Source-to-Source Optimization and Abstract Interpretation in the CLAIRE Programming Language

Yves CASEAU, François-Xavier JOSSET

Bouygues – Direction des Technologies Nouvelles

1, avenue Eugène Freyssinet
78061 Saint-Quentin-en-Yvelines cedex, France

{ycs,fxjosset}@challenger.bouygues.fr

Abstract

This paper demonstrates the use of abstract interpretation and code specialization for the translation from a high-level functional programming language into a standard object-oriented target language.

We report our experiments with CLAIRE, a high-level programming language for complex algorithms. This language includes paradigms which are usually associated with declarative languages, such as objects, sets, parametric data structures, iterators, high-order functions, production rules, and search primitives. The originality comes from the tight integration of these concepts.

We review here the use of abstract interpretation techniques in the source-to-source CLAIRE optimizer. Abstract interpretation is used for type inference, for detecting possible side effects or allocations, the possibility of run-time errors, the need for specific type checking, or for computing various complexity metrics. We use these different properties about code fragments to perform various source-to-source optimizations, such as the optimization of method calls, as well as the specialization of set iteration, which is one of the most salient features of CLAIRE. We present a generic specialization scheme based on evaluating the complexity expected for a given call. These different rewriting techniques also allow us to deal efficiently with advanced features such as procedural attachment, defeasible updates (the ability to backtrack) or the management of inverses. The multiple layers of optimization may be seen as a fixed-point computation and advocate the need for simplification rules in the generated code which we call composition polymorphism.

The CLAIRE language was originally designed to develop hybrid algorithms for combinatorial problems. These problems are complex to model and hard to solve from a computational point of view. Thus, it was necessary to combine expressiveness and efficiency. The source-to-source optimization techniques presented here aim at conciliating these two opposite goals.

Topics: Language design and implementation (7),
Analysis and design methods (1),
Reflection, adaptability, composability and reusability issues (10)

1. Introduction

CLAIRE [CJL99] is a high-level functional object-oriented language which was designed to express complex algorithms for combinatorial optimization applications (matching [CL97], scheduling [CL96b], time-tabling, routing [CL99]). These algorithms are called hybrid algorithms, because they use a blend of different techniques such as tree search (e.g., branch and bound), local search, constraint programming, linear programming or other specific search techniques or graph/flow algorithms. Designing a new hybrid algorithm is often done by reusing previous designs, but it is very difficult to isolate "black box" components and the preferred approach is the so-called "glass box" approach, using a source library of different algorithms. CLAIRE was designed to facilitate this task, by integrating useful concepts such as choice points and backtrack into the programming language, and by providing the ability to write at a higher level of abstraction (e.g., using abstract sets of objects). There was a double goal of using CLAIRE as a support tool for teaching algorithms and getting more elegant and readable source libraries of algorithms.

The key challenge when designing the CLAIRE language was to reconcile the need for efficiency that comes with the domain and the need for expressiveness that comes with the complexity of the algorithms. The technical solution to this challenge is a powerful source-to-source optimizer that allows the use of high-level expressions (which hides the implementation details and use abstract set of objects) without the expenses of unnecessary allocation (to build this sets) or dynamic interpretation (to adapt to a given data structure). CLAIRE being a reflective language, all sources are parsed into CLAIRE objects, onto which different layers of re-writing rules, implemented as methods, can be applied to produce faster but less readable source code. Once the fixed-point of rewriting is reached, target language code is generated using classical techniques (C++ or Java). The set of rewriting methods is user-extensible, giving the ability to further improve the optimizing ability of the compiler. The first version of CLAIRE was released 5 years ago; this paper describes a new version (3.0) of the system that includes significant innovations into the code optimizer.

The main contributions of the CLAIRE optimizer are the use of a rich type system (together with a flexible typing philosophy that supports static and dynamic typing) and the systematic use of abstract interpretation to derive various properties of code fragments. These more original contributions are efficiently mixed with more classical techniques (e.g., code specialization, inlining and the use of code complexity metrics). The "richer type system" means the use of complex parametric data types, second-order types (functions on types as types for functions) and meta-types (types that apply to reified expressions of the language). Abstract interpretation is used for type inference, for detecting possible side-effects or allocations, the possibility of run-time errors, the need for specific (by call) type checking, or computing various complexity metrics. This yields an interesting specialization scheme: when the complexity expected for a given call to a method is significantly lower than what is expected in general (because the type of the call parameters is more specialized than the signature of the method), we may create a specialized version of the method. This is a generic way to introduce specialization that captures many classical techniques such as constant propagation.

The paper is organized as follows. Section 2 gives an overview of the language and the architecture of the compiler. We present the CLAIRE type system that is used for objects, methods and instructions (types for reified instructions). We then explain our overall philosophy used for type inference and type checking. Section 3 presents the source-to-source optimizer and how different kinds of instructions and control structures are rewritten into lower-level and more efficient instructions. This includes the optimization of method call, as well as the specialization of set iteration, which is one of the most salient features of CLAIRE. We also show how the same techniques are used to prepare the target code generation phase, by pre-computing different properties through abstract interpretation. The last section presents a more novel use of abstract interpretation and complexity metrics to produce a generic and powerful specialization scheme. We compute the average expected complexity of a method call according to different indicators such as the number of dynamic bindings, possible run-time errors or undesired allocations. For each given call to the method, we then re-evaluate this complexity, with respect to the types of the input parameters. If there is a significant decrease in complexity, we create a specialized version of the original method. The fact that CLAIRE supports singletons as types means that most optimization schemes based on constant recognition and partial evaluation can be implemented with this approach.

2. The CLAIRE Optimizer

2.1. Overall Architecture

2.1.1. CLAIRE at a Glance

We first briefly describe the CLAIRE object-oriented data model. Classes are defined with a list of typed slots, and only single inheritance is supported. Each class is a first-class object of the system and is accessible at run-time (an optional list of instances is stored for each class). The <: operator is used to define a class (because subclasses are subtypes in CLAIRE); it consists of appending a list of new slots to the slots of the parent class. Slots support default values and unknown values, i.e., there is a special unknown object that represents the absence of value. The following two examples illustrate class definition; the second one shows a parameterized class (the usual Stack class, parameterized by the type of its elements).

```
Point <: object(x:integer = 0, y:integer = 0)
Stack[of] <: object(of:type, contents:list = nil)
```

Methods in CLAIRE are simply overloaded functions, where each parameter contributes to the overloading (the so-called multi-methods [CL94]) and each parameter can be typed with any CLAIRE type. For instance, attaching a method to a class union is useful to palliate the absence of multiple inheritance. Methods may be attached anywhere in the type lattice, not simply to the class hierarchy (more precisely, anywhere in the lattice of type Cartesian products). This makes CLAIRE closer to a language like Haskell than to C++. A method is called a restriction of the global function which itself is called a property.

It is important to notice that the class hierarchy is considered as a "law" given by the user (seen as set inclusion) and is not based on type substitution. As a consequence, there are no constraints imposed on the user when adding new methods. Covariance is enforced implicitly but not imposed explicitly (adding a new method on a subdomain may augment the range of the function on a larger domain. If we define:

```
foo(x:integer) : (0 .. 5) -> ...          // ... stands for "any expression"
foo(x:(1 .. 10)) : (5 .. 10) -> ...
```

the range of foo on integer is actually the union (0 .. 10). The range in the first definition is taken as a type restriction for this first definition, not for "foo on any integer" which is defined by the two restrictions. Covariance comes as no surprise since we are using an overloading approach, where the property is the "overloaded function" and the methods (i.e., the restrictions of the property for a given signature) are the "ordinary functions" using Castagna's terminology [Cas95].

Parameterized types can be used to produce parameterized signature. That means that we can use the value of a parameter (when it is a type) inside the signature, as in the following example:

```
push(s:Stack<X>, y:X) : void -> s.content :add y
```

Sets play an important role in CLAIRE. A set is used through testing membership, iteration, or combining with another set through a set operation. Membership is tested using the operation ∈. The for x in S e(x) is a useful control structure that evaluates an expression e for each element x that belongs to the set S. We have also introduced two convenient expressions:

- exists(x in S | P(x)) returns true if there exists an element x in the set S such that the predicate P(x) is true, and false otherwise;
- some(x in S | P(x)) returns an element x from the set S such that the predicate P(x) is true (if such a x exists), and unknown else (as in ALMA [AS97]);

For instance (refer to [CL96a] for further information), we may write:

```
x ∈ (string U list)
for y in (1 .. 10) U (20 .. 30)
        (if exists(c in (1 .. 10) | ok?(c + y))  choose(d,y))
when x := some(t in TASKS | completed?(t)) in
        register(x)                                    // t was found
else error("no completed task was found")          // x is unknown
```

New sets can be formed through selection (e.g., {x in S | P(x)}) or set image (e.g., {f(x) | x in S}) [SDD+86]. When duplicates should not be removed, a list image can be formed with list{f(x) | x in S}. Using a straightforward syntax leads to straightforward examples:

```
exists(p in {x.father | x in (man U woman)} | p.age < 20)
list{x.age | x in person}
for x in {f(y) | y in (1 .. 10)}  print(x)
```

To complete this very brief overview of CLAIRE, we need to mention the "versioning" mechanism. A version (also called a world, using the AI terminology) is a virtual copy of the state of the objects. The goal is to be able to return to a previously stored state if a hypothetical branch fails during problem solving. As a consequence, worlds in CLAIRE are organized into a stack and only two operations are allowed: one for copying the current state of the database and another for returning to the previous state. The part of the objects that supports these defeasible updates is defined by the programmer and may include slots, tables, global variables or specific data structures such as lists or arrays. Each time we ask CLAIRE to create a new world, CLAIRE saves the status of defeasible slots (and tables, variables...). Using these programming features simplifies writing search algorithms, such as branch-and-bound (with search primitives in CLAIRE or another language such as ALMA [AS97]).

CLAIRE is a reflexive language, in the sense that everything is an object (as in SMALLTALK). For a given object, much useful information is available (e.g., the class it is an instance, the position of that class in the class hierarchy). The optimizer uses systematically these reflexive features, because it can deduce a considerable amount of information that can help to produce the most accurate and optimized code.

2.1.2. The CLAIRE Compiler

The CLAIRE software is divided into three separate parts: the core of the system, the interpreter and the compiler. The core part consists of a run-time micro-kernel and a meta-declaration layer; the micro-kernel contains the implementation of run-time critical primitives (in C++ or Java), that are available as API functions in the CLAIRE side. The interpreter part gathers several basic functionalities, like the reader, the file manager, the object inspector, the pretty-printer.

The compiler is constituted of two distinct components: a source-to-source optimizer and a code generator (see Fig1). The optimizer transforms CLAIRE source code into CLAIRE optimized code (we will focus on the optimizer more in depth in Sections 3 and 4). Next, the optimized code is translated into a target language (today either C++ or Java) via the code generator. The residual, native code can then be compiled, linked with libraries, inserted into another program, and so on.

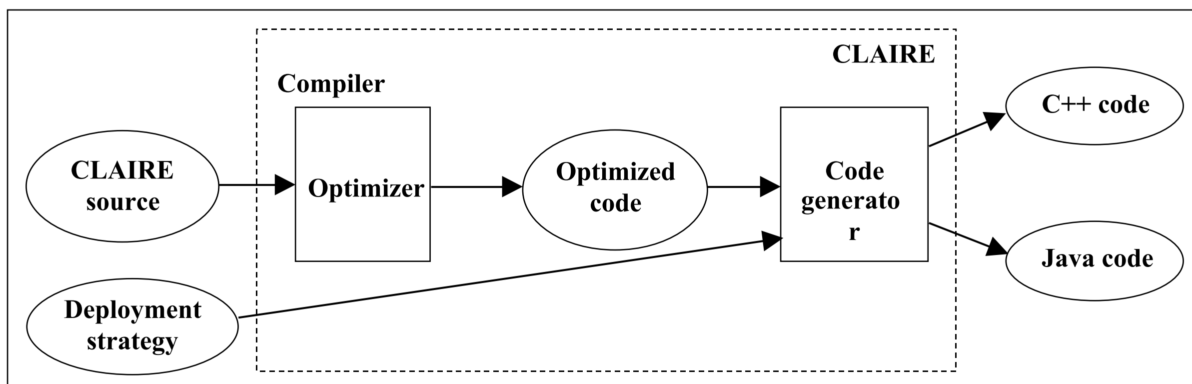

Fig1. Compilation of a CLAIRE program.

Code generators for C++ and Java both have been implemented in CLAIRE; the three-quarters are shared, known that the API functions of the two kernels have identical signatures. The differences are principally due to the target language, and some expressions are compiled in a different manner (e.g., function definitions, world primitives). For instance, the C++ code generator can insert predefined macros, while the Java code generator has to be more “expressive”.

2.1.3. Deployment Strategy: Search for Flexibility

Sometimes the code generator suffers from a lack of information that could be helpful for yielding more accurate, more efficient or more readable code. To tackle this weakness, we have introduced the possibility to specify a deployment strategy, i.e., a file containing CLAIRE statements, which is loaded just before the code generation, and is used to tell the code generator:

- What is the target language (by default, the target language depends on the CLAIRE version, and more precisely it depends on the kernel version: C++ or Java);
- How to handle the memory management (for C++ only): does the generated code have to be decorated with specific CLAIRE garbage collection macros, or do explicit allocation and deallocation primitives have to be used instead?
- How to dispatch the generated code. By default, in the C++ version a pair of files is created by module (one cpp file and its associated header file), while in the Java version one Java file is created by module and one for each class declared in the code. It is possible to modify this default behavior, in other words we can produce in the C++ version as many C++ files as classes are defined, and we are able to yield a unique Java file which contains all the generated code (classes and the remainder);
- What is the specialization context. If a binding-time analysis has been performed during the optimization phase (to isolate the static pieces of code from the dynamic ones, following the principles of partial evaluation [JGS93]), then the value of the static inputs must be known before starting the code transformation;
- …

This list is not exhaustive, it is also possible to overwrite existing (open-coded) functions or to change the value of existing objects used by the code generator.

2.2 Types in CLAIRE

A rich data type complements the simplicity of the class hierarchy. A type is either a class, a constant set of objects, the union or the intersection of two types, a parameterized class, an integer interval or a typed subset of list, array or set. A parameterized class is the subset of the class such that the parameter belongs to a given type. For instance, Stack[of:{integer,float}] is the set of stacks whose type parameter (of) is either integer or float. For convenience, we write Stack<X> to designate the set of stacks whose type parameter is exactly X.

There are three kinds of primitive collections: arrays (constant length and ordered), lists (dynamic length, ordered) and sets (dynamic length, no order and no duplicates). Both lists and sets come in two flavors: either strongly typed (list<t>, à la C++), or supporting inclusion polymorphism (list[t]), i.e., list[t1] $\subset$ list[t2] $\Leftrightarrow$ t1 $\subset$ t2. Since list<t> $\subset$ list (= list[any]), we can write generic list methods, but list subtypes are monomorphic (list<t1> $\subset$ list<t2> $\Rightarrow$ t1 = t2). Lists of type list<t> are interesting because both read and write operations can be statically type-checked [Cas95]; on the other hand, only read operation on lists of type list[t] can be statically type-checked while write operations must rely on dynamic typing.

Besides, the tuple type is used for constant, heterogeneous, typed “arrays-like” (i.e. indexed) containers; for instance, tuple(integer,integer,float) contains tuples with exactly three elements: two integers and a float. A tuple is not an array; its content cannot be changed. For instance, we use tuples in CLAIRE to return multiple values from a function.

We can summarize the CLAIRE type system as follows (as usual, integer[] is the type for arrays of integers and .. is used to define integer intervals):

```
<type> ::=    <class>                 | <class>[<parameter>:<type> *]    |
              {<object> *}            | (<integer> .. <integer>)         |
              (<type> U <type>)       | (<type> ^ <type>)                |
              set<<type>>             | list<<type>>                     |
              <type>[]                | tuple(<type> *)                  |
              subtype[<type>]
```

Types are conceptually seen as sets of objects (more precisely, as expressions that denote a set of objects) and can be used everywhere in a program, since they are reified. Types as objects are themselves typed with the subtype[...] type; for instance, subtype[integer] contains all types that are included in integer. This set-oriented view is supported by the fact that type subsumption is defined by the inclusion of the set extensions for all possible evolution of the object database. Types form an inclusion lattice with syntactical greater lower bound and subsumption (subtyping). The relationship between the type lattice and the powerset of the object domain is the basis for using an abstract interpretation scheme [CP93].

The second specificity of the CLAIRE type system is the use of second-order types that are attached to methods. A second-order type is a function, defined by a lambda abstraction, which represents the relationship between the type of the input arguments and the result. More precisely it is a function such that if it is applied to any valid type tuple for the input arguments, its result is a valid type for the result of the original function. Let us consider the two following examples:

```
Identity(x:any) : type[x] -> x
top(s:Stack<x>) : type[x] -> last(s.contents)
```

The first classical example states that Identity is its own second-order type. The second one asserts that the top of a Stack always belong to the type of the Stack elements (of parameter). In the expression e that we introduce with the type[e] construct, we can use the types of the input variables directly through the variables themselves, we may also use the extra type variables that were introduced in a parameterized signature (as in the top example) and we use the basic CLAIRE operators on types such as U (type union) or ^ (type intersection).

Last, we take advantage of the reflexivity of CLAIRE to introduce meta-types called patterns which are types that apply to language expressions (actually function calls). A pattern is a set of function calls with a given selector and a list of types of the arguments (that is a list of types to which the results of the expressions that are the arguments to the call must belong). A pattern in CLAIRE is written p[tuple(A,B,...)] and contains calls p(a,b,...) such that a is an expression of type A, and so on. Patterns have two uses: the iteration of sets represented by expressions and the optimization of function composition (including membership on the same expressions). We will see more in depth the use of patterns in the section dedicated to the optimization of iterations.

2.3 Type Inference and Type Checking

The typing philosophy of CLAIRE is different from most mainstream language. First, types are meant as data types, which are expressions that represent sets of objects. This yields a more intuitive semantics and reified types that can be used as programming paradigms. Types are used inside code fragments as set expressions that are evaluated lazily (which has proven quite useful in practice).

Next, CLAIRE supports truly generic methods that make static typing impossible (such as read(p:port) + read(p:port)). Thus CLAIRE relies on dynamic typing, along the tradition of LISP or SMALLTALK. However, the compiler is designed to statically type-check as much code as possible and goes quite far to achieve such a job. One of the obvious tools is a strong type inference component, which goes beyond mainstream compilers and exploits the richness of the CLAIRE type system (e.g., second-order types). The result is that most programs are actually statically type-checked and the resulting code is exactly as efficient as (say) C++ implementation. This feature was instrumented for getting state-of-the-art results for new combinatorial optimization algorithms developed with CLAIRE. It is interesting to notice that we have the same goal of combining parametric and subtyping polymorphism than the Theta language [DGL+95], but we use a combination of dynamic typing and code generation to circumvent the problems raised in their paper.

Type inference is seen as an abstract interpretation of a code fragment. In a previous paper [CP93], we had shown how abstract interpretation is a natural framework for type inference and how it supports the inference of second-order types. Let us recall some simple notations. If e is an expression from the language, [e] is the semantic of the expression, that is a function from some domain (usually <State> <Value> × <State>), which can be defined either with a denotational or operational semantic. The semantic functions can be abstracted into simpler functions from a simpler domain, where objects are abstracted into types, using the usual scheme from [CC77]. We write [e]* this abstraction that is precisely the definition of a valid type inference. We showed earlier that the validity of this abstract interpretation scheme is obtained (following [CC77]) from the fact that the type lattice is a sub-lattice of the powerset lattice of objects. In this paper, we will not exploit the ability to use this approach to derive second order types, so we will use [e]* to represent the type inferred for the expression e with the type context (that associates a type for each free variable in e).

For instance, the use of second-order types (when available) is straightforward. If m is a method, we use (m) to represent the signature (a Cartesian product of n types) and (m) to represent the second-order type (a function from $Type^{n-1} \rightarrow Type$). The type inferred for a call $f(a_1,\ldots,a_n)$ that uses the method m will be $([a_1]^*, \ldots, [a_n]^*)$ (static binding is explained in Section 3.1).

In CLAIRE, we restrict ourselves to the inference of data types (second-order types are indicated by the programmer), but the abstract interpretation scheme has been further refined:

- **The optimization schemes based on rewriting are taken into account: the rewriting is applied before the abstract interpretation is applied (these rewriting schemes will be detailed in Section 3);**
- **CLAIRE "system" functions are all provided with relevant second-order types;**
- **We make a more thorough use of two key features (the existence of explicit union types and singleton types) to produce more precise types (i.e., smaller sets);**
- **We use abstract interpretation to infer many other properties of a method other than its type.**

For instance, we will often refer to the status of a method, which is a set of Boolean properties that can easily be computed through abstract interpretation. The CLAIRE optimizer uses a set that contains the fact that:

- **A new allocation may be performed by the method;**
- **An update (i.e., a side-effect) is performed on an object given as a parameter. This could be either the modification of a slot or a container (list, set, array, etc.);**
- **The result of a method is one of its input argument;**
- **The result will not be garbage collected (cf. Section 3.4).**

Last, the fact that we cannot guarantee static type checking means that we generate dynamic type checks during optimization. The instruction check(x in S) return x if x belongs to S and raises an error otherwise. The number of such generated checks is one of the metrics that we use to evaluate the quality of the optimization.

3 Source-to-Source Optimization

3.1 Method Calls

The first task of the optimizer is to handle calls $f(a_1,...,a_n)$ and generate, when appropriate, method calls $m(a_1,...,a_n)$ using static binding. Notice that we use a functional syntax in both cases, f is the property (also called the selector of the message) and m is the method.

The first step is to find out if the property is closed or open. A closed property is a property for which CLAIRE will forbid any future re-definition, which is the creation of a new restriction which domain intersects an existing domain (restrictions on new classes are still allowed). The optimizer only proceeds with closed selectors, since an open selector indicates the wish to retain dynamic extensibility (the list of open properties is passed as functional parameters to the compiler).

Then the compiler infers the type for the arguments ($t_i = [a_i]^*$) and checks how many restrictions of the property f that:

(a) have a domain that intersects with $t_1 \times ... \times t_n$

(b) have a domain that contains (product order of subtyping) $t_1 \times ... \times t_n$

If the numbers are both 0, an error is raised. If the numbers are both one, static binding occurs and the optimizer proceeds to the next step. Otherwise the call $f(a_1, ..., a_n)$ will rely on dynamic binding. According to the optimizer options, it is possible to substitute an explicit case statement if the number of possible restrictions p is small – a classical trick that has proven to be worthwhile for p = 2 or 3.

The next step is to determine if a specialized optimization rule should be applied. CLAIRE keeps a dictionary of such rules, which may be extended, although a more natural approach is to use inline methods and optimization rules are intended for compiler designers. We review an example with slot updates. Whenever the parser reads an expression x.p := y, a call write(p,x,y) is generated. The optimizer then needs to resolve the following issues:

- **Type checking: if not([y]* <= range(p)), a dynamic type check must be generated.**
- **Inverse: CLAIRE supports the inverse relationship between two slots. If p has an inverse slot r, then x.p := y must trigger add(y.r,x) or y.r := x, depending on whether r contains a value or a collection of values.**
- **Rules: CLAIRE supports "if-write" demons, which are functions attached to slots. These demons are positioned by the compilation of rules [CJL99]. The optimizer will generate an explicit function call to such a demon if it exists.**
- **Worlds: defeasible updates must be stored on a trailing stack so that a backtrack may occur later.**

To achieve all this, the optimizer uses various intermediate methods that perform different levels of checking and added functionalities. The lower level methods are, in turn, recognized by the code generator to produce optimized target language instructions.

The CLAIRE optimizer uses around 20 such method optimizers, which is beyond the scope of this paper. Another interesting example is the optimization of the membership call ($x \in y$), where code substitution is used to avoid useless set or type object allocation (e.g., $x \in (1 .. 12)$ (x >= 1 & x <= 2)).

Inline methods are supported in CLAIRE and need to be dealt with by the optimizer, as opposed to letting the target language recognize their "inline" status, because of the close interaction between the macro-expansion of inline methods and the other source-to-source optimization that can be performed. For instance, the body of an inline method may contain an iteration that will, in turn, be macro-expanded into a specialized instruction that may itself contain call to other, lower level, inline methods. Such examples (iteration of a tree structure) will be shown in the next section. Inlining itself is a classical technique based on tree substitution, the only subtlety being to decide when not to perform it (recursion, side effects in the input parameters, etc.). We now describe source optimization as the fixed-point of a set of rewriting rules.

The use of multiple layers of rewriting rules leads to the need for a new paradigm to implement simplification (or reduction) rules. Consider the two following equations:

```
(A + B)[i,j] = A[i,j] + B[i,j]
x ∈ (s but t)  = (x ∈ s) & (x != t)
```

These equations may be seen as simplification rules since they respectively allow us to extract a cell of a matrix sum without computing this sum explicitly or evaluate membership to a "but" set without actually computing this set. Usually, there is no need to represent these rules because the programmer uses them implicitly. However, this is no longer the case when these sequences of source-to-source rewriting are used. For instance, there may exist the expression M[i,j] in the body of an in-line method that get transformed into (A + B)[i,j] because of the form of the input parameters. This is even more obvious with the second example if we apply a set inline method to the set expression (s but t).

Representing these simplification rules leads to the notion of composition polymorphism. What we need to represent is that the processing of f(...,x,...) may be different when x is obtained as g(....). For instance, f is get[1] and g is + in the first case; f is and g is but in the second case. This type of polymorphism might be captured with a complex abstract data type in a statically typed language using complex pattern matching rules, but is out of reach with our concrete type approach. This is why we have introduced the notion of patterns[2], which are sets of function calls. Patterns support composition polymorphism through the definition of new inline methods whose signature are made either of any (for concrete parameters) or patterns (for expressions). The compiler uses such a restriction when it finds a function call that matches its signature. Here is the application to the "matrix" example:

```
get(x:+[tuple(matrix,matrix)],i:any,j:any)
        => (args(x)[1])[i,j] + eval(args(x)[2])[i,j])
```

Suppose that we now have this situation:

```
inverseTrace(m:matrix) => sum(list{ 1.0 / M[i,i] | i in (1 .. N) })
f(m1:matrix, m2:matrix) -> inverseTrace(m1 + m2) > 0.0
```

The macro-expansion of inverseTrace(m1 + m2) will yield to an expression that involves (m1 + m2)[i,i]. This call matches the signature of the optimizing restriction that we defined previously (with x = m1 + m2 and args(x) = (m1,m2)). Thus a further reduction is performed and the result is:

```
let x := 0, i := 1 in
        (while (i >= N)
                (x :+ 1.0 / (m1[i,i] + m2[i,i]), i :+ 1),
        x > 0.0)
```

3.2 Iteration

Iteration plays an important role in CLAIRE, either in an explicit form (using the for control structure) or in an implicit form (within image/selection set expressions or within the Boolean control structures some/exists), as shown in these examples:

```
for x in person  print(x)
for y in (1 .. n)  f(y)
{x in (1 .. 10) | f(x) > 0}
{size(c) | c in (class but class)}
```

For each iteration, the optimizer uses the following translation rules so as to produce equivalent, but faster expressions:

- **for x in c:class e must generate code that traverses the class hierarchy and iterates over the instances list of each descendent of c.**
- **for x in (a .. b) e must generate a usual incremental loop if a and b are integers or chars.**
- **{y in S | P(y)} must generate a loop that iterate over S and tests for P.**
- **{f(y) | y in S} must generate a loop that gathers the images through f of the elements y of S.**

Thus, the previous four examples are optimized into the following expressions:

---

[1] get is used implicitly in the form ...[...] (e.g. M[i,j] = get(M,i,j))

[2] Here, patterns are not *design patterns*.

```
for c in person.descendents
        for x in c.instances  print(x)

let y := 1 in (while (y <= n) (f(y), y :+ 1))

let s := {}, x := 1 in
        (while (x <= 10)
                (if (f(x) > 0)  s :add x, x :+ 1),
        s)

let s := {} in
        (for c in class.instances
                (if (c != class)  s :add size(c)),
        s)
```

Since new collection classes can be defined, the compiler supports the extension of the iteration mechanism through the definition of the iterate method. For instance, we may inform the optimizer how to iterate over the Hset data structure by adding a new restriction to iterate:

```
iterate(x:Hset, v:Variable, e:any) : any
        => for v in x.content
                (if known?(v)  e)
```

This mechanism has been used to implement the lazy iteration of image or selection sets. For instance, the expression:

```
for y in {c.slots | c in (class \ relation.ancestors)}   print(y)
```

is optimized into the equivalent expression:

```
for c in class.instances
        if not(c ∈ relation.ancestors)   print(c.slots)
```

We present now two examples of the use of sets as a design and programming pattern. The first example is the concept of embedded (doubly) linked lists. Linked lists are very common and useful objects, since they provide insertion and deletion in constant time. The implementation provided by most libraries is based on cells with pointers to next/previous cells in the list. This is useful but has the inconvenience of requiring dynamic memory allocation. When performance is critical, most programmers use embedded linked list, that are implemented by adding the two next/previous slots to the objects that must be chained. This only works if few lists (e.g., one) need to be kept, since we need a pair of slots for each kind of list, but it has the double advantage of fewer memory access (twice fewer) and no dynamic allocation as the list grows. For instance, if we have objects representing tasks and if we need a list of tasks, we may define the class Task as follows:

```
Task <: object(..., next:Task, prev:Task, ...)
```

The obvious drawback is the loss of readability, since the concept of linked list is diluted into pointer chasing. In CLAIRE, we can use a pattern to represent the concept of a linked chain using these two slots as follows. We first define chain(x) which is the list of tasks obtained by following the next pointer from x on:

```
chain(x:Task) : list<Task>
        -> let l := list<Task>(x) in
                (while known?(next, x)   (x := x.next, l :add x), l)
```

We define the iteration of the chain pattern in a very similar way:

```
iterate(x:chain[tuple(Task)], v:Variable, e:any) : any
        => let v := args(x)[1] in
                (while true  (e, if known?(next, v)   v := v.next
                                        else break()))
```

We can now use the "abstract chains" as any other collection in CLAIRE:

```
sum({weight(x) | x in chain(t0)})

count(chain(t1))
```

The interest of this approach is to keep the benefit of abstraction at no cost (we do not see the pointer slots anywhere and we can later replace embedded linked list with another representation). The code produced by the optimizer is precisely the pointer-chasing loop that we would have written in the first place.

The second example shows the power of combining patterns with the concept of iterators. Iterators are a very popular solution for iterating collection in most Java and C++ libraries. However, for most collection types they induce a needless overhead (thus, we do not use them in CLAIRE). However, there are more complex data structures for which iteration can be seen as traversing the structure, and for which the iterator embodies the traversal strategy. An iterator is simply defined by two methods, start and next, which respectively provide the entry point for the iteration and the succession relationship. The following is an example of a tree iterator. We introduce a pattern (T by I) which produces the set of nodes in T using the traversing strategy of the iterator I.

```
Tree <: object(value:any, right:Tree, left:Tree)

TreeIterator <: object(tosee:list, status:boolean)

iterate(x:by[tuple(Tree,TreeIterator)], v:Variable, e:any) : any
        => let v := start(args(x)[2], args(x)[1]) in
                while (v != unknown)
                        (e, v := next(args(x)[2], args(x)[1]))
```

We can define many types of iterators corresponding to the various popular strategies for iterating a tree (Depth-First Search, Breadth-First Search, etc.):

```
TreeIteratorDFS <: TreeIterator()                       // DFS strategy
start(x:TreeIteratorDFS, y:Tree) -> ...
next(x:TreeIteratorDFS, y:Tree) ->
DFS :: TreeIteratorDFS()
TreeIteratorBFS <: TreeIterator() ...                // BFS strategy
```

We may now use the pattern (T by I) as any other collection and simply write:

```
for x in (myTree by DFS)  print(x)
sum({y.weight | y in (myTree by BFS)})
```

The value of our approach is demonstrated by the code produced by the optimizer for this last expression:

```
let d := 0, y := start(BFS,myTree) in
        (while (y != unknown)
                (d :+ y.weight, v := next(BFS, myTree)), d)
```

3.3 Other Control Structures

There are many control structures for which interesting rewriting or type inference rules may be used. For instance, when no type is supplied for a local variable defined in a let, it is inferred from the initial value. Thus the declaration:

```
let x := 12 in (x + 2)
```

is equivalent to:

```
let x:integer := 12 in (x + 2)
```

This is a small but very convenient feature of the language. A more interesting example is the rewriting of conditional statement. There are a few situations where some information may be extracted from the test expression and passed to the optimizer when the main branch (i.e., when the test is positive) is optimized:

- **If the test is an equality test with a constant (e.g., x = 12) or a membership test with a constant set which is itself a type (e.g., x ∈ (1 .. 10)), and if the first argument of the test is a variable (x), a more precise type is inferred for x (through type intersection) and used for the main branch (respectively {12} or (1 .. 10)). In the interesting case where the intersection is empty, the main branch can be skipped and the conditional instruction is rewritten into the "other" branch. This will turn quite useful for the code specialization scenario described in Section 4.**
- **If the test is a known?(x) test which checks if x is not the unknown value, the type of x may be refined if it is of the kind X ∪ {unknown}. This kind of type is used for many methods that may return either an object or the unknown value. The optimizer is able to strip the union type and restrict the type of x to X in the main branch of the conditional. This is instrumental for dealing efficiently with this unknown value, which is itself a useful feature and a clear improvement over the NULL value in more mainstream languages.**

A similar example is provided by the case structure, which is set-based in CLAIRE and looks like:

```
case X ((0 .. 10)    1,
        (20 .. 100)  2,
        integer      3,
        string U char 4,
        any          5)
```

A case is a convenient specialization of the conditional control structure, thus we may also apply the same inference rules. In each branch of the case, the type X of the main variable is restricted to the subtype that labels the branch. In addition, for any input type X of the main variable, we may infer a more precise type for the whole expression by taking the union of all the resulting types of the branches whose label types have a non-empty intersection with X. We notice that the existence of explicit union types is a convenient feature of our type system.

Our last example is the printf control structure, a very powerful tool for I/O, in particular pretty-printing or code generation. The CLAIRE printf looks very much like a C++ printf at first sight, but is actually a generalization of the LISP format structure. The expression printf(s:string,e_1,...,e_n) takes a format string s and inserts the arguments e_1, ..., e_n into the position denoted by the appropriate markers in the string. The first difference with C++ is that there is no need to use a different marker for different data types since polymorphism will play its role. The second difference is that one of the marker allows to say that e_i is the expression that will print the argument as opposed to the argument itself, making this a totally extensible scheme. This is a very convenient feature but would be inefficient if string parsing was necessary at run-time. Thus the printf statement in rewritten by the optimizer into a sequence of simple statements that contain:

- **Printing the pieces of the string that are obtained by slicing along the markers,**
- **Printing the standards argument according to the usual marker using polymorphic methods,**
- **Evaluating the other arguments.**

For instance, one may contrast:

```
printf("The price for ~A items is ~A~I.\n", order.total, order.cost,
                                              printCurrency(order))
```

with its C++ equivalent:

```
cout << "the price for " << order->total << "items is " << order->cost;
order->printCurrency(); cout << ".\n";
```

For any real example of larger size (such as the code generator!) the difference is really striking and CLAIRE printf is a dumb but really useful feature.

3.4 Supporting Target Code Generation

In this sub-section, we show how similar techniques may be used to ease the code generation phase. In both cases, pushing the solution at the optimizer level with the appropriate AI techniques solved hard problems with few lines of code.

3.4.1 GC Support

CLAIRE relies on garbage collection to free the programmer from (some) memory management problems. However, CLAIRE was designed as a "pre-processor" to C++ or Java. The later case is a non-event, but the first case left us with the choice between two approaches. The first one consists in using a public domain C++ garbage collector (such as [Boehm]) which relies on a precise understanding of the compiler strategies for allocating variables. The second approach relies on finding a portable way to implement the mark phase of our mark-and-sweep design. For various reasons that are out-of-scope for this paper (some being non-technical) we decided to stick to our own "portable" mark-and-sweep design.

The CLAIRE garbage collector is by no mean superior to other alternatives (although we benchmarked it at the same level of performance as other public/proprietary implementations), but the implementation strategy is an interesting case of using abstract interpretation to tell the compiler what should be marked.

The only tricky part is how to mark the content of the local variables inside methods once the code is compiled into C++. We use an additional stack to push the values "to-be-protected" of these variables, but this is acceptable if we only push relevant information and not too many values. Therefore, we use abstract interpretation to determine whether:

(a) The value belongs to a type of statically allocated objects (a GC-safe type),

(b) The value can be reached from a GC-safe object, or an object that was previously "protected".

Thus, whenever the optimizer sees an assignment to a local variable, it uses this information to generate a push if the value is not "GC-safe". The evaluation of (a) is straightforward, but the evaluation of (b) is actually tricky and requires the use of information such as the existence of side-effects (updates) in method body (obtained with the status abstract interpretation).

The second way to avoid these useless push operations is to find out if a garbage collection may or may not occur during a method call. Once more, we use the generic abstract interpretation scheme (status) to find out if an allocation may or may not occur (which could trigger a GC). The combination of these two approaches is sufficient to avoid more than 95% of useless value pushing, and reduces the overhead to an acceptable level.

3.4.2 Deployment Support

As we said earlier, CLAIRE tries to conciliate the strengths of dynamically and statically typed languages. Being a dynamically typed language, it needs a uniform memory representation for all objects that is used to write truly generic code. We use a classical "OID" approach (tag + offset) which may designate anything in the system. On the other hand, even statically typed code that uses the same implementation technique will never be as efficient as equivalent C++ code. This is why we also use more efficient and specialized memory representation for the case when the type of the instruction is known precisely. A sub-hierarchy of classes that uses a similar internal representation is called a sort. For instance the sort "object" uses the C++ or Java underlying representation, the sort "string" uses a pointer to an array of char (C++) or a Java string, etc. There are 6 pre-defined sorts in CLAIRE, and a mechanism for importing external data structures efficiently into the language. This allows the creation of a new class with its specialized internal representation. The sort any is added to represent the generic OID implementation strategy.

This dual representation of everything in CLAIRE (generic or specific) has two advantages over more classical strategies for dynamically typed languages (such as LAURE [CP93]):

- **The generated code is much faster;**
- **The generated code is more readable and can be debugged with a regular C++/Java debugger.**

However, it requires more work from the compiler that has to decide which implementation strategy to adopt for each code fragment. Obviously this is based on type inference; thus the optimizer makes this decision. The optimizer looks if the value contained in a variable or a parameter is a type that intersects with only one sort (mono-sort type). If this is the case, a more efficient sort-specific implementation should be used. This decision is passed to the code generator by annotating the optimized code with sort information. By default, the code generator will pick the sort associated with the type inferred for the expression. Thus, there are two types of annotation: a down cast that tells that a more precise sort may be used, or an upcast that tells that the generic sort any should be used.

4 Code Specialization through Abstract Interpretation

4.1 Metrics

One of the key ideas of software engineering is to use metrics [CK91], to monitor and reduce development costs. Because CLAIRE is a new and high-level language, there are no proven formula to relate the function points and the number of lines of code. Thus we decided to perform some measures during the optimization phase. First we measure the function points for each function, class, module, etc. to report the information at the end of the compilation. Second, we compute for each function a measure of its complexity that will be explained in the next section. Although this measure is non-standard, it is quite an accurate indicator of complexity and is used for regression testing and monitoring the software growth.

A second dimension for software metrics is to measure the “quality” of the compilation, which is a key problem for high-level languages. As a matter of fact, it is often possible to write two methods that perform the same task, that look very similar, and that have very different run-time performances. This is usually not a problem for the (mythical) “expert programmer”, but is a strong obstacle on the learning curve. Because of its expressiveness and the richness of its optimization schemes, CLAIRE is not unaffected by this problem.

Thus, the optimizer computes and presents to the programmer a measure of the quality of the optimization, which is roughly the number of occurrences of undesirable situations such as:

- Unsafe dynamic binding. A call is unsafe if the Cartesian product of the input parameter types is not included in the union of the restriction’s domains (cf. Section 3.1). An unsafe dynamic call may produce a run-time error, unless an error handler within the method correctly protects it.
- Safe dynamic binding. A dynamic binding call is also undesirable because of the performance loss at run-time. Because of its richer type system, CLAIRE relies on its own dynamic binding which is less efficient than what modern compilers can propose for C++ or Java. To reach the minimum of performance degradation promised by CLAIRE, one must monitor the use of dynamic dispatch (when performance is an issue).
- Implicit allocation. There are a few instructions that will provoke a memory allocation at run-time, although it may not be evident for the programmer. For instance the iteration of a set image, or the construction of such an image will allocate the necessary container.
- Errors (using the error raising instruction).
- Type checks, which may provoke a run-time error.

Most often, these situations are necessary, they are the consequence of the expressiveness of the language, and are totally acceptable from a performance point of view. Thus this measure is used in two ways:

- Performance tuning. These metrics are a valuable tool to fine tune a CLAIRE program to obtain better run-time performance. As observed earlier, this is much more valuable for a standard user than for an expert one, but is furthermore important.
- Regression testing. It is a very good practice to monitor the quality of the compilation throughout the evolution of different releases. It tracks most of the introductions of dangerous software patches that work but cause problems six months later because they are run with a different calling context.

4.2 Abstract Interpretation of Calls and Specialization

The optimizer also uses its own complexity metric internally, which is a linear combination of all the previous indicators including the size of the conditional tree, the number of dynamic dispatch or the possibility of run-time error. This complexity is obtained in a straightforward manner that can either be seen as an abstract interpretation or the evaluation of a function defined by structural induction. Let us call the weight function that associates to each call c the value of the complexity metric. In the case when we can directly detect an error, we will assume that $(c) =$.

To evaluate the possibility of an error, the optimizer calls the type inference (for type checks or finding whether dynamic binding is needed or it is safe). Thus the result depends on the input types. Moreover, if we apply this evaluation to the body of a function with types that are more precise than its signature, we may detect an error. Let us consider the following example:

```
f(x:any)
        -> case x (integer 1, string 2, any error("can't compute f(~S)",x)
```

The call f(a) where a is of type any is unsafe but legal, whereas the same call with a of type float is clearly an error ((f(a:float)) =). This is actually the classical problem with languages that are not statically typed: there are obviously legal programs that may cause run-time error. An interesting approach to get rid of this drawback is to evaluate each call f(x) in the program for all calling configurations. This is very close, though more expensive, to using second-order types inferred for all methods as we proposed in [CP93]. On the other hand, it is not difficult to speed up the process by restricting ourselves to calls that are potentially dangerous, since the possibility of errors is given by our metric. In CLAIRE, we use it as a separate compiler option that slows the compilation down but tracks many type errors that would occur dynamically. Our experience over 5 years with various decision support applications written in CLAIRE is that 50% of run-time error that occurred "in the field" could have been avoided with this technique that has been included in the recent version of the compiler.

In addition it is possible to extend the definition of the weight function to evaluate the difference in complexity between a call f(x) with a given set of parameter types and the complexity of the "generic" calls where parameters are variables whose types are the method's domain. We need to find the subpart of the expression tree (which defines the method) that is concerned for these input parameters and compute the ratio of the two weights in the two different type contexts (1 and 2). Notice that failing to restrict ourselves to the "covered" part of the expression tree would lead us to believe falsely that a specialized call is always much simpler than the generic call. Whereas our goal is to detect those few cases where the input parameter types makes it really much simpler, either because a dynamic binding is turned to static or because a heavy branch of a conditional may be dropped. In the current version, we apply this specialization step only if three conditions hold:

- **The final weight is small enough (less than a given constant),**
- **The ratio is more than another given constant (e.g., 3) that depends on the coefficients of the weight function,**
- **The property f is declared to be worth optimizing. This is because we do not have enough experience yet with this specialization scheme and are thus experimenting with it.**

Because we use singleton types that we associate to constant, this method of specialization covers most of what can be done by propagating constants within code fragments. It remains to be seen whether this approach will in practice yield more significant improvements than partial evaluation approaches [JGS93].

5 Conclusion

Source-to-source optimization is a challenge that several research projects are tackling. For example, the Cecil/Vortex project [CVP] deals with Vortex [DDG+96], an optimizing compiler framework that transforms Cecil, Smalltalk, C++, Java or Modula-3 programs into efficient C++ or SPARC assembly code. Vortex uses first a language-specific front-end to translate the initial object-oriented program into some Vortex RTL intermediate code. Vortex performs numerous analyses, such as static intra/inter-procedural class analysis, class hierarchy analysis, dynamic profile analysis, and other more usual analyses (e.g., side-effect analysis). With such a rich amount of information deduced from these analyses, Vortex is then able to apply many whole-program optimizations, e.g., intra/inter-procedural optimization, splitting, various inlining declinations, constant propagation, code folding, etc.

The Self project [SeP] works on the Self object-oriented language, and in particular on the dedicated Self compiler [HU94]. This compiler also performs many optimizations so as to yield efficient machine code, as inlining (based on type feedback, i.e., dynamic type analysis), customization (i.e., recompilation and duplication of methods with respect to the receiver type) and splitting. The Self compiler supports dynamic recompilation (for profiting more precise information) and polymorphic inline caches to reduce the overhead of polymorphic message sends.

Finally, the Sather project [SP] has implemented a high-level programming language called Sather [OL91]. Sather is an object-oriented language that is pretty close to CLAIRE: it proposes parameterized classes, object-oriented dispatch, statically checked typing, type inheritance, iteration abstraction, garbage collection, multiple inheritance. And Sather also handles preconditions, postconditions, and class invariants. Sather programs are compiled into efficient C code, and as with CLAIRE, the compiler is implemented in Sather. This compiler performs many optimizations (e.g., aggressive inlining). Several libraries have been implemented above Sather, for various purposes (e.g., grammar manipulation, numerical/geometric algorithms, statistics, graphics, image processing).

However, CLAIRE is a high-level, multi-paradigm programming language for writing combinatorial algorithms that is richer than the input languages of the previously mentioned approaches. The application of CLAIRE to various optimization problems has resulted in programs that are easier to read (and thus to maintain) and to reuse (or to combine). This is mostly due to the conciseness (but not at the expense of readability) of the programs, because of their high level of abstraction and the use of parameterization. We achieve two goals that may seem contradictory: on the one hand, we can write algorithms where data structures as sets are totally abstracted for the sake of simplicity. On the other hand, we can generate efficient code with no penalties compared with more traditional languages. This is made possible by three technical contributions:

- **The deep optimization of set programming, which supports the efficient use of recursively-nested set expressions;**
- **The combination of inline code substitution and features such as set iteration and higher-order functions;**
- **The use of composition polymorphism, which occurs naturally with the parameterization of code.**

CLAIRE was first released 5 years ago and has been used for various deployed industrial applications, in different companies. It was also used as a support tool for teaching "Constraints and Algorithms". CLAIRE is written in CLAIRE (10,000 lines of code), except for a small bootstrapping, run-time kernel, and the optimizer makes heavy use of CLAIRE features (including garbage collection). CLAIRE is a public domain tool, it can be downloaded from http://www.ens.fr/~caseau/claire.html.

References

[AS97] K.R. Apt, A. Schaerf. *Search and Imperative Programming*, Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), ACM Press, 1997.

[Boehm] H. Boehm Garbage Collector. ftp://ftp.parc.xerox.com/pub/gc

[Ca94] Y. Caseau. *Constraint Satisfaction with an Object-Oriented Knowledge Representation Language*. Applied Intelligence, Vol. 4, no. 2, May 1994.

[Cas95] G. Castagna. *Covariance and Contravariance: Conflict without a Cause*. ACM Transactions on Programming Languages and Systems, Vol 17, no. 3, May 1995.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints, Proceedings POPL'97, ACM Press, 1997.

[CK91] S.R. Chidamber, C.F. Kemerer. *Towards a Metrics Suite for Object-Oriented Design*, Proceedings of OOPSLA'91, ACM Sigplan Notices, A. Paepcke ed., p.197-211, 1991.

[CJL99] Y. Caseau, F.-X. Josset, F. Laburthe. *CLAIRE: Combining Sets, Search and Rules to Better Express Algorithms*, Proceedings of ICLP'99, De Schreye ed., MIT Press, Las Cruces, New Mexico, 1999.

[CL94] C. Chambers, G. Leavens. *Typechecking and Modules for Multi-Methods.* Proceedings of OOPSLA'94, ACM Sigplan Notices, Portland, 1994.

[CL96a] Y. Caseau, F. Laburthe. *Introduction to the CLAIRE Programming Language*, LIENS research report 96-16, Ecole Normale Supérieure, 1996.

[CL96b] Y. Caseau, F. Laburthe. *Cumulative Scheduling with Tasks Intervals*. Proceedings of JICSLP'96, M. Maher ed., The MIT Press, 1996.

[CL97] **Y. Caseau, F. Laburthe.** *Solving Various Matching Problems with Constraints*, proceedings of CP'97, G. Smolka eds. LNCS 1330, p. 17-31, Springer, 1997.

[CL99] Y. Caseau, F. Laburthe. *Heuristics for Large Constrained Vehicle Routing Problems*. Journal of Heuristics 5:3, Kluwer, 1999.

[CP93] Y. Caseau, L. Perron. *Attaching Second-Order Types to Methods in an Object-Oriented Language*, Proceedings of ECOOP'93, O.M. Nierstrasz Ed., p.142-160, Kaiserslautern, Germany, 1993.

[CVP] Cecil/Vortex Project. http://www.cs.washington.edu/research/projects/cecil/, Department of Computer Science & Engineering of University of Washington, Washington DC.

[DD99] S. Demayer, S. Ducasse. *Metrics, Do They Really Help?*, Proceedings of LMO'99, p. 69-82, Villefranche-sur-mer, France, 1999.

[DDG+96] J. Dean, G.Defouw, D. Grove, V. Litvinov, C. Chambers. *Vortex: An Optimizing Compiler for Object-Oriented Languages*, Proceedings of OOPSLA'96, ACM Sigplan Notices, p. 83-100, San José CA, 1996.

[DGL+95] M. Day, R. Gruber, B. Liskov, A. Myers. *Subtypes vs. Where Clauses: Constraining Parametric Polymorphism*. Proceedings of OOPSLA'95, ACM Sigplan Notices, Austin, TX, 1995.

[HS93] G.M. Hoydalsvik, G. Sindre. *On the Purpose of Object-Oriented Analysis*, Proceedings of OOPSLA'93, ACM Sigplan Notices, A. Paepcke ed., p. 240-255, Washington DC, 1993.

[HU94] U. Holzle, D. Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance, Proceedings of the ACM OOPSLA'94 Conference, Portland, OR, October 1994.

[JGS93] N.D. Jones, C. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*, International Series in Computer Science, Prentice Hall, June 1993.

[OL91] S. Omohundro and C.-C. Lim, *The Sather Language and Libraries*, Technical report TR-92-017, International Computer Science Institute, Berkeley CA, 1991.

[SDD+86] J. Schwartz, R. Dewar, E. Dubinsky, E. Schonberg. *Programming with Sets: an Introduction to SETL*, Springer, New-York, 1986.

[SeP] Self Project. http://www.cs.ucsb.edu/oocsb/self/index.html, Object-Oriented Compilers labs., Computer Science Department of University of California, Santa Barbara CA.

[SP] Sather Project. http://www.icsi.berkeley.edu/~sather/index.html, International Computer Science Institute, Berkeley CA.