

Research Proposal

Charles Alexander Stewart

31st January 1997

| | |
|--|-----------------------------|
| Programming Research Group | Tel: +44 1865 273869 (day) |
| Oxford University Computing Laboratory | Tel: +44 1865 243665 (eve.) |
| Wolfson Building | Fax: +44 1865 273839 |
| Parks Road | E-mail: cas@comlab.ox.ac.uk |
| Oxford OX1 3QD, Britain. | |

1 Motivation

Functional languages successfully provide the programmer with a framework in which it is easy to reason about the behaviour of code, so giving the programmer confidence that the performance of the code will match his expectation. This is due to the existence of a well-defined semantics that is in a natural correspondence with the actual syntax of the language.

But the success is limited to functional code regarded as ‘pure’. When code that involves control (exceptions and continuations), state (mutable structures such as references and arrays) or input/output primitives is written, predicting the behaviour of the code starts to become fraught. As all major implementations of functional programming languages implement both state and control, we see that progress in this area of theory is essential to providing the advantage of clear semantics to the kind of code demanded by real-world applications.

The central aim of the proposed research is to examine the effect of enriching a language that has local state (such as the Idealized Algol of John Reynolds [16]) with access to first class continuations, as in [12]. Approaches to implementing threaded concurrency such as in Concurrent ML [13] use these ‘impure’ features, and threaded concurrency provides a plausible basis to characterise the behaviour of input and output. Thus for the sake of this report, I will call the proposed language *Concurrent Algol*.

The virtue of this approach over the more familiar approach from Concurrent ML is that, because both Idealized Algol and μ -PCF use bindings rather than prompts, the code generated is easier to reason about for two reasons: firstly, the scope of a binding is restricted to the bound subterm. For example in Biagoni, Cline, Lee, Okasi and Stone [2], it is seen that an exception handler provided in one coroutine can be instanced by an exception in another, even though the coroutine was exited!

Secondly, prompts are implemented using closures in an irreducible manner, whereas bindings simply use the ordinary notion of context. Thus semantically, bindings are simpler than prompts.

2 Background

The author is in the final year of a three year doctorate under Luke Ong, preparing a thesis titled ‘Classical proofs and the theory of functional control’. My work examines a reformulation of Parigot’s

lambda-mu calculus from both a proof theoretic angle, as a candidate for a Curry Howard style correspondence with classical logic, and from the point of view of being a ‘toy’ programming language (*à la* PCF) in which to generalise the various control constructions in use, and investigate their operational behaviour.

2.1 Control in functional programming

There are two main ways to view functional control, one syntactic and one operational. The syntactic view is due to Felleisen [5], and is based upon his notion of an *evaluation context* which characterises the behaviour of deterministic reduction sequences: a term is either a value, or can be uniquely expressed as an evaluation context and redex: $E[R]$. A control operator is a construction whose behaviour under reduction depends upon the evaluation context. Thus $E[\text{call/cc } M]$ reduces to $E[M(\lambda x.E[x])]$.

The operational view, which is the original perspective, uses an interpretation of the reduction in terms of an abstract machine such as Landin’s SECD machine [8] or Krivine’s machine¹, a form of rewrite system in which the operation of substitution in the underlying calculus is broken down into simpler steps with a closer correspondence to how we would actually implement such a language. Integral to such an account is some form of stack which represents the unfinished portion of the computation; the evaluation of a control operator encapsulates the stack, and the continued evaluation gives some method for its’ reinvocation.

Streicher and Reus [18] give a compelling account which unites both of these views within Scott’s D_∞ model for the lambda calculus. Part of the work of my thesis is to show how this can be extended to the lambda-mu calculus, allowing it to be seen as providing first-class access to higher-order continuations, a project already started in [12].

2.2 Algol-like languages

Idealized Algol is an elegant language due to John Reynolds [16] marrying the functional computation of Scott’s PCF with the imperative features of Algol 60.

There is a distinction made between data types and ground types: for each data-type X , there are three classes of ground types, $\text{exp}[X]$, $\text{var}[X]$ and com representing the expressions yielding values of data-type X , the variables which can contain these values and the type of commands, the expressions which may have side effects.

The special constants of Idealized Algol are: $\text{seq} : \text{com} \Rightarrow \text{com} \Rightarrow \text{com}$ for sequential composition of commands, $\text{skip} : \text{com}$ for the command with no side effects, $\text{assign}_X : \text{var}[X] \Rightarrow \text{exp}[X] \Rightarrow \text{com}$ for assigning expressions to variables, $\text{deref}_X : \text{var}[X] \Rightarrow \text{exp}[X]$ for realizing variables and $\text{new}_X : \text{var}[X] \Rightarrow \text{com} \Rightarrow \text{com}$.

Despite its economy, Idealized Algol is a deceptively expressive language. Variables, assignments and variable declarations can all be the results of higher-order expressions, allowing the programmer to define his own data types, with their corresponding declarators *et al.* Reynolds’ introduction to his language Forsythe [17], an Algol 60 derivative, gives some illustrative examples of the power of this approach.

The typing of these constants guarantees that expressions are free of side effects. It is easy to liberalise this language, equating com with $\text{exp}[\text{unit}]$ where unit is a type with a single value, and generalising the above constants to the other expression types, giving *active expressions*: expressions with side-effects. However, in [15], Reynolds argues that we should not permit this, due to a

¹I believe Krivine has left this work unpublished, but see Curien’s account [3].

phenomenon known as *interference* which occurs between expressions, when a side-effect from the execution of an active expression can affect the evaluation of the other. In the call-by-name setting of Algol 60, interference leads to particularly unpredictable consequences.

Despite this, the consensus appears to be in favour of active expressions. Instead, it is believed that control of interference is better achieved using syntactic methods, where the compiler can easily verify syntactic guarantees against interference, such as that due to O’Hearn, Power, Takeyama and Tennent [11]. Forsythe, for example, lacks the restriction.

Finally, Abramsky and McCusker [1] have given a fully abstract accounts of Idealized Algol, for both variants, using game semantics. The language with active expressions is shown to be a conservative extension of the original formulation.

3 Research goals

The central task of this proposal is to provide an Algol-like language suitably extended with control features. I describe a possible program in quite some detail; of course all of the described work is tentative, and quite possibly will require substantial reworking. The duration of the described work is intended to be about one year.

3.1 Expressiveness

The central motivation for this research project was to provide a locally-scoped language capable of expressing threaded concurrency. It is therefore a minimum prerequisite that the language can formalise a useful library of coroutine operators.

All implementations of threaded concurrency in ML use the following constructions in an essential manner: references of type $\text{unit} \Rightarrow \text{unit}$, call/cc and queues. Neither of the last two will present problems given the intended nature of Concurrent Algol, but the first construction does: Idealized Algol only allows state variables of data types, not of program types.

There are two main hurdles that face the design as a consequence. The first problem concerns the behaviour of μ -names occurring in the expression part of an assignment. Consider the following:

$$\text{new}_{N \Rightarrow N} \lambda x^{\text{var}[N \Rightarrow N]}. (\text{seq } (\mu \alpha^N. \mu \gamma^N. [\alpha] \text{assign}_{N \Rightarrow N} x (\lambda y^N. [\alpha] 0)) \\ (\text{output}_N(\text{deref}_{N \Rightarrow N} x 1)) \\)$$

After creating the state variable x the first command executed is the variable assignment, which occurs in the scope of the μ -abstraction. The assigned expression itself contains the μ -name α . The next command² executed dereferences this expression, which is now meaningless as the α is no longer bound. This problem occurs in ML, but the reinvoked reference comes with a closure which guarantees the meaningfulness; we cannot allow this in our language, else we find ourselves in the domain of prompts.

The solution to the this problem lies in the following observation: the only problematic variables are those that whose binding lies within that of the binding of the state variable. We can provide a simple³ syntactic test for the correctness of the scoping.

² output_N which I assume to be a term of type $\text{exp}[N] \Rightarrow \text{com}$.

³Syntactically simple since if we use De Bruijn indices, the test is just a numeric comparison. However the implementation will be more involved; in order to avoid closures our abstract machine is going to require some peculiar features.

The second problem arises from the following question: if we allow references to store program values, then should references be to program types? The answer must be no, as otherwise we will be allowing references to references, and our solution to the scoping problem above will not work. So our data types must be able to accommodate program values, and this suggests that we want to allow function constructors in our data types. This presents syntactic and operational difficulties; to progress further we need to turn our attention to semantic issues.

3.2 Semantics of Algol with higher order state

It looks as if, given a suitable redevelopment of the starting assumptions, we can give a semantic account analogous to that of [1]. The key is that we must find a suitable way of representing the computations we may store in expressions: from our observation about scopes it seems as if the kinds of expressions we may store are slightly different from the usual kind of expression.

As the expressions stored are unfinished computations, the appropriate semantic construct is that of a *computational monad*, due to Moggi [10]. We will need two monads, one for each kind of expression.

What are the kinds of expressions? The usual expression, that of $\mathbf{exp}[X]$ corresponds to terms of type X in weak head normal form. For the stored expressions, our special requirement is that no variables corresponding to abstractions younger than the assigned state variable may occur in the expression. To ensure this, our abstract machine must perform the requisite substitutions for all intervening variables, giving a weak head normal form with an additional property. This form we call *portable head normal form*. We also note that there is a different notion of portable head normal form for each state variable; informally, we can demonstrate this requirement by indexing over state variables.

Let \mathcal{A} be our ambient category (cpo-enriched), with monads C_x expressing portable head normal forms and E expressing weak head normal forms. We notice that portable head normal forms are syntactically special cases of head normal forms, so we can provide a natural transformation $\phi_{x,X} : C_x(X) \rightarrow E(X)$; operationally this simply means interpreting the term in an environment enriched over variables that do not occur in the term.

Finally, we need a categorical analogy of the retraction in the old category. The construction here is non-trivial; the mapping corresponds to the conversion from weak head normal forms to portable head normal forms, and there are naturality conditions to verify, and so the success is stated as a conjecture:

CONJECTURE There is a natural transformation $\psi_{x,X} : E(X) \rightarrow C_x(X)$, such that for all types X , and for all appropriate state variables x , $\phi_{x,X}$ is a closure with section $\psi_{x,X}$.

Now we can develop our semantics. We no longer need data types: all types can be seen as both data types under the monad C_x and program types under E . Now we can interpret:

- $\mathbf{exp}[X] = E(X)$
- $\mathbf{var}[X] = (C(X) \Rightarrow \mathbf{com}) \times C(X)$
- $\mathbf{deref}_X = \mathbf{snd}; \phi_{x,X}$
- $\mathbf{assign}_X = \mathbf{fst}; \psi_{x,X}$

The other constants are as in McCusker and Abramsky's paper, and given the above details are correct, the semantics of **new** should proceed analogously without complication.

3.3 Semantics of Concurrent Algol

Next we must enrich our higher order Algol with the constructions of lambda-mu. Type theoretically, there is no problem at all, so we might hope that the extension will proceed without hitch, as we have designed the language so that ‘dangling’ μ -names are impossible; unfortunately there is a non-trivial complication ahead. In our Streicher and Reus style abstract machine, μ -names are pointers to continuation stacks. During the process of computation these stacks may acquire dangling λ -variables, which will need to be made ‘portable’ by realising these variables. This could be significantly more fraught than the analogous operation on the unevaluated expression; however, there appears to be no conceptual hurdle involved.

If the above program is successful, we should have an operationally sound⁴ semantics. However, the definition of the section in terms of the abstract machine leaves a place in this account where we may lose full abstraction. Similarly in the extension to Concurrent Algol, the abstract machine account is not operationally complete (due to some properties at the top level!) for $\lambda\mu$, so the whole semantics is unlikely to be.

Finally, we may address ourselves to the problem of finding easier methods for reasoning about operational properties. Syntactic control of interference is not desirable in the coroutine library: we *want* state to interfere. The task is how to reason about interference, so that we know the programmers code behaves as it should.

References

- [1] S. Abramsky and G. A. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electronic Notes in Theoretical Computer Science*, 3:13 pages, 1996.
- [2] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone. Safe-for-space threads in Standard ML. In *Proceedings, 2nd ACM SIGPLAN Workshop on Continuations*, BRICS notes series, pages 3–1 – 3–16, 1997.
- [3] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.
- [4] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings, 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.
- [5] M. Felleisen, D. P. Friedman, E. Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings, Symposium on Logic in Computer Science*, pages 131–141. IEEE Computer Society, 1986.
- [6] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
- [7] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalisation of exceptions and control in ML-like languages. In *Proceedings, ACM Conf. Functional Programming and Computer Architecture*, pages 12–23. ACM Press, 1995.

⁴Operational soundness means that any terms with equal denotation are contextually equivalent. Operational completeness is the dual condition. A semantics is fully abstract if it is operationally sound and complete, and additionally the Scott partial order on the denotational semantics coincides with the contextual preorder

- [8] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):pp. 308–320, 1964.
- [9] G. A. McCusker. *Games and full abstraction for a functional metalanguage with recursive types*. PhD thesis, Imperial College, University of London, 1996.
- [10] E. Moggi. Notions of computation and monads. *Information and Computation*, 93 (1):29, 1991.
- [11] P. W. O’Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Electronic Notes in Theoretical Computer Science*, 1:97–136, 1995.
- [12] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 215 – 217. ACM Press, 1997.
- [13] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, pages 293–305. ACM press, 1991.
- [14] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1991.
- [15] J. C. Reynolds. Syntactic control of interference. In *Proceedings, Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, 1978.
- [16] J. C. Reynolds. *The essence of Algol*, pages 345–372. North Holland, 1981.
- [17] J. C. Reynolds. Replacing complexity with generality: The programming language Forsythe. Unpublished, 1991.
- [18] T. Streicher and B. Reus. Continuation semantics: abstract machines and control operators. Submitted for publication in 1996.