In Memoriam
John C. Reynolds
Robert Harper
Carnegie Mellon University
John C. Reynolds
“The Compiler”

- In 1989 John taught the undergraduate compilers class using Twosythe, a cut-down Forsythe.
- John had to have emergency heart surgery (successfully).
“The Compiler”

- It fell to Shai Geva and me to try to recover what John was doing.
- Even with help from Andrzej Filinski, we failed to make sense of it.
- I felt very guilty, until I visited John in the hospital a few days later ...
“Hell on Wheels”

- Elegant use of categorial semantics.

- Type of compiled code is determined by the type of the code being compiled!

\[
\text{mk-argcall}_{\varphi_1} S j S^1(a_1 \in [\varphi_1] S^1) \cdots
\]

\[
\text{mk-call}_{\varphi_i} S i S^1(a_1 \in [\varphi_i] S^1) \cdots
\]

\[
S^n(a_n \in [\varphi_n] S^n) S^r(r \in (R_S)) =
\]

\[
\text{sbrs} := r [S^r_2 - S^r_4];
\]

\[
\text{call } i S_f
\]

\[
(\text{mk-subr}_{\varphi_1} S^n([\varphi_1] (S^1 \leq S^n) a_1),
\]

\[
\text{mk-subr}_{\varphi_n} S^n([\varphi_n] (S^n \leq S^n) a_n))
\]

\[
\text{mk-argcall}_{\varphi_i} S j S^1(a_1 \in [\varphi_i] S^1) \cdots
\]

\[
S^n(a_n \in [\varphi_n] S^n) S^r(r \in (R_S)) =
\]

\[
\text{sbrs} := r [S^r_2 - S^r_4];
\]

\[
\text{acall } j S_f
\]

\[
(\text{mk-subr}_{\varphi_1} S^n([\varphi_1] (S^1 \leq S^n) a_1),
\]

\[
\text{mk-subr}_{\varphi_n} S^n([\varphi_n] (S^n \leq S^n) a_n))
\]

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

\[
[\text{letrec } i \equiv p \in P_{[i]} S^\eta = [p']_{[i \in \eta]} S^\eta',
\]
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 Olen 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

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

\[
\text{comm(and) \ int(eger)\exp(ression)}
\]

\[
\text{int(eger)acc(erator) \ int(eger)var(iable)}
\]

and that the set $\Theta$ of types is the least set containing these primitive types and closed under the binary operation $\rightarrow$. We write $\leq$ for the least preorder such that

\[
\text{intvar} \leq \text{intexp} \quad \text{intvar} \leq \text{intacc}
\]

If $\theta_1 \leq \theta_1$ and $\theta_2 \leq \theta_2$ then $\theta_1 \rightarrow \theta_2 \leq \theta_1 \rightarrow \theta_2$.

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

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

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
Homages to Heroes

Chapter 9
Gödel’s T

The language $\mathcal{L}(\text{nat} \to \text{nat})$, better known as Gödel’s T, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}(\text{nat} \times \text{nat})$, which equips the naturals with some arbitrarily chosen arithmetic operators, the language $\mathcal{L}(\text{nat} \to \text{nat})$ provides a general mechanism, called primitive recursion, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an essential feature of the type language. Concepts that always refer to every program in $\mathcal{L}(\text{nat} \to \text{nat})$ exist. While this may seem a weapon that can be used to build programs, it is so powerful that it is impossible to define a function $f : \text{nat} \to \text{nat}$ by $f(n) = n + 1$.

Chapter 10
Plotkin’s PCF

This is a type language $\mathcal{L}(\text{nat} \to \text{nat})$, also known as Plotkin’s PCF, integrates functions using general recursion, a means of defining self-referential n-trial definitions in $\mathcal{L}(\text{nat} \to \text{nat})$. Expressions in $\mathcal{L}(\text{nat} \to \text{nat})$ can be evaluated: its definable functions are, in general, partial. Formally, the difference between $\mathcal{L}(\text{nat} \to \text{nat})$ and $\mathcal{L}(\text{nat} \to \text{nat})$ moves the proof of termination for an expression from if into the mind of the programmer. The type system no longer permits a wider range of functions to be evaluated, but at the cost of admitting infinite loops when the function is either incorrect or absent.

Chapter 20
Girard’s System F

The languages we have considered so far are all monomorphic in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in $\mathcal{L}(\text{nat} \to \text{nat})$ there is a distinct identity function for each type $\tau$, namely $\lambda(x : \tau. x)$, even though the behavior is the same for each choice of $\tau$. Similarly, there is a distinct composition operator for each type triple of types, namely $\circ_{\tau_1, \tau_2, \tau_3} = \lambda(f : \tau_2 \to \tau_3, g : \tau_1 \to \tau_2, x : \tau_1. f(g(x)))$.

Each choice of the three types requires a different program, even though they all exhibit the same behavior when executed.
Chapter 35

Modernized Algol

Modernized Algol, or $\mathcal{L}\{\text{nat} \rightarrow \text{cmd} \rightarrow \}$, is an imperative, block-structured programming language based on the classic language Algol. $\mathcal{L}\{\text{nat} \rightarrow \text{cmd} \rightarrow \}$ may be seen as an extension to $\mathcal{L}\{\text{nat} \rightarrow \}$ with a new syntactic sort of commands that act on assignables by retrieving and altering their contents. Assignables are introduced by declaring them for use within a specified scope; this is the essence of block structure. Commands may be combined by sequencing, and may be iterated using recursion.

$\mathcal{L}\{\text{nat} \rightarrow \text{cmd} \rightarrow \}$ maintains a careful separation between pure expressions, whose meaning does not depend on any assignables, and impure commands, whose meaning is given in terms of assignables. This ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, and allows for expressions to be manipulated much as in PCF. Commands, on the other hand, have a tightly constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of $\mathcal{L}\{\text{nat} \rightarrow \text{cmd} \rightarrow \}$ is that it adheres to the stack discipline, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline. This avoids the need for more complex forms of storage management, at the expense of reducing the expressiveness of the language.
Why Not Reynolds?

- Originally: “Reynolds’s Idealized Algol”
- I asked John to review the chapter before publication.
- But John was not pleased!
Variables vs Assignables

- In MA mathematical variables are distinguished from mutable assignables.
- To my astonishment, John was steadfastly opposed to this decision!
- “It wouldn’t be Algol if assignables weren’t variables.”
The Structure of MA

- Modal separation between
  - expressions, which are pure
  - commands, which are impure
- Modal, not monadic: commands are not expressions of command type!
- No reliance on CBN for functions.
Structure of MA

- Basic commands, m:
  - return e
  - bind (e; x.m) aka bind x=e in m

- Expressions, e:
  - cmd(m) encapsulated
  - variables x
Structure of MA

- Assignables are not variables!
  - `dcl(e,a,m)` aka `dcl a := e in m`
  - `get[a]`, `set[a](e)` are commands

- Assignables are not expressions!
  - `a+1` makes no sense
  - `bind x = cmd(get[a]) in return(x+1)`
Mobile Types

- A mobile means that no value of type A can depend on an assignable.
- Mobile values can “escape” the scope of a declaration.
- Stack discipline = assignable and returnable types must be mobile.
Statics of MA

\[
\begin{align*}
\Gamma \vdash_{\Sigma} m \sim \tau & \quad \Gamma \vdash_{\Sigma} \text{cmd}(m) : \text{cmd}(\tau) \\
\Gamma \vdash_{\Sigma} e : \tau & \quad \tau \text{ mobile} \quad \Gamma \vdash_{\Sigma} \text{ret}(e) \sim \tau \\
\Gamma \vdash_{\Sigma} e : \text{cmd}(\tau) & \quad \Gamma, x : \tau \vdash_{\Sigma} m \sim \tau' \\
\Gamma \vdash_{\Sigma} \text{bdn}(e; x. m) \sim \tau' & \\
\Gamma \vdash_{\Sigma} e : \tau & \quad \tau \text{ mobile} \quad \Gamma \vdash_{\Sigma, a \sim \tau} m \sim \tau' \\
\Gamma \vdash_{\Sigma} \text{dcl}(e; a. m) \sim \tau' & \\
\Gamma \vdash_{\Sigma, a \sim \tau} \text{get}[a] \sim \tau & \\
\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau & \quad \Gamma \vdash_{\Sigma, a \sim \tau} \text{set}[a](e) \sim \tau
\end{align*}
\]
Mobile Types

- Naturals, provided they are eager.
- Sums and products of mobile types.
- Recursive mobile types.
- Not function types.
Reference Types

- References are compatible with the stack discipline, but they are not mobile.
  - A ref contains &a for a assignable
  - `getref(e)`, `setref(e_1,e_2)`
  - `getref(&a) |-> get[a]`, etc
- Cannot build mutable data structures.
Reference Types

- Alternatively, we can deem all types to be mobile, including references.
- Dynamics must admit scope extrusion for declarations.
  - “heap allocation” of assignables.
  - mutable data structures
Hoare Logic for MA?

- Hoare logic is messy because of the confusion of variables and assignables.
- LF formulation exposes clearly.
- Heap locations represented by numbers (for disequality).
- Using assignables it’s cleaner!
Hoare Logic for MA

- No need for separation for assignables!
  - $\mu \models a \rightarrow e$ iff $\mu(a) = e$
  - $\mu \models a \rightarrow e \land a' \rightarrow e'$ iff $\mu(a) = e$ and $\mu(a') = e'$
- Distinct assignables are never aliases!
Hoare Logic for MA

- Declaration requires a “new” quantifier:
  - if $\forall a \{ P \land a \rightarrow e \} \land \{ x.Q \land a \rightarrow _{ } \}$, then $\{ P \} \ dcl \ a := e \ in \ m \ { x. Q }$
  - if $\forall a \{ P \land a \rightarrow e \} \land \{ x.Q \land a \rightarrow _{ } \}$, then $\{ P \} \ dcl \ a := e \ in \ m \ { x. Q }$ provided that $a$ not in $Q$
Hoare Logic for MA

- References require separation:
  - \( \{ e_1 \Rightarrow e_2 \} \text{getref}(e_1) \{ x.x=e_2 * e_1 \Rightarrow e_2 \} \)
  - \( \{ e_1 \Rightarrow _\_ \} \text{setref}(e_1,e_2) \{ _\_. e_1 \Rightarrow e_2 \} \)

- Here \( e_1 \) has reference type, and we have a different “contents of” primitive.
Hoare Logic for MA

- Is there a sensible substructural logic with a “new” quantifier?
- How does this relate to the representations of the heap and store in separation logic?
Chapter 35

Reynolds’s Algol

Modernized Algol, or MA, is an imperative, block-structured programming language based on the classic language Algol. MA may be seen as an extension to $L\{\text{nat} \rightarrow \}$ with a new syntactic sort of commands that act on assignables by retrieving and altering their contents. Assignables are introduced by declaring them for use within a specified scope; this is the essence of block structure. Commands may be combined by sequencing, and may be iterated using recursion.

MA maintains a careful separation between pure expressions, whose meaning does not depend on any assignables, and impure commands, whose meaning is given in terms of assignables. This ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, and allows for expressions to be manipulated much as in PCF. Commands, on the other hand, have a tightly constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of MA is that it adheres to the stack discipline, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline. This avoids the need for more complex forms of storage management, at the expense of reducing the expressiveness of the language.
John C. Reynolds