A Compiler Extension Theorem for Lucid

Christoph M. Hoffmann
Purdue University, cmh@cs.purdue.edu

Report Number:
77-222

A COMPILER EXTENSION THEOREM FOR LUCID

by

Christoph M. Hoffmann
Computer Science Department
Purdue University
West Lafayette, Ind. 47907

CSD-TR 222

February 1977

A Compiler Extension Theorem for Lucid

by

Christoph M. Hoffmann

Computer Science Department
Purdue University
West Lafayette, Ind. 47907

Abstract

The paper investigates how to extend, in a general way, compilation algorithms for subsets of the programming language Lucid, so as to handle a substantially enlarged class of programs. In particular, given an algorithm $\mathcal{A}$ which compiles correctly simple programs satisfying the syntactic restriction $\mathcal{R}$, we show how to extend $\mathcal{A}$ and $\mathcal{R}$ to compile programs which use a nesting construct. The technique does not depend on the particulars of $\mathcal{A}$ and $\mathcal{R}$, although the size of the larger class depends on $\mathcal{R}$. It constitutes an example of a compiler structuring which admits a modular correctness proof of the compiler.

1. Introduction

Proving a compiler correct is an important and non-trivial problem. Since most programs are written in a higher level language which has to be compiled, a correct compiler is a necessary part for obtaining correct results independent of a proof of the source program. The problem is amplified by the fact that very large programs, and compilers are often just that, may contain subtle mistakes which remain undetected for a long time.

The proof of a compiler requires a formal model of the semantics of the source and the object language which has to be suited both for proof purposes and for good implementation strategies. Without such models, a proof will be awkward and difficult. Perhaps for these reasons previous work on compiler correctness has dealt with languages which permit an elegant mathematical model [7, 9], or for small languages isolating a few of the constructs present in imperative languages [5, 8, 10].

We illustrate in this paper an approach to compiler correctness which reduces the complexity of the task by modularizing the proof. Morris [10] seems to have been the first to suggest this idea. Since analogous approaches to programming have been accepted principles of software design for quite some time now, it is perhaps not surprising that proof modularization appears to be an attractive method.

The source language under consideration is Lucid [1, 2, 3], a proof-oriented programming language. We define the basic language and isolate the subset of simple programs. Assuming the existence of an algorithm

$\mathfrak{A}$ which correctly compiles those simple programs which satisfy a syntactic restriction $\mathfrak{R}$, we show how $\mathfrak{A}$ may be extended to compile the larger language by incorporating language constructs not allowed in simple programs.

Existence of such algorithms may be assumed because of [6]. As only general assumptions are made about $\mathfrak{A}$ and $\mathfrak{R}$, however, our results can be used to extend any conceivable algorithm $\mathfrak{A}$ which works correctly for a formulated syntactic restriction $\mathfrak{R}$. In fact, the results also apply to interpreters such as, for example, the one reported in [4].

This degree of generality can be accomplished, because the added language constructs admit the decomposition of the source program into a collection of simple programs, related by known properties of the constructs. The decomposition result is proved first, and from it the extension strategy is derived.

The paper assumes some familiarity with the language. [1, 2, 3] are all good sources for studying Lucid, and [6] gives a specific instance of the extension technique presented here.

2. Lucid Programs and Proper Restrictions

Assuming the notation and definitions of [2], we fix in the following a particular Lucid system by choosing, without loss of generality, a standard alphabet Σ, a standard Σ structure S whose Comp(S) structure is C, and a set V of variables, and note that the results of this paper are valid in any Lucid system. As in [2], U_S denotes the range of S, i.e. a set of values.

G is the set of operation symbols in Σ, F = {first, next, fby, asa, latest, latest^{-1}} the set of Lucid functions, E the set of $(\Sigma \cup F)$ terms without quantifiers and the symbol =, E_0 the subset of $(\Sigma \cup F)$ terms without quantifiers, =, latest, and latest^{-1}. N denotes the set of natural numbers, N^N the set of infinite sequences over N.

U_C is the set of all functions from N^N into U_S. Recall the definition of Comp(S):

Definition If S is a standard Σ structure, then Comp(S) is the unique $(\Sigma \cup F)$ structure C which extends $(S^{N^N})^*$ to the larger alphabet as follows:

For $\bar{t} = t_0 t_1 t_2 \ldots$ in N^N, $\alpha, \beta, \ldots$ in U_C

(1) $(\omega_C(\alpha, \beta, \ldots))_{\bar{t}} = \omega_S(\alpha_{\bar{t}}, \beta_{\bar{t}}, \ldots)$ for all ω in G

(2) $(\text{first}_C(\alpha))_{\bar{t}} = \alpha_{0\, t_1 t_2 \ldots}$

(3) $(\text{next}_C(\alpha))_{\bar{t}} = \alpha_{t_0+1\, t_1 t_2 \ldots}$

(4) $$(\alpha \;\text{fby}_C\; \beta)_{\bar{t}} = \begin{cases} \alpha_{0\, t_1 t_2 \ldots} & \text{if } t_0 = 0 \\ \beta_{t_0-1\, t_1 t_2 \ldots} & \text{otherwise} \end{cases}$$

(5) $$(\alpha \ \underset{\sim}{\text{asa}}_C \ \beta)_{\bar{t}} = \begin{cases} \alpha_{st_1t_2\ldots} & \text{if there exists a unique } s \text{ such that } \beta_{st_1t_2\ldots} \text{ is } \underline{\text{true}} \text{ and } \beta_{rt_1t_2\ldots} \text{ is } \underline{\text{false}} \text{ for all } r<s; \\ \underline{\text{undefined}} & \text{otherwise} \end{cases}$$

(6) $$(\underset{\sim}{\text{latest}}_C\ (\alpha))_{\bar{t}} = \alpha_{t_1t_2\ldots}$$

(7) $$(\underset{\sim}{\text{latest}}^{-1}{}_C\ (\alpha))_{\bar{t}} = \alpha_{0\ t_0t_1t_2\ldots}$$

Definition A Lucid program P is a set of $(\Sigma \cup F)$ terms of the form $v = \phi_v$ where v is in V and ϕ_v is in E, and such that every variable v in P is defined in this way at most once, and the variable input is not defined.

If, furthermore, every term ϕ_v is in E_0, then P is a simple program.

The solution of a program P is the minimal (least defined) C - interpretation σ which, for a fixed interpretation α of input, satisfies P. Every Lucid program has a unique solution [2].

In order to enhance the clarity of Lucid as a programming language, certain syntactic constructs for structuring programs have been introduced in [3]. We intend to study the nature of one of these constructs, the compute clause, defined by a syntactic transformation which changes programs using compute clauses into programs of the form defined above. The clause has the following syntax

```
compute  <variable>  using  <variable list>
   <set of definitions>
end
```

and is considered a definition of the variable following the word compute, called the subject of the clause. The variables in the <variable list>

are the global variables, the ⟨set of definitions⟩ is the body of the clause. All variables which occur in the body of the clause and are not global variables, are local variables of the clause. The special variable 'result' is always a local variable and refers to the subject of the clause.

A compute clause is equivalent to the set of terms $v = \phi_v$ obtained by (1) Renaming all local variables except result with new names not occurring elsewhere in the program,

(2) Replacing every global variable X by latest X in the body,

(3) Replacing "result = ϕ" by "Y = $\text{latest}^{-1}(\phi)$", where Y is the subject,

and deleting the compute ⟨variable⟩ using ⟨variable list⟩ and end.

Example The following is a compute clause

```
compute D using M, N
   L = M fby L+M
   result =  L eq N  asa  L ge N
end
```

and is equivalent to

```
L = latest M  fby  L + latest M
D = latest⁻¹( L eq latest N  asa  L ge latest N)
```

It defines D to be the predicate "M divides N."

We can now redefine the syntax of programs. A basic assertion is a $(\Sigma \cup F)$ term of the form $v = \phi_v$ where ϕ_v is in E_0. Then

```
⟨program⟩        ::= compute output ⟨globals⟩⟨clause body⟩
⟨globals⟩        ::= ⟨empty⟩
                   | using ⟨variable list⟩
⟨clause body⟩    ::= end
                   | ⟨assertion⟩ ⟨clause body⟩
⟨variable list⟩  ::= ⟨variable⟩
                   | ⟨variable list⟩ , ⟨variable⟩
```

```
⟨assertion⟩   ::= ⟨basic assertion⟩
                | ⟨clause⟩
⟨clause⟩      ::= compute ⟨variable⟩ ⟨globals⟩ ⟨clause body⟩
```

where, in addition, the ⟨globals⟩ of output is either empty or "using input", every variable is defined at most once, either by a clause or a basic assertion, and the expression in each result definition is quiescent (see [2], Sec. 4.1), i.e. behaves as a constant.

A simple program is a ⟨program⟩ without any nested clause.

Let $\mathcal{R}$ be a set of syntactic constraints on simple programs. $\mathcal{R}$ is a proper restriction if it is decidable whether or not a simple program satisfies $\mathcal{R}$, and if there exists an algorithm $\mathcal{A}$ which correctly compiles simple programs satisfying $\mathcal{R}$.

We study how to extend $\mathcal{A}$ and $\mathcal{R}$ to programs containing nested compute clauses, after deriving certain properties of variables defined by compute clauses.

3. Properties of Compute Clauses

In a compute clause all global variables referenced are quiescent, i.e. satisfy first G = G, because of the latest implicitly applied to each global variable G. Therefore, only the current component values of global variables need to be known throughout the evaluation of the clause. Hence the global environment is 'frozen' inside a clause.

Assume that the compute clause B_2 is nested within another compute clause B_1. We prove that within B_1 the subject variable of B_2 may be considered to be defined by a point-wise operation f with arguments $G^{(1)}, \ldots G^{(r)}$ which are precisely the global variables of B_2.

<u>Theorem 3.1</u> Let R be the subject of the compute clause B_2 nested in B_1 with global variables $G^{(1)} \ldots G^{(r)}$. Then, in the clause B_1,

$$\begin{aligned} R &= f(G^{(1)}, \ldots, G^{(r)}) \\ \text{first } R &= f(\text{first } G^{(1)}, \ldots, \text{first } G^{(r)}) \\ \text{next } R &= f(\text{next } G^{(1)}, \ldots, \text{next } G^{(r)}) \end{aligned}$$

for some function f of r arguments.

<u>Proof</u> By induction on the nesting structure of clauses.

<u>Basis</u> Assume that B_2 does not contain any nested clauses. Let $G^{(1)} \ldots G^{(j)}$ be those global variables of B_2 which are local to B_1, and $G^{(j+1)} \ldots G^{(r)}$ those which are global to B_1 as well. In removing the clause structure from the program, there are i_k many latest applied to the $G^{(k)}$ in expressions in the body of B_2, transforming the $G^{(k)}$ into $\bar{G}^{(k)}$, i.e.

$$\bar{G}^{(k)} = \text{latest}^{i_k} (G^{(k)}), \qquad 1 \leq k \leq r, \; i_k > 0.$$

The interpretation $|H|$ of every expression H in B_2 may be considered a function f_H of the $|\bar{G}^{(k)}|$.

Let $\bar{t} = t_0 t_1 t_2 ..$ be in N^N, then

$$(|H|)_{\bar{t}} = (f_H)_{\bar{t}} \; (\, (|\bar{G}^{(1)}|)_{\bar{t}}, \, .., \, (|\bar{G}^{(r)}|)_{\bar{t}} \,)$$

Since latest and latest^{-1} are the only operations manipulating the $t_1 t_2 ..$ all of which are applied to the $G^{(k)}$, f_H does not depend on the $t_1 t_2 ...$ and varies with t_0 only:

$$(f_H)_{\bar{t}} = (f_H)_{t_0}$$

In particular, because of its quiescence, for the result expression E we have $\quad (f_E)_{\bar{t}} = (f_E)_{t_0} = f_E$

Therefore, in B_1,

$$\begin{aligned}(|R|)_{t_1 t_2 ...} &= (|\text{latest}^{-1} \, (f_E \, (\, \bar{G}^{(1)}, \, .., \, \bar{G}^{(r)}))|)_{t_1 t_2 ...} \\ &= (| \, f_E \, (\, \bar{G}^{(1)}, \, .., \, \bar{G}^{(r)})|)_{0 t_1 t_2 ...} \\ &= f_E \, (\, (|\bar{G}^{(1)}|)_{0 t_1 t_2 ...}, \, .., \, (|\bar{G}^{(r)}|)_{0 t_1 t_2 ...})\end{aligned}$$

Let $\bar{G}^{(k)} = \text{latest} \; \tilde{G}^{(k)}$, then

$$(|R|)_{t_1 t_2 ...} = f_E \, (\, (|\tilde{G}^{(1)}|)_{t_1 t_2 ..}, \, .., \, (|\tilde{G}^{(k)}|)_{t_1 t_2 ..})$$

Observe now that in removing the clause structure, the variables $G^{(k)}$ have i_k-1 many latest applied in expressions in the body of B_1, hence are transformed into the $\tilde{G}^{(k)}$. From this the theorem follows.

Induction Step Follows from the induction basis after all nested clauses have been replaced by the corresponding pointwise equations.

■

Corollary 3.2 Let B be a compute clause all of whose global variables are quiescent. Then the subject variable of B is quiescent in the containing block.

Proof Straight-forward.

We state the corollary because it can be used to define an optimizing transformation which reduces the nesting level of such clauses. This corresponds to a well-known compiler optimization technique known from conventional languages as moving invariant computations out of loops.

Note also, that because global variables of B_2 which are not local to the containing clause B_1 are quiescent in B_1, the function f defining the subject of B_2 depends in B_1 only on those global variables of B_2 which are local to B_1.

4. Extension of Compilation Algorithms

We investigate now how to modify a given proper restriction $\mathcal{R}$ and the associated compiling algorithm $\mathcal{A}$ so as to compile programs with nested compute clauses.

<u>Definition</u> The evaluation of $(|X|)_{\bar{t}}$ is <u>unsafe</u> if potentially $(|X|)_{\bar{t}}$ is <u>undefined</u>, i.e. may correspond to a non-terminating computation. If f is a pointwise operation and if $(|f\ (X, Y, \ldots)|)_{\bar{t}}$ is potentially <u>undefined</u> when each of the $(|X|)_{\bar{t}}$, $(|Y|)_{\bar{t}}$ ... are not, then f is <u>unsafe</u>.

Note that nested compute clauses usually correspond to unsafe pointwise computations. However, even in the case of simple programs, the interaction of non-strict and unsafe evaluations already gives rise to the 'delayed evaluation rule' for Lucid (the term is due to Vuillemin [12]), as demonstrated by the following example. In essence, delayed evaluation means that $(|X|)_{\bar{t}}$ is not to be evaluated, unless the particular value configuration requires this value.

<u>Example 4.1</u> Consider the following simple program:

```
output = X asa Y gt first input
     R = 1 asa input eq next input
     P = input eq 1
     X = 0 fby (if P then X+R else Y)
     Y = 1 fby 2*Y
```

Because of the non-strict computation of next X in conjunction with the unsafe evaluation of R, and because only a specific $\bar{t}$ component of X is needed to compute output, the evaluation of both X and R should be delayed until demanded for specific $\bar{t}$ values.

Either $\mathcal{R}$ severely curtails the interplay of non-strict and unsafe evaluations, or the algorithm $\mathcal{A}$ is sophisticated enough to handle such situations. In either case, the extension of $\mathcal{A}$ to the larger class of programs will be seen to require fairly standard methods in addition to the techniques of $\mathcal{A}$.

A proper restriction $\mathcal{R}$ analyzes syntactically a set of terms of the form $v = \phi_v$, where ϕ_v is in E_0. Given a program $\mathcal{P}$, we associate with every compute clause B_i in $\mathcal{P}$ a set P_i of terms of the above form:

Replace every compute clause B_j with subject X and global variables $G^{(1)} \ldots G^{(r)}$ which is directly nested in B_i (i.e. such that X is a local variable of B_i) by the term $X = f_X(G^{(1)}, \ldots, G^{(r)})$, where f_X represents a non-strict and unsafe pointwise operation. No other assumptions about f_X are made unless derivable syntactically by $\mathcal{R}$ from the final set P_i of terms. Also, replace every reference to a global variable G of B_i by a symbolic constant. The set P_i is now the transformed body of B_i which is evidently of the desired form, and is called the <u>simple program associated with B_i</u>.

If every simple program associated with each clause of $\mathcal{P}$ satisfies $\mathcal{R}$, then $\mathcal{P}$ satisfies $\mathcal{R}'$, the <u>proper restriction derived from $\mathcal{R}$</u>. It is easy to see that the class of programs satisfying $\mathcal{R}'$ properly includes all simple programs satisfying $\mathcal{R}$.

Consider the algorithm $\mathcal{A}$ associated with $\mathcal{R}$. Observe that $\mathcal{A}$ is either capable of implementing the delayed evaluation rule to the degree required by $\mathcal{R}$, thus can generate code evaluating a variable on demand for a particular $\bar{t}$ in N^N, or deals with programs in which delayed evaluation is required only for variables with safe evaluations. Define an algorithm $\mathcal{A}'$, the <u>extension of</u> $\mathcal{A}$, as follows.

$\mathcal{A}'$ compiles each compute clause B_i in the source program $\mathcal{P}$ into a procedure $\bar{B}_i$ which is to return the value of the result expression for a particular $\bar{t}$ in N^N when called. Because of Theorem 3.1 this is always possible. The body of B_i is compiled by $\mathcal{A}'$ in exactly the same way in which $\mathcal{A}$ compiles P_i with the following exceptions.

A reference to a variable G global to B_i is compiled into a call of a <u>parameter procedure</u> p which is to evaluate the latest value of G in the environment of its definition (usually the calling environment). Furthermore, since G is quiescent in B_i, more efficiency can be gained by compiling code which calls p at most once during each activation of $\bar{B}_i$. Methods for this are routine.

A clause B_j with subject X and global variables $G^{(1)} \ldots G^{(r)}$ which is directly nested in B_i is compiled in stages. For every one of the $G^{(k)}$ a parameter procedure p_k is compiled. Depending on the properties of the $G^{(k)}$ and the capabilities of $\mathcal{A}$, the following cases arise:

(1) $G^{(k)}$ is global to B_i as well. The code for p_k is the code for referencing a global variable of B_i as described above, i.e. a call to another parameter procedure compiled in the clause containing B_i.

(2) $G^{(k)}$ is local to B_i and has a safe evaluation. $\mathcal{A}$ may have elected to evaluate $G^{(k)}$ always (making f_X depend on $G^{(k)}$ strictly); then p_k references that value. Otherwise $\mathcal{A}$ can implement a delayed evaluation of $G^{(k)}$. In that case p_k will contain code evaluating $G^{(k)}$.

(3) $G^{(k)}$ is unsafe. Since P_i satisfies $\mathcal{R}$, $\mathfrak{A}$ can compile a delayed evaluation of $G^{(k)}$. This evaluation is the procedure p_k.

Finally, the code for evaluating $(|X|)_{\bar{t}}$ is a call of $\bar{B}_j$.

In this way, $\mathfrak{A}'$ compiles $\mathcal{P}$ into a set of procedures. A standard driving program is added which calls the procedure for the outermost clause requesting the evaluation of output. See [6] for a specific example of this extension technique.

Lemma 4.1 Let $\mathcal{R}$ be a proper restriction, $\mathcal{R}'$ the proper restriction derived from $\mathcal{R}$. Then $\mathcal{R}'$ is decidable.

Proof Evident from the construction of $\mathcal{R}$.

Let σ be the solution of the program $\mathcal{P}$ and $|X|_\sigma$ the interpretation of X in $\mathcal{P}$ according to σ. Assume that $\mathcal{P}$ satisfies $\mathcal{R}'$. The code π compiled by $\mathfrak{A}'$ for $\mathcal{P}$ consists of procedures $\bar{B}_i$ compiled from the clauses B_i of $\mathcal{P}$, and parameter procedures p_{ik} evaluating $(|G^{(k)}|_\sigma)_{\bar{t}}$ for variables $G^{(k)}$ global to B_i.

Lemma 4.2 Let π be the code compiled by $\mathfrak{A}'$ for the program $\mathcal{P}$ which satisfies $\mathcal{R}'$. If every parameter procedure p_{jk} correctly returns $(|G^{(k)}|_\sigma)_{\bar{t}}$, then the procedure $\bar{B}_i$ compiled for clause B_i with subject X correctly evaluates $(|X|_\sigma)_{\bar{t}}$.

Proof (Induction on the nesting structure of clauses)

Basis Assume that B_i contains no nested clauses. Since the parameter procedures called by $\bar{B}_i$ correctly evaluate $(|G|_\sigma)_{\bar{t}}$ by assumption, correctness follows from Theorem 3.1 and the correctness of $\mathfrak{A}$.

Step Equally straight-forward.

∎

Consequently, the correctness of $\mathfrak{A}'$ can be established by showing the correctness of the parameter procedures. Since no specific properties of $\mathfrak{A}$ and $\mathcal{R}$ can be assumed, this must be proved by a reduction to the correctness of $\mathfrak{A}$.

<u>Theorem 4.3</u> Let π be the program compiled by algorithm $\mathfrak{A}'$ for the source program $\mathcal{P}$ satisfying $\mathcal{R}'$. Given a clause B_i in $\mathcal{P}$, the procedure $\bar{B}_i$ is correct provided the parameter procedures $p_1 \ldots p_s$ of variables global to B_i are correct.

<u>Proof</u> (By induction on the nesting structure of clauses)

<u>Basis</u> B_i does not contain any nested clauses. Since the $p_1 \ldots p_s$ are the only parameter procedures called by $\bar{B}_i$, the theorem follows from Lemma 4.2.

<u>Step</u> Assume the theorem is true for all clauses B_j not containing other clauses nested beyond depth d. Let B_i be a clause which does not contain other clauses nested beyond depth d+1. We have to show that the correctness of $p_1 \ldots p_s$ implies the correctness of the parameter procedures $q_1 \ldots q_r$ compiled to evaluate variables global to clauses directly nested in B_i.

The correctness of the $p_1 \ldots p_s$ directly implies the correctness of those q_k which are to evaluate variables which are global to B_i as well. The correctness of the remaining procedures q_k evaluating local variables is seen by considering the associated simple program P_i.

Recall that every clause with subject Y local to B_i is defined in P_i by $Y = f_Y\,(G^{(1)}, G^{(2)}, \ldots)$

P_i satisfies $\mathcal{R}$ because $\mathcal{P}$ satisfies $\mathcal{R}'$. By Lemma 4.2 and the induction hypothesis, $\bar{B}_j$ correctly implements the operation f_Y. Correctness of all sequence evaluations in P_i now follows from the correctness of $\mathfrak{A}$, and from it the correctness of the procedures q_k.

∎

<u>Corollary 4.4</u> The code π compiled for $\mathcal{P}$ satisfying $\mathcal{R}'$ correctly evaluates $|\text{output}|_\sigma$.

<u>Proof</u> Since input is the only global variable a program may have, it suffices to show that the parameter procedure to evaluate it is correct. This is true since input is also global to simple programs, hence follows from the correctness of $\mathfrak{A}$.

∎

This establishes the correctness of $\mathfrak{A}'$ as extension of $\mathfrak{A}$. The result originates from the fact that $\mathcal{P}$ can be decomposed into the simple programs P_i and that the individual procedures generated are coordinated by the same techniques which $\mathfrak{A}$ employs to coordinate the evaluation of non-strict and unsafe computations. $\mathcal{R}'$ ensures that $\mathfrak{A}'$ is not overtaxed in this. Note, however, that the class of programs satisfying $\mathcal{R}'$ depends in size on $\mathcal{R}$; for less restrictive conditions $\mathcal{R}$ the class is larger. This is, of course, intuitive.

5. Conclusions

We have shown how to extend algorithms compiling simple programs to handle nested compute clauses. As the techniques for this were derived from language properties rather than particular aspects of the algorithms or their scope, our results are applicable to any compilation strategy. Of course, the 'size' of the new class of programs compilable by the extended algorithm depends on the restrictiveness of $\mathcal{R}$, i.e. on the degree of sophistication of the algorithm $\mathfrak{A}$.

Of the other language constructs proposed in [3], a good candidate for a similar extension theorem would be the mapping clause , which is a generalization of the compute clause, and can be handled in essentially the same manner. For other clauses, however, it is not clear how to decompose them into simpler concepts successfully. In particular, the full generality of the transform clause is a deep challenge to compiler writers, if at all compilable.

Intuitively, we suspect that the 'orthogonality' (cf. [11]) of a construct allows the formulation of extension theorems. Since the global environment is frozen inside a compute clause, the construct serves to substitute programs for expressions in the definition of variables. Therefore, it is of advantage to structure a compiler accordingly, both from the design aspect as well as from the point of view of proving its correctness.

It should be the case, that the modularization of a program and of its proof serves to reduce the effort invested in the development of both. Research of proof strategies aiming at exploiting this should be very fruitful.

References

1. Ashcroft, E.A., and W. Wadge
   Lucid, A Non-Procedural Language With Iteration
   forthcoming in Comm. of the ACM

2. Ashcroft, E.A., and W. Wadge
   Lucid, A Formal System for Writing and Proving Programs
   SIAM Journ. on Computing, 5(76) 336 - 354

3. Ashcroft, E.A., and W. Wadge
   Lucid, Scope Structures and Defined Functions
   TR CS-76-22, Comp. Science Dept., Univ. of Waterloo, Nov. 76

4. Cargill, T.A.
   Deterministic Operational Semantics for Lucid
   TR CS-76-19, Comp. Science Dept., Univ. of Waterloo, June 76

5. Chirica, L.M., and D.F. Martin
   An Approach to Compiler Correctness
   Intl. Conf. on Reliable Software, p 96 - 103, Los Angeles, June 76

6. Hoffmann, C.M.
   Design and Correctness of a Compiler for Lucid
   TR CS-76-20, Comp. Science Dept., Univ. of Waterloo, May 76

7. London, R.L.
   Correctness of two Compilers for a LISP Subset
   AI Memo 151, Stanford Univ., 1971

8. McCarthy, J., and J.A.Painter
   Correctness of a Compiler for Arithmetic Expressions
   Math. Aspects of Comp. Sci. Vol 19, Providence, R.I. 67

9. Milner, R., and R. Weyhrauch
   Proving Compiler Correctness in a Mechanized Logic
   Machine Intell. 7, p 51 - 71, Univ. of Edinburgh, 73

10. Morris, F.L.
    Advice on Structuring Compilers and Proving them Correct
    ACM Symp. on Princ. of Progr. Lang., p 144-152, Boston 73

11. VanWijngaarden
    Orthogonal Design and Description of a Formal Language
    MR 76, Mathematisch Centrum, Amsterdam Oct 65

12. Vuillemin, J.
    Correct and Optimal Implementation of Recursion in a Simple
    Programming Language
    5th Annl Symp on Theory of Computing, Austin 73