Lightweight and Generative Components I: Source-Level Components

Sam Kamin, Miranda Callahan, and Lars Clausen

{kamin,mcallaha,lrclause}cs.uiuc.edu
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract. Current definitions of "software component" are based on abstract data types — collections of functions together with local data. This paper addresses two ways in which this definition is inadequate: it fails to allow for lightweight components — those for which a function call is too inefficient or semantically inappropriate — and it fails to allow for generative components — those in which the component embodies a *method* of constructing code rather than actual code. We argue that both can be solved by proper use of existing language technologies, by using a higher-order *meta-language* to compositionally manipulate values of type *Code*, syntactic fragments of some *object language*. By defining a client as a function from a component to *Code*, components can be defined at a very general level without much notational overhead.
In this paper, we illustrate this idea entirely at the source-code level, taking *Code* to be `string`. Operating at this level is particularly simple, and is useful when the source code is not proprietary. In a companion paper, we define *Code* as a set of values containing machine-language code (as well as some additional structure), allowing components to be delivered in binary form.

1 Introduction

The programming world has long sought methods of dividing programs into reusable units, fundamentally altering the way in which programs are created and dramatically increasing productivity. Current notions of components as abstract data types (such as classes and, on a larger scale, COM objects and the like) have been very successful. However, we believe that to take the next step in increased productivity, two types of components will need to be accommodated that do not find a place in current component technologies. One of these is "lightweight" components — those that cannot, either due to cost or for semantic reasons, be implemented as function calls. Another are the "generative" components, those which are used to build other programs. We are particularly interested in

K. Czarnecki and U.W. Eisenecker (Eds.): GCSE'99, LNCS 1799, pp. 49–62, 2000.

the latter, as we believe the only way to effect major change in how software is constructed is to find a way to implement *methods* of computation — algorithms, idioms, patterns — rather than just programs.

The thesis of this paper is that a method for implementing such components is within our grasp, using existing programming technologies. Our proposal is both general and conceptually simple. Define a *Code* type, whose members represent the "values" of syntactic fragments of an imperative *object language*. Suppose these values can be built compositionally — that is, that the values of compound fragments can be derived from the values of their sub-fragments. Then, embed the *Code* type within a higher-order *meta-language*. The language will provide a variety of types that can be built over the *Code* type, such as lists of *Code* values, functions from *Code* to *Code*, etc. A client is then just a function from a component to *Code*, and the component can be any value that is useful to the client. Given a quotation/anti-quotation mechanism for defining constants of type *Code*, this proposal provides a powerful, general, and notationally clean mechanism for writing clients and components.

This paper introduces very little new technology. Rather, it demonstrates how existing technologies can be adapted to overcome the shortcomings of current software components. We do not pretend to present a complete solution, but rather to sketch a plausible approach, one which is marked by its simplicity and generality. A complete and practical solution would need to contend with issues of security and portability; would ideally be notationally even cleaner than what we have done; and would, of course, have a robust and efficient implementation. Some technical problems standing in the way of realizing these goals are discussed in the conclusions.

We operate here entirely at the source-code level. For us, *Code* is just "syntactic fragment" or, more simply, "string." Components are sets of values that aid in the construction of source programs. (Think of them as higher-order macros.) Source-level components can be useful in "open source" environments, and also provide the simplest setting in which to illustrate our thesis. A companion paper [6] extends this to binary-level components.

The next section elaborates on the basic idea of the paper and gives our first example. Sections 4 and 5 present two more examples. Section 6 discusses related work. The conclusions include a discussion on future research suggested by these ideas, as well as a brief synopsis of the work described in the companion paper.

2 *Code* and Components

Our central thesis is that a powerful component system can be obtained by the expedient of defining an appropriate *Code* type in a functional language (called the *meta-language*). This type represents the "value" of programs and program fragments in some *object language*. The "value" may be the code produced by that fragment, or the function it calculates (in the denotational semantics sense), or anything else, so long as values can be built compositionally — that is, the value of a compound fragment must be a function of the values of the fragments it contains.

Given the definition of *Code*, the meta-language provides a variety of useful type constructions, such as lists and functions, and is itself a powerful and concise language in which to program. A component is a value in the language, and a client is a function from the type of the component to *Code*. The implementation of a quotation/anti-quotation mechanism to facilitate the creation of *Code* values also add notational convenience.

In this paper, we give *Code* the simplest possible definition: *Code* = *string*. A more abstract approach would be to define code as abstract syntax trees, but it would make little difference for the examples in this paper; see the end of section 6 for a further discussion of using AST's.

All syntactic fragments correspond to values of type *Code*. There is no distinction made between different kinds of values: expressions, statements, etc. However, for clarity of exposition, we will use subscripts to indicate the intended kind of value. Bear in mind that *there is only one type of Code*; the subscripts are merely comments.

In our examples, the meta-language is a functional language similar to Standard ML [5], called Jr, and the object language is Java [4]. One syntactic difference between Jr and Standard ML is the anti-quotation mechanism of Jr, inspired by the one in MetaML [9]. Items occurring in double angle brackets `<<` $\cdots$ `>>` are syntax fragments in the object language. Thus, they can be thought of as constants of type *Code*, except that they may contain variable, "anti-quoted" parts, introduced by a backquote (`` ` ``); these are Jr expressions that must evaluate to *Code*.

A simple example is this "code template:"[1]

[1] The notation "`fn` *arg* `=>` *expr* `end` definitions a one-argument function in Jr. Note that the function is anonymous. It can be given a name, like any other value, by writing `val` *funname* `=` `fn` .... A more conventional-looking alternative syntax is `fun` *funname arg* `=` *expr*.

```
fn whatToPrint =>
   << public static void main (String[] args) {
         System.out.println(‘whatToPrint);
      }>>
end
```

This template is a function from an expression of type `String` to a function definition; i.e., the template has type $\mathit{Code}_{\mathit{expr.\ of\ type\ String}} \rightarrow \mathit{Code}_{\mathit{function\ def.}}$. To get the function definitions, we can apply it to a simple expression like `<<"Hello, world!">>`, or something more elaborate, like `<<args.length>0?args[0]:"">>`.

To summarize: what appears inside the double-angle brackets is in Java syntax and represents a value of type *Code*; within these brackets, the backquote introduces an anti-quoted Jr expression returning a value of type *Code*, which is spliced into the larger *Code* value.

3 A First Example: Sorting

Suppose we wish to provide a sorting function as a component. Our idea of a general-purpose component is that the programmer should be able to use the component with a minimum of bureaucracy, whether there are two data items to sort or a million; regardless of the types of the components; and with any comparison function.

To get started, here is the simplest version of this component, a procedure to sort an array of integers, using a simple but inefficient algorithm:

```
// Component:
val sortcomp =
  <<class sortClass {
      void sort (int[] A) {
        for (int i=1; i<A.length; i++) {
           int temp = A[i];
           int j = i-1;
           while ((j >= 0) && (temp < A[j]))
                { A[j+1] = A[j]; j--; }
           A[j+1] = temp;
        }
     }
   }>>
```

The client will use this component by loading it and then calling it in the usual way.[2]

[2] The `let` construct introduces a temporary name for a value. Using _ as the name simply means the expression is being evaluated for its side effects and the value will not be used. Function application is denoted by juxtaposition.

```
// Client:
let val _ = load sortcomp
in
  <<class SortClient {
      void useSortComponent() {
        int[] keys; ... sortClass.sort(keys); ...
      }
    }>>
end;
```

The `load` function turns a value of type $Code_{class}$ into an actual, executable, class definition.

To avoid the necessity of matching the type of the client's array with the type expected by the component, we abstract the type from the component:

```
// Component:
fun sortcomp typenm =
  <<class sortClass {
      void sort (`typenm[] A) {
        for (int i=1; i<A.length; i++) {
           `typenm temp = A[i];
           ... as above ...
        }
      }
    }>>;
```

The client must supply the type as an argument when loading the component, but otherwise does not change:

```
// Client:
let val _ = load (sortcomp <<int>>)
in  ... as above ...
```

To abstract out the comparison function, we could pass it as a second argument to the sort function. However, that is both inefficient and bureaucratic, particularly since Java does not allow functions as arguments; instead, we abstract the comparison function from the component as a function of type $Code_{exp} \rightarrow Code_{exp} \rightarrow Code_{comparison}$:

```
// Component:
fun sortcomp typenm compareFun =
  <<class sortClass {
     void sort (`typenm[] A) {
       ...
       while ((j >= 0) && (`(compareFun <<temp>> <<A[j]>>)))
       ...
} } >>;
```

Note that, since the arguments to `compareFun` are of type *Code*, in the body of the sort procedure they must be quoted. The client passes in an argument of the correct type:

```
// Client:
let val _ = load (sortcomp <<int>> (fn i j => <<‘i<‘j>> end))
in ... as above ...
```

The comparison function itself is a Jr function, so it is not quoted; however, it returns *Code*, so its body is quoted.

"Lightweight" components are those that, like macros, do not entail the creation of a new function or class, but simply add new code in-line. A lightweight sorting component would put a statement in the place of the call. In preparation for such an example, we now define the sort component in such a way that it provides two *Code* values: the sort procedure (optional) and a statement to be placed at the point of the call. More specifically, the second value is a function from the array being sorted to a statement. So we now think of the component as having type

$$\begin{aligned} &\mathit{Code}_{\mathit{type}} \rightarrow (\mathit{Code} \rightarrow \mathit{Code} \rightarrow \mathit{Code})_{\mathit{comparison}} \\ &\quad \rightarrow \mathit{Code}_{\mathit{fundef}} \times (\mathit{Code}_{\mathit{argument}} \rightarrow \mathit{Code}_{\mathit{call}}) \end{aligned}$$

This can accommodate either an in-line sort or a call to the provided sort routine, as above. Here is how we would provide exactly the functionality of our previous version of the component:[3]

```
// Component:
fun sortcomp typenm comparefun =
  [ <<class sortClass { ... as above ...  } >>,
    fn arg => << sortClass.sort(‘arg); >> end ];
```

Now the client must use the provided sort-calling code:

```
// Client:
fun sortclient [sortproc,sortcall] =
   let val _ = load sortproc
   in
     <<class SortClient {
          void useSortComponent() {
            ... int[] keys; ... ‘(sortcall <<keys>>) ...
       } }>>
   end;

sortclient (sortcomp <<int>> (fn e1 e2 => <<‘e1 < ‘e2>> end));
```

[3] The `[·,·]` notation creates a pair of values. The function `fn [x, y] =>` ... expects its argument to be such a pair, and binds `x` and `y` to its two elements.

In this case, the effect is exactly as in the previous version: the call to `sortClass.sort` is inserted directly into the client code (and the less-than comparison is inserted directly into the sort procedure).

Using the type given in (1), it is possible to insert in-line code beyond just a function call. We exploit this capability in the following component, adding an additional integer argument indicating the length of the array. If the length is 2 or 3, the sort is done in-line; if less than 100, the sort is done by insertion sort; otherwise, the component uses quicksort.

A[j];

```
// Component:
fun sortcomp typenm comparefun size =
  if (size < 4) // place in-line
  then
    let fun swap e1 e2 =
        <<if (!‘(comparefun e1 e2)) {
             ‘typenm temp = ‘e1; ‘e1 = ‘e2; ‘e2 = temp;
          }>>
    in [<<>>,  // no auxiliary class in this case
       fn arrayToSort =>
         if (size < 2) then <<>>
         else if (size == 2)
         then swap <<‘arrayToSort[0]>> <<‘arrayToSort[1]>>
         else // size == 3
           <<{ ‘(swap <<‘arrayToSort[0]>> <<‘arrayToSort[1]>>)
               ‘(swap <<‘arrayToSort[1]>> <<‘arrayToSort[2]>>)
               ‘(swap <<‘arrayToSort[0]>> <<‘arrayToSort[1]>>)
             }>>
       end ]
    end
  else // don’t in-line
    let val callfun =
       fn arg => << sortClass.sort(‘arg); >> end
    in [<<class sortClass {
            void sort (‘typenm[] A) {
               ‘(if (size < 100)  // use insertion sort
                 then <<... as above ...>>
                 else // size >= 100 - use quickstort
                      <<... definition of quicksort ...>>)
          } }>>,
        callfun]
    end;
```

The client calls the component just as before, but with the additional argument. Note that this argument is not a *Code* argument, but an actual integer.

Using `sortclient` defined above, the result of the call

```
sortclient (sortcomp <<int>> (fn e1 e2 => <<‘e1<‘e2>> end) 2);
```

would be to load nothing (there being no auxiliary class in this case) and to transform the client to

```
class SortClient {
  void useSortComponent() {
    ... int[] keys; ...
    if (!(keys[0] < keys[1])) {
      int temp = keys[0];
          keys[0] = keys[1];
          keys[1] = temp;
} ... } }
```

4 Example: Caching

If a programming technique can be formalized, it can be made into a component. One example is the technique of caching, which can have a dramatic impact on the running time of an algorithm.

Given a function `f`, we cannot simply define a new function, `cached_f`, to be the caching version of `f`. The reason is that `f` may have recursive calls that must be changed to calls to `cached_f`. To allow for these calls to be changed, the client must supply a function of type $Code_{recursive\ call} \rightarrow Code_{function\ body}$. In somewhat simplified form, the creator of the cached function does this:

```
                                    cached_f (x) {
                                      if (x not in cache)
ℱ = fn f => <<...‘f(x)...>> end ↦       cache entry for x =
                                          ℱ <<cached_f>>(x)
                                      return cache entry for x
```

The caching component is presented below. This version handles caching only for functions of a single integer, though versions for arbitrary arguments of arbitrary types can be developed fairly easily.

```
fun cacheComponent thecode cachesize unusedval =
   <<class cacheMaker {
      static int [] ncache = new int[‘(cachesize)];

      static int original (int x) { ‘(thecode <<cacher>>) }

      public static int cacher (int x) {
        if ((x < ‘(cachesize)) && (ncache[x] != ‘(unusedval)))
```

```
        return ncache[x];
      else {
        int newres = original (x);
        if (x < `(cachesize)) ncache[x] = newres;
        return newres;
      }
    }

    static int setupcache (int x) {
      int i=0;
      while (i < `(cachesize)) ncache[i++] = `(unusedval); }
      return cacher (x);
    }
  }>>;
```

A caching version of the Fibonacci function is shown below. The code fragment (a) shows how the use of the caching component, and fragment (b) is the resulting code.

```
(a) let fun fibonacci recursefn =
      <<if (x < 2) return x;
        else return (`recursefn(x-1) + `recursefn(x-2)); >>
    in cacheComponent fibonacci <<20>> <<-1>>
    end;

(b) class cacheMaker {
      static int [] ncache = new int[20];

      static int original (int x) {
        if (x < 2) return x;
        else return (cacher(x-1) + cacher(x-2));
      }

      public static int cacher (int x) {
        if ((x < 20) && (ncache[x] != -1)) return ncache[x];
        else {
          int newres = original (x);
          if (x < 20) ncache[x] = newres;
          return newres;
      } }

      static int setupcache (int x) {
        int i=0;
        while (i<20) { ncache[i] = -1; i++; }
        return cacher (x);
    } }
```

5 Example: Vector Operations

Figure 1 shows a component that provides vector-level operations: multiplication by a scalar, vector addition, (component-wise) vector multiplication, and vector assignment. It is very simple to provide a class containing such operations, particularly in Java, where arrays are heap-allocated. Part (a) of Fig. 1 shows such a component in outline.

The problem is that this component is quite inefficient. For one thing, it allocates a new array for each intermediate result, though this problem could be alleviated, at some cost in convenience, by providing each operation with an additional argument, the target array. The more difficult problem is that it calculates each intermediate vector separately, using a separate loop. It does no "loop fusion."

The component shown in part (b) of Fig. 1 treats each vector as a function from an index expression to a value expression. It does not perform any actual computation until a computed vector is assigned to another vector, at which point it constructs a single loop to move the calculated values to the target array. A simple client program is shown in part (c) (note that this can be a client of either version of the vector component), and parts (d) and (e) in Fig. 2 show the result of applying the client to the components in parts (a) and (b), respectively.

6 Related Work

Much of the research in programming languages can be said to concern the problem of componentizing software. Therefore, a great deal of work bears more or less directly on ours. What generally is accounted under the title of "components" nowadays are the kind of large-scale components exemplified by COM objects [8]. These are important advances in standardizing interfaces, implementing version control, and permitting component search, among other things. However, these components are not adaptable in the sense of our sort component, and by their nature, tend to be heavy-weight. Thus, though they allow for a new level of integration of software at a large granularity, they do not seem likely to change the day-to-day dynamics of programming.

We mention three other recent research efforts that are most closely related to this work: aspect-oriented programming, and Engler's 'C and Magik systems.

Aspect-oriented programming (AOP) [7] represents an attempt to change the programming process by separating *algorithms* from certain

```
(a) val adt_vectorcomp =
      [SOME <<class VectorOps {
                static double[] scale (double s, double[] A) {
                  double[] B = new double[A.length];
                  for (int i=0; i<A.length; i++) B[i] = s * A[i];
                  return B;
              } ... }>>,
       fn "copy" => fn B i => B end
        | "scale" => fn x A i => <<VectorOps.scale(‘x, ‘(A i))>> end
        | "add" => fn A B i => <<VectorOps.add(‘(A i), ‘(B i))>> end
        | "mult" => fn A B i => <<VectorOps.mult(‘(A i), ‘(B i))>> end
        | "assign" => fn A B => <<‘A = ‘(B 0);>> end
       end
      ];

(b) val fusing_vectorcomp =
      [NONE,
       fn "copy" => fn B i => <<‘B[‘i]>> end
        | "scale" => fn x A i => <<(‘x * ‘(A i))>> end
        | "add" => fn A B i => <<(‘(A i) + ‘(B i))>> end
        | "mult" => fn A B i => <<(‘(A i) * ‘(B i))>> end
        | "assign" => fn A B =>
            let val i = gensym "i"
            in <<for (int ‘i=0; ‘i<‘A.length; ‘i++)
                   ‘A[‘i] = ‘(B i);>>
            end end
       end
      ];

(c) fun vectorClient [vectorauxops,vectorfuns] =
      let val _ = loadif vectorauxops
          val copy = vectorfuns "copy"
          val scale = vectorfuns "scale"
          val add = vectorfuns "add"
          val mult = vectorfuns "mult"
          val assign = vectorfuns "assign"
      in <<class VectorClient {
             void useVectorOps (double[] A, double[] B, double[] C) {
               double[] result;
               ‘(assign <<result>>
                   (scale <<2.0>> (mult (add (copy <<A>>) (copy <<B>>))
                                        (copy <<C>>))))
               return result;
             }
           }>>
      end;
```

Fig. 1. Vector calculations: (a) Traditional; (b) Loop-fusing; (c) A client

```
(d) void useVectorOps (double[] A, double[] B, double[] C) {
      double[] result = VectorOps.scale(2.0,
                          VectorOps.mult(VectorOps.add(A, B), C));
      return result;
    }

(e) void useVectorOps (double[] A, double[] B, double[] C) {
      double[] result;
      for (int i1=0; i1<result.length; i1++)
        result[i1] = (2.0 * ((A[i1] + B[i1]) * C[i1]));
      return result;
    }
```

Fig. 2. Vector calculations (cont.): (d) The client using component (a); (e) The client using component (b)

details of their implementation — like data layout, exception behavior, synchronization — that notoriously complicate programs. These ancillary aspects of programs can be regarded as components in a broad sense, and indeed they are formally specified in their own aspect languages. Our approach is not as convenient notationally because the use of the other components cannot be implicit: the programmer has to know about, and plan for, the use of our components. On the positive side, our approach is fundamentally simpler in that it requires little new technology, and it is more general in that the mechanisms we use can handle any aspect.

The work of Engler, in two papers [2,1], is particularly close to ours. The earlier of these [2] describes a C extension, called `‘C` with a *Code* type and run-time macros. `‘C` differs from our work both in overall goals — `‘C` is intended mainly to achieve improved efficiency rather than to promote a component-based style of programming — and in technical details. There is no functional meta-language level in `‘C` — meta-computations are expressed in C itself. This implies that there are no higher-order functions or other convenient data types for manipulating code values, which we have found to be indispensable.

The more recent paper [1] describes Magik, an extension to the `lcc` [3] C compiler that gives users access to the internal abstract syntax tree form of programs. Both AOP and Magik, by virtue of their dependence on abstract syntax tree manipulations, are inherently compile-time approaches. Our use of a *Code* type with compositional semantics, together with ordinary abstractions found in any functional language, removes the dependence on access to the abstract syntax tree. This allows the same ideas to be implemented at the level of executable binaries (as shown in [6]), at the expense of some flexibility.

7 Conclusions

We have presented an approach to software components that is simple and general, and is based on well-known ideas in programming languages. Our suggestion is that higher-order macros written in a functional meta-language, to generate code in an imperative object language, can account for lightweight (i.e. in-lined) and generative components. More broadly, the idea is that the definition of a *Code* type — representing the "values" of phrases in the object language — with a quotation mechanism to allow simple construction of those values, is all the mechanism that is needed for a powerful component facility — the meta-language provides the rest. The definition of the *Code* type can vary, so long as values can be computed compositionally. In this paper, we define *Code* to be *String*, meaning that the "value" of a phrase is its textual representation. This leads to "source-level components."

Components and clients are written in a meta-language chosen for its power and conciseness, to produce code in an object language chosen by some other criterion (efficiency, portability, compatibility, etc.). The use of a functional meta-language is of critical importance, as this leads to a much more general macro facility while keeping the notational overhead reasonable. The overall approach — using higher-order, cross-language macros to create a simple, powerful, and easy-to-implement component system — has not, to our knowledge, been previously suggested.

One area for future research is to study ways to simplify the writing of components and clients. We hasten to point out that, in this view of components, higher-order functions over *Code* — like the component that returns a function the can be used by the client to invoke methods defined by the component — are essential. So a certain irreducible level of complexity is inherent in the approach. Still, some aspects of component- and client-writing could be simplified by the judicious use of types. For example, much of the quoting and anti-quoting could be made implicit if the types of various operations were known: if `f` is known to have type $Code \rightarrow Code$, then if the antiquoted fragment `‘(f <<x>>)` appears in a program, both the anti-quote symbol and the quotations on `x` can be eliminated.

Source-level components are a good way to illustrate the use of a higher-order meta-language with a *Code* type. However, in practice, binary-level components are more practical. They can be distributed in those (frequent) cases when the source code is proprietary, they are generally more efficient to use, and they allow a more dynamic use of new components. We have emphasized in the paper that the key requirement is a definition of the *Code* type that allows code values to be calculated compositionally. There is no inherent reason why machine language could not be included in such a value. In [6], we define *Code* as (roughly speaking)

$$Code = Environment \rightarrow MachineLang \times Environment$$

Here, *Environment* gives the locations of variables. This definition allows for partially-compiled components and permits the components to be distributed as executables. The examples in that paper are identical in spirit, and very similar in detail, to those given here.

References

[1] Dawson Engler. Incorporating applications semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, USA, 15–17October 1997.

[2] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg Beach, Florida, 21–24 January 1996.

[3] Chris W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[4] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[5] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML: Version 3. Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh, May 1989.

[6] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components ii: Binary-level components. September 1999.

[7] Kim Mens, Cristina Lopes, Bedir Tekinerdogan, and Gregor. Kiczales. Aspect-oriented programming. *Lecture Notes in Computer Science*, 1357:483–??, 1998.

[8] Dale Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[9] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 203–217, New York, June 12–13 1997. ACM Press.