f` above), in which the formal belongs.

Notice that the above discussion only applies to the zero-order formals of functions. As we will see, this makes the technique introduced in this section less general than the hashing-based approach. However, the hashing overhead is avoided, ensuring in this way a faster execution for many programs.

During the last step of the transformation algorithm (that is, before the translation from first-order to zero-order), the number of zero-order formal parameters of each function in the program is recorded. This information will be used to determine the size of the activation record that has to be allocated when a call to this function occurs. Moreover, for every formal parameter, an offset from the beginning of the activation record is assigned.

As we have seen, given an m-order extensional program, there are m dimensions used for its translation. Therefore, the contexts that will be used for the execution of the intensional program that results from the translation, will be m-tuples of lists of natural numbers. During execution, each component of the current context is represented as a tuple *(head, pointer to tail)*, and it is stored in the beginning of the current activation record (this will be further explained below). Therefore, a component of the context can be uniquely characterized by the address in the stack of the corresponding tuple. In other words, the current context can be characterized by m pointers (addresses) in the stack. We will call these addresses *context pointers*, denote them by $cp_1, \ldots, cp_m$, and assume that they are

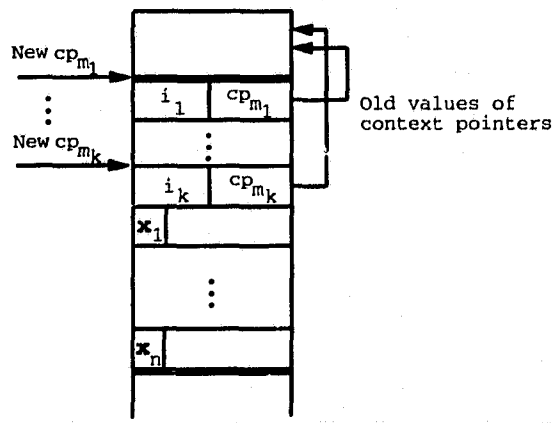
stored in corresponding registers. Just before the execution of the program starts, all pointers are set to point to the beginning of the stack.

7.2.2 Execution of Intensional Code

In the following, we describe the execution of intensional programs of nullary variables, using a stack of activation records. As before, different actions need to be performed depending on the kind of expression that is under evaluation. We consider each case separately:

Call Operator: Consider the first-order program that results just before the last step of the transformation algorithm is performed. Let $\mathbf{f}$ be a function in this program and let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be its formal parameters (which are all zero-order). Whenever during execution of the resulting zero-order program, an expression of the form $\mathbf{call}_L(\mathbf{f})$ is encountered, where $L = \{\langle m_1, i_1\rangle, \ldots, \langle m_k, i_k\rangle\}$, a new activation record is allocated in the stack (Figure 7.4). The new context pointers are set as shown in this figure.

figure 7.4 Activation Record with Context Information



Actuals Operator: Whenever an expression of the form $\mathbf{actuals}_{m,R}(E)$ is encountered during execution, the following actions are performed: the context pointer cp_m is considered, and the position in the stack that it is pointing to, is located. This position holds a tuple

that contains the head and a pointer to the tail of the m-th context component. The head is used to select the appropriate expression from the sequence E. Moreover, the set R is used in order to perform the appropriate multiple tail operation on the corresponding context pointers (that is, the pointers will be set to their previous value).

Variables: When a variable **x** is demanded during execution, the following actions are performed: if **x** is not one of the formals that was removed in the last step of the transformation, then execution continues with the body of the definition for **x**. Otherwise, the context pointer that corresponds to the last step in the transformation (i.e. cp_1), is used in order to find the activation record that corresponds to the current context. The field of the activation record that is reserved for **x**, can be located by using the offset of **x** that was recorded during compilation. If the corresponding entry is full, then the stored value is retrieved and used. Otherwise, the definition for **x** is used in order to calculate the desired value, which is then placed in its position in the activation record.

Constants: The evaluation of expressions involving constants is performed in the usual way.

The activation records are deallocated as usual, when control is returning after the evaluation of a $\mathbf{call}_L(\mathbf{f})$ expression.

The activation-record-based technique is more restricted than the hashing-based one, for two main reasons:

- It does not store all *(identifier,context,value)* triples that appear during execution. It only considers those identifiers that are eliminated during the last step of the transformation.
- There does not seem to be any straightforward way to generalize it for arbitrary intensional languages.

However, the activation-record-based approach can prove more efficient in certain cases (such as for example, first-order functional languages), because it completely avoids the hashing-related overheads.

7.3 Preliminary Implementation Results

In this section we present some preliminary implementation results regarding the activation-record based approach. The material in this section should not be taken as conclusive performance benchmarks, but instead as preliminary experimental results regarding the ideas developed in this dissertation. In the following, we present a comparison of our technique with other implementations of functional languages. Notice that all the implementations considered below are *lazy*, which means that arguments to functions are not evaluated until they are absolutely necessary [Jon87].

Following a recent trend in functional language implementation [TLA92], our implementation uses as its target machine code the C language. Such an approach ensures portability and allows several optimizations to be performed by the available C compiler. However, some performance penalty is paid for not generating native machine code directly. We consider the following lazy functional language implementations:

- **Lazy ML compiler [AJ92]**: The LML system is based on the G-machine [Aug84], [Joh84], and it is one of the fastest compilers of lazy functional languages available. It produces native code and uses a set of sophisticated optimizations.
- **Gofer [Jon91]**: It is based on supercombinators [FH88]. As it produces C code as the target code, it is a good candidate for comparison with our implementation.
- **Miranda[1] [Tur85]**: It is based on combinator graph reduction [Tur79]. Although clearly slower than some of the recently developed compilers, it is one of the most

[1] Miranda is a trademark of Research Software Ltd.

robust interpreters for lazy functional languages.

The programs used are mostly standard benchmarks for functional languages. Their main characteristic is the high number of function calls, as this is the main comparison criterion.

- **Fib(27):** Recursively computes the 27th Fibonacci number.
- **Tak(22,12,6):** The "Takeushi" benchmark.
- **Ack(6,3):** It computes the very rapidly increasing Ackermann's function.
- **Mersenne:** A program with first and second order functions. It examines a famous conjecture of number theory, which concerns the primality of a certain family of numbers [New56]. The conjecture was negatively settled in 1903.
- **Integration:** A third-order program. It uses numerical approximation methods in order to compute the area below a given curve.

The benchmarks were run on a Sun SPARCserver 690MP, with 64MB RAM, and the timings were obtained using the UNIX[2] *time* utility. Both user and system times were added. For both our system and Gofer, the same optimization settings were used during compilation. The *gcc* compiler was used with *-O2* and *-finline-functions.* The second flag allows the inlining of certain simple functions. Such functions usually result from the compilation of definitions that start with the **actuals** operator. This optimization can also be performed to the intensional code, before compiling to C. The following table shows the execution times for the above benchmarks. Entries containing "–" indicate that the corresponding execution of the program terminated abnormally.

[2]UNIX is a registered trademark of AT & T Bell Laboratories

Time (in sec)					
Program	Intensional	LML (not strict)	LML	GOFER	MIRANDA
Fib	3.0	3.3	2.0	14.0	45
Tak	3.7	2.7	2.4	16.4	54
Ack	2.3	1.3	1.0	6.0	16
Mersenne	4.8	–	1.4	15.8	43
Integration	3.6	2.3	2.3	–	19

The results indicate that code generated by our compiler runs at about half the speed of the code produced by the LML compiler. We believe that this is partly due to the fact that LML produces native code and uses many "source to source" transformations in order to produce a program for which generation of efficient code is less complicated. A similar slowdown between native code compilers and compilers that produce C code as output, has been observed by other researchers [TLA92]. For a more fair comparison, we have given the results for LML with and without strictness analysis. The comparison with Gofer and Miranda shows that the code produced by these systems is generally slower. The Gofer system, which is the best candidate for comparison as it also compiles to C code, is usually around three to four times slower. However, it is also important to stress the fact that the language we implemented is much smaller than the ones we compare it with, a fact that gives an advantage to our implementation.

7.4 Relationship with Tagged Dataflow

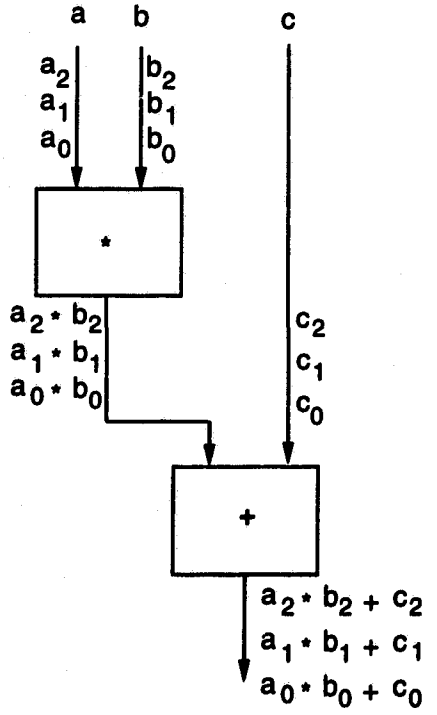
In this section, we argue that the technique developed in this dissertation, provides an elegant solution to a long lasting problem in the area of *dataflow computation.*

The basic principle of the dataflow model of computation, is that data can be processed while they are in *motion*, flowing through a *dataflow network.* A dataflow network is a

system of processing stations (or *nodes*), connected by a number of communication channels (or *arcs*). Each node may have one or more input and output arcs.

There exist two main paradigms of dataflow. In *pipeline* dataflow, data items flow along the arcs of a network, in a first-in first-out way. Therefore, the edges can be thought as queues between the nodes (see Figure 7.5).

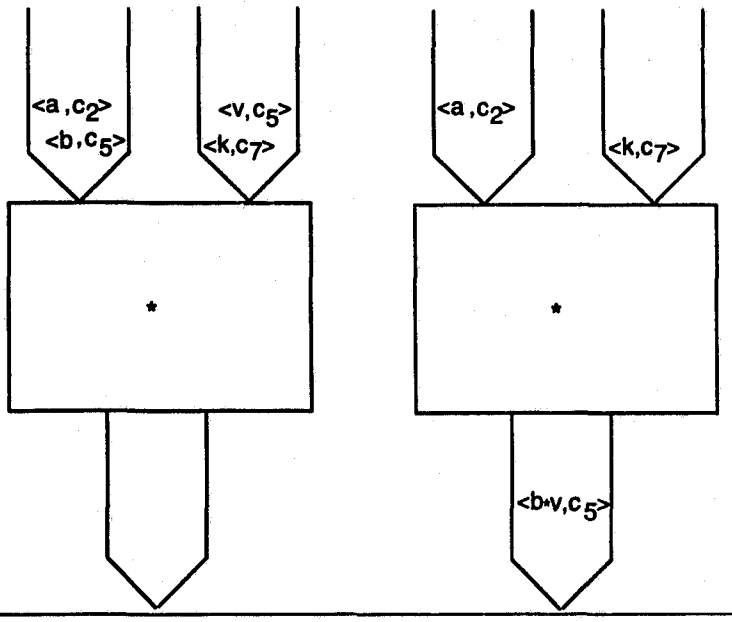
figure 7.5 A pipeline dataflow network.



In *tagged* dataflow, data items are labelled with *tags*, and edges can now be thought as sets of such items. The purpose of the tags is to impose some *conceptual* ordering on the data items. A node can execute if it finds in its input arcs data items that have identical tags. The tagged approach eliminates the need to maintain first-in first-out queues on the arcs, and in this way it offers more parallelism than the pipeline model. In the following we will denote the tagged data that flow along the arcs of a dataflow network as pairs $\langle v, c \rangle$, where c is the tag and v is the actual value. For an illustration of the above ideas, see

Figure 7.6. We use thick arrows as arcs, in order to demonstrate the difference from the pipeline model.

figure 7.6 A tagged dataflow network.



The notion of tag can be used in order to implement first-order functions in a dataflow way. The main idea is that we would like to distinguish between data items that correspond to different function invocations. This can be achieved by letting the tag uniquely characterize a particular invocation of a function. Intuitively, a tag can be thought of as a distinct *colour* [AC87] that is assigned to the data items of a function invocation, so as that we can distinguish them from those data items that belong to other invocations of the same function.

Consider now the technique for first-order programs that we formalized in Chapter 3. The contexts used during evaluation are lists of Gödel numbers of function calls. In other words, a context represents the sequence of function calls that has led to the current call. Therefore, the lists we use correspond to the "colour" idea of the dataflow community. The important contribution of Chapter 3 is that it formalizes through the use of intensional

logic, the "colouring" technique. Consequently, the execution of the intensional code that results from the transformation, is actually *demand-driven tagged dataflow*: demand-driven because the evaluator continuously asks for the value of identifiers under particular contexts, and tagged because the contexts used can be considered as labels that accompany data items during evaluation. Moreover, the **call** and **actuals** operators can be thought as operations that change the context part of the data that flow through the dataflow network.

However, the implementation of higher-order functions on a tagged dataflow framework, has never before been successful. The approach that is usually followed is to implement them using non-dataflow concepts, such as *closures* [AN90]. In other words, tagging is abandoned and data-structures are used for the implementation of closures. Apart from being inelegant, such an approach can prove quite costly in practice as it does not use the tagging capabilities of modern dataflow machines. These facts are recognized in the dataflow literature, and the implementation of higher-order functions is often cited as a problematic issue. The following quotes are relevant:

> "The general **apply** schema [for implementing higher-order functions] is of course not inexpensive" [AN90]
>
> "... [the language] Id has adopted much of the flavor of modern functional languages, including higher-order functions (which, incidentally, are not easily implemented on a dataflow machine)" [Hud91]
>
> "Another problem with the dataflow model is its inefficiency in handling data structures... A number of schemes have been proposed in the literature, but the problem of efficiently representing and manipulating data structures remains a difficult challenge" [LH94]

The transformation algorithm proposed in this dissertation, remedies the above deficiency. The class of higher-order functions we consider, can be implemented using a tagged approach, without resorting to the use of complicated data-structures. The only difference

is that multidimensional tags are now required. In dataflow terminology, a *colour* is not enough anymore to distinguish data items that belong to different function invocations. What is shown in this dissertation is that we need a *palette* of m colours, where m is the order of the program under consideration.

Concluding, we believe that the support of data structures is not a vital part of a dataflow architecture: many things that can be done using data-structures can also be done using the tagging capabilities of the dataflow machine. In this dissertation we have shown that the implementation of higher-order functions does not require the use of closures, but can instead be performed using a more sophisticated tagging scheme. It is our belief that other functional programming features can be handled in a similar purely dataflow way.

Chapter 8

Conclusions and Future Work

The main objective of this dissertation is to propose a semantics preserving transformation from extensional programs to intensional programs of nullary variables. We initially consider the transformation of first-order programs and then extend the technique to a large class of higher-order programs. A crucial point in both cases is the choice of the set of possible worlds, a choice that can be guided by our intuitions concerning the constructs of the source language.

Every major extension to the expressive power of the source language, will probably require a more powerful intensional language. There does not seem to exist any obvious "brute-force" algorithm that would "intensionalize" all language constructs: extensions to the source language have to be treated one-by-one. The main advantage of such an approach is that it gives us further insight on the nature of programming languages and the intuition behind their constructs.

In the rest of this chapter we describe the main contributions of our work and we give pointers to open problems that we consider fruitful for further research.

8.1 Contributions

The main contributions of our work can be summarized as follows:

- We propose a precisely defined transformation algorithm from first-order functional programs, to intensional programs of nullary variables. We give a rigorous correctness proof of the proposed algorithm, establishing in this way, for the first time, a semantics preserving transformation between the two formalisms. In this way we remedy the two main deficiencies of the work introduced in [Yag84].

- We define a transformation algorithm from a class of higher-order functional programs to intensional programs of nullary variables. The algorithm proceeds in steps, reducing at each step the order of the program by introducing appropriate intensional operators. The programs that result in intermediate steps of the transformation, are higher-order intensional ones. We define the synchronic semantics of these programs, which will be used in establishing the correctness proof of the transformation.

- We give a correctness proof of the algorithm for higher-order programs. In this way, we establish a rigorous transformation from a significant class of higher-order functional programs, to intensional programs of nullary variables.

- We show that the transformation algorithm developed in this dissertation, can serve as the basis for the implementation of functional languages. We propose two implementation strategies: one based on hashing and another one which extends traditional activation records to keep context-related information.

In the next section, we discuss certain problems that have not been addressed in this dissertation, and which can be the starting points for future research.

8.2 Future Work

There are certain aspects of our work that need to be further investigated, and which are described below:

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

```
result      ≐ g(8)
g(x)        ≐ twice(add(x),x)
twice(f,y)  ≐ f(f(y))
add(a)(b)   ≐ a+b
```

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 dissertation, 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:

```
result      ≐ g(8)
g(x)        ≐ (call_{⟨⟨2,l1⟩⟩}twice)(x)
twice(y)    ≐ f(f(y))
add(a)(b)   ≐ a+b
f           ≐ act_{2,R}({⟨l1, add(x)⟩})
```

where l_1 and R are obtained as usual. Notice now that the variable **x** appears free in the definition of **f**, while it is bound in the definition of **g**. The program that has resulted, can not 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).

On the other hand, it seems that the set of possible worlds that we have adopted until now, is not sufficient to accommodate programs such as the above. To understand why, we can equivalently rewrite the above source functional program as follows:

```
result      ≐ g(8)
g(x)        ≐ twice(h,x)
              where
                h(z) ≐ add(x)(z)
              end
twice(f,y)  ≐ f(f(y))
add(a)(b)   ≐ a+b
```

In the above program, the only partial application that exists, is a partially applied function name. However, we have now introduced a nested **where**-clause in the program, and our translation scheme applies only to flat programs. As it has been pointed out by Yaghi [Yag84], for first-order programs with nested **where**-clauses we need to extend the set of possible worlds to what he calls "the set of b-lists of natural numbers". Therefore, it is reasonable to conjecture that for general higher-order programs, the context space will be the set of infinite tuples of b-lists of natural numbers.

Concluding this paragraph, we should mention that a desirable but clearly much more difficult attempt would be the intensionalization of the whole untyped λ-calculus.

Efficient and General Implementation of Eduction: This is probably one of the most challenging problems in the area of intensional programming. By generality we mean that the implementation technique should be applicable to all the "reasonable" intensional languages (such as for example Lucid or the language *IL* considered in this dissertation).

The hashing based strategy presented in Chapter 7 is general in the above sense, but its efficiency is not always acceptable. However, hashing plays a very important role when implementing operators that require random access (such as for example the operation **attime** of Lucid). Therefore, a hybrid scheme that would combine the advantages of the stack and hashing based approaches might prove more appropriate.

On the other hand, an efficient implementation of eduction would require the development of powerful optimization techniques for intensional languages. Chapter 6 of [WA85] presents an introduction to this topic (for the Lucid intensional language). We believe that this an interesting area of research that may result in many important results regarding eduction and its efficiency.

Bibliography

[AC87] Arvind and D. Culler. Dataflow Architectures. In S. S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*, pages 79–101. IEEE Computer Society Press, 1987.

[AJ92] L. Augustsson and T. Johnsson. Lazy ML User's Manual. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden, 1992.

[AN90] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[Apt90] K. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 494–574. Elsevier Science Publishers, 1990.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Aug84] L. Augustsson. A compiler for Lazy ML. In *ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[AW76] E. Ashcroft and W. Wadge. Lucid - A Formal System for Writing and Proving Programs. *SIAM J. on Computing*, 5(3):336–354, September 1976.

[AW77] E. Ashcroft and W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[Bag86] R. Bagai. Compilation of the Dataflow Language Lucid. Master's thesis, Department of Computer Science, University of Victoria, 1986.

[Bar77] J. Barwise. An Introduction to First-Order Logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North Holland, 1977.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[Du91] W. Du. *Indexical Parallel Programming.* PhD thesis, Department of Computer Science, University of Victoria, Canada, 1991.

[DW90a] W. Du and W. W. Wadge. The Eductive Implementation of a Three-dimensional Spreadsheet. *Software-Practice and Experience*, 20(11):1097–1114, November 1990.

[DW90b] W. Du and W.W.Wadge. A 3D Spreadsheet Based on Intensional Logic. *IEEE Software*, pages 78–89, July 1990.

[DWP81] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics.* Reidel Publishing Company, 1981.

[EW82] E.A.Ashcroft and W.W.Wadge. Prescription for Semantics. *ACM Transactions on Programming Languages and Systems*, 4(2):283–294, April 1982.

[FH88] A. Field and P. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[FL89] A. A. Faustini and E. Lewis. Toward a Real-Time Dataflow Language. In J. Stankovic and K. Ramamrithan, editors, *Hard Real-Time Systems*, pages 139–145. IEEE, 1989.

[Gal75] D. Gallin. *Intensional and Higher-Order Modal Logic.* North-Holland, 1975.

[Gun92] C. Gunter. *Semantics of Programming Languages.* The MIT Press, 1992.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hud91] P. Hudak. Para-Functional Programming in Haskell. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 159–196. ACM Press, 1991.

[JGW85] C. Kirkham J. Gurd and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, pages 34–52, January 1985.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 58–69, 1984.

[Jon87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Jon91] Mark P. Jones. The Gofer Functional Programming Environment. Technical report, Department of Computer Science, Yale University, 1991.

[JPL93] R. Khedri J. Plaice and R. Lalement. From Abstract Time to Real Time. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 83–93, 1993.

[Knu75] D. E. Knuth. *The Art of Computer Programming (Sorting and Searching)*, volume 3. Addison-Wesley, 1975.

[LH94] B. Lee and A. Hurson. Dataflow Architectures and Multithreading. *IEEE Computer*, pages 27–38, August 1994.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LP81] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[Nel91] G. Nelan. *Firstification.* PhD thesis, Department of Computer Science, Arizona State University, U.S.A, 1991.

[New56] J. Newman. *The World of Mathematics.* Simon and Schuster, 1956.

[Org91] M. A. Orgun. *Intensional Logic Programming.* PhD thesis, Department of Computer Science, University of Victoria, Canada, 1991.

[Org94] M. Orgun. Temporal and Modal Logic Programming. *SIGART Bulletin*, 5(3), July 1994.

[OW92] M. Orgun and W. W. Wadge. Towards a Unified Theory of Intensional Logic Programming. *Journal of Logic Programming*, 13(4), 1992.

[PP94] J. Paquet and J. Plaice. On the Design of an Indexical Query Language. In *Proceedings of the Seventh International Symposium on Lucid and Intensional Programming*, pages 28–36, 1994.

[Rey72] J. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, 1972.

[Rol92] D. Rolston. *Parallel Logic Programming Using an Intensional Model of Computation.* PhD thesis, Department of Computer Science, Arizona State University, U.S.A, 1992.

[Ron92] P. Rondogiannis. Design and Implementation of an Eductive Interpreter for Higher Order Lucid. University of Victoria, Canada, August 1992.

[SE86] L. Sterling and E.Shapiro. *The Art of PROLOG.* MIT Press, 1986.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Sto79] R. Stoll. *Set Theory and Logic.* Dover Publications, 1979.

[Tao93] S. Tao. TLucid and Intensional Attribute Grammars. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 91–106, 1993.

[Tao94] S. Tao. *Indexical Attribute Grammars.* PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994.

[Ten76] R. Tennent. The Denotational Semantics of Programming Languages. *Communications of the ACM*, 19(8), August 1976.

[Ten91] R. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[Tho74] R. Thomason, editor. *Formal Philosophy, Selected Papers of R. Montague.* Yale University Press, 1974.

[TLA92] D. Tarditi, P. Lee, and A. Acharya. No Assembly Required: Compiling Standard ML to C. *ACM LOPLAS*, 1(2):161–177, June 1992.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, 9:31–49, 1979.

[Tur85] D. A. Turner. Miranda: A non-strict language with polymorphic types. In *Proceedings of IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, 1985.

[vB88] J. van Benthem. *A Manual of Intensional Logic.* CSLI Lecture Notes, 1988.

[vK76] M. H. vanEmden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, October 1976.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[Wad85] W. Wadge. Tense Logic Programming: a Sane Alternative. University of Victoria, Canada 1985.

[Wad88] W. Wadge. Tense Logic Programming: a Respectable Alternative. In *Proceedings of the First International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.

[Wad91] W. W. Wadge. Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, 1991.

[Yag84] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages.* PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

VITA

Surname: Rondogiannis | Given Names: Panagiotis

Place of Birth: Athens, Greece | Date of Birth: August 31, 1966

Educational Institutions Attended:

University of Patras	1984 to 1989
University of Victoria	1990 to 1994

Degrees Awarded:

M.Sc.	University of Victoria	1991
Ptihion	University of Patras	1989

Honours and Awards:

University of Victoria Fellowship	1990–1994
Technical Chamber of Greece Scholarship	1991–1992
Graduate Teaching Award	1990–1991
Scholarship of the Hellenic-Canadian Association	1990–1991
National Institute of Scholarships Award	1985–1986
National Institute of Scholarships Award	1984–1985

Publications:

Theses:

- P. Rondogiannis, *Detecting Deadlocks in CCS Agents Using Petri Net Reduction Techniques*, University of Victoria, November 1991.
- P. Rondogiannis, *Deadlock Detection in Distributed Systems*, University of Patras, Greece, June 1989.

Journal Publications:

- P. Rondogiannis and M. H. M. Cheng. Petri Net Based Deadlock Analysis of Process Algebra Programs, *Science of Computer Programming*, North Holland, vol. 23, no. 1, pp. 55–89, October 1994.
- P. Rondogiannis, G. Pavlides and A. Levy. A Distributed Algorithm for Communication Deadlock Detection, *Information and Software Technology*, 33(7):483–488, September 1991.

Conference Publications:

- P. Rondogiannis and W. W. Wadge, Compiling Higher-Order Functions for Tagged Dataflow, in Proceedings of the *IFIP/ACM International Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994, North Holland.
- P. Rondogiannis and W. W. Wadge, Higher-Order Dataflow and its Implementation on Stock Hardware, in Proceedings of the *ACM Symposium on Applied Computing*, March 1994.
- P. Rondogiannis and M. H. M. Cheng. DART: A Prolog System for Detecting Deadlocks in Concurrent Programs, in Proceedings of the *International Conference of Prolog Applications*, April 1992.

Symposium Publications:

- P. Rondogiannis and W. W. Wadge, A Dataflow Implementation Technique for Lazy Typed Functional Languages, In Proceedings of the *International Symposium on Lucid and Intensional Programming*, April 1993.
- P. Rondogiannis and W. W. Wadge,Transforming First-Order Functional Programs to Intensional Programs of Nullary Variables: Theoretical Foundations, In Proceedings of the *International Symposium on Lucid and Intensional Programming*, September 1994.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my dissertation to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this dissertation for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this dissertation for financial gain shall not be allowed without my written permission.

Title of Dissertation:

Higher-Order Functional Languages and Intensional Logic

Author: _____

Panagiotis Rondogiannis

December 14, 1994

Date