A Little Goes a Long Way: A Simple Tool to Support Denotational Compiler-Correctness Proofs

Mitchell Wand and Gregory T. Sullivan*

College of Computer Science
Northeastern University
Boston, MA 02115, USA
{wand,gregs}@ccs.neu.edu

September 20, 1994

Abstract

In a series of papers in the early 80's we proposed a paradigm for semantics-based compiler correctness. In this paradigm, the source and target languages are given denotational semantics in the same λ-theory, so correctness proofs can be carried out within this theory. In many cases, the proofs have a highly structured form. We show how a simple proof strategy, based on an algorithm for α-matching, can be used to build a tool that can automate the routine cases of these proofs.

1 Introduction

In a series of papers in the early 80's [19, 18, 20, 5] we proposed a paradigm for semantics-based compiler correctness, and over the last several years we have begun putting this paradigm into practice [22, 14]. In this paradigm, the source and target languages are given denotational semantics in the same λ-theory [11], so that most of the steps of the correctness proof can be done

*Work supported by the National Science Foundation under grant numbers CCR-9014603 and CCR-9304144.

within the theory. Furthermore, the compiler is given by a compositional (syntax-directed) translation, so properties of the compiler can be proved by structural induction on the grammar of the source language.

If the semantics of programs in the source language is given by $\mathcal{P}$, the semantics of the target language is given by $\mathcal{T}$, and the compiler on programs is given by $\mathcal{CP}$, then the correctness of the compiler is given by the requirement that for all programs p,

$$\mathcal{T}[\![\mathcal{CP}[\![p]\!]]\!] = \mathcal{P}[\![p]\!] \tag{1}$$

Such an equation is proven by structural induction on source programs; this typically involves an induction hypothesis for each non-terminal of the source language. The reasoning involves higher-order equational reasoning in a suitable λ-theory plus the use of the induction hypotheses.

A number of small-to-moderate sized compiler correctness proofs have been done in this way [5, 7, 14, 19, 18, 20, 22]. As proofs of congruence and soundness scale up, machine support becomes more and more important. Ideally, one would like all the simple cases to be checked by machine, perhaps leaving a few subtle cases to be done by hand.

We have now built a theorem-prover capable of proving many such theorems. The key idea in the system is the use of a very simple rule-matching technique, which we call *α-matching*, in place of more complicated techniques such as higher-order unification.

In this paper, we will describe the system and its use. Section 2 shows a typical manual proof step, and Section 3 shows how this proof is expressed in our system. Section 4 discusses the generation of a proof by structural induction from a grammar, induction hypotheses, and user-defined rules. Section 5 describes the internal organization of the system as well as some of the more interesting algorithms. Section 6 describes our experience with the system, including what it can and cannot do currently. Section 7 discusses related work, Section 8 discusses extensions to the system, and Section 9 presents our conclusions about the system.

2 Example of a Manual Proof

A theorem like equation 1 is proven by structural induction on source programs; this typically involves an induction hypothesis for each non-terminal of the source language. For example, the induction hypothesis for expressions might be

$$\mathcal{T}[\![\mathcal{CE}[\![e]\!]\Gamma t]\!]u\zeta = \mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta)) \tag{2}$$

This induction hypothesis says that if the compilation of expression e, using symbol table Γ followed by program tail t, is run on display u and stack ζ, the result is the same as that of evaluating e in the environment $(u \circ \Gamma)$[1] and a continuation that runs t on display u and the stack obtained by pushing the result v of evaluating e onto ζ.[2]

Such proofs typically have a highly structured form. As usual for structural inductions, we have one induction step for each production in the language. Let us consider the case of addition expressions. The semantics of an addition expression might be given by

$$\mathcal{E}[\![e_1 + e_2]\!] = \lambda\rho\kappa.\, \mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\, \mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\, \textit{cps-add}\; v_1 v_2 \kappa)) \tag{3}$$

This is standard continuation semantics.

A typical production in the target language might be

$$t ::= \mathbf{add}\, t$$

representing an **add** instruction followed by additional target code, with semantics specified by

$$\mathcal{T}[\![\mathbf{add}\, t]\!] = \lambda u\zeta.\, \textit{cps-add}(\zeta \downarrow 2)(\zeta \downarrow 1)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}(\zeta \dagger 2))) \tag{4}$$

Here u is the run-time display and ζ is a stack on which the instruction acts; $\zeta \downarrow i$ denotes the i-th element of the stack, and $\zeta \dagger i$ represents the stack from which the first i elements have been removed. Thus the **add** instruction works by adding the first two elements of the stack, and invoking the code t on the stack obtained by replacing the first two elements of ζ by their sum; the display register u is left unchanged.

The compiler on addition expressions might be given by

$$\mathcal{CE}[\![e_1 + e_2]\!]\Gamma t = \mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma(\mathbf{add}\, t)) \tag{5}$$

[1] We model a symbol table Γ as a map from variables to lexical addresses, and a display as a map from lexical addresses to values, so their composition $(u \circ \Gamma)$, functions as an environment, mapping variables to values.

[2] The reader familiar with [5] will note some differences. We have eliminated both Clinger's accumulator ϵ and the continuation register κ. This entails no loss of generality, as these quantities can be cached at the top and bottom of the stack ζ.

Here we write $\mathcal{CE}$ rather than $\mathcal{CP}$ since we are compiling an expression rather than a program. The equation says that to compile e_1+e_2 (followed by target code t), compile e_1, using symbol table Γ, followed by the compilation of e_2, followed by an **add** instruction, followed by t. This is a reasonable compilation strategy for a stack machine.

To show that the induction hypothesis holds for every source language expression e, we will use structural induction [3]. We will have one induction step for each production in the grammar. For the production for addition expressions, we will want to show that if the induction hypothesis is true at expressions e_1 and e_2, then it is true at the expression $e_1 + e_2$ as well. To do this, we start with a goal equation of

$$\mathcal{T}[\![\mathcal{CE}[\![e_1 + e_2]\!]\Gamma t]\!]u\zeta = \mathcal{E}[\![e_1 + e_2]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta)) \tag{6}$$

and we calculate as follows:

$$
\begin{array}{l}
\mathcal{T}[\![\mathcal{CE}[\![e_1 + e_2]\!]\Gamma t]\!]u\zeta \\
= \quad \text{[definition of } \mathcal{CE} \text{ (5)]} \\
\mathcal{T}[\![\mathcal{CE}[\![e_1]\!]\Gamma(\mathcal{CE}[\![e_2]\!]\Gamma(\mathbf{add}\, t))]\!]u\zeta \\
= \quad \text{[induction hypothesis (2) at } e_1\text{]} \\
\mathcal{E}[\![e_1]\!](u \circ \Gamma)(\lambda v_1.\, \mathcal{T}[\![\mathcal{CE}[\![e_2]\!]\Gamma(\mathbf{add}\, t)]\!]u(v_1{::}\zeta)) \\
= \quad \text{[induction hypothesis (2) at } e_2\text{]} \\
\mathcal{E}[\![e_1]\!](u \circ \Gamma)(\lambda v_1.\, \mathcal{E}[\![e_2]\!](u \circ \Gamma)(\lambda v_2.\, \mathcal{T}[\![\mathbf{add}\, t]\!]u(v_2{::}(v_1{::}\zeta)))) \\
= \quad \text{[definition of } \mathcal{T} \text{ (4), properties of stack functions]} \\
\mathcal{E}[\![e_1]\!](u \circ \Gamma)(\lambda v_1.\, \mathcal{E}[\![e_2]\!](u \circ \Gamma)(\lambda v_2.\, \mathit{cps\text{-}add}\; v_1 v_2(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta)))) \\
= \quad \text{[definition of } \mathcal{E} \text{ (3)]} \\
\mathcal{E}[\![e_1 + e_2]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta))
\end{array}
$$

as desired. Here we have labelled each equality with its justification.

In doing this proof, we used only ordinary β-conversion, plus the knowledge of the induction hypotheses. We used the fact that if $M = N$ is provable in the λ-calculus, so is any substitution instance of $M = N$. This allowed us to use a complex substitution for κ when we applied the induction hypothesis at e_2. These rules of inference are shown in Figure 1. It is easy to show that these rules are a conservative extension of the ordinary λ-calculus (or of any λ-theory) [23].

This proof has a very definite structure: one reduces the left-hand term, using ordinary β-reduction and the induction hypotheses, and similarly reduces the right-hand term, using β-reduction and the definition of the original semantics. We can formalize this notion by converting the equational

$$\Delta \vdash (\lambda x.M)N = M[N/x] \quad \beta$$

$$\frac{(M = N) \in \Delta}{\Delta \vdash M\theta = N\theta} \quad \text{subst}$$

$$\frac{M =_\alpha N}{\Delta \vdash M = N} \quad \text{refl}$$

$$\frac{\Delta \vdash M = N}{\Delta \vdash N = M} \quad \text{sym}$$

$$\frac{\Delta \vdash M = N \quad \Delta \vdash N = P}{\Delta \vdash M = P} \quad \text{trans}$$

$$\frac{\Delta \vdash M = M' \quad \Delta \vdash N = N'}{\Delta \vdash MN = M'N'} \quad \text{cong}$$

$$\frac{\Delta \vdash M = N}{\Delta \vdash \lambda x.M = \lambda x.N} \quad \xi$$

Figure 1: Equational rules for λ-calculus with additional hypotheses

$$C[(\lambda x.M)N] \rightarrow_\Delta C[M[N/x]]$$

$$\frac{(M = N) \in \Delta}{C[M\theta] \rightarrow_\Delta C[N\theta]}$$

Figure 2: Rewriting rules for λ-calculus with additional hypotheses

rules of Figure 1 to the rewriting rules of Figure 2, where $C[]$ denotes any context (λ-term with a hole), and where α-congruent terms are treated as identical (following [1, Convention 2.1.12]). These rules are clearly sound with respect to the equations of Figure 1, and their transitive, reflexive, symmetric closure is equivalent to the system of Figure 1.

To prove $\Delta \vdash M = N$, we attempt to find a term Z such that $M \rightarrow^*_\Delta Z$ and $N \rightarrow^*_\Delta Z$. We do this by reducing M and N as much as possible using the reduction relation $\rightarrow_\Delta$. This strategy will fail in general, as $\rightarrow_\Delta$ is neither strongly normalizing nor (for suitable choice of Δ) Church-Rosser. Fortunately, the particular terms M, N and rules Δ that arise in practice (except for terms involving fixed-point operators) have both properties, so this strategy works on the cases of interest.

3 A Mechanized Proof

Our theorem-prover is able to automatically verify this example and others like it. Our example is for a toy source language with only addition and variable reference. We hope that these constructs are enough to give the reader an adequate feel for using the system. The input to the theorem-prover is shown in Figures 3, 4 and 9, and the output is shown in Figures 5–8.

Figures 3 and 4 show four lists of rules. The first three lists give the definitions of the compiler, the target-language semantics, and the source-language semantics. The last list, `misc-rules`, gives the relevant rules for evaluation of operations on lists as well as for composition and for `cps-add`.

Each rule is a three-element list with a left-hand side, a right-hand side, and a label. In the rules, constants are represented by quoted symbols (`'compile-exp`, `'plus`, etc.); other symbols (`exp1`, `tail`, etc.) are variables. Since our system is tuned for dealing with semantics, all applications and abstractions are internally curried. The unparsing routine automatically merges consecutive `lambda`'s and omits unnecessary parentheses, as shown in Figures 5–8. We use explicit bound variables rather than deBruijn indices in the internal representation; this makes the algorithms clearer.

Figures 5–8 show the output from the an automated proof of the induction hypothesis for the `Exp` nonterminal and the `add` production. The prover applies user-supplied rules, beta-reduction, and the induction hypotheses as rewrite rules as often as possible to both sides of the desired conclusion. The prover then tests the resulting terms for congruence to determine whether the proof was successful. We have edited the output only to improve the pretty-printing and by replacing some of the more obvious intermediate term displays by the string `[...]`. Notice how closely the steps correspond to the human-generated proof in Section 2.

4 From Grammars to Proofs

The proof of an induction hypothesis IH involves one induction step for each production of the source language. For example, for the production

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle$$

the induction step is

$$IH(e_1) \ \wedge \ IH(e_1) \Rightarrow IH(e_1 + e_2).$$

In order to prove this implication, we generate two local rules

$$\begin{aligned} \mathcal{T}[\![\mathcal{CE}[\![\,'e_1]\!]\Gamma t]\!]u\zeta &= \mathcal{E}[\![\,'e_1]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta)) \\ \mathcal{T}[\![\mathcal{CE}[\![\,'e_2]\!]\Gamma t]\!]u\zeta &= \mathcal{E}[\![\,'e_2]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta)) \end{aligned}$$

and then use these to prove the conclusion

$$\mathcal{T}[\![\mathcal{CE}[\![\,'e_1 + \,'e_2]\!]\Gamma t]\!]u\zeta = \mathcal{E}[\![\,'e_1 + \,'e_2]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta))$$

via rewriting.

Here $'e_1$ and $'e_2$ are constants; asserting the induction hypothesis

$$\mathcal{T}[\![\mathcal{CE}[\![e]\!]\Gamma t]\!]u\zeta = \mathcal{E}[\![e]\!](u \circ \Gamma)(\lambda v.\, \mathcal{T}[\![t]\!]u(v{::}\zeta))$$

as a rewrite rule would enable us to reach the conclusion immediately and erroneously via rule (subst). Introducing new constants deals correctly with this quantification.

```
(define example-compiler-rules
  '((('compile-prog ('prog exp) symtab tail)
     ('compile-exp exp symtab tail)
     "CP of 'prog")
    (('compile-exp ('plus exp1 exp2) symtab tail)
     ('compile-exp exp1 symtab
       ('compile-exp exp2 symtab
         ('add-inst tail)))
     "CE of +")
    (('compile-exp ('var id) symtab tail)
     ('fetch-inst (symtab id) tail)
     "CE of 'var ")))

(define example-machine-denotation-rules
  '((('target-semantics ('add-inst tail) u stack)
     ('cps-add ('cadr stack) ('car stack)
       (lambda (v)
         ('target-semantics tail u ('cons v ('cddr stack)))))
     "T of 'add-inst")
    (('target-semantics ('fetch-inst addr tail) u stack)
     ('target-semantics tail u
       ('cons (u addr) stack))
     "T of 'fetch-inst")))

(define example-source-semantics-rules
  '((('source-prog-semantics ('prog exp) env cont)
     ('source-exp-semantics exp env cont)
     "P of 'prog")
    (('source-exp-semantics ('plus exp1 exp2) env cont)
     ('source-exp-semantics exp1 env
       (lambda (v1)
         ('source-exp-semantics exp2 env
           (lambda (v2) ('cps-add v1 v2 cont)))))
     "E of 'plus")
    (('source-exp-semantics ('var id) env cont)
     (cont (env id))
     "E of 'var")))
```

Figure 3: Syntax and Rules for the addition example (part 1)

```
(define example-misc-rules
  '((('car ('cons a d))
     a
     "car")
    (('cadr ('cons a1 ('cons a2 dd)))
     a2
     "cadr")
    (('cddr ('cons a1 ('cons a2 dd)))
     dd
     "cddr")
    (('compose f g)
     (lambda (v) (f (g v)))
     "compose")
    (('cps-add v1 v2 cont)
     (cont ('+ v1 v2))
     "cps-add")))
```

Figure 4: Syntax and Rules for the addition example (part 2)

In general, we will have several induction hypotheses for each non-terminal in our grammar. For each production

$$A ::= \ldots B \ldots C \ldots D \ldots$$

we generate a constant for each non-terminal on the right-hand side of the production, and then introduce a local rule for each induction hypothesis appropriate for each constant. We then use these rules to prove each of the induction hypotheses associated with the left-hand side. Thus, in this example, if the non-terminal A was associated with two induction hypotheses IH_1 and IH_2, and B was associated with three induction hypotheses IH_3, IH_4, and IH_5, we would generate new constants *'b*, *'c*, and *'d*. We would then generate rules $IH_3('b)$, $IH_4('b)$, and $IH_5('b)$, and similar rules for *'c* and *'d*. We would then use these rules to prove

$$IH_1(\ldots 'b \ldots 'c \ldots 'd)$$

and

$$IH_2(\ldots 'b \ldots 'c \ldots 'd)$$

```
Reducing left-hand side...
('target-semantics
   ('compile-exp ('plus 'exp_1 'exp_2) symtab tail)
   u
   stack)
[by CE of +]:
('target-semantics
   ('compile-exp
      'exp_1
      symtab
      ('compile-exp 'exp_2 symtab ('add-inst tail)))
   u
   stack)
[by IH at 'exp_1]:
('source-exp-semantics
   'exp_1
   ('compose u symtab)
   (lambda (v)
      ('target-semantics
         ('compile-exp 'exp_2 symtab ('add-inst tail))
         u
         ('cons v stack))))
[by compose]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('target-semantics
         ('compile-exp 'exp_2 symtab ('add-inst tail))
         u
         ('cons v stack))))
[by IH at 'exp_2]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('source-exp-semantics
         'exp_2
         ('compose u symtab)
         (lambda (v_1)
            ('target-semantics
               ('add-inst tail)
               u
               ('cons v_1 ('cons v stack)))))))
```


Figure 5: Machine-generated proof for the addition example (part 1)

```
[by T of 'add-inst]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('source-exp-semantics
         'exp_2
         ('compose u symtab)
         (lambda (v_1)
            ('cps-add
               ('cadr ('cons v_1 ('cons v stack)))
               ('car ('cons v_1 ('cons v stack)))
               (lambda (v_2)
                  ('target-semantics
                     tail
                     u
                     ('cons v_2 ('cddr ('cons v_1 ('cons v stack))))))))))
[by car]:
[...]
[by cadr]:
[...]
[by cddr]:
[...]
[by compose]:
[...]
[by cps-add]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v_1)
            ((lambda (v_2) ('target-semantics tail u ('cons v_2 stack)))
             ('+ v v_1))))))
[by beta]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v_1)
            ('target-semantics tail u ('cons ('+ v v_1) stack))))))
```


Figure 6: Machine-generated proof for the addition example (part 2)

```
Reducing right-hand side...
('source-exp-semantics
   ('plus 'exp_1 'exp_2)
   ('compose u symtab)
   (lambda (v) ('target-semantics tail u ('cons v stack))))
[by E of 'plus]:
('source-exp-semantics
   'exp_1
   ('compose u symtab)
   (lambda (v1)
      ('source-exp-semantics
         'exp_2
         ('compose u symtab)
         (lambda (v2)
            ('cps-add
               v1
               v2
               (lambda (v) ('target-semantics tail u ('cons v stack))))))))
[by compose]:
[...]
[by compose]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v1)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v2)
            ('cps-add
               v1
               v2
               (lambda (v) ('target-semantics tail u ('cons v stack))))))))
[by cps-add]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v1)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v2)
            ((lambda (v) ('target-semantics tail u ('cons v stack)))
             ('+ v1 v2))))))
```


Figure 7: Machine-generated proof for the addition example (part 3)

```
[by beta]:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v1)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v2)
            ('target-semantics tail u ('cons ('+ v1 v2) stack))))))

lhs reduces to:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v_1)
            ('target-semantics tail u ('cons ('+ v v_1) stack))))))

rhs reduces to:
('source-exp-semantics
   'exp_1
   (lambda (v) (u (symtab v)))
   (lambda (v1)
      ('source-exp-semantics
         'exp_2
         (lambda (v) (u (symtab v)))
         (lambda (v2)
            ('target-semantics tail u ('cons ('+ v1 v2) stack))))))

Checking for alpha-congruence... true
```

Figure 8: Machine-generated proof for the addition example (part 4)

```
(define example-abstract-syntax
  '(prog          ;;; prog is the start symbol
     ((prog ('prog exp))
      (exp ('plus exp exp))
      (exp ('var id)))))

(define example-induction-hypotheses
  '((prog p       ;;; prog is nonterminal, p is induction var.
          ('target-semantics
            ('compile-prog p symtab tail) u stack)
          ('source-prog-semantics p
            ('compose u symtab)
            (lambda (v)
              ('target-semantics tail u ('cons v stack)))))
    (exp e
         ('target-semantics
           ('compile-exp e symtab tail) u stack)
         ('source-exp-semantics
           e ('compose u symtab)
           (lambda (v)
             ('target-semantics tail u ('cons v stack)))))))

(define prove-example
  (lambda ()
    (let ((results
            (structural-induction
              example-abstract-syntax
              example-induction-hypotheses
              (append
                example-compiler-rules
                example-machine-denotation-rules
                example-source-semantics-rules
                example-misc-rules))))
      (report results))))
```

Figure 9: Syntax, Induction Hypotheses, and Top-level driver for the addition example

```
;;; Basic types

tac = success-cont * fail-cont -> ans
success-cont = value * fail-cont -> ans
fail-cont = () -> ans
tacfn = term -> tac

;;; Types of basic operations on tactics

succeed : term -> tac
fail    : () -> tac
and-then : tac * (term->tack) -> tac
or-else  : tac * tac -> tac
repeat   : tacfn -> tacfn
or-else-use* : tacfn* -> tacfn
```

Figure 10: Basic operations on tactics

5 Organization of the Prover

The prover is a straightforward tactically-driven theorem-prover, using α-*matching* to determine the applicability of all rules other than β-reduction.

A tactic is implemented as a Scheme procedure that takes two arguments: a success continuation and a failure continuation. A success continuation takes a value and a failure continuation and returns an answer, and a failure continuation is thunk that returns an answer.

The interface for the basic operations on tactics is shown in Figure 10. Most operations involve `tacfn`'s, which are maps from terms to tactics. `succeed` is a tacfn that always succeeds and sends its argument to its success continuation. `(fail)` produces a tactic that always fails. `(and-then tac1 tacfn1)` applies `tac1`; if it succeeds, its value is sent to `tacfn1`. `or-else` establishes a choice point. `or-else-use*` tries a list of tacfns in sequence. `repeat` applies its argument (a tacfn) until it fails, and then succeeds. The definitions are familiar and are shown in Figure 11.

A more interesting operation on tactics is `leftmost`. Given a tacfn and a term, it finds the leftmost place in a term where the tacfn applies and applies it there. The definition, using our operations on tactics, is shown in Figure 12.

```
(define succeed                         ; val -> tac
  (lambda (v)
    (lambda (succ fail)
      (succ v fail))))

(define fail                            ; () -> tac
  (lambda ()
    (lambda (succ fail)
      (fail))))

(define and-then                        ; tac * (val->tac) -> tac
  (lambda (tac tacfn)
    (lambda (succ fail)
      (tac (lambda (v backtrack) ((tacfn v) succ backtrack))
           fail))))

(define or-else                         ; tac * tac -> tac
  (lambda (t1 t2)
    (lambda (succ fail)
      (t1 succ (lambda () (t2 succ fail))))))

(define or-else-use*                     ; (val->tac)* -> (val->tac)
  (lambda l
    (let or-else-use*-loop ((l l))
      (if (null? l)
        (lambda (v) (fail))
        (lambda (v)
          (or-else ((car l) v)
            ((or-else-use*-loop (cdr l)) v)))))))

(define repeat                          ; (val->tac) -> (val->tac)
  (lambda (tacfn)
    ;; keep doing (tacfn v) until it fails
    (lambda (v)
      (or-else
        (and-then
          (tacfn v)
          (repeat tacfn))
        (succeed v)))))
```

Figure 11: Definitions of basic operations on tactics

```
(define leftmost                        ; (term->tac) -> (term->tac)
  (lambda (tacfn)
    (lambda (m)
      ;; find leftmost place in m where tac applies, and apply it,
      ;; rebuilding the term around the result
      (let leftmost-loop ((m m))
        (or-else
          (tacfn m)
          (record-case m
            (app (rator rand)
                 (or-else
                   (and-then
                     (leftmost-loop rator)
                     (lambda (rator1)
                       (succeed (make-app rator1 rand))))
                   (and-then
                     (leftmost-loop rand)
                     (lambda (rand1)
                       (succeed (make-app rator rand1))))))
            (abs (bv body)
                 (and-then
                   (leftmost-loop body)
                   (lambda (body)
                     (succeed (make-abs bv body)))))
            (else (fail)))))))
```

Figure 12: Definition of `leftmost`

5.1 α-matching

The main problem in building the theorem-prover was handling the user-defined rules, which serve as the additional hypotheses Δ. In most modern higher-order theorem-provers (eg. Isabelle [15]), the applicability of a rule to a given term is determined by some powerful mechanism like higher-order unification. Humans, on the other hand, typically use much weaker strategies to determine if a rule is applicable. In the hand-generated proof above, for example, a human can easily check each step, because at every step some subterm of the term being manipulated is α-congruent (that is, identical up to a change of bound variables) to some instance of the cited rule.

We therefore fixed on the problem of *α-matching*, which is defined as follows:

Definition 1 (α-matching) *Given two λ-terms, a pattern P and a subject S, find the most general substitution θ such that $P\theta$ and S are α-congruent.*

The algorithm for this problem is a fairly straightforward adaptation of a conventional unification algorithm, keeping careful track of bound variables so that no substitution is ever made on a bound variable. The main function of the algorithm is *match*. *match* takes four arguments: two λ-terms P (the pattern), and S (the subject), and two lists of identifiers Σ_1 and Σ_2. The output of *match* is the most general substitution θ such that $(\lambda\,\Sigma_1.P)\theta =_\alpha (\lambda\,\Sigma_2.S)$. Figure 13 gives the algorithm for *match* as an inference system. For each case, we give a brief explanation.

- If P and S are the same constant, succeed with the empty substitution; if they are different constants, then fail.
- If P and S are both variables, check to see if they occur in the same position in Σ_1 and Σ_2. If they do, then succeed with the empty substitution (second rule). If both are free, return the substitution that substitutes one of the variables for the other (third rule). If they are bound in different positions, or if one is free and the other is bound, then fail— the terms cannot be matched.
- If P is a free variable v and S is not, then check to see if any of the free variables of S are bound in Σ_2. If so, then fail– no substitution in P can have a variable captured. Otherwise, succeed with the substitution

$$\frac{c_1 \in \mathit{Const},\ c_2 \in \mathit{Const},\ c_1 = c_2}{\mathit{match}(c_1, c_2, \Sigma_1, \Sigma_2) \to \{\}}$$

$$\frac{v_1 \in \mathit{Var},\ v_2 \in \mathit{Var},\ \exists i \text{ s.t. } \Sigma_1[i] = v_1 \text{ and } \Sigma_2[i] = v_2}{\mathit{match}(v_1, v_2, \Sigma_1, \Sigma_2) \to \{\}}$$

$$\frac{v_1 \in \mathit{Var},\ v_2 \in \mathit{Var},\ v_1 \notin \Sigma_1,\ v_2 \notin \Sigma_2}{\mathit{match}(v_1, v_2, \Sigma_1, \Sigma_2) \to \{(v_1 \mapsto v_2)\}}$$

$$\frac{v \in \mathit{Var},\ v \notin \Sigma_1,\ \mathit{fv}(S) \cap \Sigma_2 = \emptyset}{\mathit{match}(v, S, \Sigma_1, \Sigma_2) \to \{(v \mapsto S)\}}$$

$$\frac{\mathit{match}(P_1, S_1, \Sigma_1, \Sigma_2) \to \theta,\ \mathit{match}(P_2\theta, S_2, \Sigma_1, \Sigma_2) \to \theta'}{\mathit{match}(P_1 P_2,\ S_1 S_2, \Sigma_1, \Sigma_2) \to \theta \circ \theta'}$$

$$\frac{\mathit{match}(P_1, S_1, (\Sigma_1 : x), (\Sigma_2 : y)) \to \theta}{\mathit{match}(\lambda x.P_1,\ \lambda y.S_1, \Sigma_1, \Sigma_2) \to \theta}$$

Figure 13: Rules for α-matching

mapping the variable v to S. If S is a variable and P is not, then fail– substitutions are allowed only in the pattern. Note that this is the only case that would need to be changed to do unification rather than matching.

- If P is an application P_1P_2 and S is an application S_1S_2, let $\theta = match(P_1, S_1, \Sigma_1, \Sigma_2)$. Then let $\theta' = match(P_2\theta, S_2, \Sigma_1, \Sigma_2)$. If both these steps succeed, return the composition of θ followed by θ'. We recur on applications in the usual way.
- If P is an abstraction $\lambda x.P_1$ and S is an abstraction $\lambda y.S_1$, then add the bound variables to the bound-variable lists, and recur on the bodies.
- Otherwise fail: P and S are in different syntactic categories.

It is easy to see, by induction on the height of the proof tree, that if $(\lambda\Sigma_1.P)$ and $(\lambda\Sigma_2.S)$ have no free variables in common, then this algorithm satisfies its specification. [3] The algorithm halts by an easy induction on the size of the subject S.

The code for the matching algorithm is shown in Figures 14–15. It is coded as a procedure that takes four arguments and returns a tactic. If a match exists, the substitution is sent to the success continuation; otherwise the tactic fails.

5.2 Top-Level Procedures

The top-level procedure for doing a proof by structural induction, `structural-induction`, converts the given grammar, induction hypotheses, and user-defined rules into internal form and then invokes `induct-prod` for each production-induction-hypothesis pair. See Figures 16–18 for the top-level code. The function `induct-prod` derives the "induction rules" based on the given production and induction hypothesis, and then builds a top-level tactic which tries the user-defined rules, beta-reduction, the induction rules, and finally eta-expansion. The rules are applied until none succeed.

The resulting system is very small: about 1000 lines of Scheme, including comments. Because it searches only to find rewrite sites (as in `leftmost`), it

[3] This restriction is always satisfied for our application. To eliminate this restriction, we would simply need to add an occurs check $v \notin fv(S)$ to the fourth rule.

```
(define match
  (lambda (pat subj)
    (let loop ((pat pat) (subj subj) (s1 '()) (s2 '()))
      (cond
        ((and (const? pat) (const? subj))
          (if (equal?
                (const->datum pat)
                (const->datum subj))
            (succeed (make-empty-subst))
            (fail)))
        ((and (var? pat) (var? subj))
          (match-two-vars pat subj s1 s2))
        ((var? pat)
          (match-one-var pat subj s1 s2))
        ((var? subj)                    ; if subj is var and pat
                                        ; isn't, then fail
          (fail))
        ((and (app? pat) (app? subj))
          (and-then
            (loop (app->rator pat) (app->rator subj)
              s1 s2)
            (lambda (theta)
              (and-then
                (loop (subst (app->rand pat) theta)
                  (app->rand subj)
                  s1 s2)
                (lambda (theta2)
                  (succeed (comp-subst theta theta2)))))))
        ((and (abs? pat) (abs? subj))
          (loop (abs->body pat) (abs->body subj)
            (cons (abs->bv pat) s1)
            (cons (abs->bv subj) s2)))
        (else (fail))))))
```

Figure 14: Code for α-matching

```
(define match-two-vars
  (lambda (v10 v20 s10 s20)             ; v1 is pattern variable
    (let ((v1 (var->sym v10))
          (v2 (var->sym v20)))
      (let loop ((s1 s10) (s2 s20))
        (cond
          ((and (null? s1) (null? s2))
           (succeed (make-unit-subst v1 v20)))
          ((or (null? s1) (null? s2))
           ;; don't fail here, raise an error immediately
           (error 'match-two-vars
             "bound-variable lists had different lengths: ~s and ~s"
             s10 s20))
          ((and (eq? v1 (car s1)) (eq? v2 (car s2)))
           (succeed (make-empty-subst)))
          ((or (eq? v1 (car s1)) (eq? v2 (car s2)))
           (fail))
          (else
            (loop (cdr s1) (cdr s2))))))))

(define match-one-var
  (lambda (vm1 subj s1 s2)
    (let ((v1 (var->sym vm1)))
      (if
        (and
          (not (memq v1 s1))
          (not (non-empty-intersection?
                 s2
                 (fv subj))))
        (succeed (make-unit-subst v1 subj))
        (fail)))))
```

Figure 15: Code for α-matching, continued

```
(define structural-induction
  (lambda (grammar-up ih-l-up rules-up) ; up="unparsed"
    ;; parse everything
    (let* ((grammar (parse-grammar grammar-up))
           (ih-l (parse-ihs ih-l-up))
           (rules (parse-rules rules-up))
           (rewriting-tacfn            ; glom rules into a tacfn (val->tac)
             (or-else-use
               (append
                 (rules->report-tacfns rules)
                 (list (report-tac "beta" (leftmost beta)))))))
      ;; do each production
      (let prod-loop
        ((results '())
         (prods (grammar-productions grammar)))
        (if (null? prods)
          results
          ;; prove IH's for this production
          (induct-prod (car prods) grammar ih-l rewriting-tacfn
            (lambda (more-results)
              (prod-loop (append results more-results)
                (cdr prods))))))))))
```

Figure 16: Top-level Code for Proof by Structural Induction

```
(define (induct-prod prod grammar ih-l tacfn cont)
  (let* ((ordsigma (build-ordsubst prod))
         (s (apply-ordsubst ordsigma (prod-rhs prod)))
         (nonterm (prod-lhs prod))
         (nt-ihs (find-nt-ihs nonterm ih-l))) ; all IH's for this nonterm
    ;; prove correctness for each IH for this nonterm
    (let ih-loop
      ((results '())
       (ihs nt-ihs))
      (if (null? ihs)
        (cont results)
        ;; prove correctness for this ih for this prod
        (let* ((ih (car ihs))
               (ih-subst (make-unit-subst (var->sym (ih-var ih)) s))
               ;; generate induction equation goal (step (2) in alg.)
               ;; by applying ih-subst to both sides of ih.
               (goal-lhs (subst (ih-lhs ih) ih-subst))
               (goal-rhs (subst (ih-rhs ih) ih-subst))
               ;; build new "induction" rules, specialized by ordsigma:
               ;; then make them into tacfns
               (new-ih-tacfns
                 (build-ih-tacfns ih-l ordsigma))
               ;; glom ih-tacfns onto input tacfn.
               ;; Also add eta.
               (new-rewrite-tacfn
                 (repeat*
                   (or-else-use
                     (cons tacfn
                           (append new-ih-tacfns
                             (list eta-tacfn)))))))
          ;; do the proof, finally
          (prove
            (list (unparse goal-lhs) (unparse goal-rhs))
            new-rewrite-tacfn
            (lambda (result)
              (if result
                (ih-loop results (cdr ihs))
                (ih-loop
                  (cons (cons (unparse goal-lhs) (unparse goal-rhs))
                        results) (cdr ihs))))))))))
```

Figure 17: Top-level Code for Proof by Structural Induction, continued

```
(define (prove goal-up tacfn cont)
  (let ((goal-lhs (parse (car goal-up)))
        (goal-rhs (parse (cadr goal-up))))
    (printf "Reducing left-hand side...~%")
    (pretty-print (unparse goal-lhs))
    ((tacfn goal-lhs)
     (lambda (lhs-goesto backtrack)
       (printf "~%Reducing right-hand side...~%")
       (pretty-print (unparse goal-rhs))
       ((tacfn goal-rhs)
        (lambda (rhs-goesto backtrack2)
          (printf "~%lhs reduces to:~%")
          (pretty-print (unparse lhs-goesto))
          (printf "~%rhs reduces to:~%")
          (pretty-print (unparse rhs-goesto))
          (printf "~%Checking for alpha-congruence... ")
          (let ((answer
                  (alpha-congruent? lhs-goesto rhs-goesto)))
            (if answer
              (printf "true~%")
              (printf "false~%"))
            (cont answer)))
        (lambda ()
          (printf "~%rhs failed.")
          (cont #f))))
     (lambda ()
       (printf "~%lhs failed.")
       (cont #f)))))
```

Figure 18: Top-level Code for Proof by Structural Induction, continued

operates quickly: generation of a proof like the one in Figures 5–8 requires about 3 seconds on our Sparc-10 processor.

Because the algorithms were tightly determined by the mathematics and we used some simple coding standards for data structures, the system has been remarkably reliable: only two errors have been discovered since the system passed its initial tests.

6 Experience with the prover

We have exercised the theorem-prover on a variety of examples of the sort sketched above. Our examples have included conditional expressions, closure creation and procedure calls with lexical addressing and multiple arguments, correct compilation of tail-recursive code [5, 14], and a small parallel language [21].

Another set of examples proved by the theorem-prover arise from checking the operational semantics of the target machine. In our compiler-correctness framework, we require that the λ-term representing the meaning of the target code has reduction behavior that closely mimics the behavior of typical machine instructions. This means that the target language instructions are easily implementable by translation to single instructions or short sequences of instructions on conventional machines. For the compilers we have considered, requiring the meaning of each target-machine instruction to be a tail-form supercombinator [6, 10] turns out to be a reasonable choice.

By examining this reduction behavior, we can obtain an operational semantics for the target language. The soundness of this operational semantics must then be proved. For example, the operational semantics of the addition instruction might be given by

$$\langle (\mathbf{add}\, t), u, (\langle \mathbf{int}, v_2\rangle{::}(\langle \mathbf{int}, v_1\rangle{::}z))\rangle \Longrightarrow \langle t, u, (\langle \mathbf{int}, (v_1 + v_2)\rangle{::}z)\rangle$$

We then have to prove that the denotation of the left and right sides of this rule are the same. Our theorem-prover has been able to prove such soundness results for the examples cited above.

The theorem-prover's main current limits are induced by the fact that it depends on the strong normalization and Church-Rosser properties of the terms and rules we have used. Attempting to reason about non-normalizing terms like the fixed-point combinator Y, or adding rules that can be applied in both directions will lead very quickly to infinite loops. We hope to explore more complex tactics as a way around some of these difficulties.

7 Related Work

As mentioned above, our initial experiments were performed in an early version of Isabelle [4]. Isabelle was chosen because it seemed suited to this kind of higher-order equational reasoning, and it allowed for the introduction of new theories to manage the induction hypotheses. The results of the experiment were unsatisfactory, in part because the transcription of our problem into Isabelle was less transparent than we would have liked, and in part because it was more difficult than expected to control the search strategy.

Looking back at this experience, we now believe that the biggest problem was that the built-in proof strategies in Isabelle were too powerful. Isabelle used higher-order unification to locate possible applications of a rule. Humans, on the other hand, typically use much weaker strategies to determine if a rule is applicable. This realization led us to base our system on α-matching.

Another system we explored was Elf [16]. Elf is a powerful system for finding proofs in the LF logical framework [9]. Hannan has encoded in Elf the correctness of a compiler from the lambda-calculus to code for the categorical abstract machine, using the operational semantics of call-by-value as a specification of the source language [8]. Although the encodings are fairly readable once one understands the principles involved, the translation still requires some effort.

While there has been considerable work on rewriting systems and unifications for higher-order terms, most of it has considered unification or pattern up to β-convertibility rather than α-congruence [12, 13] Most of this work is also restricted to typed terms, a restriction that is unnecessary for us. The work closest to ours that we are aware of is the v-unification of Talcott [17]. Talcott gives an algorithm for deciding α-unification for a class of binding structures that generalize conventional λ-terms with contexts and holes. We conjecture that her algorithm specializes to ours when restricted to the class of terms we consider.

It should also be noted that the operational proofs in [14, 22] are much more first-order in flavor than the proofs considered here, and a powerful first-order theorem-prover like Boyer and Moore's Nqthm [2] may be more appropriate for these proofs. We have not yet investigated this issue.

Because successful theorem-provers and proof-checkers are large, complicated systems, we long resisted the temptation to build our own. We turned to construction only when we realized that we could take advantage of the extra structure in our problem to build a small system tuned to our needs.

8 Current Work

It would be useful to have a type checker which could be applied to proposed rules. Another possible development is the application of the theorem-prover to other kinds of inductions over lambda terms, such as those typically found in fixed-point reasoning or in reasoning about sequences.

To supplement the theorem prover, we are developing a consistency checker for patterns which automatically derives patterns not explicitly covered by a set of rules. This pattern completion tool is useful for soundness proofs, where it is important to examine and prove results for all possible cases.

9 Conclusions

We have built a small theorem-prover to do higher-order equational reasoning in the presence of induction hypotheses and other general rules. We were pleasantly surprised to find that such a simple strategy sufficed to do large portions of denotationally-based compiler correctness proofs.

Acknowledgements

Bolek Ciesielski struggled heroically with the Isabelle metalogic and with various systems problems to produce [4]. Conversations with Tobias Nipkow helped us isolate α-matching as the key problem.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[2] Robert S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[3] Rod M. Burstall. Proving Properties of Programs by Structural Induction. *Computer Journal*, 12(1):41–48, February 1969.

[4] Boleslaw Ciesielski and Mitchell Wand. Using the Theorem Prover Isabelle-91 to Verify a Simple Proof of Compiler Correctness. Technical Report NU-CCS-91-20, Northeastern University College of Computer Science, December 1991.

[5] William Clinger. The Scheme 311 Compiler: An Exercise in Denotational Semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[6] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge, MA, 1992. Also McGraw-Hill, Chicago, 1992.

[7] J. D. Guttman, V. Swarup, and J. Ramsdell. The VLISP Verified Scheme System. *Lisp and Symbolic Computation*, 1994. Submitted for publication.

[8] John Hannan and Frank Pfenning. Compiler Verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, 1992.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993. preliminary version appeared in *Proc. 2nd IEEE Symposium on Logic in Computer Science*, 1987, 194–204.

[10] R. J. M. Hughes. Super Combinators: A New Implementation Method for Applicative Languages. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

[11] Albert R. Meyer. What Is a Model of the Lambda Calculus? *Information and Control*, 52:87–122, 1982.

[12] G. Nadathur and D. Miller. An Overview of λ-Prolog. In Kowalski and Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 810–827, Cambridge, MA, 1988. MIT Press.

[13] Tobias Nipkow. Nigher-Order Critical Pairs. In *Proc. 6th IEEE Symposium on Logic in Computer Science*, pages 342–349, 1991.

[14] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP Verified PreScheme Compiler. *Lisp and Symbolic Computation*, 1994. submitted for publication.

[15] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[16] Frank Pfenning. Logic Programming in the LF Logical Framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[17] Carolyn Talcott. A Theory of Binding Structures and Applications to Rewriting. *Theoretical Computer Science*, 112:99–143, 1993.

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

[19] Mitchell Wand. Semantics-Directed Machine Architecture. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 234–241, 1982.

[20] Mitchell Wand. Loops in Combinator-Based Compilers. *Information and Control*, 57(2–3):148–164, May/June 1983.

[21] Mitchell Wand. Compiler Correctness for Parallel Languages. in preparation, August 1994.

[22] Mitchell Wand and Dino P. Oliva. Proving the Correctness of Storage Representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.

[23] Mitchell Wand and Zheng-Yu Wang. Conditional Lambda-Theories and the Verification of Static Properties of Programs. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 321–332, 1990.