PROGRAM UNDERSTANDING WITH THE LAMBDA CALCULUS

Stanley Letovsky
Department of Computer Science
Yale University
New Haven, CT 06520

ABSTRACT

A prerequisite of any attempt to build intelligent tools to assist in the programming process is a representation language for encoding programming knowledge. Languages that have been used for this purpose include the predicate calculus [5] and various program-schema languages [1,4]. This paper advocates a new candidate which is as expressive as the predicate calculus but more intimately connected with programming: the lambda calculus. Its advantages lie in its close resemblance to conventional programming languages, and in a straighforward model of inference by rewriting, which can be applied to automatic programming and program understanding. The use of the lambda calculus in an automatic program understander is described.

1 INTRODUCTION

The general goal of research in AI/Software Engineering (AI/SW) is to construct tools which automate aspects of the software development process that are presently carried out by expert programmers. Such tools need knowledge bases which encode the expertise currently possessed only by programmers. One important category of programming knowledge is the standard plans of programming: this category includes algorithms, data structures and associated operations, and simple cliches such as summing and counting. This paper describes an approach to notating these plans, and to performing mechanical inference with them.

II METHODS OF REPRESENTING PLANS

One might think that a programming language would suffice to notate the plans used by programmers: after all, programming languages are languages for notating programming knowledge. There are two problems with this idea. First, the syntax and semantics of programming languages tends to be rather too clunky and complex for the needs of mechanical inference. More importantly, the types of composition of concepts supported by programming languages, via language constucts such as subroutines, packages, abstract data types, or objects, correspond to only a subset of the ways that concepts can combine in a programmer's mind.

These limitations on the expressivity of programming languages have led AI/SW researchers to use more expressive languages for writing down the knowledge. These include a variety of program schema languages and the predicate calculus. A schema language (eg.,[4,1]) is usually a programming language of some sort augmented with pattern-matching variables. Plans are represented as incomplete programs, with variable parts. This approach lends itself well to syntactic matching on programs written in the base programming language, but the reasoning is subject to various types of errors, because the syntactic pattern matching can generate semantic nonsense. In particular, schema variables can be instantiated with code segments which destroy the dataflow relationships assumed by other parts of the schema.

Another approach is to represent the knowledge in the predicate calculus (PC). This method requires a PC representation of the semantics of the target programming language: axioms which state how variables behave, what conditionals do, and so forth. These axioms can be used to translate programs into complex assertions which decribe the behavior of the program. Plans can also be represented by complex axioms. [5,6] Problems with this approach are first, the difficulties of doing general purpose mechanical inference in the predicate calculus, and second, the unreadability of PC translations of programs. This unreadability makes it hard for programmers or designers of knowledge-based tools to write knowledge bases or to follow intermediate stages of a tool's reasoning.

In the remainder of the paper I describe another approach to representing programming knowledge using the lambda calculus (λC). The lambda calculus possesses both the semantic precision missing from the schema languages and the readability and ease of inference missing from the predicate calculus. It has been my experience in conctructing a series of prototype program analyzers based on each of the above types of representations that the lambda calculus lends itself far more easily than the other two candidates to the development of simple and powerful inference tools.

III THE LAMBDA CALCULUS

The lambda calculus is basically an idealized programming language. It was invented by Church [3] to analyze the mathematical underpinnings of computation. Since then it has been used to analyse the semantics of programming languages [9] and as the explicit model for several programming languages, including LISP, and more recently T [7] and Scheme [8]. The λ-calculus is a language of functional expressions over a set of symbols, eg., (+ 1 3) or (f x), together with λ, an operator for creating functions. For example, (λ(x)(+ x 1)) denotes a function which adds 1 to its argument.

The two most important kinds of inference in λC are λ-abstraction and its opposite, β-reduction.

$$(+\ 2\ 1) \Leftrightarrow ((\lambda(x)(+\ x\ 1))\ 2)$$

β-reduction ⇔ λ-abstraction

[1]Thli research was supported by NSF under 1ST grant #8505019.

Abstraction creates new functions, whereas β-reduction simplifies functions away. Interpreters for λC-based languages perform simulated β-reduction, usually augmented with primitive arithmetic procedures and the like. Compilers are also largely based on β-reduction. Abstraction, by contrast, has little mechanical application at present; constructing abstractions is the business of programmers. The program analyzer developed by the author relies heavily on λ-abstraction for understanding programs.

It should be noted that parameters of λ-expressions play a role analogous to schema variables of schema languages, and to universally quantified variables in the predicate calculus. The λ-calculus is expressive enough to represent all quantification; no additional constructs are needed [2].

IV PROGRAM ANALYSIS IN THE λ-CALCULUS

In reasoning about programs, two important tasks are program analysis and synthesis, which are the analogs of classical AI models planning and understanding in the programming domain. In these models a planner or understander has a knowledge base of *plans* which are used to solve a given problem or analyze a given solution. Planning consists of replacing (or expanding) top level goals with selected plans from the knowledge base. Understanding is the opposite process: a set of observed actions is identified as being an instance of some library plan. Both the planning and understanding processes produce the same final data structure: a hierarchical representation of the design of a program. The top layer of this hierarchy is called the *specification*, the bottom layer is the code.

In the author's program analyzer, the code, plans, and specifications are all represented using λC. The code and specifications are both sets of function definitions, which are semantically equivalent for a correct program. The design hierarchy is represented as a set of λC rewritings which transform one to the other. This section briefly describes the techniques used to represent code, specs, and plans, and the inference mechanisms used in analysis.

A. Representing Code

Representing code in λC requires a translation procedure from the target language into λC; because of λC's similarity to programming languages, such translation is not difficult. Loops are represented by recursions; GOTO's by function calls as described in [8]. Imperative language constructs related to dataflow by side effect, such as reference or assignment to variables, pose a more difficult problem. Memory allows causal effects to propagate during program execution in a way which the notation reflects only implicitly; to understand what is going on in such programs an analyzer needs to make the flow of data through memory explicit so it can be reasoned about. In the analyzer this is done by translating the imperative code into λC-expressions which represent memory explicitly, an approach which is sometimes used in the analysis of programming language semantics [9]. Memory is viewed as a function from *pointers* to values. Pointers are arbitrary objects: symbols, say, or numbers. States of memory will be represented by λ-expressions of the form:

```
(λ (pointer)
  (if (= pointer pointer1) value1
      (if (= pointer pointer2) value2)
      ...))
```

which I will refer to as *M-expressions*. The value of a pointer in some state (i.e., a variable reference) is found by applying the M-expression for that state to the pointer. Functions which depend on state must take an M-expression as an argument; actions which modify state must take an M-expression as an argument and return one. Actions are thus functions from states to states.

Within this framework assignment is defined as follows:

```
(define (:= loc M value)
  (λ (pointer)
    (if (= pointer loc) value (M pointer))))
```

The function := takes 3 arguments rather than the usual 2: the extra argument M is the M-expression describing memory just prior to the assignment. := returns a new M-expression which describes the state of memory after the assignment.

As an example, consider a program to compute the *triangle* of a number n, that is, the sum of the integers from 1 through n. In FORTRAN, it would look something like this:

```
      subroutine TRIANGLE ( n , answer)
      answer = 0
      do 100 i = 1,n
100           answer = answer + i
      return
      end
```

The translation of this into λC is as follows. Symbols beginning with M are M-expressions.

```
(define (triangle M0 n answer)
  (define (do-loop M)
    (if (> (M i)(M n)) M
        (let ((M1 (:= answer M (+ (M answer)
                                  (M i)))))
          (let ((M2 (:= i M1 (+ (M1 i) 1))))
            (do-loop M2)))))
  (triangle1 (:= i (:= answer M 0) 1)) )
```

Notice the way consecutive assignments get transformed to nested `lets` or `sets`; this pattern replaces the statement-block constructs of conventional languages. Issues relating to the scoping of the variable i are ignored here in the interests of brevity.

B. Analysing Code

The goal of analysis is to recognize code in terms of standard plans. Analysis may thus be viewed as a process of *factoring* out known plans from a given program, or rewriting the program in the most concise possible form, using standard plans in the knowledge base. Several types of rewritings are used in the course of analysis:

Knowledge-Based λ-Abstraction: The knowledge base contains definitions of standard plans, data objects and algorithms. When a subexpression of the target program matches the body of such a definition, the subexpression is replaced by a call to the library function, with arguments dervied from the match bindings. So for example, if the knowledge base contains the definition

```
(define add1 (λ (x)(+ x 1)))
```

then knowledge-based abstraction entitles us to rewrite the expression `(+ a 1)` to `(add1 a)`.

Rewrite Rules: The knowledge base contains rewrite rules of the general form

(λ (params) lhs) ⇒ (λ (params) rhs)

which encode a variety of simplifications and theorems. When a code subexpression can be parametrized to match *lhs*, it is rewritten to the *rhs* form.

β-Reduction: Used to simplify the target program in a variety of ways so that other kinds of analysis can proceed. In particular, reduction of complex M-expressions simplifies away dataflow through memory.

Recursion Elimination: The simple methods just described fail on recursive definitions, because recursive definitions can occur in arbitrarily many syntactic forms, and it is not feasible to anticipate each of these forms in the plan library. Recursion elimination transforms a recursive definition into an equivalent definition which is expressed in terms of operations on streams of data values – the sequence of values taken on by the loop variables during loop execution. Definitions expressed in this form can be further analyzed using the above methods.

The recursion elimination transformation is a λC adaptation of a technique developed by Waters [10] to analyze loops in the plan calculus. He showed that most loops can be viewed as compositions of a few primitive looping plans: enumerators, accumulators, filters, terminators, and maps. These plans can be viewed as operations on streams of data. For example, an *enumerator* generates a stream by repeatedly applying an operation to the previous element in the stream: an example is counting, which enumerates using the add1 operation, starting with 1 as an intial value. A running total is an accumulation with the operator +; accumulation is the same as APL's *scan* operator.

In the analysis of TRIANGLE, a few applications of β-reduction simplify the body of the do-loop subdefinition significantly. In the resulting definition, the argument to the recursive call is a simple M-expression which specifies the values of each loop variable at the end of each iteration as a function of the values at the start of the iteration. The recursion elimination algorithm is applied to definitions of this form. Space constraints permit only a brief sketch of the algorithm.

The first step is to construct *semi-steam* expressions for each loop variable: these are the potentially infinite (*semi*-finite) streams of values which each variable would take on if the loop never terminated. The semi-stream expressions are formed from the basic looping plans, plus fragments of the original definition, λ-abstracted according to the pattern of interdependence among the variables. For the do-loop in TRIANGLE, the semi-streams are:

```
(define i-semi-stream
   (enumerate (λ (a)(+ a 1)) (M i)))

(define answer-semi-stream
   (accumulate + (M answer) i-semi-stream))
```

These semi-streams must be truncated as dictated by the exit tests to yield streams. Each semi-stream must be truncated once for each exit test. There are two basic truncate operations: truncate and extrude. Truncate truncates a stream at the point where a predicate is satisfied, while extrude extrudes a stream to a specified length. Variables which are not explicitly tested by an exit test are extruded up to a length determined by the variables which are tested. The predicates supplied to these operations are formed by λ-abstracting the exit tests in the original definition. The streams for do-loop are:

```
(define i-stream
   (truncate (λ (a)(> a (M n))) i-semi-stream))

(define answer-stream
   (extrude (length i-stream) answer-semi-stream))
```

A new nonrecursive definition is constructed from these elements, in which each loop variable is set to the last element of its corresponding stream.

```
(define (do-loop M)
   (:= i (:= answer M (last answer-stream))
         (last i-stream)))
```

Application of the other rewrite methods can then proceed until the following specification is produced for the original program:

```
(define (triangle M n answer)
      (:= answer M (sum (integers-between 1 n))))
```

V CONCLUSIONS

The λ-calculus is a useful vehicle for encoding the planning knowledge needed by intelligent programming tools. An important strength of the language is the existence of a notation which is similar to conventional programming languages, but more expressive and more amenable to machine manipulation. A program analyser based on these ideas has been constructed which is currently able to transform simple programs into compositions of standard plans.

References

[1] David Barstow. *Knowledge-Based Program Construction.* Elsevier North Holland Inc., 1979.

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

[3] Alonio Church. The calculi of lambda conversion. ***Annals of*** *Mathematical Studies,* 6, 1951.

[4] W. L. Johnson and E. Soloway. Proust: knowledge-based program understanding. In *Proceedings of the 7th International Conference* ***on Software Engineering,*** IEEE, Orlando, Florida, 1983.

[5] Charles Rich. A formal representation of plans for the programmer's apprentice. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* pages 1044-1053, IJCAI, Vancouver, B.C., 1981.

[6] Charles Rich. *Inspection Methods in Programming.* Technical Report AI-TR-604, MIT AI Lab, 1981.

[7] Stephen Slade. *The T Programming Language: A Dialect of* ***LISP.*** Prentice Hall Inc., 1987.

[8] Guy Lewis Steele and Gerald Jay Sunman. ***The Revised Report*** *on SCHEME, a Dialect of LISP.* Technical Report AI-Memo-452, MIT AI Lab, January 1978.

[9] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[10] Richard C. Waters. A method for analysing loop programs. *IEEE Transactions on Software Engineering,* SE-5(3):237-247, 1979.