Jana Dunfield (jcd@cs.pdx.edu) 31 May 2000

Thesis: Semi-Automatic Verification of Purely Functional Programs

1. Introduction

Since the 1960s, computer scientists have been interested in the verification of computer programs, by hand or automatically. Checking that a program obeys its specification is tantamount to showing it is bug-free (provided the specification is right!). Verification by hand is tedious and prone to error. Automatic verification presents itself as a solution to both problems (if one is willing to believe that the verifier itself is correct).

Fully automatic verification, however, is not practical. Even if we content ourselves with merely proving assertions about parts of the program, rather than showing a full specification, some assertions will be hard to prove automatically. So one must interact with the verifier. Theorem provers, such as HOL [3], are programs that assist in the creation of formal mathematical proofs. If a theorem prover is used, the verification occurs in a separate context; the machine maintains no links between the program and the verification. When the program is changed—and real programs change often!—it is not clear which verifications (if any) were rendered invalid. We would like that information to be maintained by the machine, and so a system built into the programming environment is desirable. Furthermore, the mapping from the program to the language of the theorem prover may not be trivial, and one must understand that mapping to interact effectively with the prover. Finally, one must understand how to use the theorem prover; theorem provers are not known for their ease of use.

Thus, we propose to build a tool that is an integrated part of the programmer's environment, and speaks one language well: the underlying language in which the program is written. It follows that the language of the assertions we desire to prove should match the underlying language: similar syntax, similar semantics. So-called predicates for use in assertions, such as `sorted` to determine whether a list of numbers is in increasing order, can be written as ordinary functions in the underlying language.

The underlying language is functional and pure. By "functional" we mean a language that is expression- and function-oriented, rather than command-oriented. A lack of program state and side effects, known as purity, allows clean equational reasoning about the meanings of programs. Since functions cannot depend on state, `x = y` implies `f(x) = f(y)`. (In addition, if `f(x)` does not terminate, `f(y)` won't either—an equivalent meaning.) If the language had side effects, we would need to supplement the assertion language with a way to talk about those effects, or else substantially limit the verifier's applicability.

Complete automation is not practical, but a worthwhile degree of automation is achievable. Existing tools, such as the Stanford Validity Checker (SVC), can check the validity of certain classes of logical formulae without guidance by the user. Using such a system as a "back end" expedites the implementation of the proof tool. SVC knows nothing of such things as `case`

expressions, but it can handle the subset of the language which corresponds to first-order logic (and integer relations). Section 4.3 describes how the system could interface with SVC.

The following example illustrates some of the essential properties of the design. Suppose we have a function `min` to return the minimum of two integers:

```
fun min (x, y): int * int -> int =
    if x <= y then x else y end
...
```

This declares a function of type `int * int -> int`, that is, taking two integer arguments (named `x` and `y`) and returning an integer result. The return value is defined by the expression following '=': `if x <= y then x else y end`, which returns `x` if `x` is less than or equal to `y`, and `y` otherwise.

We decide to prove that, for all `x` and `y`, `min(x, y) <= x`. Within the programming editor, we enter—just as we would any other piece of text—a corresponding assertion.

```
fun min (x, y): int * int -> int =
    if x <= y then x else y end
...
{x: int, y: int. min(x, y) <= x}
```

The `x: int, y: int.` is a universal quantification over `x` and `y`: "for all *x*, *y* of integer type, $min(x, y) \leq x$". The system converts the newly entered assertion to a form acceptable by its back end, and then invokes the back end. However, since the system will not "open up" or look inside a function definition without user instruction, all the back end sees is a call of an unknown function "min." Clearly, without the body of `min`, `min(x, y) <= x` can't be proved. The failure is reported by a question mark next to the assertion:[1]

```
{x: int, y: int. min(x, y) <= x}?
```

Taking a cursory glance at the definition (body) of `min`, we think something like "it should follow from the definition of `min`." So we move the pointer over the assertion and option-click[2] `min`, directing the system to open up the call to `min` in the assertion.

```
fun min (x, y): int * int -> int =
    if x <= y then x else y end {_ <= x}?
...
{x: int, y: int. min(x, y) <= x}
```

An assertion has been created inside the body of `min`. It contains an underscore ('_', pronounced "this"), representing the value of the preceding expression. Furthermore, the system has moved the **?** from the original assertion to the new assertion. The original assertion hasn't been proved, so it has no ✓, but since *the original assertion follows from the new assertion,* it lacks a **?**. Thus, the symbols reflect the proper focus of the user's attention.

---

[1]The real system will likely use different assertion colors in addition to the symbols **?**, ✓.

[2]The Macintosh interface, with one mouse button, is assumed; users accustomed to other windowing systems may read "option-click" as "right-click".

Now we apply the (IfJoin) rule. There are several rules that can be applied to `if` expressions (see Section 4.4). We could choose (IfJoin) from a menu, but it's more convenient to option-click the `end` keyword.

```
fun min (x, y): int * int -> int =
    if x <= y then x {_ <= x}✓ else y {_ <= x}? end {_ <= x}
...
{x: int, y: int. min(x, y) <= x}
```

The assertion at the `then`-expression is proved (✓): `{_ <= x}` means `{x <= x}`, which is a simple tautology. The next assertion is not proved, because `{y <= x}` is not a tautology. To show that `y <= x`, we apply (IfElse), again by option-clicking a keyword, `else`.

```
fun min (x, y): int * int -> int =
    if x <= y then x {_ <= x}✓ else y {x > y}{_ <= x}✓ end {_ <= x}✓
...
{x: int, y: int. min(x, y) <= x}✓
```

The new assertion `{x > y}` is the negation of `{x <= y}`. `x > y` implies `y <= x`, so it is proved. Together these imply the assertion following `end`, which implies our goal `min(x, y) <= x`.

It's important to note that assertions assume termination. An assertion {P} claims only that, if evaluation of every involved expression terminates, then P must be true. We are proving partial correctness, not total correctness.

Section 2 describes the language of the examples. Section 3 discusses the user interface and gives a more involved example. Section 4 considers aspects of the design of the proposed system's internal structure: the inference graph, the interface to the back end, and some rules of inference. Section 5 discusses related work, and Section 6 concludes.

2. A small functional language

2.1. Overview

The design goals for our "toy" language are to simplify presentation (and eventual implementation) by avoiding superfluous features, while preserving most of the interesting aspects common to actual pure functional languages. The grammar is shown in Figure 2.1; {*a*} denotes zero or more repetitions of *a*, [*a*] indicates that either *a* or the empty string may appear. The right column gives brief descriptions.

The tokens of the language need little explanation. Integer literals consist of one or more decimal digits; identifiers ('id') begin with a letter and can be followed by one or more letters, digits, or underscores; there are several keywords (`fun`, `if`, and so on).

Assertions aren't really part of the language, but they are included in the grammar and typing rules to precisely specify their allowed positions and type behavior. Grammar productions only relevant to assertions are marked with '*'.

program	::=	{function-dec} in expr	Program
function-dec	::=	**fun** id args '->' type '=' expr	Function declaration
args	::=	'(' id {',' id} ')' ':' type	Names and types of arguments
type	::=	**int**	Integers
	|	**bool**	Booleans
	|	type **list**	Lists
	|	'(' type {'*' type} ')'	Tuples
expr	::=	integer-literal	Integer constant
	|	**true | false**	Boolean literals
	|	id	Named value
	|	expr binary-op expr	see binary-op
	|	**not** expr	Boolean ¬
	|	'(' expr ')'	Grouping
	|	id expr	App
	|	expr '::' expr	Construct a list: head::tail
	|	**nil**	The empty list
	|	'(' expr ',' expr {',' expr} ')'	Tuple
	|	'#' integer-literal expr	Selection from a tuple
	|	**if** expr **then** expr **else** expr **end**	If/then/else
	|	**case** expr **of nil** '=>' expr	
		'|' id '::' id '=>' expr **end**	Case on lists
	|	**let** id = expr **in** expr	Bind identifier to expression
	|	expr '{' assertion '}'	Assertion*
	|	'{' **assume** assertion '}' expr	Assumption*
	|	'_'	"This" (expr. to left of assertion)*
assertion	::=	[quantification] expr	Assertion*
quantification	::=	id ':' type {',' id ':' type} '.'	Universal quantification*
binary-op	::=	'+' | '–' | '*' | '/'	Integer arithmetic
	|	'<' | '>' | '<=' | '>='	Integer relations
	|	'=' | '<>'	Comparison (on any type)
	|	**or** | **and** | '=>'	Boolean ∨, ∧, ⇒

Figure 2.1: Grammar

A program consists of a series of function declarations, the keyword `in`, and an expression *e*. The result of a program is the result of evaluating *e*. Each declared function can apply (call) any *previously* declared function, including itself; furthermore, nested functions are not allowed by the syntax. This restriction disallows mutual recursion. The program body *e* appears after all functions and can apply any of them. Each function has one or more arguments.

Binary operators (`+`, `-`, `and`, etc.) are infix and have the usual precedences. Note that function application, `f x`, has higher precedence than any operator: `f x * 2` is equivalent to `(f x) * 2`.

2.2. Types

We define the type system of the language thus ($Val(\tau)$ denotes the set of values of type τ).

(int) 1. *int*, taking values from the set {…, -2, -1, 0, 1, 2, …}, is a type.

(bool) 2. *bool*, taking values from the set {`true`, `false`}, is a type.

(Tuples) 3. If $\tau_1, \tau_2, \ldots, \tau_n$ are non-arrow types, $(\tau_1 * \ldots * \tau_n)$ is a type. $Val(\tau_1 * \ldots * \tau_n) = (Val(\tau_1) \times \ldots \times Val(\tau_n))$, where × is Cartesian product.

(Arrows) 4. If α and β are non-arrow types, the "arrow" or function type α –> β is a type, with values from the set of functions from α to β.

(Lists) 5. If α is a non-arrow type, α *list* is a type whose values are drawn from $\{nil\} \cup (Val(\alpha) \times Val(\alpha\ list))$.

The "non-arrow" conditions forbid functions that take functions as arguments (higher-order functions). Integers can be of arbitrary size, to avoid consideration of overflow.

This is an example of a type rule:

$$\frac{\Gamma \vdash e_1 : \text{int} \quad e_2 : \text{int}}{\Gamma \vdash e_1 \ \textbf{\textit{op}}\ e_2 : \text{int}} \text{(ArithOp)}$$

where ***op*** is '+', '–', '*', or '/'

It states that, if an expression e_1 has type 'int' and an expression e_2 has type 'int', the type of e_1 ***op*** e_2 is also'int'. Figure 2.2 gives the type rules.

2.3. Semantics

Lisp-style lists are provided: `nil` is the empty list, `::` is the "cons" operator. The `case` construct permits the extraction of the head and tail. Algebraic datatypes (discriminated unions) are omitted: they can be simulated by lists and our proof techniques for lists could be generalized.

If evaluation of any of a user-defined function's arguments fails to terminate, evaluation of the function does not terminate: functions are strict, as in traditional imperative languages. The built-in operators are also strict, with the exceptions of `and`, `or`, and `=>` (implication), which are "short-circuiting", like SML's `andalso`/`orelse` or C's `&&`/`||`. `a => b` is shorthand for `(not a) or b`, so `b` is evaluated iff *a* evaluates to `true`.

let $x = e_1$ **in** e_2 binds an identifier x to an expression e_1 in e_2. It is exactly equivalent to e_2 with (e_1) substituted for all free occurrences of x in e_2.

if e_1 **then** e_2 **else** e_3 **end** evaluates a boolean expression e_1; if the result is **true**, e_2 is returned; otherwise, e_3 is returned.

case e **of nil** => e_1 | x::xs => e_2 **end** evaluates a list e. If e is **nil**, e_1 is the result; otherwise, e_2 is evaluated with x bound to the head of e and xs to e's tail.

'#' selects a component from a tuple. For example, #2 (10, 20, 30) = 20; #1 (x, y) = x.

The various binary operators behave as one would expect; the result of division by zero is undefined.

$\vdash n :$ int where n is an integer literal (Int)

$\vdash$ **true** : bool (True)

$\vdash$ **false** : bool (False)

$\Gamma \vdash x :$ a where $\Gamma(x) =$ a (Var)

$$\frac{\Gamma \vdash e_1 : \text{int} \quad e_2 : \text{int}}{\Gamma \vdash e_1 \; \textbf{\textit{op}} \; e_2 : \text{int}}$$ (ArithOp), where ***op*** is '+', '–', '*', or '/'

$$\frac{\Gamma \vdash e_1 : \text{int} \quad e_2 : \text{int}}{\Gamma \vdash e_1 \; \textbf{\textit{op}} \; e2 : \text{bool}}$$ (RelOp), where ***op*** is '<', '>', '<=', or '>='

$$\frac{\Gamma \vdash e_1 : \text{T} \quad e_2 : \text{T}}{\Gamma \vdash e_1 \; \textbf{\textit{op}} \; e_2 : \text{bool}}$$ (EqOps), where ***op*** is '=' or '<>', and T is any non-arrow type

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad e_2 : \text{bool}}{\Gamma \vdash e_1 \; \textbf{\textit{op}} \; e_2 : \text{bool}}$$ (LogOp) where ***op*** is 'or', 'and', or '=>'

$$\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash (\textbf{not } e) : \text{bool}}$$ (Not)

$$\frac{\Gamma \vdash e1 : \text{a}}{\Gamma \vdash \text{'('} \; e1 \; \text{')'} : \text{a}}$$ (Grouping)

$$\frac{\Gamma \vdash f : \text{a -> b} \quad expr : \text{a}}{\Gamma \vdash (f \; expr) : \text{b}}$$ (App)

$$\frac{\Gamma \vdash e_1 : \text{T}, e_2 : \text{T list}}{\Gamma \vdash (e_1 \text{ '::' } e_2) : \text{T list}}$$ (Cons)

$\vdash$ **nil** : α list (Nil)

$$\frac{\Gamma \vdash e_1 : t_1 \; \ldots \; e_n : t_n}{\Gamma \vdash \text{'('} \; e_1 \text{ ','} \ldots \text{','} \; e_n \text{ ')'} : t_1 * \ldots * t_n}$$ (Tuple)

$$\frac{\Gamma \vdash e : (t_1 * \cdots * t_n)}{\Gamma \vdash \text{'\#'} \; k \; e : t_k}$$ (Selection) where $1 \le k \le n$

$$\frac{\Gamma \vdash e_1 : \text{bool}, e_2 : \text{T}, e_3 : \text{T}}{\Gamma \vdash (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ end}) : \text{T}}$$ (IfThenElse)

$$\frac{\Gamma \vdash e : \text{a list}, \; e_1 : \text{b} \quad \Gamma + [x : \text{a}, xs : \text{a list}] \vdash e_2 : \text{b}}{\Gamma \vdash (\textbf{case } e \text{ of } \textbf{nil} \Rightarrow e_1 \mid x::xs \Rightarrow e_2 \textbf{ end}) : \text{b}}$$ (Case)

$$\frac{\Gamma \vdash e1 : \text{t1} \quad \Gamma + [x : \text{t1}] \vdash e2 : \text{t2}}{\Gamma \vdash (\textbf{let } x = e1 \textbf{ in } e2) : \text{t2}}$$ (Let)

$$\frac{\Gamma + [f : (T_1 * \cdots * T_n) \text{ -> } T_{result}, x_1 : T_1, \ldots, x_n : T_n] \vdash fbody : T_{result} \quad \Gamma + [f : (T_1 * \cdots * T_n) \text{ -> } T_{result}] \vdash e : T_e}{\Gamma \vdash (\textbf{fun } f \text{ '('} \; x_1 \text{ ':'} \; T_1 \text{ ','} \ldots, x_n \text{ ':'} \; T_n \text{ ')' '->'} \; T_{result} \text{ '='} \; fbody \; \textbf{in } e) : T_e}$$ (Fun) where for all i s.t. $1 \le i \le n$, T_i is not an arrow type

$$\frac{\Gamma \vdash e : T_e \qquad \Gamma + [\text{'\_'} : T_e, v_1 : T_1, \ldots, v_n : T_n] \vdash a : \text{bool}}{\Gamma \vdash e \text{ '\{'} \; v_1, \ldots, v_n \text{ . } a \text{ '\}'} : T_e}$$ (Assertion)

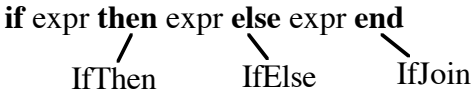
$$\frac{\Gamma \vdash e : T_e \qquad \Gamma + [v_1 : T_1, \ldots, v_n : T_n] \vdash a : \text{bool}}{\Gamma \vdash \text{'\{'} \; \textbf{assume } v_1, \ldots, v_n \text{ . } a \text{ '\}'} \; e : T_e}$$ (Assumption)

Figure 2.2: Typing rules

3. User interface

The user interface is central to the system: if the tool is not easy to use effectively, it will not be used. Therefore, there should be no needless "context switch" between editing a program and proving something about it. One consequence is that we have no special command for adding an assertion; the user enters it in the same way she would an identifier or function call or `if`-expression: by typing it.

Internal machinery should remain invisible whenever possible; for example, the user shouldn't have to think in terms of named inference rules. Hence, the extensive use of option-clicking (as seen in the 'min' example). The effect of option-clicking "end" and the effect of the menu command "IfJoin" must be learned in much the same way, but option-clicking is faster (one click, versus one click to select and a menu selection) and more direct. Conceptually, the mapping is between parts of grammar productions and menu commands:



We envision commands for showing and hiding all assertions, top-level assertions, assertions inside a particular function, assertions that depend on a particular assertion, and assertions that a particular assertion depends on. We assume the environment can present several views of the same program text. The displayed set of assertions is specific to each view. If an assumption has been introduced in a view, opening a new view allows one to proceed without that assumption or with a different set of assumptions. Thus, one can work on several proofs in parallel.

3.1. Example: Merge sort

This section describes a session with the goal of proving that a function implementing the classic mergesort algorithm fulfills part of its specification. Specifically, we prove that the result is sorted, but do not prove that it is a permutation of the input.

We begin with the program in Figure 3.1.

```
fun sorted (L): int list -> bool =
    case L of
      nil => true
    | x::xs => case xs of
                 nil => true
               | x2::xs => (x <= x2) and sorted (x2::xs)
               end
    end

fun merge (L1, L2): int list * int list -> int list =
    case L1 of
      nil => L2
    | x::xs =>
        case L2 of
          nil => L1
        | y::ys =>
            if x <= y then
                x::(merge (xs, y::ys))
            else
                merge (y::ys, x::xs)
            end
        end
    end

fun split (L): int list -> int list * int list =
    case L of
        nil => (nil, nil)
      | x::xs =>
          case xs of
              nil => (L, nil)
            | x2::xs =>
                let s = split xs
                in (x::#1 s, x2::#2 s)
          end
    end

fun mergesort (L): int list -> int list =
    case L of
      nil => nil
    | x::xs =>
        case xs of nil => L
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
        end
    end
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}?
```

Figure 3.1. Mergesort program, with the goal assertion.

We start by option-clicking **`mergesort`** to "open it up", placing `{sorted _}` after the function body:

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil
    | x::xs =>
        case xs of nil => L
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
        end
    end {sorted _}?
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

Option-clicking **`end`** applies the (CaseJoin) rule:

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil {sorted _}?
    | x::xs =>
        case xs of nil => L
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
        end {sorted _}?
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

There's a nested `case`, so we apply (CaseJoin) again, obtaining:

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil {sorted _}?
    | x::xs =>
        case xs of nil => L {sorted _}?
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

We option-click on the first `{sorted _}` assertion. The editing window is automatically split into two panels to show the definitions of `mergesort` and the newly opened `sorted`. The top panel displays `sorted`, with an assumption about `L` inserted:

```
fun sorted (L): int list -> bool =
  {assume L = nil}
    case L of
      nil => true
    | x::xs => case xs of
                 nil => true
               | x2::xs => (x <= x2) and sorted (x2::xs)
               end
    end {_}?
```

The `{_}` looks odd, but merely asserts that the previous expression (`case L … end`) is true. This is our goal. Since `L` is `nil`, it's logical to suppose we only have to prove that the `nil` arm satisfies the goal, i.e., is true. We type `{_}` in the appropriate spot. It's proved automatically (the system replaces the _ with the expression to the left, and the result—`true`—is trivially proved by the back end).

```
fun sorted (L): int list -> bool =
  {assume L = nil}
    case L of
      nil => true {_}✓
    | x::xs => case xs of
                 nil => true
               | x2::xs => (x <= x2) and sorted (x2::xs)
               end
    end {_}?
```

Above, in `mergesort`, we had a `case` and didn't know if the list was `nil`, so we applied the (CaseJoin) rule, which says that if an assertion is true of each case arm it's true of the whole `case`. Here, we know `L` is `nil`. We don't need to prove anything about the `x::xs` arm—we can use the (CaseNilArm) rule: if `L` is `nil`, and we can show something is true of the `nil` arm, it's true for the whole `case`.

(CaseNilArm) may be applied by option-clicking the **`nil`** of the case arm. But suppose we are confused and option-click the `x::xs` instead, which applies (CaseConsArm). (CaseConsArm) is the corresponding rule for the `::` case arm, applicable when `L <> nil`. `{L <> nil}` isn't an assumption or proved assertion, and a message appears near the bottom of the window: "CaseConsArm requires L <> nil".

We realize that's not what we wanted, and option-click `nil`.

```
fun sorted (L): int list -> bool =
  {assume L = nil}
    case L of
      nil => true {_}✓
    | x::xs => case xs of
                 nil => true
               | x2::xs => (x <= x2) and sorted (x2::xs)
               end
    end {_}✓
```

And in `mergesort`, we see a new checkmark:

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}? *
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

The assertion marked * can be proved in a very similar fashion. Now we show the recursive case. We do this by induction, assuming that `mergesort L1` and `mergesort L2` are sorted. We click on the recursive call `mergesort L1` and choose "Add Inductive Assumption" from a menu. The system inserts an induction assumption `{assume arg …}` and a version of the assumption specialized to `L1`.

```
fun mergesort (L): int list -> int list =
  {assume arg: int list. length arg < length L =>
                          sorted (mergesort arg)}
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}✓
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
                         {length L1 < length L => sorted (mergesort L1)}✓
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

Deciding to take `length L1 < length L` on faith, we *legislate* it: we type `{length L1 < length L}`, then choose "Legislate" from a menu. This is rather dangerous, but it may at times be the most effective way to proceed. The "legislation" is indicated by a **!**:

```
fun mergesort (L): int list -> int list =
  {assume arg: int list. length arg < length L =>
                          sorted (mergesort arg)}
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}✓
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
                         {length L1 < length L}!
                         {length L1 < length L => sorted (mergesort L1)}✓
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

With that done, the system can easily prove `{sorted (mergesort L1)}`. We repeat for `mergesort L2`, and arrive at the rather cluttered stage shown below. Note that the **!**'s propagate, so assertions that depend on "legislation" are clearly marked as such.

```
fun mergesort (L): int list -> int list =
  {assume arg: int list. length arg < length L =>
                          sorted (mergesort arg)}
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}✓
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         merge (mergesort L1, mergesort L2)
                         {length L1 < length L}!
                         {length L1 < length L => sorted (mergesort L1)}✓
                         {sorted (mergesort L1)}!
                         {length L2 < length L}!
                         {length L2 < length L => sorted (mergesort L2)}✓
                         {sorted (mergesort L2)}!
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
```

Now, if we can show that `merge` of two sorted lists produces a sorted list, we can prove `{sorted _}`. Or can we? Before heading off to prove `{L1,L2. sorted L1 and sorted L2 => sorted (merge (L1, L2))}`, we verify that our goal would follow if we did. After asking the system to hide some of the assertions, we add the assertion shown in bold:

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}✓
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         {sorted (mergesort L1)}!
                         {sorted (mergesort L2)}!
                         merge (mergesort L1, mergesort L2)
                         {sorted _}?
        end {sorted _}
    end {sorted _}
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}
  {L1: int list, L2: int list.
        sorted L1 and sorted L2 => sorted (merge (L1, L2))}?
```

Now we "legislate" it.

```
...
  {L1: int list, L2: int list.
        sorted L1 and sorted L2 => sorted (merge (L1, L2))}!
```

For technical reasons (SVC's inability to handle nested quantifiers), quantified assertions such as the one just added can't be used directly. `L1` and `L2` have to be instantiated to specific values (the system selected the values automatically for the induction assumptions). To instantiate, we select `mergesort L1` and drag it onto any of the `L1`'s in the assertion; the system adds a new assertion:

```
            ...
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         {sorted (mergesort L1)}!
                         {sorted (mergesort L2)}!
                          merge (mergesort L1, mergesort L2)
          {L2: int list. sorted (mergesort L1) and sorted L2 =>
                     sorted (merge (mergesort L1, L2))}!
                         {sorted _}?
            ...
```

Note the propagated **!**. We now drag `mergesort L2` onto `L2`.

```
fun mergesort (L): int list -> int list =
    case L of
      nil => nil {sorted _}✓
    | x::xs =>
        case xs of nil => L {sorted _}✓
                 | y::ys =>
                     let r = split L  in
                     let L1 = #1 r    in
                     let L2 = #2 r    in
                         {sorted (mergesort L1)}!
                         {sorted (mergesort L2)}!
                         merge (mergesort L1, mergesort L2)
 {sorted (mergesort L1) and sorted (mergesort L2) =>
                     sorted (merge (mergesort L1, mergesort L2))}!
                         {sorted _}!
        end {sorted _}!
    end {sorted _}!
in
   mergesort [2, 5, 17, 6, 7]
{L: int list. sorted (mergesort L)}!
  {L1: int list, L2: int list.
        sorted L1 and sorted L2 => sorted (merge (L1, L2))}!
```

The system has proved the ultimate goal (that is, if our legislated assertions hold). It's worth noting that even though we hid the inductive assumption

```
  {assume arg: int list. length arg < length L =>
                          sorted (mergesort arg)},
```

it remained in the system's internal representation. Also notice that the conclusion isn't guarded by the inductive assumption: we have

```
{L: int list. sorted (mergesort L)},
```

not

```
{(arg: int list, L: int list. length arg < length L =>
                                sorted (mergesort arg)) =>
 (L: int list. sorted (mergesort L))}.
```

One might be skeptical that this could really happen so conveniently. After all, we began our proof of `{sorted nil}` the same way we began our proof of `{L: int list. sorted (mergesort L)}`, by opening up a function, and so one would expect that the same, *non*-inductive, rule would be applied in both cases. But in the second case, we *did* invoke the "Add Inductive Assumption" command—an obvious sign of intent to prove the result by induction.

Another objection might be: but we didn't explicitly indicate a base case! No, but in the traditional two-step process of well-founded induction:

1. Prove P for all minimal elements;
2. Assuming P holds for all predecessors of *n*, prove P for *n*.

the first step is an instance of the second: if *n* is a minimal element, it has no predecessors, and the assumption says nothing.

Thus, there is an equivalent one-step formulation:

> Assuming P holds for all predecessors of *n*, prove P for *n*.

The system uses this single-step process, and so the base case(s) need not be indicated.

We omit the remainder of the example; it requires similar techniques (including induction to prove `sorted L1 and sorted L2 => sorted (merge (L1, L2))`).

4. System design

4.1. Introduction

The design centers around an *inference graph*, which encodes dependencies among assertions. In addition, it includes information associating assertions and positions in the program. To maintain that association, the prover must track changes as they are made. This imposes some requirements on the programming environment.

At minimum, the environment must be able to give the prover an accurate description of program changes as they occur. Changes can be effected by editing operations within the environment, of course, but also via another text editor, text-processing tools, and so on. The scope of editing within the environment can be ascertained easily: when the system inserts or deletes such-and-such a piece of text, it can pass on the text range and text to the prover. Changes outside the environment are trickier; the environment must save an additional copy of the text when it closes the file and, when it opens the file, compare its copy to the newer version. Then it can report the differences between the two to the prover.

Beside those minimal requirements, however, it is helpful for the environment to maintain concrete and abstract syntax trees—preferably including the result of a type analysis[3]—of the program. The prover needs a parsed version, and (to typecheck the assertions) the types of identifiers; if the environment produces and maintains this information, it can be used by other tools (such as an integrated browser or debugger). There are algorithms for incremental parsing and other means of maintaining syntax trees and type information; the details are beyond the scope of this paper.

The next section describes the inference graph itself.

4.2. The inference graph

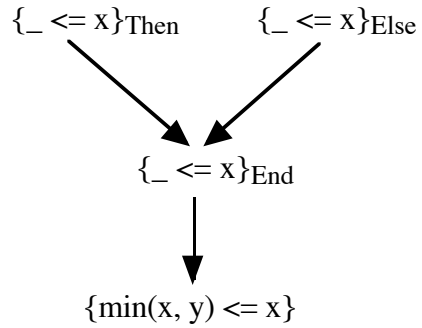
Each node corresponds to an assertion (or assumption). The directed edges indicate the dependencies between the assertions: if a node Q has predecessors P_1, …, P_n, then Q can be proved using (P_1 and … and P_n). Take the example from the Introduction (subscripts have been added to distinguish between the three {_ <= x} assertions):

---

[3]The language used in this paper is not hard to type, but full-blown functional languages have more elaborate type systems.

```
fun min (x, y): int * int -> int =
    if x <= y then x {_ <= x}Then else y {_ <= x}Else end {_ <= x}End
...
{min(x, y) <= x}
```

Here is the inference graph:

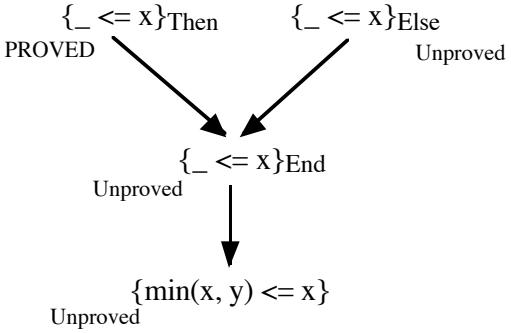


The graph does *not* claim that $\{\_ <= x\}_{Then}$ alone implies $\{\_ <= x\}_{End}$; it says that $\{\_ <= x\}_{Then}$ together with $\{\_ <= x\}_{Else}$ implies $\{\_ <= x\}_{End}$.

Each node has some associated information: the text of the assertion, a description (such as an offset into the text) of the assertion's position in the program, a status indication, an indication of whether the assertion has been "legislated", and (unless no attempt has been made to prove the assertion) the inference rule which justifies the assertion. The status indication is either "Proved" or "Unproved". Every assertion with one or more "Unproved" predecessors is "Unproved". Assertions with no predecessors may be "Proved" (by the back end, or because they follow directly from the program structure), or "Unproved".

The ✓ and **?** marks seen by the user are closely related to the internal status indication. The set of assertions marked ✓ is the same as the set that are "Proved". On the other hand, "Unproved" assertions may be marked either by **?** or not at all. To the user, the interesting unproved assertions are those whose nodes don't have predecessors. In the situation below, we know {min(x, y) <= x} can be proved on the basis of $\{\_ <= x\}_{End}$—the interesting assertion at this stage in the proof $\{\_ <= x\}_{Else}$. So nodes with no predecessors are marked **?**, and nodes *with* predecessors are unmarked.

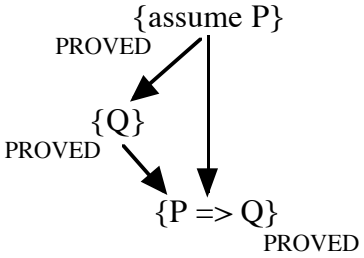
```
fun min (x, y): int * int -> int =
    if x <= y then x {_ <= x}Then✓ else y {_ <= x}Else? end {_ <= x}End
...
{min(x, y) <= x}
```



Assumptions, too, appear as nodes in the inference graph. Assumptions never have any predecessors and always have "Proved" status. The user or system might ask this question: what assumptions is an assertion contingent upon? It would appear that if there is a path from an assumption node, {assume P}, to an assertion node {Q}, then {Q} is contingent upon {assume P}. This is correct, except where we eventually lift {Q} out of the assumption (with the (Deduce) inference rules), resulting in {P => Q}. Although {P => Q} depends on {assume P} and {Q}, in the sense that the proof of {P => Q} is the proof of {Q} assuming P, {P => Q} is *not* contingent on P. It exists in a context where P has not been assumed. The (Induction) rule is similar: its conclusion is *not* made the induction assumption.

To handle this case, we modify the contingency test slightly:

If there is a path from an assumption node, {assume P}, to an assertion node {Q}, such that neither {Q} nor any intermediate nodes in the path are justified by the (Deduce) or (Induction) rules, then {Q} is contingent upon {assume P}.



Above, {Q} is contingent on P, but as {P => Q} is justified by (Deduce), it is not contingent.

Legislated assertions are similar to assumptions. Like assumptions, they have no predecessors and are always "Proved". The question of contingency on a legislated assertion (which is instrumental in determining if a **!** should be displayed) is analogous to the same question about an assumption. The difference is that legislated rules have no exceptional cases: there is no way to discharge a legislated assertion.

Finally, note that the commands to show and hide assertions depending on a selected assertion, and the assertions the selected assertion depends on, can be implemented by simple graph traversal.

4.3. Verification

An external verification engine ("back end") proves assertions that don't follow directly from the program structure. The back end should directly support all the language's types: arbitrary-size integers, booleans, tuples, and lists. It should produce intelligible counterexamples for invalid propositions (the system itself would not use such counterexamples, but the user might). It should never require user intervention. Without user intervention, it's unclear whether a full-blown theorem prover would be substantially more useful than a "lightweight" checker such as SVC [1].

SVC has some shortcomings. True, it can deal with integer equations and inequalities, but only if no overflow (arithmetic results outside $\pm 2^{31}$ or so) occurs. Non-linear equations are beyond its

capabilities. It has a theory of flat records (good for tuples), but it cannot handle instances of *recursive* datatypes like lists. If it deems a formula invalid, it reports a counterexample (a set of values for free variables such that the formula is false), but does not give the kind of information offered by a theorem prover. If it is clear from the counterexample what additional premise(s) must be included, this level of information is adequate. Finally, if a formula strays outside its domain, it fails with little grace: "`**** Result of Addition out of Range  Abort (core dumped)`." However, the system could shield the user from these messages. Despite these flaws, SVC remains a viable back end. Source code is provided, and adding recursive type handling and other features appears feasible.

The remainder of this section sketches a procedure for transforming assertions into a form acceptable to a back end that supports all the language's types (e.g. an extended SVC).

Suppose the system asks the back end to check an assertion {Q} which follows an expression *e*: *e* {Q}. The proposition passed to the back end has the form {Z}

$$\forall\, v_1, \ldots, v_m \,.\, \big((P_1 \wedge \ldots \wedge P_n) \Rightarrow C\big)$$

where the P's are the "premises", C is the "conclusion", and $v_1, \ldots, v_m$ are the free variables of C. To transform Q into C, we:

1. Replace any occurrences of'_' with *e*.
2. Expand every let-expression: `let x =` e_1 `in` e_2 becomes e_2[`x` / e_1].
3. Replace each `if` and `case` by a unique identifier *v*, unless the `if` or `case` is syntactically identical to another `if` or `case` already assigned an identifier. (Thus,

   `(if x = 0 then 1 else 2 end) = (if x = 0 then 1 else 2 end)`

   becomes `(`v_1 `=` v_1`)`, which the back end can show is valid.)

The premises $P_1, \ldots, P_n$ can be drawn from available unquantified assertions/assumptions in the program. If too few are included, the back end may lack a proposition needed to prove the conclusion. If too many are included, several issues arise. An assertion {A} may be included which is *not* needed to prove C, creating an inference graph with a superfluous edge. This confuses the inference graph displayed to the user, and it leads to unnecessary verifier activity if {A} is subsequently removed or rendered invalid. Most dangerous, though, is this situation. Say we'd like to prove {Z}, and we've decided that a lemma {Q} is needed. If Z is among the premises given to the back end to prove Q, a cycle in the graph results:

Some things are always safe to include in the list of premises. Assumption nodes never have incoming edges, so circularity can't result. Assertions in "Proved" status won't create a cycle, either. A good first approximation (until we have experience with the tool), then, is to include all available unquantified assumptions/assertions, *except* those at the top level of the program.

Quantified premises can't be used because they lead to a proposition with nested quantifiers:

$$\forall\, v_1, \ldots, v_m \,.\, \big(((\forall w \,.\, \mathrm{P}_1) \wedge \ldots \wedge \mathrm{P}_n) \Rightarrow \mathrm{C}\big),$$

which the back end cannot accept.

Once the list of premises has been determined, each is converted by the means described above.

4.4. Inference rules

The inference rules introduce valid assertions about programs based on other valid assertions. The user chooses (often by option-clicking) which rules to apply and where to apply them.

For example, this is a rule pertaining to `let` expressions:

$$\frac{\textbf{let } x = e_1 \ \{\mathrm{P}\} \ \textbf{in } e_2}{\textbf{let } x = e_1 \ \textbf{in } e_2 \ \{\mathrm{P}[x/\_]\}} \quad \begin{matrix}\text{(Let)} \\ \text{Axiom}\end{matrix}$$

Read it as "if P is valid and available after the first expression (e_1) of a **let**, then the assertion P[*x*/_]—P with *x* for 'this'—is valid following the second expression (e_2)". The precise meaning of "is available" will be defined shortly.

Implicit in each inference rule is that evaluation of all the expressions and assertions mentioned in the rule would terminate. So, to apply (Let), evaluating e_1, P, P[*x*/_], and e_2 should terminate. If they don't, the rule is not guaranteed to work (but the system, which does not reason about termination, will allow its application!).

In many cases, an assertion not at the given position above the line can be used as though it were. Such an assertion, at another position *p*, is said to be "available" at the given position *q*. The necessary criterion for availability is that the new assertion be valid at *q* if and only if it is valid at *p*. (Two assertions following each other, as *e* {P} {Q}, have the same position; both are trivially available just after *e*.) Assertions containing '_' are suspect; assertions inside `if`'s and `case`'s, as in the following example, are never available in surrounding scopes.

```
fun length xs : int list -> int =
    case xs of
       nil => 0
     | y::ys => (length ys + 1) {xs <> nil}
    end {xs <> nil}
```

Here, the first and second assertions are textually identical, but the first is valid and the second isn't. In the first, `xs` can't be `nil`: if it were, the `nil` case arm would be evaluated, not the `y::ys` arm. In the second, `xs` can be `nil`. Clearly, the first `{xs <> nil}` is not available following `end`.

An assertion {P} at position *p* is available at *q* if 1) it does not contain '_', and 2) it could be moved from *p* to *q* in the manner described below.

Rules for moving assertions have a natural form:

$$\frac{(e\ \{\mathrm{P}\})}{(e)\ \{\mathrm{P}\}} \qquad \frac{(e)\ \{\mathrm{P}\}}{(e\ \{\mathrm{P}\})}$$

Taken together, these mean that an assertion inside parentheses can be moved outside parentheses, and vice versa. For compactness, however, we use a different notation for most of the rules: the above two rules become

$$(e\ \{\mathrm{P}\})\ \{\mathrm{P}\}$$

Here are the rules:

$(e\ \{\mathrm{P}\})\ \{\mathrm{P}\}$

$(e_1\ \{\mathrm{P}\}\ op\ (e_2\ \{\mathrm{P}\}))\ \{\mathrm{P}\}$ where *op* is for any binary operator, including '::'

not $(e\ \{\mathrm{P}\}))\ \{\mathrm{P}\}$

#1 $(e\ \{\mathrm{P}\}))\ \{\mathrm{P}\}$ and likewise for #2, #3, etc.

$((f\ \{\mathrm{P}\})\ (e\ \{\mathrm{P}\}))\ \{\mathrm{P}\}$

$((e_1\ \{\mathrm{P}\})\ (e\ \{\mathrm{P}\}))\ \{\mathrm{P}\}$

$(e_1\ \{\mathrm{P}\}, e_2\ \{\mathrm{P}\})\ \{\mathrm{P}\}$ and likewise for *n*-tuples

$(e_1\ \{\mathrm{P}\}, e_2\ \{\mathrm{P}\})\ \{\mathrm{P}\}$

if $e_1\ \{\mathrm{P}\}$ **then** e_2 **else** e_3 **end** $\{\mathrm{P}\}$

case $e_1\ \{\mathrm{P}\}$ **of nil** => e_2 | x::xs => e_3 **end** $\{\mathrm{P}\}$

if $e_1\ \{\mathrm{P}\}$ **then** e_2 **else** e_3 **end** $\{\mathrm{P}\}$

let $x = e_1\ \{\mathrm{P}\}$ **in** $e_2\ \{\mathrm{P}\}$ where x does not appear in P

$$\frac{\textbf{fun}\ f\ xs : \ldots = e\ \ldots\ \textbf{in}\ program\ \{\mathrm{P}\}}{\textbf{fun}\ f\ xs : \ldots = e\ \{\mathrm{P}\}\ \ldots\ \textbf{in}\ program}$$ where no x in xs appears in P

$$\frac{\textbf{case}\ e_1\ \textbf{of nil} => e_2 \mid x::xs => e_3\ \textbf{end}\ \{\mathrm{P}\}}{\textbf{case}\ e_1\ \textbf{of nil} => e_2\ \{\mathrm{P}\} \mid x::xs => e_3\ \{\mathrm{P}\}\ \textbf{end}}$$

$$\frac{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}\ \{\mathrm{P}\}}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \{\mathrm{P}\}\ \textbf{else}\ e_3\ \{\mathrm{P}\}\ \textbf{end}}$$

An assumption {assume P} *e* is available iff an assertion following *e* would be available.

That concludes our treatment of availability; we now give the regular inference rules. All rules carry the unstated condition that the top and bottom are type-correct, according to the rules in Section 2. For clarity, we omit type declarations (for function arguments and quantified variables in assertions).

What follows is not necessarily a complete listing of all the rules. Rules that follow from the semantics of the language are marked "Axiom", rules that can be derived from other rules are marked "Derived". The "Axiom" rules should be proved sound with respect to a formal semantics of the language, and the "Derived" rules proved sound by constructing derivations from the "Axiom" rules, but we have not done so.

Basic rules of quantifier-free first-order logic could be specified here, but the back end is quite capable of such feats as inferring `Q` from `P => Q` and `P`. Instead of those rules, we give (Backend):

$$\frac{e\ \{P_1\}\ \{P_2\}\ \ldots\ \{P_n\}}{e\ \{Q\}}$$ (Backend)
where the back end deems $((P_1 \wedge \ldots \wedge P_n) \Rightarrow Q)$ to be valid.

For assumptions, we have

$$\frac{\{\text{assume } P\}\ e\ \{Q\}}{e\ \{P \Rightarrow Q\}}$$ (Deduce)
Axiom

Function calls are characterized thus (the double line indicates an equivalence, rather than a one-way inference: the top can be inferred from the bottom, as well as the bottom from the top):

$$\frac{\text{fun } f\ (x1, \ldots, xn) = body \text{ in } e\ \{P\}}{\begin{array}{c}\text{fun } f\ (x1, \ldots, xn) = body \text{ in } e \\ \{P\ [body[a1/x1, \ldots, an/xn]\ /\ (f\ (a1, \ldots, an))]\}\end{array}}$$ (Beta)
Axiom

$$\frac{\text{fun } f\ xs = e\ \{P\} \text{ in } e2}{\text{fun } f\ xs = e \text{ in } e2\ \{xs.\ P[f\ xs/\_]\}}$$ (Lift)
Axiom

$$\frac{e\ \{v_1, \ldots, v_n \,.\, P\}}{e\ \{v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \,.\, P\ [e/v_i]\}}$$ (InstantiateOne)
Axiom

$$\frac{e\ \{v_1, \ldots, v_n \,.\, P\}}{e\ \{P\ [e_1/v_1, \ldots, e_n/v_n]\}}$$ (InstantiateAll)
Derived from (InstantiateOne)

$$\frac{e\ \{v_1, \ldots, v_n \,.\, P\}}{e\ \{w_1, \ldots, w_n \,.\, P[w_1/v_1, \ldots\ w_n/v_n]\}}$$ (RenameQuantifiers)
where the w's contain no duplicates (i.e., for all i, j, if $i \neq j$, $w_i \neq w_j$)
Axiom

$$\frac{e\ \{P\}}{e\ \{P[\_/e]\}}$$ (This1)
Axiom

$$\frac{e\ \{P\}}{e\ \{P[e/\_]\}}$$ (This2)
Axiom

$$\frac{\text{let } x = e_1\ \{P\} \text{ in } e_2}{\text{let } x = e_1 \text{ in } e_2\ \{P[x/\_]\}}$$ (Let)
Axiom

$$\frac{\text{let } x = e_1 \text{ in } e_2}{\text{let } x = e_1 \text{ in } e_2 \{x = e_1\}}$$ (LetEq) Derived from (Let): write _ = e_1 for P

$$\frac{\text{if } e \text{ then } e_1 \{X\} \text{ else } e_2 \{Y\} \text{ end}}{\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ end } \{(e \Rightarrow X) \text{ and } (\text{not } e \Rightarrow Y)\}}$$ (If) Axiom

$$\frac{\text{if } e \text{ then } e1 \text{ else } e_2 \text{ end}}{\text{if } e \text{ then } e1 \{e\} \text{ else } e_2 \text{ end}}$$ (IfThen) Axiom

$$\frac{\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ end}}{\text{if } e \text{ then } e_1 \text{ else } e_2 \{\text{not } e\} \text{ end}}$$ (IfElse) Axiom

$$\frac{\text{if } e \text{ then } e_1 \{P\} \text{ else } e_2 \{P\} \text{ end}}{(\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ end}) \{P\}}$$ (IfJoin) Derived from (If)

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \{P\} \\ \quad \mid h::t \Rightarrow e_2 \{Q\} \text{ end}\end{array}}{\begin{array}{c}\text{case … end} \\ \{((e = \text{nil}) \Rightarrow P) \text{ and } ((e <> \text{nil}) \Rightarrow Q)\}\end{array}}$$ (Case) where Q contains neither h nor t Axiom

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \\ \quad \mid h::t \Rightarrow e_2 \text{ end}\end{array}}{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \{e = \text{nil}\} \\ \quad \mid h::t \Rightarrow e_2 \text{ end}\end{array}}$$ (CaseNil) Axiom

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \\ \quad \mid h::t \Rightarrow e_2 \text{ end}\end{array}}{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \\ \quad \mid h::t \Rightarrow e_2 \{e = h::t\} \\ \text{end}\end{array}}$$ (CaseCons) Axiom

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \{P\} \\ \quad \mid h::t \Rightarrow e_2 \{P\} \text{ end}\end{array}}{\text{case … end } \{P\}}$$ (CaseJoin) Derived from Case

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \{P\} \\ \quad \mid h::t \Rightarrow e_2 \text{ end } \{e = \text{nil}\}\end{array}}{\text{case … end } \{P\}}$$ (CaseNilArm) Derived from Case

$$\frac{\begin{array}{l}\text{case } e \text{ of nil} \Rightarrow e_1 \\ \quad \mid h::t \Rightarrow e_2 \{P\} \text{ end } \{e <> \text{nil}\}\end{array}}{\text{case … end } \{P\}}$$ (CaseConsArm) Derived from Case

4.4.1. Induction

Reasoning about recursive functions requires a rule based on well-founded induction:

$$\frac{\begin{array}{l}\text{fun } f\ x : t_x \text{ -> } t_e = \{\text{assume } y : t_x \,.\, m(y) < m(x) \text{ => } P[y/x]\} \\ \qquad\qquad e\ \{P[\_/f\ x]\} \\ \qquad\qquad \text{in } e_2\ \{z : t_x \,.\, m(z) >= 0\}\end{array}}{\text{fun } f\ x = e \text{ in } e_2\ \{x \,.\, P\}}\ \text{(Induction)}$$

where m has type t_x -> int

The function m is an *induction measure* of the argument; its result type is `int`, but must always be nonnegative: $\{z : t_x \,.\, m(z) >= 0\}$. m yields a well-founded relation R on t_x: $x\ R\ y$ iff $m(x) < m(y)$.

For a function f taking a list as its argument, a common induction measure is the length of the list. In that case, the inductive assumption is $\{\text{assume } y \,.\, length(y) < length(x) \text{ => } P[y/x]\}$, and the rule says that if—using the inductive assumption that P holds for all lists shorter than x—we can show that P holds for x, we can conclude that P holds for all x.

The base case, $m(x) = k$ (where $m(x)$ is always at least k), is implicit. When $m(x) = k$, there is no y such that $m(y) < m(x)$ (recall that m's result is always nonnegative). So the induction assumption can't be used for the base case.

> **Note:** A base case is guaranteed to exist: Since $m(x) \geq 0$ for all x, the image of m has a lower bound. Frequently—in the case of *length*, for example—the lower bound is 0, but it may be any nonnegative integer.

The choice of m depends on the function f, but there are often "standard" m's for particular types. (If f has multiple arguments, think of them as constituting a single tuple argument.) For lists of any kind, *length* is a good candidate; for tuples, the sum of the standard m's for each component (so a function taking two lists could have as its induction measure the sum of the lengths).

5. Related Work

A significant precursor is an unpublished proposal by Mark Jones and colleagues at the Oregon Graduate Institute. They propose extending the Haskell language with a method for stating top-level properties. A programming environment would be extended with tools for keeping track of dependencies among properties, with the ability to attach "certificates" to the properties. One form of certificate would be a verification by an external theorem prover. They do not propose verifications with properties (assertions) *inside* functions, as we do.

The idea of assertions inside functions seemed very natural, but was probably inspired by a traditional method of proving imperative programs, in which assertions are placed between program statements [4]. Igarishi *et al.* [5] described a verifier for a subset of Pascal augmented with an assertion language; their system produced "verification conditions" which were to be given to a theorem prover. The conditions generated are very involved; it is not clear if they could be

simplified to the point where a human could provide useful assistance to the theorem prover, or whether using a pure functional language instead of Pascal would have a substantially positive effect.

Functional programs' similarity to the languages of theorem provers like HOL [3] has led to efforts to verify them within the theorem prover [9]. However, it's not especially easy to embed recursive functions in the theorem prover's language.

De Millo *et al*. take a skeptical view of the prospects for verification [2]. Among their many objections is the enormous length of verifications (not verification conditions, but they would doubtless object to the length of those as well), rendering them impossible for a human to check. We can't claim innocence on this score; our intended "back end", SVC, just reports "VALID" if it thinks a proposition is valid. We may only be making a virtue out of necessity, but if the system has only limited ability to figure things out for itself, so to speak, the propositions proven by the back end should remain fairly simple—simple enough to re-prove them by hand, if we wish.

Reps' dissertation [8] on language-based programming environments briefly mentioned the possibility of environment support for verification.

Meyer's object-oriented, non-functional Eiffel language [6, 7] includes assertion support. The assertions document the program, and can be written as uninterpreted comments or interpreted expressions. The latter have the same form (with two minor extensions, `old` and `strip`) as any Boolean-valued Eiffel expression. So they can be (quite helpfully) checked at runtime.

6. Conclusion

We believe an approach to verification which is integrated with the language and the environment will result in a highly usable, effective system. We have sketched a rather convenient user interface. The degree of automation described is quite low, but may already be adequate; more automation could be achieved through the addition of a simple "tactic", an algorithm for choosing and applying inference rules. For example, figuring out that `{sorted nil}` can be proved with the (CaseNilArm) rule is straightforward: notice that the body of `sorted` is a `case`, ask the back end to prove `{L = nil}`, get a "VALID" result, and apply (CaseNilArm).

The back end's inability to accept nested quantifiers is perhaps its most significant impact on the rest of the system. From a user interface standpoint, instantiating by hand appears somewhat inconvenient (to what degree remains to be seen); perhaps automatic instantiation would be feasible in certain cases.

Scalability can be considered along two dimensions: program size and language sophistication. In large programs, assertions could be part of module interfaces; ideally, the assertions would completely specify the module behavior, so one could prove client code correct without ever going inside the called module. To extend the system to a "realistic" pure language, such as the currently popular Haskell, appropriate treatment would have to be found for language features such as polymorphic types, higher-order functions, mutual recursion, and lazy evaluation. (Wadler [10]

popularized the fact that certain theorems about polymorphic and higher-order functions can be deduced "for free" from the types of the functions. One of these would simplify a part of the `mergesort` verification not included in the example.) Defining a formal semantics for the small language of this paper, and proving the inference rules sound, would not be unreasonable; for a realistic language, defining a formal semantics would be a major project in itself.

There are, no doubt, user interface issues and points of design we have not discovered, and that will be found only through experience with an actual system.

References

[1] Barrett, Clark, et al. Stanford Validity Checker home page.
`http://verify.stanford.edu/SVC/`

[2] De Millo, Richard A., Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Comm. ACM* **22**(5), 1979.

[3] Gordon, M.J.C., and T.F. Melham, eds. Introduction to HOL: a theorem-proving environment for higher-order logic. Cambridge Univ. Press, 1993.

[4] Gries, David. The science of programming. Springer-Verlag, 1981.

[5] Igarishi, S., R.L. London, and D.C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Inf.* **4** 145-182, 1975.

[6] Meyer, Bertrand. Eiffel: the language. Prentice Hall, 1992.

[7] Meyer, Bertrand. Object-oriented software construction (2nd ed.). Prentice Hall, 1997.

[8] Reps, Thomas W. Generating language-based environments. MIT Press, 1984.

[9] Slind, Konrad. Function definition in higher-order logic. In *Proc. 9th Intl. Conf. on Theorem Proving in Higher Order Logics*, LNCS v. 1125. Springer-Verlag, 1996.

[10] Wadler, Philip. Theorems for free! In *Proc. 4th Int'l Symposium on Functional Prog. Lang. and Comp. Arch.,* 1989.