Using Functor Categories to Generate Intermediate Code *

John C. Reynolds

Department of Computing
Imperial College
London SW7 2BZ, Great Britain
jr@doc.ic.ac.uk

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
john.reynolds@cs.cmu.edu

Abstract

In the early 80's Oles and Reynolds devised a semantic model of Algol-like languages using a category of functors from a category of store shapes to the category of predomains. Here we will show how a variant of this idea can be used to define the translation of an Algol-like language to intermediate code in a uniform way that avoids unnecessary temporary variables, provides control-flow translation of boolean expressions, permits online expansion of procedures, and minimizes the storage overhead of calls of closed procedures. The basic idea is to replace continuations by instruction sequences and store shapes by descriptions of the structure of the run-time stack.

1 Introduction

To construct a compiler for a modern higher-level programming language, one needs to structure the translation to a machine-like intermediate language in a way that reflects the semantics of the language. Little is said about such structuring in compiler texts that are intended to cover a wide variety of programming languages. More is said in the literature on semantics-directed compiler construction [1], but here too the viewpoint is very general (though limited to languages with a finite number of syntactic types). On the other hand, there is a considerable body of work using the continuation-passing transformation to structure compilers for the specific case of call-by-value languages such as Scheme and ML [2, 3].

In this paper, we will describe a method of structuring the translation of Algol-like languages that is based on the functor-category semantics developed by Reynolds [4] and Oles [5, 6].

An alternative approach using category theory to structure compilers is the early work of F. L. Morris [7], which anticipates our treatment of boolean expressions, but does not deal with procedures.

*This research was sponsored in part by National Science Foundation Grant CCR-8922109 and in part by a fellowship from the Science and Engineering Research Council.

2 Types and Syntax

An Algol-like language is a typed lambda calculus with an unusual repertoire of primitive types. Throughout most of this paper we assume that the primitive types are

$$\mathbf{comm}(\text{and}) \quad \mathbf{int}(\text{eger})\mathbf{exp}(\text{ression})$$

$$\mathbf{int}(\text{eger})\mathbf{acc}(\text{eptor}) \quad \mathbf{int}(\text{eger})\mathbf{var}(\text{iable}),$$

and that the set Θ of types is the least set containing these primitive types and closed under the binary operation $\to$.

We write $\leq$ for the least preorder such that

$$\mathbf{intvar} \leq \mathbf{intexp} \qquad \mathbf{intvar} \leq \mathbf{intacc}$$

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'.$$

When $\theta \leq \theta'$, θ is said to be a *subtype* of θ'.

A type assignment is a mapping from some finite set of identifiers into types; we write Θ^* for the set of type assignments. Then we write the *typing* $\pi \vdash p : \theta$ to indicate that the phrase p has type θ under the type assignment π.

We omit both the definition of the syntax of phrases and the inference rules for typings, beyond noting that phrases include identifiers and the lambda-calculus operations of application and abstraction, and the inference rules include the standard rules for the typed lambda calculus with subtypes.

3 Functor-Category Semantics

The basic assumption that an Algol-like language is a species of typed lambda calculus is captured by using a cartesian closed category to provide its semantics. More specifically, we assume that there is a functor $[\![-]\!]$, from Θ (preordered by the subtype relation and viewed as a category) to a cartesian closed *semantic category* $\mathcal{K}$, that interprets the type constructor $\to$ as the exponentiation operation of $\mathcal{K}$:

$$[\![\theta \to \theta']\!] = [\![\theta]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\theta']\!].$$

(If the type system includes type constructors for tuples or records, these will be interpreted by products in $\mathcal{K}$. Intersection types would be interpreted by pullbacks, as discussed in [8, 9, 10].)

The functor $[\![-]\!]$ interprets types as objects of the semantic category $\mathcal{K}$. In addition, whenever θ is a subtype of θ' (i.e. $\theta \leq \theta'$), it maps the unique morphism from θ to θ' into an "implicit conversion morphism", which we denote by $[\![\theta \leq \theta']\!]$, from the meaning of θ to the meaning of θ'.

The meaning of type assignments is specified by a functor $[\![-]\!]^*$, from Θ^* (preordered pointwise and viewed as a category) to $\mathcal{K}$, that maps each type assignment into an appropriate product in the semantic category:

$$[\![\pi]\!]^* = \prod_{\iota \in \mathrm{dom}\,\pi}^{\mathcal{K}} [\![\pi\iota]\!] .$$

When $\pi \vdash p : \theta$, the semantics of the phrase p, with respect to π and θ, is a morphism from the meaning of π to the meaning of θ that we denote by

$$[\![p]\!]_{\pi\theta} \in [\![\pi]\!]^* \xrightarrow[\mathcal{K}]{} [\![\theta]\!] .$$

(This redundant use of emphatic brackets is saved from ambiguity by the subscripts, which qualify the semantics of a phrase but not the functorial use of the brackets.)

Throughout this paper, we will write $A \xrightarrow[\mathcal{C}]{} B$ to denote the set of morphisms from A to B in the category $\mathcal{C}$, and $A \Longrightarrow_{\mathcal{C}} B$ to denote the exponentiation of B by A in $\mathcal{C}$. When the subscript is omitted, the relevant category is either that of domains and continuous functions or of predomains and continuous functions. (Of course, $\to$ is also used as the type constructor for procedural types.)

The semantics $[\![p]\!]_{\pi\theta}$ is specified by giving a semantic equation for each syntactic construct of the language, plus an equation describing the effect of implicit conversions:

$$[\![p]\!]_{\pi\theta'} = [\![p]\!]_{\pi\theta} ; [\![\theta \le \theta']\!] \qquad \text{when } \theta \le \theta' , \qquad (1)$$

where the semicolon denotes the composition of morphisms in $\mathcal{K}$ (in diagrammatic order).

The above equation is not syntax-directed, i.e. it does not define the semantics of p in terms of the semantics of its subphrases, but rather defines one semantics of p in terms of another. As discussed in [10], this means that $[\![p]\!]_{\pi\theta}$ must be defined by structural induction on the proof of the typing $\pi \vdash p : \theta$ rather than on the syntactic structure of p. This has the advantage that semantics (and similarly translation) is only defined for type-correct programs, but it introduces the requirement of *coherence*, i.e. that different proofs of the same typing must not lead to different meanings. Fortunately, the proof of coherence given in [10], for a language using intersection types, carries over to the much simpler language discussed in this paper.

If we were defining a purely functional, call-by-name language, we could take the semantic category $\mathcal{K}$ to be the category of domains (c.p.o.'s with least elements) and continuous functions. It is less obvious, however, how to provide a clear semantics of languages that include assignment. In the early 80's, Frank Oles and I devised such a semantics that makes the block structures of Algol-like languages explicit. Our basic idea was that the meaning of a type should be a family of domains parameterized by state sets (which Oles called "store shapes"):

$$[\![\mathbf{comm}]\!]S = S \to S_\perp$$

$$[\![\mathbf{intexp}]\!]S = S \to \mathcal{Z}_\perp$$

$$[\![\mathbf{intacc}]\!]S = \mathcal{Z} \to (S \to S_\perp)$$

$$[\![\mathbf{intvar}]\!]S = [\![\mathbf{intacc}]\!]S \times [\![\mathbf{intexp}]\!]S .$$

Similarly, the semantics of a phrase was a family of continuous functions parameterized by state sets:

$$[\![p]\!]_{\pi\theta} S \in [\![\pi]\!]^* S \to [\![\theta]\!]S .$$

Consider, for example, the Algol-like program

new x: **intvar in** (x := x + 1 ; . .

new y: **intvar in** (y := x + y ; x := x + 1 ; . . .)) .

In the outer block, where only a single variable is declared, the appropriate set of states is the set of integers, while in the inner block, where a second variable is declared, the appropriate set of states is the set of pairs of integers. Thus the semantics of the two occurrences of x:=x+1 are provided by different members of the family $[\![\mathsf{x} := \mathsf{x} + 1]\!]_{\pi,\mathrm{comm}}$. The member of this family appropriate to the occurrence in the outer block is

$$[\![\mathsf{x} := \mathsf{x} + 1]\!]_{\pi,\mathrm{comm}} \mathcal{Z} \in [\![\pi]\!]^* \mathcal{Z} \to (\mathcal{Z} \to \mathcal{Z}_\perp) ,$$

which maps an environment appropriate to states that are integers into a state-transition function on integers. If η is an environment specifying that x denotes the integer that is the entire state, then $[\![\mathsf{x} := \mathsf{x} + 1]\!]_{\pi,\mathrm{comm}} \mathcal{Z} \eta$ will be the function that increases an integer by one.

On the other hand, the member of $[\![\mathsf{x}:=\mathsf{x}+1]\!]_{\pi,\mathrm{comm}}$ that is appropriate to the occurrence in the inner block is

$$[\![\mathsf{x} := \mathsf{x} + 1]\!]_{\pi,\mathrm{comm}} (\mathcal{Z} \times \mathcal{Z}) \in$$
$$[\![\pi]\!]^* (\mathcal{Z} \times \mathcal{Z}) \to ((\mathcal{Z} \times \mathcal{Z}) \to (\mathcal{Z} \times \mathcal{Z})_\perp) ,$$

which maps an environment appropriate to states that are pairs of integers into a state-transition function on such pairs. If η is an environment specifying that x denotes the first component of the state, then $[\![\mathsf{x}:=\mathsf{x}+1]\!]_{\pi,\mathrm{comm}} (\mathcal{Z} \times \mathcal{Z}) \eta$ will be the function mapping a pair $\langle x, y \rangle$ into $\langle x + 1, y \rangle$.

In both cases, command execution is described by a state transition that preserves the shape of the state. Indeed, this is generally true since, for any command c,

$$[\![c]\!]_{\pi,\mathrm{comm}} S \in [\![\pi]\!]^* S \to (S \to S_\perp)$$

implies that $[\![c]\!]_{\pi,\mathrm{comm}} S \eta$ preserves the shape S.

From the viewpoint of category-theoretic semantics, parameterization by state sets is realized by taking the semantic category to be the functor category

$$\mathcal{K} = \mathrm{PDOM}^\Sigma ,$$

where Σ is a category whose objects are state sets and PDOM is the category of predomains and continuous functions. Of course, this implies that the meanings of types are functors that act on morphisms of Σ as well as objects, and that the semantics of phrases are natural transformations between such functors. In this brief synopsis, however, we will only remark that a morphism in $S \xrightarrow[\Sigma]{} S'$, called an "expansion" in [4, 5, 6], shows how a small state in S can be extracted from a large state in S' and how a small-state transition in $S \to S_\perp$ can be extended to a large-state transition in $S' \to S'_\perp$.

It is shown in [5, 6] that the functor category PDOM^Σ is cartesian closed (actually for any Σ). In particular, exponentiations are functors whose action on objects of Σ is

$$(F \Longrightarrow_{\mathcal{K}} G) S = \mathrm{hom}_\Sigma S \times F \xrightarrow[\mathcal{K}]{} G .$$

(Here pointwise ordering is used to regard the set on the right as a predomain, hom_Σ is the curried hom-functor for the category Σ, and $\mathrm{hom}_\Sigma S S' = S \xrightarrow[\Sigma]{} S'$ is regarded as a discretely ordered predomain.)

To see how this exponentiation captures the interaction of procedures and block structure, suppose

$$p \in [\![\theta_1 \to \theta_2]\!]S = ([\![\theta_1]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\theta_2]\!])S = \hom_\Sigma S \times [\![\theta_1]\!] \underset{\mathcal{K}}{\to} [\![\theta_2]\!],$$

is the meaning of a procedure of type $\theta_1 \to \theta_2$. Then p is a natural transformation such that

$$pS' \in (S \underset{\Sigma}{\to} S') \times [\![\theta_1]\!]S' \to [\![\theta_2]\!]S' .$$

Here S is the state set that is appropriate to the point in the program where the procedure is defined (and that contains states specifying the values of any variables occurring globally in the procedure). For example, if p were the meaning of the procedure in

new x: **intvar in**

let silly $\equiv \lambda$c: **comm**. (c ; x := x + 1 ; c) **in**

new y: **intvar in** silly(x := x + y)

then S would be a set of integers specifying the global variable x.

However, as illustrated by the procedure call in the above program, the procedure with meaning p can be called from an inner block where a set S' of larger states is appropriate, and these larger states, rather than the members of S, may be needed to specify variables in the actual parameter of the call. Thus the state set S' must be given to p as a "hidden" argument, and both the explicit argument of p (the meaning of the actual parameter) and the result of p (the meaning of the call) must be appropriate to S'. In addition, p must be supplied with a further hidden argument, which is a morphism in Σ showing how a state in S can be expanded to a state in S'.

In the above program, for example, the meaning of the procedure call would be $pS'\langle \iota, a \rangle$, where S' is the set of pairs of integers, ι is an expansion identifying the integers in S with one component of the pairs in S', and $a \in [\![\mathbf{comm}]\!]S'$ is the meaning of the actual parameter x := x + y.

For simplicity, we have used direct semantics in this introduction. However, the method applies even more elegantly to continuation semantics, without any change in the category $\mathcal{K}$. One introduces two new types:

compl(etion) **int**(eger)**compl**(etion)

whose meanings are command continuations and integer continuations, respectively. More precisely,

$$[\![\mathbf{compl}]\!]S = S \to O \qquad [\![\mathbf{intcompl}]\!]S = \mathcal{Z} \to (S \to O) ,$$

where O is an unspecified domain of "outputs". The remaining types are then defined by exponentiation and (pointwise) products in $\mathcal{K}$:

$$\begin{aligned} [\![\mathbf{comm}]\!] &= [\![\mathbf{compl}]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{compl}]\!] \\ [\![\mathbf{intexp}]\!] &= [\![\mathbf{intcompl}]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{compl}]\!] \\ [\![\mathbf{intacc}]\!] &= [\![\mathbf{compl}]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{intcompl}]\!] \\ [\![\mathbf{intvar}]\!] &= [\![\mathbf{intacc}]\!] \times [\![\mathbf{intexp}]\!] \\ [\![\theta \to \theta']\!] &= [\![\theta]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\theta']\!] . \end{aligned} \tag{2}$$

In fact, as one might expect from the prevalence of continuations in compiler design, it is this continuation semantics that will be the starting point for our development of an intermediate-code generator. (Strictly, it is a continuation semantics with respect to the imperative aspects of our illustrative language, but still a direct semantics with respect to the call-by-name procedural aspects.)

4 Stack Descriptors

During program execution, the variables and other information accessible to the program will lie in a sequence of contiguous blocks, called *frames*, contained within stack; when this sequence has length n, we will denote its members by *frame counts* between 0 and $n-1$, in order of their their position from the bottom to the top (most recently allocated and highest addressed portion) of the stack. We assume that the frames are organized as a linked list called a *static chain*, specifically that a register SR points to the base of frame $n-1$ and that the first (least addressed) word in each frame except frame 0 points to the base of the previous frame.

For simplicity, we also assume that integer variables and pointers both occupy single words, and that addressing is by words. (In fact, our approach extends straightforwardly to more complex cases where different kinds of variables require fields of different sizes, and these fields must be aligned on different-sized word boundaries.)

During compilation, for each variable the compiler will know the count S_f of the frame containing the variable, and the *displacement* S_d, which is the distance from the base of the containing frame to the location of the variable. This pair $S = \langle S_f, S_d \rangle$ of nonnegative integers is called a *stack descriptor*.

As one would expect, the stack descriptors of variables will be embedded (implicitly) in a compile-time environment describing the free identifiers of the phrase to be compiled. However, compilation will also be influenced by more general information about the stack that is not particular to any variable; in the simple case considered in this paper this compile-time information consists of the total number of frames (minus one) and the size of the top frame. We call this pair of integers, which will depend upon position in the intermediate code being compiled, the *current stack descriptor* S^{curr}. Note that the current stack descriptor describes the beginning of the free portion of the stack, i.e. the position of the next variable to be allocated.

Stack descriptors are ordered lexicographically:

$$\langle S_f, S_d \rangle \le \langle S'_f, S'_d \rangle \text{ iff } S_f < S'_f \text{ or } (S_f = S'_f \text{ and } S_d \le S'_d) .$$

Thus the effect of pushing the stack, either by enlarging the current top frame or by adding a new frame, is to increase the current stack descriptor. We also define the addition or subtraction of a stack descriptor and an integer by

$$\langle S_f, S_d \rangle \pm n \stackrel{\text{def}}{=} \langle S_f, S_d \pm n \rangle .$$

Thus the requirement that a variable described by S^v must lie within a frame in the currently active portion of the stack implies that $S^v \le S^{\text{curr}} - 1$.

The key to moving from a functor-category description of semantics to an analogous description of of intermediate-code generation is to replace Oles's the category Σ of store shapes by the ordered set of stack descriptors (viewed as a category), which we will also denote by Σ.

5 The Intermediate Language

The intermediate language into which we translate programs can be described by an abstract van Wijngaarden grammar with four stack-descriptor-indexed families of nonterminals: lefthand sides $\langle \mathrm{L}_S \rangle$, simple righthand sides $\langle \mathrm{S}_S \rangle$, righthand sides $\langle \mathrm{R}_S \rangle$, and instruction sequences $\langle \mathrm{I}_S \rangle$. The intent of the indexing is that a member of $\langle \mathrm{I}_S \rangle$ is an instruction sequence that can be meaningfully executed when the current stack descriptor is S:

$\langle \mathrm{L}_S \rangle ::= S^v$ when $S^v \leq S - 1$

| **sbrs**

$\langle \mathrm{S}_S \rangle ::= \langle \mathrm{L}_S \rangle$ | **lit** ⟨integer⟩

$\langle \mathrm{R}_S \rangle ::= \langle \mathrm{S}_S \rangle$ | ⟨unary operator⟩$\langle \mathrm{S}_S \rangle$

| $\langle \mathrm{S}_S \rangle$⟨binary operator⟩$\langle \mathrm{S}_S \rangle$

$\langle \mathrm{I}_S \rangle ::=$ **stop**

| $\langle \mathrm{L}_{S+\delta} \rangle := \langle \mathrm{R}_S \rangle\,[\delta]\,;\,\langle \mathrm{I}_{S+\delta} \rangle$

| **if** $\langle \mathrm{S}_S \rangle$⟨relation operator⟩$\langle \mathrm{S}_S \rangle\,[\delta]$ **then** $\langle \mathrm{I}_{S+\delta} \rangle$ **else** $\langle \mathrm{I}_{S+\delta} \rangle$

| **adjustdisp** $[\delta]\,;\,\langle \mathrm{I}_{S+\delta} \rangle$

} when $S_d + \delta \geq 0$

| **popto** $S'\,;\,\langle \mathrm{I}_{S'} \rangle$ when $S' \leq S$

(Additional forms will be introduced later.) Here **sbrs** denotes a register used to communicate the result of function procedures that are implemented by closed subroutines, and **lit** ⟨integer⟩ is a constant (or in compiler jargon, a literal). Notice that neither a right operand of the assignment operator := nor a relation following **if** can contain more than one operator.

In various instructions here, the bracketed integers δ are *displacement adjustments*, indicating an amount to be added to the current stack descriptor when the instruction is executed, because of the allocation or deallocation of either program variables or temporary variables. This adjustment of the current stack descriptor is the only effect of the **adjustdisp** instruction. The final instruction **popto** S' causes the current stack descriptor to be reset to S'; it is used to reduce the number of frames and causes a change in the register SR during program execution.

Although a change in the frame count of the current stack descriptor causes a change in the register SR during program execution, adjustments of the displacement have no effect during program execution, since displacements are not actually computed at run time. However, the compiler must keep track of these adjustments in translating intermediate code into machine language.

Strictly speaking, the $\langle \mathrm{I}_S \rangle$ are *domains* of instruction sequences, formed by completing the sets described by the above grammar in the sense of Scott's lattice of flow diagrams [11]. (Equivalently, the $\langle \mathrm{I}_S \rangle$ are components of the carrier of an initial continuous algebra [12] whose many-sorted signature is specified by the grammar.) Fortunately, the only infinite or partial instruction sequences that arise during compilation can be represented by data structures with loops, which can be implemented using references (in the sense of ML).

More generally, references can be used to avoid duplicating code: Whenever an instruction sequence i may be duplicated by the compiler, it is replaced by a unique reference whose value is i. For example, an instruction sequence of the form **if** $\cdots$ **then** $c\,;i$ **else** $d\,;i$ (which might arise from the compilation of a conditional command followed by another command), would be represented by **if** $\cdots$ **then** $c;r$ **else** $d;r$ where r is a unique reference whose value is i. In the conversion to actual machine code, all but at most one occurrence of such a reference is replaced by a jump to the code obtained from its value, rather than a copy of such code.

From the viewpoint this paper, however, the treatment of loops, the avoidance of code duplication, and the distinction between instructions and their addresses are questions of representation. The mathematics of compilation is much cleaner if we abstract away from finite instruction sequences with references or jumps, to the possibly infinite sequences that they represent.

6 From Semantics to Compilation

To apply functor-category semantics to intermediate code generation, in addition to taking Σ to be the ordered set of stack descriptors, one must change the meaning of the basic types. The translation of a phrase of type **compl** appropriate to the stack descriptor S is an instruction sequence in $\langle \mathrm{I}_S \rangle$. Thus

$$[\![\mathbf{compl}]\!]S = \langle \mathrm{I}_S \rangle\,.$$

The translation of a phrase of type **intcompl** is more complex: Roughly speaking, it is a function from righthand sides to instruction sequences (which one can think of as an instruction sequence containing a hole to be filled by a righthand side), but more precisely it is an exponential object in the functor category:

$$[\![\mathbf{intcompl}]\!] = \mathcal{R} \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{compl}]\!]\,,$$

where $\mathcal{R}$ is the functor such that

$$\mathcal{R}S = \langle \mathrm{R}_S \rangle\,.$$

The remaining types are defined by Equations (2) in exactly the same way as in the functor-category continuation semantics described earlier.

However, since Σ is a preorder (actually a total order) viewed as a category, the operation of exponentiation in $\mathcal{K}^\Sigma$ can be simplified. We have seen that, if

$$p \in (F \underset{\mathcal{K}}{\Longrightarrow} G)S = \mathrm{hom}_\Sigma S \times F \underset{\mathcal{K}}{\longrightarrow} G\,,$$

then

$$pS' \in (S \underset{\Sigma}{\longrightarrow} S') \times FS' \to GS'\,.$$

But the morphism set $S \underset{\Sigma}{\longrightarrow} S'$ contains a single morphism when $S \leq S'$ and is empty otherwise. Thus a simpler but equivalent condition is that pS' belongs to $FS' \to GS'$ when $S \leq S'$ and is the empty function otherwise. We indicate this by

$$p(S' \geq S) \in FS' \to GS'$$

or

$$p(S' \geq S)(x \in FS') \in GS'\,.$$

We are skirting over the requirement that both the morphisms in $\mathcal{K}$ and the members of $(F \underset{\mathcal{K}}{\Longrightarrow} G)S$ should be natural transformations. In fact, this requirement must be relaxed: Where naturality would require pairs of instruction sequences to be equal, we will only require them to have the

same denotational behavior, i.e. to denote the same functional continuation from states to final outputs. In some cases these instruction sequences will differ operationally, say by popping the stack at different steps.

A similar situation holds with regard to coherence. Just as with semantics, the translation $[\![p]\!]_{\pi\theta}$ is defined by structural induction on the proof of the typing $\pi \vdash p : \theta$. However, different proofs of the same typing are not required to lead to the same translation, but merely to translations with the same denotational behavior. Such translations may vary in the points where implicit conversions are invoked.

Except for completions, which are translated into instruction sequences, and integer variables, which are translated into acceptor-expression pairs, each phrase of the input language is translated into a functional value in the compiler that will be applied at compile-time to produce instruction sequences. Moreover, the type of the phrase in the input language will determine the type of its translation within the compiler. However, this categorical type discipline uses dependent function spaces that cannot be expressed in most languages (such as ML) in which the compiler might be written. In such languages, one must give translations a single type (e.g. a recursive functional data type in ML) that includes all the kinds of translations described above, as well as a variety of nonsensical translations that are guaranteed by the categorical discipline not to occur during any compilation.

7 Commands

If the phrase c has type **comm** under the type assignment π, then

$$[\![c]\!]_{\pi,\mathrm{comm}} \in [\![\pi]\!]^* \xrightarrow[\mathcal{K}]{} [\![\mathbf{comm}]\!] ,$$

so that

$$[\![c]\!]_{\pi,\mathrm{comm}} S(\eta \in [\![\pi]\!]^* S) \in [\![\mathbf{comm}]\!] S = ([\![\mathbf{compl}]\!] \Longrightarrow_{\mathcal{K}} [\![\mathbf{compl}]\!]) S ,$$

and thus

$$[\![c]\!]_{\pi,\mathrm{comm}} S(\eta \in [\![\pi]\!]^* S)(S' \geq S)(\kappa \in \langle \mathrm{I}_{S'} \rangle) \in \langle \mathrm{I}_{S'} \rangle .$$

Thus the translation of c is a function that accepts an environment η appropriate to the stack descriptor S and an instruction sequence κ appropriate to a possible larger stack descriptor S', and returns an instruction sequence appropriate to S'. In the absence of jumps, κ will describe the computation to be performed after c, so that the result of this function will be obtained by prefixing instructions for performing c to the sequence κ. We will call κ (like its semantic counterpart) a continuation.

The translation of **skip** returns its continuation argument κ without change:

$$[\![\mathbf{skip}]\!]_{\pi,\mathrm{comm}} S\eta S'\kappa = \kappa .$$

On the other hand, the translation of $c_1 \,;\, c_2$ first prefixes instructions for c_2 to κ, and then prefixes instructions for c_1. Put the other way round, the translation of $c_1 \,;\, c_2$ is the translation of c_1 using a continuation that is the translation of c_2 using the continuation κ that is to be performed after $c_1 \,;\, c_2$:

$$[\![c_1 \,;\, c_2]\!]_{\pi,\mathrm{comm}} S\eta S'\kappa = [\![c_1]\!]_{\pi,\mathrm{comm}} S\eta S'([\![c_2]\!]_{\pi,\mathrm{comm}} S\eta S'\kappa) .$$

The translation of assignment commands is described by an equation that is formally similar to the previous one:

$$[\![a := e]\!]_{\pi,\mathrm{comm}} S\eta S'\kappa = [\![e]\!]_{\pi,\mathrm{intexp}} S\eta S'([\![a]\!]_{\pi,\mathrm{intacc}} S\eta S'\kappa) ,$$

except for the typing, since the subphrase $[\![a]\!]S\eta S'\kappa$ belongs to $[\![\mathbf{intcompl}]\!]$ rather than $[\![\mathbf{compl}]\!]$. The value of this subphrase is an instruction sequence with a "hole", the result of $[\![e]\!]S\eta S'(\cdots)$ is obtained by filling this hole with a righthand side that will evaluate to the value of e, and then prefixing any instructions needed to set temporary variables appearing in the righthand side.

The close connection between this approach to compilation and functor-category semantics is exemplified by the fact that all three of the above equations for command translations are identical to the analogous semantic equations in functor-category continuation semantics. This pleasant situation occurs for a surprising number of language constructs. However, there must be exceptions — somewhere something must actually compute intermediate-language instructions. In fact, there are only three kinds of constructs whose translation is a nontrivial deviation from functor-category semantics: expressions, where temporary variables must be allocated, variable declarations, where program variables must be allocated, and closed and/or recursive procedure declarations, where calling sequences must be generated.

A similar situation holds for implicit conversions. Both the general Equation (1) for implicit conversions and the specific equations for the conversions from variables to acceptors and expressions:

$$[\![\mathbf{intvar} \leq \mathbf{intacc}]\!] S\langle a, e\rangle = a$$

$$[\![\mathbf{intvar} \leq \mathbf{intexp}]\!] S\langle a, e\rangle = e$$

are the same for translation as for semantics. However, if we extended our illustrative language to include a conversion from, say, integer to real expressions, then the corresponding equation for converting translations would explicitly describe the intermediate code for changing numerical representation.

8 Integer Expressions

If the phrase e has type **intexp** under the type assignment π, then

$$[\![e]\!]_{\pi,\mathrm{intexp}} S(\eta \in [\![\pi]\!]^* S)(S' \geq S)(\beta \in [\![\mathbf{intcompl}]\!] S') \in \langle \mathrm{I}_{S'} \rangle ,$$

where $\beta \in [\![\mathbf{intcompl}]\!] S'$ implies

$$\beta(S'' \geq S')(r \in \langle \mathrm{R}_{S''} \rangle) \in \langle \mathrm{I}_{S''} \rangle .$$

In essence, the translation of e must fill the hole in β by applying β to a righthand side r that will evaluate to the value of e, and then prefix to the resulting instruction sequence any instructions needed to set up temporary variables in r.

The translation of a constant, when given an integer continuation β, simply fills the "hole" in β by applying it to an appropriate literal:

$$[\![7]\!]_{\pi,\mathrm{intexp}} S\eta S'\beta = \beta S'(\mathbf{lit}\ 7) .$$

On the other hand, the translation of a unary expression such as $-e$ is obtained by applying the translation of the subexpression e to an altered integer continuation β':

$$[\![-e]\!]_{\pi,\mathrm{intexp}} S\eta S'\beta = [\![e]\!]_{\pi,\mathrm{intexp}} S\eta S'\beta' .$$

Here the effect of β' depends upon whether the righthand side r to which it is applied is a simple righthand side. If so, then $-r$ is a righthand side that evaluates to the value of $-e$ and contains the same temporary variables as r, so that the original β can be filled with $-r$:

$$\beta' S'' r = \beta S''(-r) \qquad \text{when } r \in \langle \mathrm{S}_{S''} \rangle \,.$$

Otherwise, however, $-r$ would contain more than one operator, so that a temporary variable must be used instead. Then the effect of β' is to fill β with the negation of the temporary, and to prefix to the resulting instruction sequence an assignment of r to the temporary:

$$\beta' S'' r = S^v := r[S'''_d - S''_d] \,;\, \beta S'''(-S^v) \quad \text{when } r \notin \langle \mathrm{S}_{S''} \rangle \,,$$

where

$$S^v = S' \qquad \text{and} \qquad S''' = S^v + 1 \,.$$

In the latter case, $[\![-e]\!]_{\pi,\mathbf{intexp}} S\eta S'\beta$ will give an instruction sequence of the form

$$\underset{S'}{\uparrow} \boxed{\begin{array}{c}\text{instructions}\\ \text{for setting}\\ \text{temporaries in } r\end{array}} \;;\; S^v := r[S'''_d - S''_d] \;;\; \beta S'''(-S^v) \,.$$
$$\qquad\qquad\qquad\qquad\qquad\quad \underset{S''}{\uparrow} \qquad\qquad\qquad \underset{S'''}{\uparrow}$$

It is important to understand the roles of the various stack descriptors here:

- S is appropriate to the environment η and is simply passed along with η. It describes a portion of the stack containing any program variables that may be accessed during the evaluation of $-e$.
- S' will be the current stack descriptor before $-e$ is evaluated. It may be larger than S since the computation to be done after evaluating $-e$ may refer to variables higher in the stack than the portion described by S.
- $S^v = S'$ describes the temporary variable used to store the value of e, which is placed on the stack immediately above the portion described by S'.
- S'' will be the current stack descriptor just before the assignment $S^v := r$. It may be larger than S' since the stack at this point may include temporaries occurring in r.
- $S''' = S^v + 1$ will be the current stack descriptor just after the assignment $S^v := r$. It is just large enough to include the temporary S^v.

Thus, the effect of the displacement increment $[S'''_d - S''_d]$ is to deallocate any temporaries occurring in r and allocate the temporary S^v.

The equation for $[\![-e]\!]$ can be written succinctly as

$$[\![-e]\!]_{\pi,\mathbf{intexp}} S\eta S'\beta = [\![e]\!]_{\pi,\mathbf{intexp}} S\eta S'(\text{usetmp } S'(\lambda S''.\ \lambda r.\ \beta S''(-r))) \,,$$

where usetmp is a function encapsulating the use of temporary variables:

$$\text{usetmp } S'\beta S'' r = \begin{cases} \beta S'' r & \text{when } r \in \langle \mathrm{S}_{S''} \rangle \\ S^v := r[S'''_d - S''_d] \,;\, \beta S''' S^v & \text{when } r \notin \langle \mathrm{S}_{S''} \rangle \,. \end{cases}$$

Similarly, as the reader may verify, the function usetmp can be used to handle temporary variables in binary expressions:

$$[\![e_1 + e_2]\!]_{\pi,\mathbf{intexp}} S\eta S'\beta = [\![e_1]\!]_{\pi,\mathbf{intexp}} S\eta S'(\text{usetmp } S'(\lambda S''.\ \lambda r_1.\ [\![e_2]\!]_{\pi,\mathbf{intexp}} S\eta S''(\text{usetmp } S''(\lambda S'''.\ \lambda r_2.\ \beta S'''(r_1 + r_2))))) \,.$$

9 Variable Declarations

The other construct for which translation involves storage allocation is the variable declaration. Suppose c is a command in which free occurrences of the identifier ι have type **intvar**. Then the translation of the command **new** ι: **intvar in** c gives an instruction sequence that allocates a new variable, initializes it (say, to zero), executes c, deallocates the new variable, and finally executes the continuation to be done after the whole command. Notice that the deallocation is done by an **adjustdisp** instruction prefixed to the continuation:

$$[\![\mathbf{new}\ \iota\mathbf{:intvar\ in}\ c]\!]_{\pi,\mathbf{comm}} S\eta(S' \geq S)(\kappa \in \langle \mathrm{I}_{S'} \rangle) = S^v := \mathbf{lit}\ 0\,[1] \,;\, [\![c]\!]_{[\pi|\iota:\mathbf{intvar}],\mathbf{comm}} S''\hat{\eta}S''(\mathbf{adjustdisp}[-1] \,;\, \kappa) \,.$$

Here

$$S^v = S' \qquad \text{and} \qquad S'' = S^v + 1 \,,$$

so that the new variable is placed just above the stack described by S', and S'' describes the extended stack containing this variable. The environment used to translate c is

$$\hat{\eta} = [\,[\![\pi]\!]^*(S \leq S'')\eta \mid \iota\mathbf{:}\langle a, e\rangle\,] \,,$$

which is the extension of η that maps ι into an acceptor-expression pair describing the new variable. (We will explain shortly why $[\![\pi]\!]^*(S \leq S'')\eta$ occurs here rather than η.)

The expression component $e \in [\![\mathbf{intexp}]\!]S''$ fills the hole in β with the stack descriptor S^v for the new variable:

$$eS'''\beta = \beta S''' S^v \,,$$

while the acceptor component $a \in [\![\mathbf{intacc}]\!]S''$ prefixes to its continuation κ' an assignment of a hole to S^v:

$$aS'''\kappa' S''''r = S^v := r[S'''_d - S''''_d] \,;\, \kappa' \,.$$

Notice that the functor-category discipline insures that S''' is larger than S'', so that the new variable lies within the stack described by S'''. The descriptor S'''' may be still larger, since it must include any temporaries occurring in r.

Since the environment $\hat{\eta}$ maps ι into $\langle a, e\rangle \in [\![\mathbf{intvar}]\!]S''$, it must belong to $[\![[\,\pi \mid \iota\mathbf{:intvar}\,]]\!]^*S''$. Thus it cannot be an extension of $\eta \in [\![\pi]\!]^*S$, but must be an extension of some related environment in $[\![\pi]\!]^*S''$. In fact, this environment is obtained by applying the "raising function" $[\![\pi]\!]^*(S \leq S'')$ to η.

When F is a functor that is the meaning of a type or type assignment, we write $F(S \leq S')$ for the application of the morphism part of F to the unique morphism in Σ from S to a larger stack descriptor S'. In general, such an application gives a function that serves to "raise" a translation or environment appropriate to S to a similar entity appropriate to S'. For the type **compl**, this operation prefixes

an appropriate adjustment of the stack to the instruction sequence that is its argument:

$$[\![\mathbf{compl}]\!](S \le S')(\kappa \in \langle \mathrm{I}_S \rangle) =$$
$$\begin{cases} \mathbf{adjustdisp}[S_d - S'_d]\,;\kappa & \text{when } S'_f = S_f \\ \mathbf{popto}\ S\,;\kappa & \text{otherwise}\,. \end{cases}$$

For exponentiations, the function $f \in (F \underset{\mathcal{K}}{\Longrightarrow} G)S$, whose domain is the set of stack descriptors greater than S, is restricted to the set of stack descriptors greater than S':

$$(F \underset{\mathcal{K}}{\Longrightarrow} G)(S \le S')f = f\,\lceil\{\, S'' \mid S'' \ge S' \,\}\,.$$

For products, the morphism parts are defined componentwise:

$$(F \times_{\mathcal{K}} G)(S \le S')\langle a, e\rangle = \langle F(S \le S')a, G(S \le S')e\rangle\,.$$

Similarly, the morphism parts of the meanings of type assignments, which are products of meanings of types over sets of identifiers, are also defined componentwise:

$$[\![\pi]\!]^*(S \le S')\eta\iota = [\![\pi\iota]\!](S \le S')(\eta\iota)\,.$$

10 Boolean Expressions and Conditionals

It would be straightforward to translate boolean expressions in the same manner as integer expressions. However, it is more interesting, and in most cases more efficient, to provide a "control-flow" translation, in which boolean expressions are compiled into trees of branch instructions.

To describe this approach, we extend our illustrative language with the new types **boolexp** and **boolcompl**. The meaning of boolean expressions is defined analogously to that of integer expressions:

$$[\![\mathbf{boolexp}]\!] = [\![\mathbf{boolcompl}]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{compl}]\!]\,,$$

but boolean completions are defined quite differently:

$$[\![\mathbf{boolcompl}]\!] = [\![\mathbf{compl}]\!] \times [\![\mathbf{compl}]\!]\,.$$

The translation of a boolean expression b accepts a pair $\langle\kappa, \bar\kappa\rangle$ of continuations and produces an instruction sequence that branches to κ when b is true or to $\bar\kappa$ when b is false.

In this approach the translation of constants is trivial:

$$[\![\mathbf{true}]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle = \kappa$$

$$[\![\mathbf{false}]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle = \bar\kappa\,,$$

while the translation of relations gives rise to test instructions, with temporary variables being handled in the same way as with binary arithmetic operations:

$$[\![e_1 \le e_2]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle =$$
$$[\![e_1]\!]_{\pi,\mathbf{intexp}} S\eta S'(\text{usetmp } S'(\lambda S''.\ \lambda r_1.$$
$$[\![e_2]\!]_{\pi,\mathbf{intexp}} S\eta S''(\text{usetmp } S''(\lambda S'''.\ \lambda r_2.$$
$$\mathbf{if}\ r_1 \le r_2[S'_d - S'''_d]\ \mathbf{then}\ \kappa\ \mathbf{else}\ \bar\kappa))))\,.$$

On the other hand, the translations of boolean operations and conditional commands simply compose or rearrange the trees produced by subexpressions:

$$[\![\mathbf{not}\ b]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle = [\![b]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\bar\kappa,\kappa\rangle$$

$$[\![b_1\ \mathbf{or}\ b_2]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle =$$
$$[\![b_1]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa, [\![b_2]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle\kappa,\bar\kappa\rangle\rangle$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]_{\pi,\mathbf{comm}} S\eta S'\kappa =$$
$$[\![b]\!]_{\pi,\mathbf{boolexp}} S\eta S'\langle [\![c_1]\!]_{\pi,\mathbf{comm}} S\eta S'\kappa, [\![c_2]\!]_{\pi,\mathbf{comm}} S\eta S'\kappa\rangle\,.$$

Notice that the second equation describes "short-circuit" evaluation for **or**.

11 Open Procedures

The functor-category semantics of the lambda-calculus aspects of Algol-like languages is described by the following semantic equations, which are determined by the cartesian closed nature of $\mathcal{K}$ and the definition of **let** $\iota \equiv p$ **in** p' by the redex $(\lambda\iota.\ p')p$:

$$[\![\iota]\!]_{\pi,\pi\iota} S\eta = \eta\iota$$

$$[\![p_1 p_2]\!]_{\pi\theta'} S\eta = [\![p_1]\!]_{\pi,\theta\to\theta'} S\eta S([\![p_2]\!]_{\pi\theta} S\eta)$$

$$[\![\lambda\iota{:}\,\theta.\ p]\!]_{\pi,\theta\to\theta'} S\eta S'a = [\![p]\!]_{[\pi|\iota\cdot\theta],\theta'} S'[\,[\![\pi]\!]^*(S \le S')\eta \mid \iota{:}\,a\,]$$

$$[\![\mathbf{let}\ \iota \equiv p\ \mathbf{in}\ p']\!]_{\pi\theta'} S\eta = [\![p']\!]_{[\pi|\iota\cdot\theta],\theta'} S[\,\eta \mid \iota{:}\,[\![p]\!]_{\pi\theta} S\eta\,]\,.$$

There is also an equation for implicit conversion from one procedural type to another:

$$[\![\theta_1 \to \theta_2 \le \theta'_1 \to \theta'_2]\!] S(f \in [\![\theta_1 \to \theta_2]\!] S)$$
$$(S' \ge S)(a \in [\![\theta'_1]\!] S') =$$
$$[\![\theta_2 \le \theta'_2]\!] S'(fS'([\![\theta'_1 \le \theta_1]\!] S'a))$$
$$\text{when } \theta'_1 \le \theta_1 \text{ and } \theta_2 \le \theta'_2\,.$$

These equations all carry over to compilation, where they describe the translation of (nonrecursive) procedures into open or "inline" code. (They have been written above in the simplified form that is appropriate when Σ is a partial order.)

Essentially, these equations describe a compiler where the lambda-calculus aspects of the source language are completely reduced at compile-time, leaving target code that is purely imperative. This is in pleasant contrast with conventional approaches to compiling conventional languages, where inline implementation of procedures is notoriously hard to get right.

12 Closed Subroutines

Closed subroutines are necessary for the implementation of procedures, and other types of phrases, that are defined recursively. (It would be straightforward to also provide a "**letclosed**" definition for nonrecursive entities that are to be implemented by closed subroutines.)

We use the term *procedure* for lambda expressions and their meanings, and the term *subroutine* or *closed subroutine* for instruction sequences that may be *called* from several points in the intermediate-code program. It is important

to distinguish these concepts for two reasons: As we have already seen, a procedure may be implemented by inline expansion rather than by a subroutine, and on the other hand, because of the use of call by name, a phrase such as a command or an expression, even though it is not a procedure, will be implemented by a subroutine if it is defined by a recursive definition or is a parameter to a procedure that is implemented by a subroutine.

Similarly, we will distinguish between *parameters*, which are source-language phrases passed to procedures, and *arguments*, which are instruction sequences (actually subroutines) passed to subroutines.

While procedures are classified by types, subroutines are classified by *simple types*:

$$\langle \text{simple type} \rangle ::= \mathbf{compl} \mid \mathbf{intcompl} \mid \langle \text{simple type} \rangle \to \langle \text{simple type} \rangle$$

The exact connection between types and simple types will be explained in Section 14. Roughly speaking, however, a simple type is obtained from a type by replacing

$$\begin{array}{rcl} \mathbf{comm} & \text{by} & \mathbf{compl} \to \mathbf{compl} \\ \mathbf{intexp} & \text{by} & \mathbf{intcompl} \to \mathbf{compl} \\ \mathbf{intacc} & \text{by} & \mathbf{compl} \to \mathbf{intcompl} \,. \end{array}$$

For example, a procedure of type

$$\theta_1 \to (\cdots(\theta_n \to \mathbf{comm})\cdots)$$

would be implemented by a subroutine of simple type

$$\varphi_1 \to (\cdots(\varphi_n \to (\mathbf{compl} \to \mathbf{compl}))\cdots)\,,$$

(where each φ_i is the simple type corresponding to θ_i), which would accept $n+1$ arguments. The extra argument of simple type **compl** plays the role of a return address.

Thus in the $n = 0$ case, even a command is implemented by a subroutine accepting a "return address" argument of simple type **compl**. However, such an argument, or any other subroutine of simple type **compl**, accepts no arguments.

Calling a subroutine is a more complex operation than merely jumping to an instruction sequence, since it may be necessary to switch context from a frame list appropriate to the calling program to a frame list appropriate to the subroutine, and since arguments may be passed by placing them in a vector accessible from the new frame list. In general, to call a subroutine one must specify

1. the subroutine to be called,
2. the *global* frame list to be used during execution of the subroutine,
3. a list of the arguments, which are themselves subroutines.

To execute the call, when there are one or more arguments, one switches context to a frame list that is formed by adding a new frame (allocated on top of the current stack) to the global frame list, and then sends control to the subroutine. The context switch causes the register SR to increase by an amount δ that is the current frame displacement S_d^{curr} at the time of the call.

The new frame contains two words: the lower word points to the global frame list, while the upper word points to a *call block*, which in turn contains the argument list and the quantity δ. The latter is the distance between the base of the frame pointing to the call block and the old frame list, which is in turn the global frame list to be used when calling the arguments.

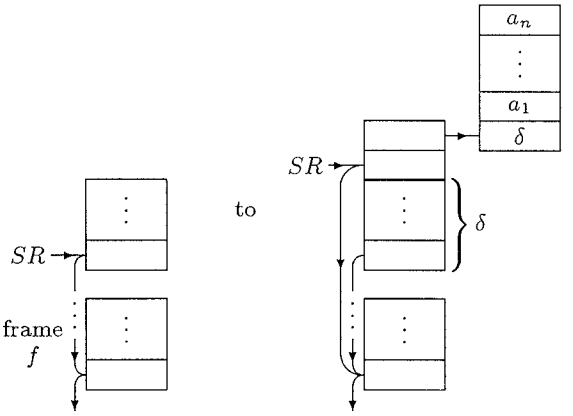
Notice that the contents of the call block are the same for all executions of a particular call, and are known at compile-time. Thus a single copy of the call block can be placed in code space, rather than placing multiple copies (in the case of a recursive call) on the stack.

It is important to distinguish the special case when the subroutine being called takes no arguments, since in this case the information in the new frame is vacuous, so that it is more efficient simply to take the new frame list to be the global frame list. This will remove from the stack everything above the most recent frame of the global frame list, but it is easy to see that this data is inaccessible (since all pointers in the stack point downward). Indeed this is the "stack pop" that occurs when a subroutine passes control to its return address.

The specific method of achieving all this depends upon whether the subroutine being called is the result of compiling a definition, or is an argument to a subroutine containing the call. In the first case, the identity of the subroutine being called is known at compile time, and its global frame list is a tail of the current frame list. If there are arguments, then the call is performed by executing the instruction sequence

$$\mathbf{call}\ i\ f\ (a_1, \ldots, a_n)\,,$$

where i is the subroutine being called, f is the frame count of the (top frame of the) global frame list, and $a_1, \ldots, a_n$ are the arguments. This instruction sequence changes

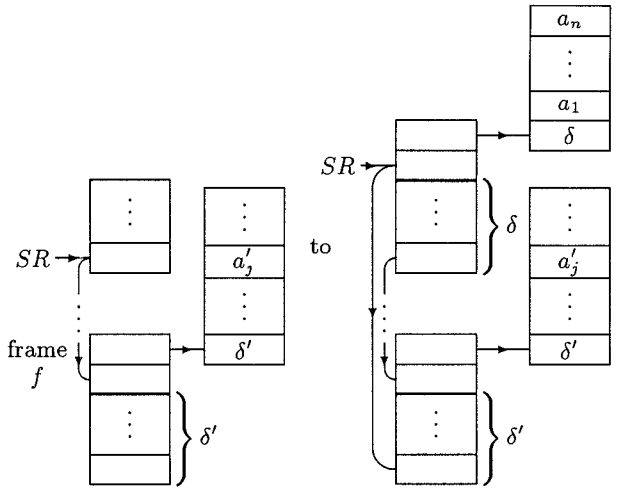


and sends control to i. If the subroutine being called takes no arguments, however, then in place of **call** i f () one simply resets SR to point to frame f and sends control to i.

On the other hand, if the subroutine being called is an argument then it will occur, say in position j, in a call block pointed to by a frame, say with frame count f, in the current frame list, and the relevant global frame list will also be pointed to by the frame f. If the subroutine being called takes arguments, then the call is performed by the instruction sequence

$$\mathbf{acall}\ j\ f\ (a_1, \ldots, a_n)\,,$$

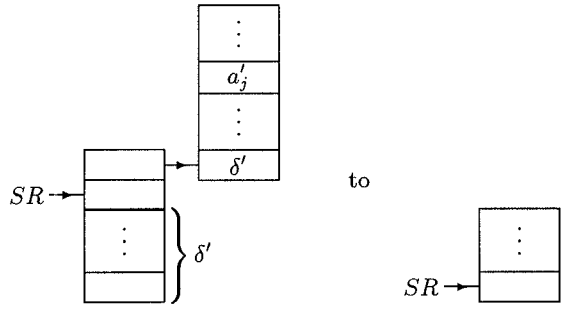
which changes



and sends control to a'_j. If the subroutine being called takes no arguments, however, then in place of **acall** j f () one resets SR to point to frame f and then executes

$$\textbf{ajump}\ j\,,$$

which changes



and sends control to a'_j.

In summary, three new forms of instruction sequence have been introduced for calling subroutines:

$$\begin{aligned}\langle \mathrm{I}_S\rangle ::= {}& \textbf{call}\,\langle \mathrm{I}_{f+}\rangle\, f\,(\langle \mathrm{SR}_S^{\varphi 1}\rangle,\ldots,\langle \mathrm{SR}_S^{\varphi n}\rangle)\\ \mid {}& \textbf{acall}\ j\ f\,(\langle \mathrm{SR}_S^{\varphi 1}\rangle,\ldots,\langle \mathrm{SR}_S^{\varphi n}\rangle)\\ \mid {}& \textbf{ajump}\ j\end{aligned}$$

where $f \le S_f$, $f^+ = \langle f+1, 3\rangle$, $n \ge 1$, and $j \ge 1$. Here SR_S^φ denotes a subroutine of simple type φ whose global frame list is described by the stack descriptor S:

$$\begin{aligned}\langle \mathrm{SR}_S^\varphi\rangle ::= {}& \langle \mathrm{I}_S\rangle \qquad \text{when } \varphi \in \{\textbf{compl}, \textbf{intcompl}\}\\ \langle \mathrm{SR}_S^\varphi\rangle ::= {}& \langle \mathrm{I}_{S^+}\rangle \qquad \text{otherwise}\,,\end{aligned}$$

where $\langle S_f, S_d\rangle^+ = \langle S_f + 1, 3\rangle$.

In passing, we note that the calling conventions described here differ from those used in traditional Algol compilers [13, 14] in that all arguments to a subroutine are associated with the same global frame list, instead of each argument having its own global frame list. The advantage of our approach is that it reduces the amount of stack storage used to call subroutines. The disadvantage is that, when a recursive call has an actual parameter that is identical with the corresponding formal parameter, e.g. the parameter x in

$$\textbf{letrec}\ \mathsf{p} \equiv \lambda \mathsf{n}.\ \lambda \mathsf{x}.\ \cdots\ \mathsf{p}\,(\mathsf{n}-1)\,\mathsf{x}\,\cdots\ \textbf{in}\ \cdots\,,$$

an nth-level evaluation of the parameter will invoke a chain of n subroutines. However, this inefficiency can be avoided if the programmer (or an optimizing compiler) eliminates the parameter by using a global identifier.

In any event, although we have not pursued the matter, we expect that more traditional calling conventions should also be expressible within the functor-category framework.

13 Compiling Subroutines and their Calls

Subroutines and their calls are compiled by three families of functions indexed by simple types. The translation of a phrase is mapped into a subroutine by

$$\text{mk-subr}_\varphi S \in [\![\varphi]\!]S \to \langle \mathrm{SR}_S^\varphi\rangle\,,$$

while a subroutine is mapped into a call (more precisely into a function that, when applied to appropriate arguments yields a call) by

$$\text{mk-call}_\varphi S \in \langle \mathrm{SR}_S^\varphi\rangle \to [\![\varphi]\!]S\,.$$

Finally, there is a family of functions that produce calls of arguments:

$$\text{mk-argcall}_\varphi S \in \mathcal{N} \to [\![\varphi]\!]S\,,$$

where $\mathcal{N}$ denotes the set of positive integers. Specifically, $\text{mk-argcall}_\varphi S\, j$ gives a call of an argument of simple type φ that is the jth argument in the call block accessed from the top frame of the frame list described by S.

These three families of functions are defined by mutual induction on simple types, reflecting the fact that compiling a subroutine involves compiling calls of its arguments, and compiling a call involves compiling subroutines for arguments:

$$\begin{aligned}\text{mk-subr}_{\mathbf{compl}} S\,\kappa &= \kappa\\ \text{mk-call}_{\mathbf{compl}} S\, i &= i\\ \text{mk-argcall}_{\mathbf{compl}} S\, j &= \textbf{ajump}\ j\,,\end{aligned}$$

and when $\varphi = \varphi_1 \to (\cdots \to (\varphi_n \to \mathbf{compl})\cdots)$ for some $n \ge 1$:

$$\begin{aligned}&\text{mk-subr}_\varphi S(c \in [\![\varphi]\!]S) =\\ &\quad c\,S^+(\text{mk-argcall}_{\varphi_1} S^+1)\cdots S^+(\text{mk-argcall}_{\varphi_n} S^+n)\\ &\text{mk-call}_\varphi S\, i\, S^1(a_1 \in [\![\varphi_1]\!]S^1)\cdots S^n(a_n \in [\![\varphi_n]\!]S^n) =\\ &\quad \textbf{call}\ i\ S_f\\ &\qquad \big(\text{mk-subr}_{\varphi_1} S^n([\![\varphi_1]\!](S^1 \le S^n)a_1),\\ &\qquad\quad \vdots\\ &\qquad \text{mk-subr}_{\varphi_n} S^n([\![\varphi_n]\!](S^n \le S^n)a_n)\big)\\ &\text{mk-argcall}_\varphi S\, j\, S^1(a_1 \in [\![\varphi_1]\!]S^1)\cdots S^n(a_n \in [\![\varphi_n]\!]S^n) =\\ &\quad \textbf{acall}\ j\ S_f\\ &\qquad \big(\text{mk-subr}_{\varphi_1} S^n([\![\varphi_1]\!](S^1 \le S^n)a_1),\\ &\qquad\quad \vdots\\ &\qquad \text{mk-subr}_{\varphi_n} S^n([\![\varphi_n]\!](S^n \le S^n)a_n)\big)\,.\end{aligned}$$

Further equations deal with the cases where φ ends in **intcompl** rather than **compl**, which arise when function procedures or expressions are compiled into closed subroutines. Here the special register **sbrs** is used to transmit integer results. Let saveres $\in [\![\mathbf{intcompl}]\!] \underset{\mathcal{K}}{\Longrightarrow} [\![\mathbf{compl}]\!]$ be the function such that

$$\text{saveres}\, S\, \beta = S^v := \mathbf{sbrs}\,[S'_d - S_d]\,;\, \beta\, S'\, S^v\,,$$

where

$$S^v = S \qquad \text{and} \qquad S' = S^v + 1\,.$$

Then

$$\text{mk-subr}_{\text{intcompl}} S\, \beta = \text{saveres}\, S\, \beta$$

$$\text{mk-call}_{\text{intcompl}} S\, i\, S'\, r = \mathbf{sbrs} := r\,[S_d - S'_d]\,;\, i$$

$$\text{mk-argcall}_{\text{intcompl}} S\, j\, S'\, r = \mathbf{sbrs} := r\,[S_d - S'_d]\,;\, \mathbf{ajump}\ j\,,$$

and when $\varphi = \varphi_1 \to (\cdots \to (\varphi_n \to \mathbf{intcompl})\cdots)$ for some $n \geq 1$:

$$\text{mk-subr}_\varphi S(\beta \in [\![\varphi]\!]S) = \text{saveres}\, S^+\big(\beta\, S^+(\text{mk-argcall}_{\varphi_1} S^+1)\cdots S^+(\text{mk-argcall}_{\varphi_n} S^+n)\big)$$

$$\text{mk-call}_\varphi S\, i\, S^1(a_1 \in [\![\varphi_1]\!]S^1)\cdots S^n(a_n \in [\![\varphi_n]\!]S^n)S'(r \in \langle \mathrm{R}_{S'}\rangle) =$$

$$\mathbf{sbrs} := r\,[S^n_d - S'_d]\,;\ \mathbf{call}\ i\ S_f\ \big(\text{mk-subr}_{\varphi_1} S^n([\![\varphi_1]\!](S^1 \leq S^n)a_1),\ \vdots\ \text{mk-subr}_{\varphi_n} S^n([\![\varphi_n]\!](S^n \leq S^n)a_n)\big)$$

$$\text{mk-argcall}_\varphi S\, j\, S^1(a_1 \in [\![\varphi_1]\!]S^1)\cdots S^n(a_n \in [\![\varphi_n]\!]S^n)S'(r \in \langle \mathrm{R}_{S'}\rangle) =$$

$$\mathbf{sbrs} := r\,[S^n_d - S'_d]\,;\ \mathbf{acall}\ j\ S_f\ \big(\text{mk-subr}_{\varphi_1} S^n([\![\varphi_1]\!](S^1 \leq S^n)a_1),\ \vdots\ \text{mk-subr}_{\varphi_n} S^n([\![\varphi_n]\!](S^n \leq S^n)a_n)\big)\,.$$

Using these functions, it is straightforward to translate recursive definitions:

$$[\![\mathbf{letrec}\ \iota \equiv p\ \mathbf{in}\ p']\!]_{\pi\theta'} S\eta = [\![p']\!]_{[\pi|\iota:\varphi],\theta'} S\eta'\,,$$

where

$$\eta' = [\,\eta \mid \iota\colon \text{mk-call}_\varphi S i\,]$$
$$i = \text{mk-subr}_\varphi S([\![p]\!]_{[\pi|\iota:\varphi],\varphi} S\eta')\,.$$

Here η' and i are mutual fixed points. This can be represented in the compiler by making the instruction sequence i a loop, e.g. by making the i-field of the **call** instruction generated by mk-call a reference whose value is eventually set to i.

The reader may wonder why the contents of the register **sbrs** is stored in a temporary variable immediately upon return from a subroutine that produces a result. The reason, illustrated by the integer expression

$$\mathbf{letrec}\ \mathsf{x} \equiv \cdots\ \mathbf{in}\ \mathbf{letrec}\ \mathsf{y} \equiv \cdots\ \mathbf{in}\ \mathsf{x} + \mathsf{y}\,,$$

is that **sbrs** may be reset by later calls before its contents is used.

14 Subroutines for Product Types

The method for compiling closed subroutines described in the previous section does not deal with integer variables or with boolean expressions. Specifically, the mapping from types to simple types does not apply to types containing **intvar**, **boolcompl**, or **boolexp**.

The key to filling this lacuna is that the meanings of a type such as

$$\theta_1 \to (\cdots(\theta_n \to \mathbf{intvar})\cdots)$$

are isomorphic to pairs of meanings of the types

$$\theta_1 \to (\cdots(\theta_n \to \mathbf{intacc})\cdots)$$

$$\theta_1 \to (\cdots(\theta_n \to \mathbf{intexp}) \quad)\,.$$

Thus a procedure of the first type can be implemented by a pair of subroutines corresponding to the second and third types.

The general situation is that a type is mapped into a *sequence* of simple types by the function Γ such that

$$\Gamma\,\mathbf{compl} = \mathbf{compl}$$

$$\Gamma\,\mathbf{intcompl} = \mathbf{intcompl}$$

$$\Gamma\,\mathbf{comm} = \mathbf{compl} \to \mathbf{compl}$$

$$\Gamma\,\mathbf{intexp} = \mathbf{intcompl} \to \mathbf{compl}$$

$$\Gamma\,\mathbf{intacc} = \mathbf{compl} \to \mathbf{intcompl}$$

$$\Gamma\,\mathbf{intvar} = \mathbf{compl} \to \mathbf{intcompl},\ \mathbf{intcompl} \to \mathbf{compl}$$

$$\Gamma\,\mathbf{boolcompl} = \mathbf{compl},\ \mathbf{compl}$$

$$\Gamma\,\mathbf{boolexp} = \mathbf{compl} \to \mathbf{compl} \to \mathbf{compl}\,,$$

and if

$$\Gamma\,\theta = \varphi_1, \ldots, \varphi_m \qquad \text{and} \qquad \Gamma\,\theta' = \varphi'_1, \ldots, \varphi'_n$$

then

$$\Gamma(\theta \to \theta') = \varphi_1 \to \cdots \to \varphi_m \to \varphi'_1,\ \ldots\,,\ \varphi_1 \to \cdots \to \varphi_m \to \varphi'_n\,.$$

Although the details are too tedious to record in this paper, it is straightforward to define two functions $\Phi_\theta S$ and $\Psi_\theta S$ such that, when $\Gamma\,\theta = \varphi_1, \ldots, \varphi_n$,

$$[\![\theta]\!]S \underset{\Psi_\theta S}{\overset{\Phi_\theta S}{\rightleftarrows}} [\![\varphi_1]\!]S \times \cdots \times [\![\varphi_n]\!]S$$

is an isomorphism.

When $\Gamma\,\theta = \varphi_1, \ldots, \varphi_n$, this isomorphism can be used to define the compilation of a phrase p of type θ into a collection of n subroutines of simple types $\varphi_1, \ldots, \varphi_n$. In place of the equations at the end of the previous section, we have:

$$[\![\mathbf{letrec}\ \iota \equiv p\ \mathbf{in}\ p']\!]_{\pi\theta'} S\eta = [\![p']\!]_{[\pi|\iota:\theta],\theta'} S\eta'\,,$$

where

$$\eta' = [\,\eta \mid \iota\colon \Psi_\theta S(\text{mk-call}_{\varphi_1} S i_1, \ldots, \text{mk-call}_{\varphi_n} S i_n)\,]$$

$$i_1 = \text{mk-subr}_{\varphi_1} S\, a_1$$

$$\vdots$$

$$i_n = \text{mk-subr}_{\varphi_n} S\, a_n$$

$$(a_1, \ldots, a_n) = \Phi_\theta S([\![p]\!]_{[\pi \mid \iota:\theta],\theta} S\eta')\,.$$

Here η' and $i_1, \ldots, i_n$ are mutual fixed points that can be represented in the compiler by using a reference for each of the instruction sequences $i_1, \ldots, i_n$.

It is clear that the method outlined in this section could also be used to deal with source language types, such as record or object types, that are defined by products. To treat binary products, for example, one would take

$$\Gamma(\theta \times \theta') = (\Gamma\theta) \circ (\Gamma\theta')\,,$$

where $\circ$ denotes concatenation of sequences of simple types.

15 Completions and Iteration

Although the type **compl** plays a major role in our approach to compilation, it does not occur in the original type structure of the illustrative language. In contrast, the Algol-like language Forsythe [9] includes phrases of type **compl** (though not **intcompl** or **boolcompl**) that provide a capability similar to **goto**'s and labels in Algol 60.

To extend our illustrative language similarly, we introduce the type **compl** into the language, as a subtype of **comm**, with the implicit conversion

$$[\![\textbf{compl} \le \textbf{comm}]\!] S\kappa S'\kappa' = [\![\textbf{compl}]\!](S \le S')\kappa\,.$$

A completion can be constructed by joining a command and a completion with a sequencing operator:

$$[\![c_1\,;\,c_2]\!]_{\pi,\text{compl}} S\eta = [\![c_1]\!]_{\pi,\text{comm}} S\eta S([\![c_2]\!]_{\pi,\text{compl}} S\eta)\,,$$

or by joining two completions with a conditional:

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]_{\pi,\text{compl}} S\eta =$$
$$[\![b]\!]_{\pi,\text{boolexp}} S\eta S\langle [\![c_1]\!]_{\pi,\text{compl}} S\eta, [\![c_2]\!]_{\pi,\text{compl}} S\eta\rangle\,.$$

We also provide an **escape** command that binds an identifier to a completion that causes an exit from the **escape** command:

$$[\![\textbf{escape } \iota \textbf{ in } c]\!]_{\pi,\text{comm}} S\eta S'\kappa =$$
$$[\![c]\!]_{[\pi \mid \iota:\text{compl}],\text{comm}} S'[\,[\![\pi]\!]^*(S \le S')\eta \mid \iota\colon \kappa\,]S'\kappa\,.$$

All of these equations are the same for translation as for semantics.

Once the language includes completions, various iterative constructs can be defined a syntactic sugar. A trivial example is a recursive definition of the completion **loop** c that repeats the command c ad infinitum (until c executes an escape):

$$\textbf{loop } c \stackrel{\text{def}}{=} \textbf{letrec } \mathsf{k} \equiv c\,;\,\mathsf{k} \textbf{ in } \mathsf{k}\,,$$

where k is an identifier not occuring free in c. By substituting this syntactic definition into the translation equations, and using the definitions of mk-subr and mk-call, one obtains the translation

$$[\![\textbf{loop } c]\!]_{\pi,\text{compl}} S\eta = i \qquad \text{where } i = [\![c]\!]_{\pi,\text{comm}} S\eta S i\,.$$

A less trivial example is the **while** command:

$$\textbf{while } b \textbf{ do } c \stackrel{\text{def}}{=} \textbf{escape e in}$$
$$\textbf{letrec k} \equiv \textbf{if } b \textbf{ then } (c\,;\,\mathsf{k}) \textbf{ else e in k}\,,$$

where e and k are identifiers not occuring free in b or c. This definition leads to the translation

$$[\![\textbf{while } b \textbf{ do } c]\!]_{\pi,\text{comm}} S\eta S'\kappa = i\,,$$

where

$$i = [\![b]\!]_{\pi,\text{boolexp}} S'([\![\pi]\!]^*(S \le S')\eta) S'$$
$$\langle [\![c]\!]_{\pi,\text{comm}} S'([\![\pi]\!]^*(S \le S')\eta) S' i, \kappa\rangle\,.$$

This is a correct translation, but

$$i = [\![b]\!]_{\pi,\text{boolexp}} S\eta S'\langle [\![c]\!]_{\pi,\text{comm}} S\eta S' i, \kappa\rangle$$

gives a simpler translation with the same denotational behavior that sometimes pops the stack sooner.

16 An Example

The following example illustrates many of the aspects of translation discussed in this paper. Let P be the command

new x: **intvar in**

letrec pr $\equiv \lambda$c: **comm.** (c ; x := x + 1 ;

if x $\le$ 10 **then** pr(c ; c) **else skip**)

in new y: **intvar in** pr(y := y + x × x)\,.

Then $[\![P]\!]_{[\,],\text{comm}}$, applied to appropriate arguments for translating a complete program, is

$$[\![P]\!]_{[\,],\text{comm}}\langle 0,0\rangle\,[\,]\,\langle 0,0\rangle\,\textbf{stop} =$$
$$\underset{\langle 0,0\rangle}{\langle 0,0\rangle} := \textbf{lit } 0\,[1]\,;\ \underset{\langle 0,1\rangle}{\langle 0,1\rangle} := \textbf{lit } 0\,[1]\,;$$
$$\textbf{call } i\ 0\ \Big(\underset{\langle 1,3\rangle}{\langle 1,3\rangle} := \langle 0,0\rangle \times \langle 0,0\rangle\,[1]\,;$$
$$\underset{\langle 1,4\rangle}{\langle 0,1\rangle} := \langle 0,1\rangle + \langle 1,3\rangle\,[-1]\,;\ \textbf{ajump } 1,\ \underset{\langle 1,3\rangle}{}$$
$$\underset{\langle 0,2\rangle}{\textbf{adjustdisp}}[-1]\,;\ \underset{\langle 0,1\rangle}{\textbf{adjustdisp}}[-1]\,;\ \underset{\langle 0,0\rangle}{\textbf{stop}}\Big)\,,$$

where i is

$$\underset{\langle 1,3\rangle}{\textbf{acall}}\ 1\ 1\ \Big(\underset{\langle 1,3\rangle}{\langle 0,0\rangle} := \langle 0,0\rangle + \textbf{lit } 1\,[0]\,;$$
$$\underset{\langle 1,3\rangle}{\textbf{if } \langle 0,0\rangle} \le \textbf{lit } 10\,[0] \textbf{ then}$$
$$\textbf{call } i\ 0\ \Big(\underset{\langle 1,3\rangle}{\textbf{acall}}\ 1\ 1\ (\underset{\langle 2,3\rangle}{\textbf{acall}}\ 1\ 1\ (\underset{\langle 2,3\rangle}{\textbf{ajump}}\ 1)),$$
$$\underset{\langle 1,3\rangle}{\textbf{ajump}}\ 2\Big)$$
$$\textbf{else } \underset{\langle 1,3\rangle}{\textbf{ajump}}\ 2\Big)\,.$$

Under each instruction sequence in the above displays, we have placed the stack descriptor that will be current when the sequence begins execution.

17 Conclusions

In the spring of 1994, the basic approach described here was used in an undergraduate compiling course to construct an intermediate code generator, written in Standard ML, for a simple Algol-like language. In the coming months, we hope to extended it for use in implementing the language Forsythe.

A major research question is to what extent the approach can be extended to generate more efficient intermediate-language code. We suspect that a richer form of stack descriptor can be devised that will provide information about the caching of variables in registers and the use of displays (groups of registers pointing directly to active frames). On the other hand, the approach depends so heavily on the use of continuations that it may be difficult to vary evaluation order (say, by interleaving the evaluation of subexpressions) with sufficient flexibility, especially for RISC machines.

A second question is whether the approach will lend itself to a proof of compiler correctness. One would expect that the close connections between functor-category semantics and our approach to code generation would lead to simple proofs of the relationship between semantics and compilation. However, we have not yet pursued this topic beyond intuitive arguments.

References

[1] Jones, N. D. and Schmidt, D. A. *Compiler Generation from Denotational Semantics.* in: **Semantics-Directed Compiler Generation**, Proceedings of a Workshop, Aarhus, Denmark, January 14–18, edited by N. D Jones. **Lecture Notes in Computer Science**, vol. 94, Springer-Verlag, Berlin, 1980, pp. 70–93.

[2] Steele Jr., G. L. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).* no. AI–TR–474, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1978, iii+272 pp.

[3] Wand, M. *Deriving Target Code as a Representation of Continuation Semantics.* **ACM Transactions on Programming Languages and Systems**, vol. 4 (1982), pp. 496–517.

[4] Reynolds, J. C. *The Essence of Algol.* in: **Algorithmic Languages**, Proceedings of the International Symposium on Algorithmic Languages, Amsterdam, October 26–29, edited by J. W. de Bakker and J. C. van Vliet. North-Holland, Amsterdam, 1981, pp. 345–372.

[5] Oles, F. J. *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph. D. Dissertation. Syracuse University, August 1982, vi+240 pp.

[6] Oles, F. J. *Type Algebras, Functor Categories, and Block Structure.* in: **Algebraic Methods in Semantics**, edited by M. Nivat and J. C. Reynolds. Cambridge University Press, Cambridge, England, 1985, pp. 543–573.

[7] Morris, F. L. *Correctness of Translations of Programming Languages — An Algebraic Approach*, Ph. D. Dissertation. Stanford University, August 1972.

[8] Reynolds, J. C. *Conjunctive Types and Algol-like Languages (Abstract of Invited Lecture).* in: **Proceedings Symposium on Logic in Computer Science**, Ithaca, New York, June 22–25. 1987, p. 119.

[9] Reynolds, J. C. *Preliminary Design of the Programming Language Forsythe.* Report, no. CMU–CS–88–159, Carnegie Mellon University, Computer Science Department, June 21, 1988.

[10] Reynolds, J. C. *The Coherence of Languages with Intersection Types.* in: **Theoretical Aspects of Computer Software**, International Conference TACS '91, Proceedings, Sendai, Japan, September 24–27, 1991, edited by T. Ito and A. R. Meyer. **Lecture Notes in Computer Science**, vol. 526, Springer-Verlag, Berlin, 1991, pp. 675–700.

[11] Scott, D. S. *The Lattice of Flow Diagrams.* in: **Symposium on Semantics of Algorithmic Languages**, edited by E. Engeler. **Lecture Notes in Mathematics**, vol. 188, Springer-Verlag, Berlin, 1971, pp. 311–366.

[12] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. *Initial Algebra Semantics and Continuous Algebras.* **Journal of the ACM**, vol. 24 (1977), pp. 68–95.

[13] Dijkstra, E. W. *Recursive Programming.* **Numerische Mathematik**, vol. 2 (1960), pp. 312–318.

[14] Naur, P. *The Design of the GIER ALGOL Compiler, Part I.* **BIT**, vol. 3 (1963), pp. 124–140. *Reprinted in Goodman, Richard, editor, Annual Review in Automatic Programming, Vol. 4, Pergamon Press, Oxford (1964) 49–85.*