

EXTENDING THE INTENSIONALIZATION ALGORITHM TO A BROADER CLASS OF HIGHER-ORDER PROGRAMS

P. RONDOGIANNIS AND W. W. WADGE

Department of Computer Science, University of Victoria

P.O.Box 3055, Victoria, BC, Canada V8W 3P6

E-mail: {prondo,wwadge}@lucy.uvic.ca

ABSTRACT

Recent research^{5,3,4,2} has demonstrated that a broad class of higher-order functional programs can be transformed into zero-order intensional ones, in a semantics preserving way. The main practical benefit of this investigation, is that the programs that result from the transformation can be efficiently executed based on the *eduction* implementation technique¹. From a theoretical point of view, this research provides the formal basis for establishing the links between functional languages and the *dataflow* model of computation. At its present form, the transformation can be used in order to compile a significant and well-defined subset of any higher-order functional language. However, it still remains an interesting problem to extend the algorithm to apply to a fully higher-order functional language. In this paper we make a contribution in this direction, by proposing a technique that extends the class of compilable programs.

1. Introduction

Recent research^{5,3,4,2} has demonstrated that a broad class of higher-order functional programs can be transformed into zero-order intensional ones, in a semantics preserving way. The interest for such an investigation is twofold:

- From a functional programming point of view, this research provides a promising dataflow approach for the implementation of general-purpose functional languages.
- From an intensional programming point of view, this research allows the investigation and experimentation with higher-order Lucid and other intensional languages and systems.

At its present form, the transformation into intensional form can be used in order to compile a significant and well-defined subset of any higher-order functional language. However, it still remains an interesting problem to extend the algorithm to apply to a fully higher-order language. In this paper we make a contribution in this direction, by proposing a technique that extends the class of compilable programs.

The rest of the paper is organized as follows: Section 2 presents background information on the intensionalization algorithm in its present form. Sections 3 and 4 illustrate how the algorithm can be extended in order to apply to a broader class of higher-order functional programs. Finally, Section 5 discusses the limitations of the new scheme as well as the reasons that cause them.

2. Intensionalization: an Overview

The main idea of the algorithm for higher-order programs 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 is done in the first-order case⁶. The same procedure can then be repeated for the new program, until an intensional program of nullary variables is obtained. In the following we give an example of how the algorithm works in practice.

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: call_i and actuals for the first step, call'_i and

actuals' for the second, and so on. 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      ≐ (call0(apply))(8)
apply(x)    ≐ f(x)
inc(y)      ≐ y+1
f           ≐ actuals(inc)

```

We see that in the program above all the functions have zero-order arguments. The only exception is the definition of `f` which is an equation between function expressions. This can be changed by introducing a formal parameter `z` for `f`:

```

result      ≐ (call0(apply))(8)
apply(x)    ≐ f(x)
inc(y)      ≐ y+1
f(z)        ≐ (actuals(inc))(z)

```

Notice now that the above program is not yet first-order: the operators `call` and `actuals` have as arguments higher-order objects. We must therefore transform the program into one in which the operators are first-order.

For the `actuals` operator, this can be done by passing the variable `z` inside the scope of the operator. However, the `actuals` operator alters (*shortens*) the contexts. In order for `z` to be evaluated in the outer context, it has to be appropriately “advanced” before entering it inside the scope of `actuals`. This advancement can be achieved using the complementary `call` operator.

For the `call` operator, this can be done by passing the constant `8` inside the scope of the operator. The `call` operator *lengthens* the contexts. In order for `8` to be evaluated in the outer context, it has to be appropriately “taken back” before entering it inside the scope of `call`. This can be achieved using the complementary `actuals` operator.

The above ideas are illustrated below:

```

result      ≐ call0(apply(actuals(8)))
apply(x)    ≐ f(x)
inc(y)      ≐ y+1
f(z)        ≐ (actuals(inc(call0(z))))

```

This completes the first step in the transformation. Now, we have a first-order intensional program, and we can apply the same procedure getting the following intensional program of nullary variables:

```

result      ≐ call0(call'0(apply))
apply       ≐ call'0(f)
inc         ≐ y+1
f           ≐ actuals(call'0(inc))
z           ≐ actuals'(x)
y           ≐ actuals'(call0(z))
x           ≐ actuals'(actuals(8))

```

The evaluation of a program starts with a sequence that contains m empty lists, where m is the order of the source functional program. Operators that have been introduced at different steps of the transformation,

affect different lists in the sequence. Execution proceeds as in the first-order case⁶, the only difference being that the appropriate list within the sequence is accessed every time.

3. Extending the Technique

The intensionalization algorithm described in the last section can only be applied to a specific class of higher-order programs. This class can be intuitively described as follows:

1. The programs are flat, i.e., they do not contain nested where-clauses. Every functional program can be easily transformed into flattened form.
2. All functions in the program have zero-order result type. This is not a restriction because every function can easily take this form by adding an appropriate number of variables in both sides of its definition.
3. The only partially applied objects that can appear as parameters in function calls are function names.

Many programs can be excluded under the above conditions. For example:

```

result      ≐ g(2)
g(x)        ≐ twice(add(x),x+1)
twice(f,x)  ≐ f(f(x))
add a b     ≐ a+b

```

This is not a legitimate program, since the function `twice` is called with the partial application `add(x)` as an argument. Indeed, when trying to apply the intensionalization algorithm to the above program, we get:

```

result      ≐ g(2)
g(x)        ≐ call0(twice)(x+1)
twice(x)    ≐ f(f(x))
f           ≐ actuals(add(x))
add a b     ≐ a+b

```

The resulting program is not semantically equivalent to the initial one: the variable `x` is bound in the definition of `g` while it is free in the definition of `f`.

The cause of the problems is clearly the existence of partial applications (like `add(x)`) in the source program. In such applications, higher-type variables and lower-type ones coexist, and the algorithm is forced to process them at the same time. The problems would be avoided if we could somehow “separate” the higher from the lower type variables. For the program under consideration, this can be performed as follows:

```

result      ≐ g(2)
g(x)        ≐ twice1(add,x,x+1)
twice1(h,z,w) ≐ h z ( h z w)
add a b     ≐ a+b

```

The partial application `add(x)` has been transformed into two different arguments, and the definition of `twice` has been altered in order to reflect this fact. The new program is a “legitimate” one, and can be intensionalized in the usual way.

In general, it is possible that the above process gives rise to different forms of a function. For example, consider the program:

```

result      ≐ twice(g,5)
g(x)        ≐ twice(add(x),x+1)
twice(f,x)  ≐ f(f(x))
add a b     ≐ a+b

```

In this program we have two different calls to the `twice` function. The transformed program will be:

```

result      ≐ twice(g,2)
g(x)        ≐ twice1(add,x,x+1)
twice1(h,z,w) ≐ h z (h z w)
twice(f,x)  ≐ f(f(x))
add a b     ≐ a+b

```

Notice that the fact that in the source program `twice` is called in two different ways, forces us to create two different forms of the `twice` function in the resulting program.

4. Handling Recursive Functions

The technique described in the last section can also be used in the case of recursively defined functions. For example, consider the following program:

```

result      ≐ f(sq,3)
f(s,x)      ≐ if (x<1) then s(x) else f(add (s x), x-1)
add a b     ≐ a+b
sq y        ≐ y*y

```

Following the guidelines of the last section, we can transform the above program in the following way:

```

result      ≐ f(sq,3)
f(s,x)      ≐ if (x<1) then s(x) else f1(add, (s x), x-1)
f1(h,z,w)   ≐ if (w<1) then (h z w) else f1(h, (h z w), w-1)
add a b     ≐ a+b
sq y        ≐ y*y

```

Again, the resulting program is a “legitimate” one, and can be transformed in the usual way.

5. Discussion

The technique developed in this paper works for a class of programs that we were not able to handle until now. However, there still exist higher-order programs for which the transformation we propose is not applicable. We give one such example and then explain the possible reasons that create the problems in this case. Consider the program:

```

result      ≐ f(sq,3)
f(s,x)      ≐ if (x<1) then s(x) else f(h s x, x-1)
h g z w     ≐ g(z)+w
sq y        ≐ y*y

```

In the function call `f(h s x, x-1)`, the function `h` has the same order as the function `f`. Therefore, if we rewrite this call as `f1(h,s,x,x-1)`, the function `f1` will have order higher than the order of `f`. In other words, while trying to get rid of the illegitimate partial applications, we increase the order of the source program, and the transformation never ends.

For the moment, it is not clear how large is the class of programs that the transformation can handle. However, the case of function calls of the form `f(..., (h a1 ... an), ...)` where `f` and `h` are of the same order, appears to be problematic (as described above). For treating this case a different approach has to be devised.

6. References

1. A. A. Faustini, E. A. Ashcroft and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.
2. P. Rondogiannis. *Higher-Order Functional Languages and Intensional Logic*. PhD thesis, Department of Computer Science, University of Victoria, Canada, December 1994.
3. P. Rondogiannis and W. W. Wadge. Higher-Order Dataflow and its Implementation on Stock Hardware. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, March 1994.
4. P. Rondogiannis and W. W. Wadge. Compiling Higher-Order Functions for Tagged-Dataflow. In *Proceedings of the IFIP International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, North-Holland, August 1994.
5. W. W. Wadge. Higher-Order Lucid. In *Proceedings of the International Symposium on Lucid and Intensional Programming*, 1991.
6. A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.