Compiling Higher-Order Functions for Tagged Dataflow*

Panos Rondogiannis William W. Wadge

Abstract

The implementation of higher-order functions on tagged-dataflow machines has always been a problematic issue. This paper presents and formalizes an algorithm for transforming a significant class of higher-order programs into a form that can be executed on a dataflow machine. The meaning of the resulting code is described in terms of Intensional Logic, a mathematical formalism which allows expressions whose values depend on hidden contexts.

Keyword Codes: D.1.1; D.3.1; F.4.1

Keywords: Applicative (Functional) Programming; Programming Languages, Formal Definitions and Theory; Mathematical Logic

1 Introduction

One of the most appealing features of the dataflow model of computation is its close relationship with functional programming. This is evidenced by the fact that all the well-known dataflow languages are functional in nature [1]. Moreover, it is generally easy to implement first-order functions on a tagged-token dataflow machine. This is achieved through the use of appropriate tag-manipulation operations, which can be thought of as having a "coloring" effect on tokens [2]. However, this scheme fails when higher-order functions are considered. In practice, higher-order functions are implemented using non-dataflow mechanisms such as *closures* [3], an approach which is against the basic principles and spirit of tagged dataflow.

This paper considers the implementation of a significant subset of a higher-order functional language, using only simple dataflow concepts (such as *tags*). Given a program of order N, the technique gradually transforms it into a zero-order program extended with appropriate context (tag) manipulation operators. The algorithm is initially presented at an informal level and illustrated by examples. The last sections of the paper contain a formal definition of the transformation algorithm. The paper concludes by briefly discussing implementation issues and possible extensions.

2 The First-Order Case

Before considering higher-order programs, we outline the approach we adopt for the first-order case; this was initially developed in [16] and also described in [4]. The algorithm given in [16] transforms a first-order program into a set of zero-order definitions that contain context-manipulation operations. As the semantics of the resulting code is based on Montague's Intensional Logic [13], the resulting definitions are also referred to in [14] as *intensional* definitions. The functional language adopted in [16] is ISWIM [9]. Programs are initially flattened using a technique similar to lambda-lifting [7]. The following algorithm is then applied to this flattened code.

1. Let f be a function appearing in the program. Number the textual occurrences of calls to f starting at 1.

*Published in Michel Cosnard, Guang R. Gao, Gabriel M. Silberman, editors, *IFIP PACT*, North-Holland, pages 269–278, 1994.

2. Replace the i-th call of f by $call_i(f)$.

3. Remove the formal parameters from the definition of f, so that f is defined as an ordinary individual variable.

4. Introduce a 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. The actual parameters are listed in the order in which the calls are numbered.

As an example, consider the following program:

$$\begin{aligned} result &= f(5) \\ f(x) &= inc(x+1) \\ inc(y) &= y+1 \end{aligned}$$

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

$$\begin{aligned} result &= call_1(f) \\ f &= call_1(inc) \\ inc &= y+1 \\ x &= actuals(5) \\ y &= actuals(x+1) \end{aligned}$$

An execution model is established by considering the $call_i$ and *actuals* as operations on finite lists of natural numbers (referred to from now on as *tags* or *contexts*). Execution of the program starts by demanding the value of the variable *result* of the intensional program under the empty tag $[\,]$. The operator $call_i$ augments a tag t by prefixing it with i. On the other hand, *actuals* takes the head i of a tag and uses it to select its i-th argument. Formally, the semantic equations, as introduced in [16], are:[1]

$$\begin{aligned} \big(call_i(A)\big)_t &= A_{[i|t]} \\ \big(actuals(A_1,\ldots,A_n)\big)_{[i|t]} &= (A_i)_t \\ c(A_1,\ldots,A_n)_t &= c\big((A_1)_t,\ldots,(A_n)_t\big) \end{aligned}$$

where A, A_1, ..., A_n are values in the target (intensional) language and c is an n-ary operation symbol. In particular, individual constants in the target language have the same value under any tag, e.g., $(2)_t = 2$ for all t. The evaluation of the program proceeds by applying the above semantic rules; every time a variable is encountered, it is replaced by its defining expression (for an example execution, see [11]). The technique just described has been extensively used in the implementations of the Lucid functional dataflow language [15] as well as in other Lucid-related languages and systems.

3 The Higher-Order Case

The main idea for the generalization of the technique to higher-order programs was initially proposed in [14] and has since been extended and formalized in [10]. In this section we outline how the technique can be applied to a significant class of higher-order programs. Intuitively, the language we adopt allows a Pascal-like use of higher-order objects:

1. Function names can be passed as parameters but not returned as results.

2. Operation symbols are first-order.

[1] The notation $[i \mid t]$ denotes a list with head i and tail t.

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

The idea of tags is now more general: for a program of order N, a tag is an N-tuple of lists, where each list corresponds to a different order of the program. The operators are also more general as they have to manipulate the new, more complicated contexts. As the transformation for the higher-order case consists of a number of stages, we use a different set of operators for each stage. For the first step we use the operators $actuals_N$ and $call_{\langle N,i\rangle}$, where i ranges as in the first-order case. For the second step, we use $actuals_{(N-1)}$ and $call_{\langle N-1,i\rangle}$, and so on.

The code that results from the transformation can be executed following the same basic principles as in the first-order case. In the following, we present the transformation algorithm and describe the semantics of the generalized operators. Consider the following simple second-order program:

$$\begin{aligned} result &= apply(inc, 10) \\ apply(f, x) &= f(x) \\ inc(y) &= y + 1 \end{aligned}$$

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

$$\begin{aligned} result &= \big(call_{\langle 2,1\rangle}(apply)\big)(10) \\ apply(x) &= f(x) \\ inc(y) &= y + 1 \\ f &= actuals_2(inc) \end{aligned}$$

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. We can easily change this by introducing an auxiliary variable z:

$$\begin{aligned} result &= \big(call_{\langle 2,1\rangle}(apply)\big)(10) \\ apply(x) &= f(x) \\ inc(y) &= y + 1 \\ f(z) &= \big(actuals_2(inc)\big)(z) \end{aligned}$$

It is necessary to pass z inside the *actuals* before performing the next stage of the transformation. However, the *actuals* operator alters (shortens) the tags. In order for z to be evaluated in the outer tag, it has to be appropriately advanced before entering the scope of *actuals*. This is done as follows:

$$\begin{aligned} result &= \big(call_{\langle 2,1\rangle}(apply)\big)(10) \\ apply(x) &= f(x) \\ inc(y) &= y + 1 \\ f(z) &= actuals_2\big(inc(call_{\langle 2,1\rangle}(z))\big) \end{aligned}$$

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:

$$\begin{aligned} result &= call_{\langle 1,1\rangle}\big(call_{\langle 2,1\rangle}(apply)\big) \\ apply &= call_{\langle 1,1\rangle}(f) \\ inc &= y + 1 \\ f &= actuals_2\big(call_{\langle 1,1\rangle}(inc)\big) \\ z &= actuals_1(x) \\ y &= actuals_1\big(call_{\langle 2,1\rangle}(z)\big) \\ x &= actuals_1(10) \end{aligned}$$

The (informal) algorithm for the higher-order case consists of repeating the following steps until the program becomes zero-order.

- Let f be a function of the current highest order d. Number the textual occurrences of calls to f starting at 1.
- Remove from the i-th call to f all the actual parameters of order $(d-1)$. Prefix the call to f with $call_{\langle d,i\rangle}$.
- Remove from the definition of f the formal parameters of order $(d-1)$.
- For every formal parameter x of f that was eliminated, introduce an $actuals_d$ definition, using the same procedure as in the first-order case.
- Introduce new variables according to the type of x. Add the variables to both sides of the definition of x, appropriately advancing them before they enter the scope of $actuals_d$.

In the execution model for a program of order N, tags are N-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 $\langle t_1, \ldots, t_N\rangle$ to denote a tag. The operators *call* and *actuals* can now be thought of as operations on these more complicated tags. The semantics of $call_{\langle d,i\rangle}$ can be described as follows: given a tag, d is used in order to select the corresponding list from the tag. The list is then prefixed with i and returned to the tag.

On the other hand, $actuals_d$ takes from the tag the list corresponding to d, uses its head i to select the i-th argument of *actuals*, and returns the tail of the list to the tag. The new semantic equations are:

$$\begin{aligned} \big(call_{\langle d,i\rangle}(A)\big)_{\langle t_1,\ldots,t_d,\ldots,t_N\rangle} &= A_{\langle t_1,\ldots,[i|t_d],\ldots,t_N\rangle} \\ \big(actuals_d(A_1,\ldots,A_n)\big)_{\langle t_1,\ldots,t_d,\ldots,t_N\rangle} &= \big(A_{head(t_d)}\big)_{\langle t_1,\ldots,tail(t_d),\ldots,t_N\rangle} \end{aligned}$$

For n-ary operation symbols c, the semantic equation is the same as in the first-order case. The evaluation of a program starts with an N-tuple of empty lists, one for each order. Execution proceeds as in the first-order case, the only difference being that the appropriate list within the tuple is accessed every time.

4 Extensional and Intensional Languages

In this section, the syntax of the source and target languages are introduced. Based on this material, a formal definition of the transformation algorithm is presented in the next section.

4.1 The Source Functional Language

The source language adopted is a simple functional language, of recursive equations, whose syntax satisfies the two requirements introduced in the previous section. The source language will also be referred as the *extensional* language, to distinguish it from the target *intensional* one. We proceed by defining the set of allowable types of the language and then presenting its formal syntax.

Definition 1. *The set* **Typ** *of extensional types is ranged over by* τ *and is recursively defined as follows:*

- *The base type* ι *is a member of* **Typ**.
- *If* $\tau_1, \ldots, \tau_n$ *are members of* **Typ**, *then* $(\tau_1 \times \cdots \times \tau_n) \to \iota$ *is a member of* **Typ**.

For simplicity, when $n = 0$, we assume that $(\tau_1 \times \cdots \times \tau_n) \to \iota$ is the same as ι. The meaning of the base type ι is a given domain $\mathbf{V}$, which will be called the *extensional domain.*

Definition 2. *The syntax of the extensional functional language Fun is described by the following rules:*

- $\mathbf{Var}^\tau$ *are given sets of* variable *symbols of type* τ *and are ranged over by* f^τ.
- $\mathbf{Con}^\tau$ *are given sets of* operation *symbols of type* τ, *ranged over by* c^τ.
- **Exp** *is a set of* expressions *ranged over by* E *and given by:*

$$\begin{aligned} E ::= & \ f^\tau \\ & \mid \left(c^{(\iota \times \cdots \times \iota) \to \iota}(E_1^\iota, \ldots, E_n^\iota)\right)^\iota \\ & \mid \left(f^{(\tau_1 \times \cdots \times \tau_n) \to \iota}(E_1^{\tau_1}, \ldots, E_n^{\tau_n})\right)^\iota \end{aligned}$$

  The set of expressions of type τ *is denoted by* $\mathbf{Exp}^\tau$.

- **Def** *is a set of* definitions *ranged over by* D *and given by:*

$$D ::= \left(f^{(\tau_1 \times \cdots \times \tau_n) \to \iota}(x_1^{\tau_1}, \ldots, x_n^{\tau_n}) \doteq E^\iota\right)$$

- **Prog** *is a set of* programs *ranged over by* P. *A program is a set of definitions,exactly one of which defines the nullary variable result. The variables appearing in the program (including formal parameters), are distinct from each other. Functiondefinitions do not contain global nullary variable symbols in their body.*

The order of a function is the order of its type. Formally:

Definition 3. *The* order *of a type is recursively defined as follows:*

$$\begin{aligned} order(\iota) &= 0 \\ order\big((\tau_1 \times \cdots \times \tau_n) \to \iota\big) &= 1 + max\big(\{order(\tau_k) \mid k = 1..n\}\big) \end{aligned}$$

4.2 The Target Intensional Language

Intensional Logic [13, 6] is a mathematical formal system which allows expressions whose values depend on hidden contexts. From the informal exposition given in §2–3, it is clear that the programs that result from the transformation can be evaluated with respect to a context. The zero-order variables that appear in the final program are not ordinary variables that have a fixed data value. In fact, they can intuitively be thought of as tree-like structures that have individual data values at their nodes. Such variables are also called *intensions* [6].

Another way one can think about intensions is to consider them as functions from contexts to data values, i.e., as members of $(\mathbf{W} \to \mathbf{V})$, where $\mathbf{W}$ is a set of contexts (also called *possible*

worlds) and $\mathbf{V}$ is a data domain. Given an intension and a context, the value of the intension at this context can be computed. Moreover, operations like $+$, $*$, `if`–`then`–`else`, and so on, are now operations that take as arguments intensions. For example, if x and y are intensions, then $(x + y)$ can be thought of as a new intension, whose value at every context is the sum of the values of the intensions x and y at this same context. In other words, operations on intensions are defined in a pointwise way using the corresponding (extensional) operations on the data domain $\mathbf{V}$. The above discussion is formalized by the following definitions:

Definition 4. *The set* **ITyp** *of intensional types is ranged over by* τ *and is recursively defined as follows:*

- *The type* ι^* *is a member of* **ITyp**.
- *If* $\tau_1, \ldots, \tau_n$ *are members of* **ITyp** *then* $(\tau_1 \times \cdots \times \tau_n) \to \iota^*$ *is a member of* **ITyp**.

Let $\mathbf{W}$ be a set of *possible worlds* (or *contexts*, or *tags*) and let $\mathbf{V}$ be an extensional domain. Then the meaning of the base type ι^* is the domain $\mathbf{I} = (\mathbf{W} \to \mathbf{V})$, called the *intensional domain*. Members of $\mathbf{I}$ are called *intensions*. The set $\mathbf{W}$ plays the most crucial rôle in the specification of an intensional language. In our case, $\mathbf{W} = (\mathbb{N} \to \mathbf{List}(\mathbb{N}))$, i.e., $\mathbf{W}$ is the set of functions from natural numbers to lists of natural numbers. Given an extensional language *Fun* over the data domain $\mathbf{V}$, an intensional language *In*(*Fun*) can be created as follows:

Definition 5. *The syntax of the intensional language In*(*Fun*) *is described by the following rules:*

- $\mathbf{IVar}^\tau$ *are given sets of* intensional variable *symbols ranged over by* f^τ.
- $\mathbf{ICon}^\tau$ *is a given set of continuous* intensional operation *symbols, ranged over* c^τ. *Members of* $\mathbf{ICon}^\tau$ *are also called* pointwise *operations, because their meaning is defined in a pointwise way in terms of their extensional counterparts.*
- **INp** *is a set of non-pointwise operation symbols given by:*

$$\{call_{\langle d,i\rangle} \mid d, i \in \mathbb{N}\} \cup \{actuals_d \mid d \in \mathbb{N}\}$$

- **IExp** *is a set of* intensional expressions *ranged over by* E *and given by:*

$$\begin{aligned} E ::= {} & f^\tau \\ & \mid \big(c^{(\iota^* \times \cdots \times \iota^*) \to \iota^*}(E_1^{\iota^*}, \ldots, E_n^{\iota^*})\big)^{\iota^*} \\ & \mid \big(E_0^{(\tau_1 \times \cdots \times \tau_n) \to \iota^*}(E_1^{\tau_1}, \ldots, E_n^{\tau_n})\big)^{\iota^*} \\ & \mid \big(call_{\langle d,i\rangle}(E^\tau)\big)^\tau \\ & \mid actuals_d\big(\{i_1 : E_{i_1}^\tau, \ldots, i_n : E_{i_n}^\tau\}\big)^\tau, d, i, i_1, \ldots, i_n \in \mathbb{N} \end{aligned}$$

- **IDef** *is a set of* intensional definitions *ranged over by* D *and given by:*

$$D ::= \big(f^{\tau_1 \times \cdots \times \tau_n) \to \iota^*}(x_1^{\tau_1}, \ldots, x_n^{\tau_n}) \doteq E^{\iota^*}\big)$$

- **IProg** *is a set of* intensional programs *ranged over by* P. *The same assumptions are adopted as in the case of extensional programs.*

Notice that the *actuals* operator is more general than the one described in §3. The new semantic equation is:

$$actuals_d\big(\{i_1 : A_{i_1}, \ldots, i_n : A_{i_n}\}\big)\big(\langle t_1, t_2, \ldots\rangle\big) = A_{hd(t_d)}\big(\langle t_1, t_2, \ldots, tl(t_d), \ldots\rangle\big)$$

The order of intensional functions can be defined in the same way as in the extensional case.

5 A Formal Definition of the Transformation

In this section, we describe the transformation of a d-order program into a $(d-1)$-order one. The following conventions are adopted:

Definition 6. *Let P be a program and let f be a function symbol defined in P. Then:*

- *The notation $\overline{f}$ is used to denote an expression of the form $(call_{\langle d_1,i_1\rangle} \cdots call_{\langle d_k,i_k\rangle}(f))$, $k \geq 0$. For $k = 0$, $\overline{f}$ denotes the identifier f itself.*
- *Given $d \in \mathbb{N}$, $pos(f, d)$ is the list of positions of the formal parameters of f that have order less than $(d-1)$. For example, $pos(apply, 2) = [2]$, because the second argument of the apply function of §3 has order less than 1.*
- *The set of $(d-1)$-order formal parameters of f is represented by $high(f, d)$.*
- *The set of subexpressions appearing in definitions of the program P is denoted by $Sub(P)$, and the set of function identifiers defined in P is denoted by $func(P)$.*

Moreover, we assume the existence of a function $\lceil\cdot\rceil$ that assigns unique natural-number identities to expressions, i.e., if $E_1 \neq E_2$ then $\lceil E_1\rceil \neq \lceil E_2\rceil$. The elimination of the $(d-1)$-order arguments from function calls is accomplished with the $\mathcal{M}_d$ function, defined below:

$$\frac{E = \overline{f}}{\mathcal{M}_d(E) = \overline{f}}$$

$$\frac{E = c(E_1, \ldots, E_n)}{\mathcal{M}_d(E) = c\big(\mathcal{M}_d(E_1), \ldots, \mathcal{M}_d(E_n)\big)}$$

$$\frac{E = (\overline{f^\tau})(E_1, \ldots, E_n), order(\tau) = d, pos(f, d) = [i_1, \ldots, i_k]}{\mathcal{M}_d(E) = \big(call_{\langle d,\lceil E\rceil\rangle}(\overline{f})\big)\big(\mathcal{M}_d(E_{i_1}), \ldots, \mathcal{M}_d(E_{i_k})\big)}$$

$$\frac{E = (\overline{f^\tau})(E_1, \ldots, E_n), order(\tau) \neq d}{\mathcal{M}_d(E) = (\overline{f})\big(\mathcal{M}_d(E_1), \ldots, \mathcal{M}_d(E_n)\big)}$$

$$\frac{E = actuals_{d'}\big(\{i_1 : E_{i_1}, \ldots, i_n : E_{i_k}\}\big)}{\mathcal{M}_d(E) = actuals_{d'}\big(\{i_1 : \mathcal{M}_d(E_{i_1}), \ldots, i_n : \mathcal{M}_d(E_{i_n}\}\big)}$$

The first and second rules above are self-explanatory. The third rule applies in the case where a function call is encountered, and the corresponding function is d-order. In this case, the arguments that cause the function to be d-order (i.e., the $(d-1)$-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 $\mathcal{M}_d$ identical. The fourth rule applies in the case where the function under consideration is not d-order. In this case, the translation proceeds with the actual parameters of the function call. The final rule concerns the *actuals* operator, and is also straightforward.

The elimination of the $(d-1)$-order formal parameters from the function definitions is accomplished by the function $\mathcal{R}_d$, defined below:

$$\frac{D = \big(f(x_1, \ldots, x_n) \doteq E\big), pos(f, d) = [i_1, \ldots i_k]}{\mathcal{R}_d(D) = \big(f(x_{i_1}, \ldots, x_{i_k}) \doteq \mathcal{M}_d(E)\big)}$$

For every formal parameter that has been eliminated, a new definition is added to the program. This is achieved using the function $\mathcal{A}_d^f$:

$$\mathcal{A}_d^f(P) = \bigcup_{x_k \in high(f,d)} \{x_k \doteq actuals_d(\{(\lceil F \rceil : \mathcal{M}_d(E_k)) \mid F \equiv (\overline{f})(E_1, \ldots, E_n) \in Sub(P)\})\}$$

The overall translation of a d-order program into a $(d-1)$-order one is performed by the function $\mathcal{T}_d$:

$$\mathcal{T}_d(P) = \Big(\bigcup_{f \in func(P)} \mathcal{A}_d^f(P) \Big) \cup \Big(\bigcup_{D \in P} \{\mathcal{R}_d(D)\} \Big)$$

The translation described above introduces certain *actuals* definitions, whose result type is not first order. Let

$$f^\tau \doteq actuals_d\big(\{i_1 : E_{i_1}, \ldots i_n : E_{i_n}\}\big)^\tau$$

be one of them, and suppose that $\tau = \tau_1 \times \cdots \times \tau_n \to \iota^*$. We introduce n new variables $z_1^{\tau_1}$, ..., $z_n^{\tau_n}$ and equivalently rewrite the above definition as:

$$f(z_1, \ldots, z_n) \doteq actuals_d\big(\{i_1 : E_{i_1}, \ldots, i_n : E_{i_n}\}\big)(z_1, \ldots, z_n)$$

It can be shown [10] that the above definition is equivalent to the following one in which the parameters have entered the scope of *actuals*:

$$f(z_1, \ldots, z_n) \doteq actuals_d\Big(\big\{i_1 : E_{i_1}\big(call_{\langle d,i_1\rangle}(z_1), \ldots, call_{\langle d,i_1\rangle}(z_n)\big), \ldots, i_n : E_{i_n}\big(call_{\langle d,i_n\rangle}(z_1), \ldots, call_{\langle d,i_n\rangle}(z_n)\big)\big\}\Big)$$

In the program that results following this approach, all the definitions have first-order result types. The transformation algorithm can therefore be applied repeatedly, until a zero-order program is obtained.

6 Discussion and Conclusions

An algorithm for transforming higher-order functional programs into zero-order intensional ones has been presented. The resulting code can be evaluated relative to a context, giving in this way a method of implementing a class of higher-order functions in a purely dataflow manner.

Two preliminary implementations have been undertaken by the authors. In [11], a hashing-based approach is adopted. More specifically, identifiers, together with their context and value, are saved in a Value Store. If the identifier is encountered again under the same context during execution, the Value Store is looked up using a hashing function, and the corresponding value is returned. The measurements reported in [11] indicate that the technique can compete with modern graph-reduction [8] implementation techniques for functional languages. However, hashing is reported as a factor that can cause a severe bottleneck to the implementation.

In [12], a technique is proposed in which contexts are appropriately incorporated in the headers of activation records. A similar approach is reported in [5]. The main advantage of an activation-record based technique is that hashing is avoided, and a significant gain in execution time is obtained. The results presented in [12] indicate that the tag-based code outperforms many well-known reduction-based systems.

Future work includes investigating optimizations such as *strictness analysis*, as well as intensional code transformations. Also, we are currently considering the extension of the technique for functions with higher-order result types.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and a University of Victoria Graduate Fellowship.

References

[1] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982.

[2] Arvind and David E. Culler. Dataflow architectures. In *Selected Reprints on Dataflow and Reduction Architectures*, pages 79–101. IEEE Press, 1987.

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

[4] E. A. Ashcroft, A. A. Faustini, and R. Jagannathan. An intensional language for parallel applications programming. In *Parallel functional languages and compilers*, pages 11–49, New York, NY, USA, 1991. ACM.

[5] David E. Culler and Gregory M. Papadopoulos. The explicit token store. *J. Parallel Distrib. Comput.*, 10(4):289–308, 1990.

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

[7] Thomas Johnsson. Lambda lifting: Treansforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

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

[9] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[10] P. Rondogiannis. *Higher-Order Functional Languages and Intensional Logic*. PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994.

[11] P. Rondogiannis and William W. Wadge. A dataflow implementation technique for lazy typed functional languages. In *6th International Symposium on Lucid and Intensional Programming*, Québec, Canada. Département d'Informatique, Université Laval.

[12] Panos Rondogiannis and William W. Wadge. Higher-order dataflow and its implementation on stock hardware. In *SAC*, pages 431–435, 1994.

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

[14] William W. Wadge. Higher-order Lucid. In *4th International Symposium on Lucid and Intensional Programming*, Menlo Park, CA. SRI International.

[15] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

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