
The Binder Library

version 1.1

A C++ Template Library
User's Guide and Reference Manual

Jaakko Järvi
Turku Centre for Computer Science
Lemminkäisenkatu 14 A, FIN-20520 Turku
Finland
jaakko.jarvi@cs.utu.fi

January 19, 2000

Copyright

Copyright (C) 1999, 2000 Jaakko Järvi (jaakko.jarvi@cs.utu.fi)

The Binder Library is free software; Permission to copy and use this software is granted, provided this copyright notice appears in all copies. Permission to modify the code and to distribute modified code is granted, provided this copyright notice appears in all copies, and a notice that the code was modified is included with the copyright notice.

This software is provided "as is" without express or implied warranty, and with no claim as to its suitability for any purpose.

Contents

1	Introduction	1
1.1	Contact Information	2
1.2	Installing the library	2
1.3	Portability	2
2	Tuples	2
2.1	Introduction	2
2.2	Tuple types	2
2.3	Constructing tuples	3
2.3.1	make_tuple	4
2.4	Accessing tuple elements	5
2.5	Copy construction and tuple assignment	5
2.5.1	Tiers	6
2.6	Efficiency of tuples	6
2.6.1	Effect on Compile Time	7
3	Binders	7
3.1	Introduction	7
3.2	Binders in the BL	8
3.3	Bind expression	8
3.3.1	Binder objects and the Standard Library	9
3.4	Target function	9
3.4.1	Argument types	9
3.4.2	Member functions as targets	10
3.4.3	Nonconst member functions as targets	11
3.5	About placeholders	12
3.6	Function composition with binders	12
3.6.1	Examples of function composition	12
3.6.2	Binding nullary functions	13
4	About using the library	13
4.1	Error messages	13

1 Introduction

The Binder Library (BL in the sequel) is a C++ template library, which implements

- a versatile argument binding mechanism for function objects
- generic tuple types

The library contains no binary library files, all template definitions are in a set of header files. The library code is standard C++.

1.1 Contact Information

The Binder Library home page is <http://www.cs.utu.fi/BL>. It is also available at the C++ Boost site: <http://www.boost.org>.

1.2 Installing the library

The library consists of a set of `.hpp` files. Place all these files into a subdirectory named `bl`. The library uses `#include <bl/foo.hpp>` to include the internal files. This means that the `bl` directory, or a symbolic link to it, should be located in a directory which is searched for include files while compiling.

Include `<bl/binder.hpp>` to use binders and `<bl/tuple.hpp>` to use tuples. The rest of the header files in the package are for internal use. Note further that `<bl/binder.hpp>` includes `<bl/tuple.hpp>`.

All definitions are placed in the namespace `boost`. If you prefer another namespace, define `BL_NAMESPACE` with some other name. Define `BL_NO_NAMESPACE` if you want the definitions in the global namespace.

1.3 Portability

The BL is written in standard C++, so it works with any standard conforming compiler. However, the current version has been tested only with GCC 2.95 (the library has some workarounds for noncompliant features in GCC). The author of the library is grateful for any information concerning problems or success in using the library with other compilers.

2 Tuples

2.1 Introduction

In general, a tuple (or n -tuple) is a fixed size collection of elements. Pairs, triples, quadruples etc. are tuples. In a programming language, a tuple is a data object containing other objects as elements. These element objects may be of different types. Tuples are convenient in many circumstances. In particular, tuples make it easy to define functions that return more than one value.

Some programming languages, such as ML, Python and Haskell, have built-in tuple constructs. Unfortunately C++ does not. To compensate this, the BL implements a tuple construct with templates.

2.2 Tuple types

A tuple type is an instantiation of the tuple template. The template parameters specify the types of the tuple elements. The current BL version supports tuples with 0-10

elements. If necessary, the upper limit can be increased up to, say, a few dozens of elements. The element can be of any C++ type, except:

- a volatile qualified type (not supported in the current version of the BL)
- a type that cannot be copied, i.e.:
 - classes which do not have a public copy constructor
 - arrays

However, a reference to such a type is a valid element type.

For example, the following definitions are valid tuple instantiations, where A, B and C are some user defined classes.

```
tuple<int>
tuple<double&, const double&, const double, double*, const double*>
tuple<A, int(*)>(char, int), B(A::*)(C&), C>
tuple<std::string, std::pair<A, B> >
tuple<A*, tuple<const A*, const B&, C>, bool, void*>
```

The following shows some invalid tuple instantiations:

```
class Y {
    Y(const Y&);
public:
    Y();
};
tuple<Y> // not allowed, objects of type Y can not be copied
...
tuple<char[10]> // not allowed: no array copy
```

Note however that `tuple<Y&>` and `tuple<char(&)[10]>` are valid instantiations.

2.3 Constructing tuples

The tuple constructor takes the tuple elements as arguments. For an n -element tuple, the constructor can be invoked with k arguments, where $0 \leq k \leq n$. Default values are used for the $n - k$ last unspecified arguments. For example:

```
tuple<int, double>()
tuple<int, double>(1)
tuple<int, double>(1,3.14)
```

Note that an argument can only be left unspecified, if it has a default initialisation.

```
class X {
    X();
public:
    X(std::string);
};
tuple<X,X,X>() // error: no default constructor for X
tuple<X,X,X>(string("Jaba"), string("Daba"), string("Duu")) // ok
```

In particular, reference types do not have default initialisation:

```
tuple<double&>() // error: a reference must be initialised explicitly
double d = 5; tuple<double&>(d) // ok
...
tuple<double&>(d+3.14) // error: can not initialise a reference
// with a temporary
tuple<const double&>(d+3.14) // ok, but dangerous: the element
// may become a dangling reference
```

In sum, the tuple construction is semantically just a group of individual elementary constructions.

2.3.1 make_tuple

Tuples can also be constructed using the `make_tuple` (cf. `std::make_pair`) helper functions. This makes the construction more convenient, saving the programmer from explicitly specifying the element types:

```
tuple<int, int, double> add_multiply_divide(int a, int b) {
    return make_tuple(a+b, a*b, double(a)/double(b));
}
```

By default, the element types are deduced to the plain nonreference types. E.g:

```
void foo(const A& a, B& b) {
    ...
    make_tuple(a, b);
}
```

The `make_tuple` invocation results in a tuple of type `tuple<A, B>`.

Sometimes the plain nonreference type is not desired, e.g. if the type cannot be copied. The programmer can control the type deduction and state that a reference to const or reference to nonconst type should be used. This is accomplished with two helper template functions: `ref` and `cref`. Any argument can be wrapped with these functions to get the desired type. The mechanism does not compromise const correctness, since an attempt to wrap a const object with `ref` is rejected. For example:

```
A a; B b; const A ca = a;
make_tuple(cref(a), b); // creates a tuple<const A&, B>
make_tuple(ref(a), b); // creates a tuple<A&, B>
make_tuple(ref(a), cref(b)); // creates a tuple<A&, const B&>
make_tuple(ref(ca)); // error, can't wrap a const with ref
```

Arrays as `make_tuple` arguments are deduced to reference to const types by default, so there is no need to wrap them with `cref`. For example:

```
make_tuple("Donald", "Daisy");
```

creates an object of type `tuple<const char (&)[5], const char (&)[6]>`¹. If a nonconst array type is needed, `ref` must still be used.

¹Note that the type of a string literal is an array of const characters, not `const char*`.

2.4 Accessing tuple elements

Tuple elements are accessed with the expression:

```
t.get<N>()
```

or

```
get<N>(t)
```

where `t` is a tuple object and `N` is a constant integral expression specifying the index of the element to be accessed. Depending on whether `t` is `const` or not, `get` returns the `N`th element as a reference to `const` or nonconst type. The index of the first element is 1. `N` must be between 1 and k , where k is the number of elements in the tuple. Violations of these constraints are detected at compile time. Examples:

```
double d = 2.7; A a;
tuple<int, double&, const A&> t(1, d, a);
const tuple<int, double&, const A&> ct = t;
...
int i = get<1>(t); i = t.get<1>(); // ok
int j = get<1>(ct); // ok
get<1>(t) = 5; // ok
get<1>(ct) = 5; // error, can't assign to const
...
double e = get<2>(t); // ok
get<2>(t) = 3.14; // ok
get<3>(t) = A(); // error, can't assign to const
A aa = get<4>(t); // error: index out of bounds
...
++get<1>(t); // ok, can be used as any variable
```

2.5 Copy construction and tuple assignment

A tuple can be copy constructed from another tuple, provided that the element types are element-wise copy constructible. Analogously, a tuple can be assigned to another tuple, provided that the element types are element-wise assignable. For example:

```
class A;
class B : public A {};
struct C { C(); C(const B&); }
struct D { operator C() const; }
tuple<char, B*, B, D> t;
...
tuple<int, A*, C, C> a(t); // ok
a = t; // ok
```

In both cases, the conversions performed are: `char`→`int`, `B*`→`A*` (derived class pointer to base class pointer), `B`→`C` (a user defined conversion) and `D` to `C` (a user defined conversion).

Note, that assignment is also defined from `std::pair` types:

```
tuple<float, int> a = std::make_pair(1, 'a');
```

2.5.1 Tiers

Tiers are tuples, where all elements are of nonconst reference types. They are constructed with a call to `tie` function template (cf. `make_tuple`):

```
int i; char c; double d;
    ...
tie(i, c, a);
```

The above `tie` function creates a tuple of type `tuple<int&, char&, double&>` (the same result could be achieved with the call `make_tuple(ref(i), ref(c), ref(a))`). A tuple, such as this, that contains nonconst references as elements can be used to 'unpack' another tuple into variables. E.g.:

```
int i; char c; double d;
tie(i, c, d) = make_tuple(1,'a', 5.5);
cout << i << " " << c << " " << d;
```

This code prints `1 a 5.5` to the standard output stream. A tuple assignment operation like this is found for example in ML and Python. It is convenient when calling functions which return tuples.

The tying mechanism works with `std::pair` templates as well:

```
int i; char c; tie(i, c) = std::make_pair(1, 'a');
```

2.6 Efficiency of tuples

Tuples are efficient. All functions are small inlined one-liners and a decent compiler will eliminate any extra cost. Particularly, there is no performance difference between this code:

```
class hand_made_tuple {
    A a; B b; C c;
public:
    hand_made_tuple(const A& aa, const B& bb, const C& cc)
        : a(aa), b(bb), c(cc) {};
    A& getA() { return a; };
    B& getB() { return b; };
    C& getC() { return c; };
};
hand_made_tuple hmt(A(), B(), C());
hmt.getA(); hmt.getB(); hmt.getC();
```

and this code:

```
tuple<A, B, C> t(A(), B(), C());
t.get<1>(); t.get<2>(); t.get<3>();
```

The tier mechanism may have a small performance penalty compared to using nonconst reference parameters as a mechanism for returning multiple values from a function. For example, suppose that the following functions `f1` and `f2` have equivalent functionalities:

```
void f1(int&, double&);
tuple<int, double> f2();
```

Then, the call #1 may be slightly faster than #2 in the code below:

```
int i, double d;
...
f1(i,d);           // #1
tie(i,d) = f2();  // #2
```

In vast majority of cases, this difference is of no significance. See [6, 7] for more in-depth discussions about efficiency.

2.6.1 Effect on Compile Time

To compile tuples quite many template instantiations must be performed. This may increase compile time. Depending on the compiler and the tuple length, it may be more than 15 times slower to compile a tuple construct, compared to compiling an equivalent explicitly written class, such as the `hand_made_tuple` class in section 2.6. However, as a realistic program is likely to contain a lot of code in addition to tuple definitions, the difference is probably unnoticeable: Compile time increases of 5 to 10 percentages were measured for programs which used tuples very frequently. With the same test programs, memory consumption of compiling increased between 22% to 27%. See [6, 7] for details.

3 Binders

3.1 Introduction

The Standard Template Library (STL) [2], now part of the C++ Standard Library [3], is a generic container and algorithm library. Typically, STL algorithms operate on container elements via function objects. These function objects are passed as arguments to the algorithms.

Any C++ construct, which can be called with the function call syntax, is a function object. The STL contains predefined function objects for some common cases (such as `plus`, `less` and `negate`). In addition, it contains *adaptors* for creating function objects from pointers to unary and binary functions, as well as from pointers to nullary and unary member functions. It also contains *binder* templates for creating a unary function object from a binary function object by fixing one of the arguments to a constant value. Some STL implementations contain function composition operations as extensions to the standard [4].

Despite these tools, the possibilities to use member and nonmember functions as function objects in STL algorithms are limited. The binding mechanism is restricted as well, allowing only one argument of a binary function to be bound. Further, the use of adaptors, binders and function composition operations together leads to lengthy expressions, which may be difficult to comprehend. The BL provides a solution to these problems:

- arbitrary arguments of practically any C++ function can be bound
- syntax is easy and intuitive
- function composition is supported with the same binding syntax

3.2 Binders in the BL

Binding is an operation, which creates a k -argument function object, a *binder object*, from an n -argument *bindable function object*, such that $k \leq n$. Some of the arguments of the original function object are bound to fixed values, the remaining k unbound arguments are called *free*. The free arguments are specified with special *placeholders* `free1` and `free2`.

To get the general idea, consider this example:

```
double gaussian(double x, double mean, double variance);
vector<double> y, z; double m, v;
...
transform(y.begin(), y.end(), z.begin(), bind(gaussian, free1, m, v));
```

Let `gaussian` be a function computing values from the Gaussian distribution. The `bind` expression creates a unary function object which calls `gaussian` every time it is invoked. Parameters `mean` and `variance` are bound to `m` and `v`, whereas `x` is left open. Obviously, the `transform` invocation computes the value of Gaussian function with mean `m` and variance `v` for each point in `y` and places the results in `z`.

3.3 Bind expression

The form for a bind expression is:

```
bind(f, bind_args)
```

where f is an n -argument function object (*target function*), and $bind_args$ is a list of n arguments. In the current version of the BL, $0 \leq n \leq 10$ must hold. $bind_args$ must be a valid argument list for f , except that any argument can be replaced with `free1` or `free2` (or another bind expression, see section 3.6). In addition, the BL defines two additional placeholder objects: `free1_ptr` and `free2_ptr`. Their usage is explained in section 3.4.2.

The result of a bind expression is either a nullary², unary or binary function object. The prototype of this function object is defined as follows:

- (i) The return type is the same as the return type of the target function.
- (ii) If $bind_args$ contains no placeholders, the function object that results is nullary.
- (iii) If $bind_args$ contains `free1` placeholders, but no `free2` placeholders, the resulting function object is unary. The argument type is the type of the k th parameter of the target function, where k is the index of the first occurrence of `free1` in $bind_args$.
- (iv) If $bind_args$ contains `free1` and `free2` placeholders, the resulting function object is binary. The first argument type is the type of the k_1 th parameter of the target function, where k_1 is the first occurrence of `free1` in $bind_args$. The second argument type is the type of the k_2 th parameter of the target function, where k_2 is the first occurrence of `free2` in $bind_args$.
- (v) If $bind_args$ contain `free2` placeholders, but not `free1` placeholders, the expression leads to a compile time error.

²A zero-argument function.

3.3.1 Binder objects and the Standard Library

Binder objects conform to the requirements of adaptable function objects, i.e., they define appropriate member typedefs. This means that they can be used as arguments to standard adaptors. For example, `bind1st(bind(f, free1, 1, 2), 3)` is a valid expression, though not very reasonable. A more realistic example of this is: `not1(bind(f, 1, free1))`.

3.4 Target function

The target function can be:

- a pointer to a nonmember or a static member function.
- a pointer to a const or nonconst member function.
- an STL adaptable function object: the class of the function object must define the typedefs:
 - `result_type`
 - `argument_type` (unary function objects only)
 - `first_argument_type`, `second_argument_type` (binary functions objects only)
- a BL bindable function object: the class of the function object must define the typedefs:
 - `result_type`
 - `argument_types` (a tuple type containing the argument types)

In the current version of the BL, the target function can not be a pointer to volatile qualified member function.

3.4.1 Argument types

Internally, the arguments (excluding the target argument) of the bind expression are stored as a tuple object. Consequently, the same constraints and rules apply to parameters of target functions which apply to tuple element types and constructing tuples with `make_tuple` (see 2.3.1):

- Volatile qualified types are not supported in the current version of the BL.
- References to nonconst objects are allowed, but the corresponding actual arguments in the bind expression must be wrapped with `ref`.
- References to const objects that cannot be copied are allowed, but the corresponding actual arguments in the bind expression must be wrapped with `cref`.

Hence, by default, the bound arguments are stored in the binder object as copies. The `cref` (or `ref`) wrapper state, that instead of a copy a const (or nonconst) reference to the argument should be stored. For example:

```

void foo(int& i); int i;
bind(foo, i); // error
bind(foo, ref(i)); // ok
...
class A { A(const A&); ... };
void bar(const A& a);
bind(bar, A()); // error: A can't be copied
bind(bar, cref(A())); // ok

```

Note that free arguments must not be wrapped with `ref` or `cref`.

3.4.2 Member functions as targets

If the target function is a member function, the first argument of *bind_args* is special. It is the object, whose member function is to be called. This *object argument* may be bound or free, like any other argument.

A bound object argument can be a reference or pointer to the object, the BL supports both cases with a uniform interface:

```

bool A::foo(int); A a; vector<int> ints;
...
find_if(ints.begin(), ints.end(), bind(&A::foo, a, free1)); // reference is ok
find_if(ints.begin(), ints.end(), bind(&A::foo, &a, free1)); // pointer is ok

```

The functionality is identical in both cases.

Similarly, if the object argument is free, the resulting binder object can be called both via a pointer or a reference.

```

bool A::foo(int); list<A> refs; list<A*> pointers;
...
find_if(refs.begin(), refs.end(), bind(&A::f, free1, 1));
find_if(pointers.begin(), pointers.end(), bind(&A::f, free1, 1));

```

Actually in both cases, the deduced type of the free argument is a reference rather than a pointer. The resulting binding object can, however, be called with a pointer, which is dereferenced when needed. In some rare cases this may not be the desired effect. Therefore the BL defines two additional placeholders: `free1_ptr` and `free2_ptr`. They can be used within object arguments to explicitly state that the free parameter should be deduced to the appropriate pointer type. For example:

- If the same placeholder is used in place of some other argument as well, the reference type may not be suitable. For example:

```

void A::foo(A);
void A::bar(A*);
A a;
...
bind(&A::foo, free1, free1)(a);
bind(&A::foo, free1, free1>(&a);
...
bind(&A::bar, free1, free1>(&a); // error
bind(&A::bar, free1_ptr, free1>(&a);
...
bind(&A::foo, free1_ptr, free1)(a); // error
bind(&A::foo, free1_ptr, free1>(&a); // error

```

The first two lines invoke the `foo` function as `a.foo(a)`. The third line is an error, as the library tries to call `a.bar(a)`. The fourth line is again ok: the `_ptr` variants of the placeholders state explicitly, that the free object variable is a pointer instead of a reference. The resulting call is `(&a)->bar(&a)`. The last two lines are erroneous. The first of them fails since `a` is not a pointer. The second fails, since it tries to invoke `(&a)->foo(&a)`.

- If the `bind` expression is used as an argument to some standard adaptor (say `not1`), the reference type may not be suitable. For example:

```
bool A::foo(int) const; A a;
...
bind(&A::foo, free1, 1)(a); // ok
not1(bind(&A::foo, free1, 1))(a); // ok
...
bind(&A::foo, free1, 1>(&a); // ok
not1(bind(&A::foo, free1, 1))(&a); // but this is an error
not1(bind(&A::foo, free1_ptr, 1))(&a); // this is ok
```

3.4.3 Nonconst member functions as targets

The binding mechanism is safe with respect to constness of member functions, analogously to constness of arguments. There is a small difference, though. While bound arguments of type reference to nonconst type must be wrapped with the `ref` function (see section 3.4.1), this is not necessary with the object argument of a nonconst member function. For example:

```
void A::foo_const(int&) const;
void A::foo(int&);
A a; const A const_a = a; int i;
...
bind(&A::foo_const, const_a, ref(i)); // ok
bind(&A::foo_const, a, ref(i)); // ok
bind(&A::foo, const_a, ref(i)); // error: nonconst member
// called with const object
bind(&A::foo, a, ref(i)); // ok: ref is not needed in the object argument
```

Note, however, that `cref` must still be used if the object argument can not be copied:

```
class B {
    B(const B&);
public:
    foo() const;
    bar();
};
...
B b; const B& cb = b;
...
bind(&B::foo, cb); // error, B cannot be copied
bind(&B::foo, cref(cb)); // ok
bind(&B::bar, b); // ok
```

3.5 About placeholders

The placeholders `free1`, `free2`, `free1_ptr` and `free2_ptr` are variables defined in unnamed namespace in `bl/binder.hpp`. The BL does not support any additional placeholders (such as `free3`, `free4`, ...), since no STL algorithms accept function objects taking more than two parameters.

3.6 Function composition with binders

An argument in the *bind_args* list of a bind expression can be another bind expression. This results in a powerful function composition mechanism.

A nested bind expression is of the form:

```
bind(f, args1, bind(g, args2), args3)
```

That is, any argument in the *bind_args* list can be a bind expression. This holds recursively: the level of nesting can be arbitrary.

The interpretation of a nested bind expression is a kind of a deferred function call. An argument of the *bind_args* list is replaced with a bind expression, which creates a function object. Each time the value of the argument is needed, the function object is called to provide it. In the above bind expression, each invocation of the binder object calls *g* and passes the result as a parameter to *f*.

Section 3.3 stated the requirements of a valid argument list for a bind expression. With nested bind expressions these rules do not have to hold for individual bind expressions. Instead, a combined argument list is formed by collecting the arguments of the full bind expression from left to right. In the example above, the combined argument list is *args1*, *args2*, *args3*. The combined list must fulfil the conditions for *bind_args* stated in section 3.3.

Here is another example. The expression

```
bind(f, free1, bind(g, a, bind(h, free1, free2)))
```

creates a binary function object. The combined *bind_args* list is (*free1*, *a*, *free1*, *free2*). When the binder object is invoked, say, with parameters *x* and *y*, it makes the call:

```
f(x, g(a, h(x, y)))
```

where the substitutions `free1=x` and `free2=y` have been done.

3.6.1 Examples of function composition

The example below converts a list of strings (*sl*) containing integral numbers to a vector of integers (*iv*):

```
list<string> sl; vector<int> iv;
...
transform(sl.begin(), sl.end(), iv.begin(),
          bind(atoi, bind(&string::c_str, free1)));
```

As another example, suppose that `point` is a function for drawing points on a canvas using some colour:

```
void canvas::point(double x, double y, colour c);
canvas* canv; vector<double> x, y;
```

To draw the points (in green) specified by the coordinates in vectors `x` and `y`, one could write:

```
for_each(x.begin(), x.end(), y.begin(),
         bind(&canvas::point, canv, free1, free2, green));
```

Note, that this two-sequence `for_each` algorithm is not part of the C++ standard library, it is however straightforward to write [5, p. 532]. Now, to represent the y-coordinates in logarithmic scale, the standard function `double log(double)` can be bound as follows:

```
for_each(x.begin(), x.end(), y.begin(),
         bind(&canvas::point, canv, free1, bind(log, free2), green));
```

3.6.2 Binding nullary functions

For completeness, nullary functions can be bound as well. This is convenient with function composition. For instance, the following code creates a vector of 100 random number pairs:

```
vector<pair<int, int> > pvec;
generate_n(back_inserter(pvec), 100,
          bind(&make_pair<int, int>, bind(rand), bind(rand)));
```

4 About using the library

4.1 Error messages

One problem with the BL is related to error messages. The BL is type safe, meaning that the compiler will catch such errors as wrong parameter types etc. While compiling BL bind expression, the diagnostic messages reporting errors tend to be lengthy and somewhat cryptic. The reason for this is that the error may be encountered only after several (recursive) template instantiation steps.

References

- [1] Järvi J.: *C++ Function Object Binders Made Easy*, In Generative and Component Based Software Engineering, to appear in Lecture Notes in Computer Science, 1999.
- [2] Stepanov, A. A. and Lee, M.: *The Standard Template Library*, Hewlett-Packard Laboratories Technical Report HPL-94-34(R.1), April 1994, <http://www.hpl.hp.com/techreports>.
- [3] *International Standard, Programming Languages – C++*, ISO/IEC:14882, 1998.
- [4] The SGI Standard Template Library, Silicon Graphics Computer Systems Inc., <http://www.sgi.com/Technology/STL>.
- [5] Stroustrup, B.: *The C++ Programming Language - Third Edition*, Addison-Wesley, Reading, Massachusetts, 1997.

- [6] Järvi J.: *Tuples and multiple return values in C++*, submitted for publication, see TUCS Technical Report No 249, 1999, <http://www.tucs.fi/publications>.
- [7] Järvi J.: *ML-Style Tuple Assignment in Standard C++ — Extending the Multiple Return Value Formalism*, TUCS Technical Report No 267, 1999, <http://www.tucs.fi/publications>.