An Optimising Compiler for a Modern Functional Language

ADRIENNE BLOSS*, P. HUDAK AND J. YOUNG†
Yale University, Department of Computer Science, New Haven, CT 06520, USA

One of the factors hindering the use of functional languages has been their relatively poor performance in comparison to more traditional languages such as C and Pascal. During the last decade tremendous progress has been made in building implementations of functional languages but the approaches adopted have employed specialist hardware and/or compiler optimisations that have been developed specifically for functional languages. Building specialist hardware may be the best long-term solution but in the short run it is possible to increase the use and acceptance of functional languages by exploiting the performance of commercially available machines. The goal of the project described in this paper has been to design an optimising compiler that produces fast code for functional languages on conventional sequential and parallel machines.

Received November 1988

1. INTRODUCTION

One of the factors hindering the use of functional languages has been their relatively poor performance in comparison to more traditional languages such as C and Pascal. However, during the last decade tremendous progress has been made in building efficient implementations of functional languages. Much of this progress has come about through the realisation that the performance problem lies less with functional languages themselves than with attempts to implement them using conventional architectures and compilation strategies. Most modern approaches employ specialised hardware and/or compiler optimisations that have been developed specifically for functional languages.

Elsewhere in this issue the reader will find discussions of novel *computation models*, or *abstract machines*, that support the evaluation of functional languages directly. Most of these are variants of *graph reduction*, which mimicks the reduction rules of the lambda calculus. A model such as graph reduction can be implemented either by building specialised hardware that supports the model directly, or by mapping the model onto conventional hardware. Building specialised hardware may be the best long-term solution, but for the short run we would like to increase the use and acceptance of functional languages by exploiting the performance of commercially available machines. Thus our goal has been to design an optimising compiler that produces fast code for functional languages on conventional sequential and parallel machines.

Of course, the design of such a compiler for conventional uniprocessors is non-trival, to say the least. Imperative languages have their roots in the so-called 'von Neumann architecture', or uniprocessor, whose structure is reflected in constructs such as the assignment statement. But functional languages do not have such a machine bias – their roots are more closely tied to abstract models such as the lambda calculus – and their constructs bear little resemblance to traditional hardware. Thus it is not surprising that mapping graph reduction onto conventional hardware is difficult. Furthermore, many of the optimisation techniques that are successful for imperative languages are either ineffective or simply not applicable for functional languages. On the bright side, certain kinds of optimisations that would normally be considered intractable for imperative languages become viable with functional languages because of their simpler and more uniform semantics. For both these reasons, new methods are required to optimise the performance of functional languages.

Yale's Lisp and Functional Programming Research Group has advanced the state of the art of compiler technology for uniprocessors to the point where Lisp and functional programs are rapidly approaching the performance of conventional programs on standard benchmarks. Our technology is best described as a combination of low-level implementation techniques that take advantage of conventional hardware, and a host of high-level optimisations that reflect the unique characteristics of modern functional languages. The work can be summarised in two major areas:

1. An efficient implementation of Lisp. T[24] is a lexically scoped dialect of Scheme that was developed at Yale; Orbit,[17,18] the T compiler, produces very efficient code for conventional uniprocessors through a combination of conventional compilation techniques (such as representation strategies, register allocation, and memory management), and an innovative compilation strategy for higher-order functions (i.e. closures). The code generation strategy for closures views registers, stacks, and heap-allocated cells uniformly as different representations of *environments*. This uniform viewpoint permits the construction of *nested* environments, in contrast to 'flat' strategies for implementing environments that begin with lambda-lifted code or 'super-combinators'.

2. A T-based implementation of a non-strict functional language. ALFL[9] is a non-strict functional language, similar to SASL, that was developed at Yale for both instructional and research purposes. The ALFL compiler translates ALFL programs into T code, thus taking advantage of Orbit's ability to compile higher-order

* Current address: Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA.
† Current address: Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

functions and exploit the low-level features of conventional machines. Of course, ALFL's non-strict, purely functional semantics requires (alternatively, provides opportunities for) additional optimisations, and we have implemented the following:

(*a*) *Strictness analysis and termination analysis*, which allow the compiler to overcome the overhead of lazy evaluation by evaluating arguments before function calls whenever possible.
(*b*) A flexible *call interface strategy* that matches function/argument strictness properties depending on context.
(*c*) *Path analysis*, which permits the detection of 'single-threaded data structures' which may be updated through destructive operations rather than by copying.
(*d*) *Partial evaluation*, which unfolds functions and simplifies expressions.
(*e*) *Uncurrying*, which at the expense of 'fully' lazy evaluation vastly improves the efficiency of most function calls.

The ALFL compiler also has a flexible user interface that allows the user to control the optimisations listed above and that generally aids in program development.

One of the unique aspects of modern functional language implementations is captured in the *methodology* that underlies many of the new compilation strategies. In conventional languages one normally relies on an *operational semantics* to invent and justify particular optimisations or code generation strategies. But with functional languages it is often *denotational semantics* that provides such insights and justifications. In particular, *abstract interpretation* has proven to be a very powerful tool; it forms the theoretical basis for many of the optimisations described in this paper.

Although reasonably robust and currently in use at several institutions, the ALFL compiler will soon be abandoned in favour of a version that will compile HASKELL, the newly proposed functional language standard. The new compiler will use the core of the current ALFL compiler, including all of the optimisations discussed in this paper, but because HASKELL is a strongly typed language we expect the new compiler to perform even better than the current one.

In this paper we will concentrate on the ALFL compiler itself, rather than the Orbit compiler (details of which may be found in refs 17 and 18). In particular, we will concentrate on the optimisation techniques which are crucial to good functional program performance on conventional uniprocessors. Despite this concentration, the paper is only a summary of results – space limitations preclude detailed discussion – but there are ample references for the interested reader. In addition, there are many aspects of the ALFL compiler that we do not discuss at all. Among these, the most important is code generation strategies for 'thunks' (described in detail in ref. 13), but we also omit more standard analyses and transformations such as type inference (which we use to improve our code generation strategies) and translations for pattern-matching and list comprehensions. An excellent summary of standard techniques may be found in ref. 23.

For the sake of consistency and clarity, all of our programming examples will be written in HASKELL instead of ALFL (the primary semantic difference being that HASKELL is statically typed, whereas ALFL is not). In addition, we will use Scheme instead of T in describing the result of our compilation strategies.

2. A SUMMARY OF OPTIMISATIONS

For each of the optimisations described in the introduction, we present in this section a motivating example in HASKELL, an intuitive description of how the optimisation works and why it helps, and a discussion of its implementation status in the ALFL compiler. In the next section we will discuss a unifying framework, namely abstract interpretation, through which the analyses necessary to support each of the optimisations can be expressed and implemented.

2.1 Strictness Analysis

2.1.1 Motivation

The non-strict semantics of ALFL requires *lazy evalution*, in which expressions are not evaluated until (and unless) their values are demanded. Lazy evaluation increases the expressive power of a language in two ways:

(1) *It increases the power of functional abstraction.* Suppose while programming we notice the repeating patterns* 'p x1 ⇒ x2; x3' and 'p y1 ⇒ y2; y3'. Using good abstraction principles we may decide to replace these by 'f x1 x2 x3' and 'f y1 y2 y3', respectively, where:

$$f\,a\,b\,c = p\,a \Rightarrow b;\ c.$$

However, note that with a strict language an expression such as 'f x1 x2 x3' will result in the evaluation of all three arguments, whereas in the original program only one of x2 and x3 would get evaluated. Thus if x is 0 the call 'f (x = = 0) x (1/x)' would cause an error, while the original program would not. The implication is significant: with a strict language, *functional abstraction can change the semantics of a program.* Using lazy evaluation in a non-strict language, however, f's arguments are not evaluated until they are demanded; in the example above the value of 1/x will never be demanded, and so the use of f does not affect the semantics of the program.

(2) *It allows the definition of infinite structures.* Since expressions are not evaluated until their values are required, we can define constructs such as infinite lists. For example, the infinite list of ones is defined below:

$$ones = 1 : ones.$$

In a strict language evaluation of ones would of course not terminate, but with laziness each element of the list is evaluated only as it is required, and the 'infinite' part of the list always remains unevaluated.

The easiest way to implement lazy evaluation is to delay the evaluation of any expression which will not be immediately needed, including all arguments passed to user-defined functions, and insert appropriate code to force the evaluation of the delayed expressions when they are needed. However, this can be extremely inefficient. Consider a compute-intensive program such as tak, a benchmark from the Gabriel suite.[6] This function is

* "p ⇒ c; a" is like "if p then c else a" in a conventional language.

highly recursive, and resembles the inner loops of many functional programs:

```
tak 18 12 6
where tak x y z = ~(y < x) ⇒ z;
                  tak   (tak (x−1) y z)
                        (tak (y−1) z x)
                        (tak (z−1) x y)
```

Note the call-intensive nature of this program – if we were to compile this into Scheme using no optimisations, the program would spend most of its time delaying and forcing values:

```
(LETREC ((tak (LAMBDA (Dx Dy Dz)
                (IF    (NOT (< (FORCE Dy) (FORCE Dx)))
                       (FORCE Dz)
                       (tak (DELAY (tak (DELAY (− (FORCE Dx)) 1)) Dy Dz))
                            (DELAY (tak (DELAY (− FORCE Dy) 1)) Dz Dx))
                            (DELAY (tak (DELAY (− FORCE Dz) 1)) Dx Dy)))
                ))))
(tak (DELAY 18) (DELAY 12) (DELAY 6)))
```

where DELAY is a syntactic form for delaying the evaluation of its argument, and FORCE is the inverse syntactic form which induces the evaluation of its (delayed) argument.*

However, it is possible to determine that tak will always need to evaluate all three of its arguments – it is *strict* in x, y, and z, and so no value need ever be delayed. Thus we can produce the following code:

```
(LECTREC ((tak (LAMBDA (x y z)
                 (IF (NOT (< y x))
                 z
                 (tak (tak (− x 1) y z)
                      (tak (− y 1) z x)
                      (tak (− z 1) x y))))))
(tak 18 12 6))
```

which is essentially what one would write if programming directly in Scheme.

2.1.2 *Analysis and optimisation*

Intuitively, strictness analysis[15,19,20] is an interprocedural analysis that determines when an argument to a function is actually needed to compute the result of a call to that function. If a particular argument will always be needed, then we can be certain that it is safe to evaluate it before the call – thus avoiding the delay.

Formally, we say that f is *strict* in its first argument if $f\perp = \perp$; that is, the application of f to a divergent argument also diverges. This is slightly stronger than the intuitive notion of 'need', but having proven this it is surely safe to pass an argument to f by value, since either the argument converges – and we have done nothing untoward – or the argument diverges, and the function call would have too.

It may come as a surprise to learn that it is often easier to prove that a function is strict rather than that the function will always evaluate its argument. For instance, in the case of tak presented earlier, it is clear that x and y must always be evaluated, but it would be difficult to establish that 'eventually' y < x would be true, and so z must also be evaluated. Assuming z is $\perp$, however, we can prove that the recursive calls '(tak (y−1) z x)' and '(tak (z−1) x y)' will not terminate, so the value of 'tak x y $\perp$' is either $\perp$ or 'tak (tak (x−1) y $\perp$) $\perp$ $\perp$'. From this it follows by induction that for any x and y, 'tak x y $\perp$' diverges. Of course, proving divergence of a program in the general case is undecidable; we discuss our approximation techniques (which lose precision while gaining computability) in sections 3 and 4.

* The obvious way to delay and force values in Scheme is to use 'nullary closures', but there are more sophisticated ways as well, which are beyond the scope of this paper; see ref. 13 for details.

2.1.3 *Implementation status*

Strictness analysis is fully implemented in the ALFL compiler for both first-order and higher-order functions. While the value of higher-order strictness analysis when not combined with a 'collecting interpretation' has been disputed,[5] the increased complexity and analysis time that comes with a full collecting interpretation has so far discouraged its incorporation into the strictness analyser. Collecting interpretations are discussed further in Section 3.3.

2.2 Termination analysis

The previous optimisation depended heavily on the ability to prove that a particular function application diverged. Similarly, if we can prove that an expression is guaranteed to *terminate*, then other optimisations are available to us. In particular, it is safe (although not necessarily more efficient) to evaluate the expression rather than delay it.

2.2.1 *Motivation*

Suppose we are in a context in which we know that x has already been evaluated, but we are unable to prove that f is strict. Now consider the call 'f (x+1)'. Since for integers it is clear that x+1 will terminate, we could compile this call as:

```
(LET ((x1 (+ x 1)))
  (f (TRIVIAL-DELAY x1)))
```

where TRIVIAL-DELAY is like DELAY except that it takes advantage of the fact that its argument is already evaluated and can thus avoid, for example, the overhead of capturing the current environment. In most situations this will be more efficient.

2.2.2 *Analysis and optimisation*

We say that an expression e *terminates* if the expression is guaranteed to converge under any circumstances.

More formally, an expression e *terminates* if there exists no environment in which the value of e is $\perp$.

As the previous example demonstrates, it is also possible that an expression might be particularly well-behaved under other assumptions we have made in compiling the program at hand. In particular, we may have decided through strictness analysis that the variable x is to be passed in by value to a particular function. In this case, it is not fair to say that within the function the expression x+1 does not terminate just because the value of x+1 is $\perp$ in an environment in which x is bound to $\perp$, because we have made the assumption that x will be passed in by value, and thus will never take on the value $\perp$. We call this *conditional termination*, because it depends upon other assumptions being made by the compiler.

2.2.3 Implementation status

Both unconditional and conditional termination analyses have been fully implemented in the ALFL compiler. In addition, although termination analysis could be implemented in a local manner, we in fact do full interprocedural analysis, using the same approximation techniques as we use for strictness analysis

2.3 Collected termination

2.3.1 Motivation

Delayed values are not only expensive *to create*, but are also expensive to *access* in comparison to undelayed values which can be accessed 'for free'. In order to decrease the use of delayed values in ALFL loops, we use a third optimisation called *collected termination*.

For example, in our source language the effect of a Fortran 'do loop' might be obtained as follows:

```
do 10 init_a1 ... init_an
where do i a1 ... an =
        (i = = 0) ⇒ finalize a1 ... an;
                    do (i−1)  (next_a1 a1 ... an i)
                              ...
                              (next_an a1 ... an i)
```

Clearly do is strict in i, but it is only strict in aj if finalise is strict in its jth argument and some next_ai is strict in its jth argument, where do is also strict in ai. While it is not likely that this circular condition will hold, it is possible that some of the next_ai expressions can be easily proven to terminate. If both init_ai and next_ai always terminate, then ai can safely be passed in by value. In general, if all occurrences of a particular argument to a function can be shown to terminate, then it is safe not only to evaluate that argument before entering the function, but also to use an undelayed representation when passing it to the function.

2.3.2 Analysis and optimisation

For 'collected termination' we compute the set of termination properties (e.g. 'terminates' or 'diverges') of all the expressions which might be passed in as an argument to a particular function. This is an example of a general technique known as a *collecting interpretation* as described in Section 3.3.

2.3.3 Implementation status

Collected termination is fully integrated into the ALFL system. We have found that in many instances of inner loops it determines that arguments may be passed by value. We are currently investigating ways of improving its performance as well as the possibility of generating the same optimised code without an analysis by producing different versions of the same function.

2.4 Destructive aggregate updating

2.4.1 Motivation

Arrays in ALFL can be 'incrementally updated'. The primitive operation upd is defined to take an array a, an index i, and a value x, and return a new array a′ which is identical to a except that its ith element is the value x. Thus, conceptually at least, a′ is a *copy* of a except in its ith element.

Now consider a function to initialise an n-element array to some value x:

```
init a i x = (i = = 0) ⇒ a; init (upd a i x) (i−1) x
```

Using a copying update, initialising an n-element array causes it to be copied n times, requiring $O(n^2)$ space and time! The initial copy ensures that the original array argument is not mutated by init, while every other copy preserves a partially initialised version of a. Yet each of these intermediate copies is discarded at the recursive call to init; only the fully initialised array is returned as the value of the function. Thus after the first update, each update could be done *destructively*, actually changing the value of its array argument, without affecting the semantics of the program. Furthermore, if the original array that is passed to init is not used anywhere else, then even the initial update could safely be done in place.

2.4.2 Analysis and optimisation

In general, detecting when it is safe to do an update destructively requires knowing whether or not the array being updated will be accessed again after the update, which in turn requires knowledge of when objects are used and evaluated. This information is not readily available in the presence of lazy evaluation, but it can be inferred using a compile-time technique called *path analysis*.[2] Given this order-of-evaluation and order-of-use information, whether or not an update of an array a in a function f can be done destructively depends on three things:

(1) Does f use a after updating it?
(2) Does f call another function that either updates a before f uses it, or uses a after f updates it?
(3) Is a aliased in such a way that f's use of another object after its update of a is actually a use of a?

The answer to (1) requires a *local analysis*, since it depends only on information found directly in f; (2) requires an *interprocedural analysis*, since it requires knowledge of how other functions behave; and (3) requires a *collecting interpretation*, since it depends on the context in which f is used.

2.4.3 Implementation status

Although the three analyses just described have been implemented, we currently have only a research implementation of path analysis, and so destructive aggregate updating is not yet fully incorporated into the ALFL compiler. Nevertheless, the implementation has been sufficient to produce numerous benchmarks, including those in Section 4. The primary difficulty with the analysis has been its complexity, which grows in the worst case as the factorial of the number of arguments to a function. Although it appears that the average-case complexity is in fact much better, and that a first-order analysis is tractable, the higher-order analysis appears to be intractable for all but very simple programs, and heuristics are currently used to handle higher-order functions

2.5 Uncurrying

2.5.1 Motivation

All functions in ALFL are *curried*; that is, if a function f of n arguments is applied to $k < n$ arguments, $x_1 \ldots x_k$, it will return a new function f′ of n–k arguments such that $f' x_{k+1} \ldots x_n = f x_1 \ldots x_n$. Thus the Scheme code for f looks like this:

```
(lambda (x1)
  (lambda (x2)
    ...
      (lambda (xn) body-of-f)...))
```

However, in the common case where the function is fully applied, this introduces substantial overhead in the creation and application of $n-1$ extra lambda-expressions compared to the more efficient form:

```
(lambda (x1 x2 ... xn) body-of-f)
```

The reason this is more efficient is that the arguments can be considered as a *tuple*, allowing them to be more compactly represented, and there is only *one* closure instead of n. This efficiency is not only gained in conventional compilation schemes, but also in many models of graph reduction.

Although more efficient, this strategy does not provide the flexibility of partial application that the other strategy has. On the other hand, this capability can be recovered easily as follows: When the uncurried version of f is applied to k arguments, $e_1 \ldots e_k$, the following code is generated:

```
(lambda (xk+1)
  (lambda (xk+2)
    ...
      (lambda (xn)
        (f e1 ... ek xk+1 ... xn))...))
```

Thus the expense of currying is incurred only when it is used. This optimisation is called *uncurrying*.

2.5.2 Analysis and optimisation

No analysis is required to do this form of uncurrying, but its price is the loss of *fully lazy evaluation*.[16] Lazy evaluation guarantees that an actual parameter to a function is evaluated at most once; fully lazy evaluation guarantees this even for the case of shared partial applications. For example, consider the following functions:

```
f g x y = g x + g y
h x y = x + y
```

Now consider the evaluation of f (h e1) e2 e3. With fully lazy evaluation e1 is evaluated only once, but with the uncurrying transformation given earlier it is evaluated twice, once for each call to g inside of f. Fully lazy evaluation can be preserved if a *sharing analysis* is first done[7,8] to determine when a partial application may be shared. However, we believe that shared applications occur infrequently in practice, and thus we have implemented uncurrying as described above.

2.5.3 Implementation status

Uncurrying is fully implemented in the ALFL compiler.

2.6 Partial evaluation

2.6.1 Motivation

The simplest form of partial evaluation is what is traditionally called *constant folding*, where expressions like 1+2 and head (x:y) are reduced to 3 and x, respectively, at compile time. But constant folding is really only a rather limited form of partial evaluation – more sophisticated simplifications are possible, including 'unfolding' function calls, as will be explained below. But first, some motivation.

Proponents of functional programming languages generally encourage the use of many sorts of *abstractions* for the sake of improving the clarity of programs. *Data abstraction* is one common technique, indeed not unique to functional languages, and there are very well understood ways for implementing data abstraction mechanisms. In languages such as HASKELL, for example, all data types are known at compile-time, so all run-time checks can be eliminated and the representation problem is fairly trivial.

Other kinds of abstractions, however, do not necessarily lend themselves to efficient code. For example the simple concept of *functional abstraction* (*procedural abstraction* in imperative languages) presents fundamental difficulties in that the cost of the function call appears to be unavoidable. This is unfortunate, since such efficiency concerns may lead one to avoid abstractions that would otherwise improve the clarity and succinctness of one's program.

This problem is perhaps more serious with functional languages in that one would like to use *higher-order* functional abstractions, yet higher-order functions are notoriously difficult to implement efficiently. For example, starting with a definition of the curried function plus:

```
plus x y = x + y
```

we may then define functions such as:

```
add1 = plus 1
sub1 = plus -1
```

When one of these functions is used in an expression, say add1 x, then we must incur the overhead of not only the call to add1, but also the call to plus and the creation of the closure representing plus 1. Although, as mentioned

earlier, we have come a long way in our ability to compile closures, that effort is surely not going to be as effective as reducing the above expression to x+1!

Some imperative languages solve this problem through the use of *macros* (as in Lisp or C), but these are typically rather *ad hoc* and semantically inconsistent with the rest of the language. In particular, because macros amount to *syntactic* abstractions rather than semantic abstractions, they almost always use a certain degree of dynamic binding rather than static binding. (It is also worth noting that one of the most common uses of macros is to avoid the evaluation of arguments, which of course is obtained for free in modern functional languages through lazy evaluation.)

Another technique is the use of *annotations* or *pragmas* that indicate to the compiler which procedures and functions should be 'integrated', 'unfolded', or 'coded in-line' (all different terminologies for the same thing). This has the benefit of being semantically consistent, but the disadvantage of requiring user involvement in the decision-making process.

Perhaps the best technique of all is an *automatic* strategy for unfolding function calls, that is, a general technique for inducing partial evaluation. Some compilers for imperative languages (including the Orbit compiler) attempt to do this, but doing so for such languages is decidedly more difficult than for functional languages. There are two simple reasons for this: conventional imperative languages have *side-effects*, and they are *strict*. A single example suffices to demonstrate both points: Consider the expression head (e1 : e2). In a modern functional language this could be immediately partially evaluated to yield e1. Not so in an imperative language, however, since in the original expression e2 would get evaluated, whereas in the reduced expression it would not. Thus if e2 did not terminate, or if it induced observable side-effects, very different results might appear.

2.6.2 *Analysis and optimisation*

Despite the apparent ease with which partial evaluation can be applied to functional languages, in general it cannot be applied indiscriminately. In particular, in an untyped language such as ALFL, termination of the partial evaluation process is not guaranteed! The most common example of this is the following:

```
f f  where  f x = x x
```

Note first of all that this expression is not recursive; yet one step of partial evaluation yields exactly the same expression, and thus the process does not terminate. In general deciding which terms are safe to reduce is recursively unsolvable, reducing trivially to the halting problem.

Fortunately, in languages such as HASKELL that employ a Hindley–Milner type system, programs such as the above are rejected by the type system. In fact, using such a type system non-recursive programs are *strongly normalisable*, meaning that a normal form exists and can always be found. It is only through explicit recursion that arbitrarily long computations may be invoked. Thus as long as partial evaluation is done *after* type inference, and explicit recursive functions (easily detectable) are not unfolded, then arbitrary degrees of partial evaluation are possible with *no* danger of non-termination. This is what is done in the ALFL compiler (as an option to the programmer, of course).

There is, however, yet another problem. Even though the above partial evaluation strategy is safe, it may induce a certain degree of 'code explosion'. It may be very efficient with respect to execution time to unfold many calls to a particular function, but it may be very *inefficient* with respect to code size. Some degree of restraint is often required, although our experience has been that the degree is not as high as one might think. Further experience is necessary to determine what the 'right' level of partial evaluation should be.

2.6.3 *Implementation status*

Thus we see that partial evaluation of typed functional programs can be used quite extensively, from none at all to performing as much as possible up to recursion. In the ALFL compiler the user can optionally choose between these extremes (even though ALFL is not a typed language). The new HASKELL compiler will allow a third option which does partial evaluation based on heuristics that try to avoid code explosion.

The current partial evaluator in the ALFL compiler works by translating the ALFL program into an equivalent combinator expression, partially evaluating the combinator expression, and then converting back into ALFL while doing common subexpression elimination. The reason combinators were chosen as the medium through which to do partial evaluation is that the combinator reduction rules can be specified simply and uniformly to accomplish all forms of partial evaluation. For example, the rule:

$$+\,1\,2 \Rightarrow 3$$

specifies the constant-folding of integer summation. Similarly, the rule:

$$hd\,(pair\,x\,y) \Rightarrow x$$

specifies the constant-folding of taking the head of a list. In a lambda-calculus based partial evaluator these same kinds of rules would be used, but in addition rules such as β-reduction with its associated notion of substitution need to be specified. With combinators β-reduction is replaced with simple rules such as:

$$S\,f\,g\,x \Rightarrow (f\,x)\,(g\,x)$$

which look no different from the 'constant-folding' rules above, and can be implemented with the same general mechanism.

Despite this uniformity, the translation to and from combinators is time-consuming, and thus a lambda-calculus based partial evaluator is being designed for the new HASKELL compiler. It will also incorporate other useful reduction rules, such as ones having type-specific knowledge, and will employ heuristics to reduce the degree of code-explosion, as discussed earlier.

3. SEMANTIC ANALYSIS

For presentational purposes, the optimisations above were discussed in an intuitive manner. They can also be described formally in a way similar to that of denotational semantics. However, for them to be useful in a real compiler their semantic domains and analyses must be

implemented, and the implementations must be *computable*. In this section we describe the role of denotational semantics in describing interesting properties of programs, and how *abstract interpretation* mathematically approximates the otherwise uncomputable semantic properties needed to perform our optimisations.

3.1 Semantics and abstraction

Denotational semantics is a formal way of describing the meaning of a program in terms of mathematical domains that properly capture our intuition about program behaviours. In functional languages the 'standard' meaning, or *standard interpretation* of an expression is what we intuitively think of as its *value*, whether that be a number, list, function, or whatever. However, in some applications a less precise meaning may be sufficient. For example, suppose we wish to know the sign of the product of two integers; we could perform the multiplication and then extract the sign of the result, or we could deduce its sign directly from the signs of the operands. The latter approach is arguably the easiest path to finding the desired result, in that manipulating the signs directly requires only part of the information required to do the actual multiplication. For example, extracting $(-)$ from the result of $(+7)*(-5)$ takes more work than having a simple rule that says '$(+)*(-) = (-)$'.

An approximation to a value, such as the sign of an integer, is called an *abstraction*, and a computation over such abstract values is called an *abstract interpretation*.

Abstract interpretation has recently become popular as a general and effective compiler optimisation technique, primarily in functional language circles, but also in other areas. The reasons for its popularity include the fact that it is a *formal* methodology that can be related directly back to the denotational semantics of the source language. This allows one to prove the correctness of an optimisation at an abstract level, independently of operational concerns.*

From a practical perspective, abstract interpretation provides a convenient methodology for expressing compile-time analyses in a relatively language-independent manner. An abstraction is completely specified by just three things: the *abstract domain*, the *primitive functions* of the language as implemented in this domain, and an *approximation direction*. In *strictness analysis*, the abstract domain is $\mathbf{2} = \{\bot, \top\}$, where $\bot \sqsubseteq \top$. $\bot$ represents non-termination; $\top$ represents termination. To be safe, a compiler should only optimise a program when a function is *guaranteed* to be strict. Thus in our analysis we should err in the direction of non-strictness, meaning that we want $f\bot$ to return $\top$, and thus the approximation direction for strictness is 'upward', where $\top$ is above $\bot$. On the other hand, termination analysis requires that we approximate towards bottom, or non-termination, for safety.

If we treat $\bot$ and $\top$ as *true* and *false*, respectively, then the primitive functions of the language can be redefined for strictness analysis to compute over this new domain. For example:

$$\hat{+}\, x\, y = x \vee y$$
$$\hat{if}\, p\, c\, a = p \vee (c \wedge a)$$

The first definition reflects the fact that strict operators will diverge if either of their arguments diverges. The second indicates that the conditional will diverge if the predicate diverges or if *both* arms diverge. The conjunction is required because we do not know the exact value of p, so we can only rely on information that is contained in both c and a.

Given these redefined primitives, a recursive program induces an abstract interpretation which properly expresses the strictness properties of interest. Computing these properties requires computing fixpoints of functionals over the abstract domain; methods for doing this are described in Section 3.5.

3.2 Non-standard semantics

Strictness and termination analysis are examples of abstractions of the standard interpretation. However, not all optimisations can be performed using only the information contained in the standard semantics; some optimisations require information about the *operational* aspects of a program's computation. One could at this point abandon the methodology of abstract interpretation, and resort instead to some kind of operational semantics, but we have found it useful to retain the utility of abstract interpretation by starting with a *non-standard* denotational semantics that captures the operational semantics of interest. For example, if the program's behaviour with respect to storage allocation is needed, one could give a store semantics for a functional language in much the same way that one gives a store semantics for an imperative language. Given this non-standard semantics one can then perform abstractions in the same way as for the standard semantics, yielding the information that is desired. This strategy is used in ref. 12 for computing at compile-time the approximate *reference counts* of call-by-reference objects.

The prime example of abstract interpretation of a non-standard semantics in the ALFL compiler is path analysis, which provides the order-of-evaluation information required for destructive aggregate updating (see Section 2.4). The order-of-evaluation properties of a program are first described by an exact non-standard semantics from which an abstract semantics is derived to produce an analysis that is computable at compile-time.

3.3 Collecting interpretations

A *collecting interpretation*†[14] is a semantic analysis which associates with each variable a *set* of possible values which it could take on during program execution. It is similar to data flow analysis in conventional compilers, and has obvious utility.

As an example, consider the typical definition of map such that map f xs builds a new list from xs by applying f to each of xs's elements. Higher-order strictness analysis

* The formal theory of abstract interpretation has itself witnessed remarkable growth. Beginning with the Cousots' seminal work,[3,4] then Mycroft's reformulation in an applicative (i.e. functional) idiom,[20,19] and more recently Nielsons'[21,22] work have laid down a useful framework in which rather general theorems can be re-cast in particular application domains. The recent textbook edited by Abramsky and Hankin[1] is an excellent source of state-of-the-art developments in this area.

† Some researchers refer to this as a 'sticky' interpretation.

tells us strictness properties of map, but *only as a function of* f*'s strictness properties*. Thus despite strictness analysis, the compiler-writer is not free to turn f's application in the body of map into call-by-value, because at compile-time f is unknown. However, a collecting interpretation might be able to help in two different ways:

(1) It could determine that all possible functions bound to f *in the body of* map were strict, thus allowing the optimisation mentioned.
(2) It could determine that all possible functions bound to f *at a particular application of* map were strict, thus allowing an optimised version of map to be used there, and presumably a more conservative map to be used elsewhere.

Despite this advantage, we have not yet implemented 'collected strictness analysis' in the ALFL compiler. However, we have implemented collecting interpretations of termination analysis and path analysis, as described earlier.

3.4 Semantic toolbox

Once the semantic analyses are designed, they must be implemented. Most of the analyses mentioned previously (strictness analysis, termination analysis, and collected termination), as well as several routines within the ALFL compiler proper, are implemented using a layered system of libraries for semantic analysis. These libraries include an embedded language for the specification of abstract syntax trees and functions upon them and a highly parameterised generic semantic evaluator for ALFL abstract syntax trees. The evaluator includes several general methods for calculating and approximating fixpoints of semantic functions (see below). In addition, several utility libraries are a part of the toolbox.

3.5 Finding fixpoints

The ordered nature of the abstract domain combined with the presence of recursion in our language requires the ability to compute or approximate the least fixpoint of a recursive semantic equation on the abstract domain. For example, strictness analysis amounts to computing fixpoints of recursive monotone boolean formula. This is analogous to finding solutions to general dataflow equations such as found in traditional optimising compilers. Our fixpoint-finding toolbox implements several different techniques, including *direct evaluation*, in which recursive equations are expanded until an answer is found; *Kleene chains*, in which iteratively $\perp$ and the subsequent approximations are substituted for the recursively defined value until a fixpoint is found; and *pending analysis*, in which evaluation proceeds until a recursive loop is identified, and then some approximating action is taken. Unfortunately, on an infinite domain, none of these techniques is guaranteed to terminate, and so we also implement *depth bounding*, in which an appropriate value is returned when a finite resource is exhausted.

Most of the semantic analyses in the ALFL compiler use pending analysis and a depth bound of 30. We are actively investigating other fixpoint approximation techniques.

4 PUTTING IT ALL TOGETHER

In the previous section we described the role of semantic analyses in formalising the optimisations detailed in Section 2, and how they are combined and implemented in the ALFL compiler. In addition, the ALFL system has a flexible *user interface* that provides control over the degree of optimisation performed by the compiler, plus an *interpreter* for interactive program development. In this section we will describe this user interface and present benchmarks that show the impact of the various compiler optimisations.

There are essentially three incarnations of the ALFL compiler, which we will refer to as ALFL, XALFL, and NALFL, listed in order of development and increasing sophistication. Most of the distributed ALFL systems are XALFL version 3.0, although our current in-house development is with NALFL.

4.1 User interface

The ALFL system is built upon the T interactive programming environment in which, typical of most Lisp environments, T expressions are read, evaluated, and printed in a loop called the REPL. Major ALFL activities generally fall into two categories: 'programming' – interactively defining functions and evaluating expressions – and 'file manipulation' – compiling and loading files. An interactive ALFL session begins when the user types (NALFL) to the T REPL. ALFL then enters its own REPL. Depending on the first token read, the ALFL system takes one of three actions:

(1) LET *defns*. The keyword LET introduces a set of top-level definitions (functions and constants) which are compiled and installed in the global ALFL environment.
(2) *Keyword*. Certain keywords print or change the state of the NALFL compiler. For example, timeon enables the printing of the execution time of each expression; timeoff disables it. help prints a list of the possible keywords and actions; state prints out a complete list of the state of the compiler, including the status of each optimisation phase (see below).
(3) *Expression*. Any form not beginning with a known keyword is interpreted as an ALFL expression and is immediately evaluated and the result is printed.

All file manipulation commands are entered to the T REPL. (NALFL-COMPILE *filename*) reads the ALFL program in '*filename*.ALF' and writes out the translated T program to '*filename*.T'. If an error is encountered while parsing the program, the ALFL system does its best to locate the error for the user. On an Apollo workstation, a window will appear automatically with the cursor at the place where the error occurred.‡ On other machines, if ALFL is run under Emacs, two keystrokes suffice to tell Emacs to read in a buffer with the offending file and locate the cursor at the problem. (NALFL-LOAD *filename*) loads a compiled ALFL program into the ALFL system, and (NALFL *filename*) both compiles and loads a file.

The NALFL compiler is organised into a series of *stages* which are performed in sequence. After the source code

‡ We are indebted to Phil Wadler for suggesting this feature to us, who first used it in the Orwell interpreter.[25]

is converted into lexemes, a parse tree is built. Several optimisations are then performed upon the parse tree, including *partial evaluation*, *dependency analysis*, *type inference*, *strictness analysis*, *uncurrying*, *termination analysis*, and *collected termination*. Because the ALFL system is first and foremost a research vehicle, each optimisation stage may be individually turned on and off. A stage is turned on using (ENABLE *stage*) and turned off using (DISABLE *stage*).

4.2 Benchmarks

For comparison with other systems we first present the standard benchmarks NFIB and 8QUEENS. These were run on a Sun 350 workstation (MC68020 CPU; 16 MHz clock) with 4 megabytes of main memory.

Program	NALFL+	Pascal	T
QUEENS 8	14.87	2.75	3.00
NFIB 20	0.17	0.067	0.06

Execution time in seconds.

The NALFL+ compiler includes strictness and termination analysis and collected termination. The T column indicates the execution for a 'comparable' T program – one in which all function calls are strict and lists are eager.

We now compare the execution time of several programs under various versions of the compiler. Except as noted, all benchmarks were run in a fully loaded NALFL system on an Apollo DN3000 workstation (MC68020 CPU; 12 MHz clock) with 8 metabytes of main memory and a local disc. Since these were intentionally small benchmarks, paging and garbage collection are not a factor (with exceptions as noted).

Program	XALFL	NALFL	NALFL+	T
TAK	43.262	0.893	0.886	0.748
MM	0.101	0.101	0.083	0.010
DERIV	0.091	0.090	0.085	0.002
TFIB 100 40	3.524	1.203	0.018[a]	0.016[b]

Execution time in seconds.
[a] Estimated, 5000 iterations.
[b] Estimated via (tfib 10000 40).

Program	NALFL	NALFL*	T
QSORT[a]	28.70	0.73	0.73
INIT[b]	1.35	0.04	0.02

Execution time in seconds.
[a] QSORT and INIT were run on an Apollo DN3000 with 4 megabytes of main memory and a non-local disc.

In the first table, XALFL refers to the vanilla compiler with no optimisation of lazy evaluation; the NALFL compiler includes strictness and termination analysis; and the NALFL+ compiler includes collected termination as well.

In the second table, NALFL refers to the NALFL compiler described above with the addition of an array construct implemented with *trailers*, a technique that allows only the element being updated to be copied. The NALFL* compiler includes the destructive aggregate update analysis based on path analysis.

The benchmarks were taken in part from a set of T benchmarks compiled for the Orbit compiler[18] from several sources, including the Gabriel suite of benchmarks for Lisp systems.[6] Where the T benchmarks used fixed precision integers, we allowed the ALFL compiler also to use them (using an unreleased feature of the ALFL system, but which would happen automatically in a Haskell implementation because of the static type system). The particular benchmarks are:

- TAK: The Takeuchi function from the Gabriel suite. A highly recursive function.
- MM: Matrix multiplication using lists and higher-order functions.
- DERIV: Symbolic derivative from the Gabriel suite. List-intensive.
- TFIB i n: Tail-recursive fibonacci of n executed i times. This is essentially a doubly-nested do loop.
- QSORT: Quicksort using arrays.
- INIT: The initialisation function described in Section 2.4. Note that INIT is lazy in its third argument, which accounts for T's better execution time.

5 CONCLUSIONS

We have outlined the design of the major optimisations in the ALFL compiler, discussed the underlying semantic analysis methodology, and provided benchmarks that indicate that functional languages are indeed becoming competitive with conventional languages. As an overview, we have necessarily had to skip many of the finer details; most of these can be found in the references.[2,13,26] In addition, we have said nothing about our efforts in building *parallel* implementations of Lisp and functional languages; suitable references for that work include refs 10 and 11.

Much work remains. Better analyses, new optimisations, improved code generation, and more efficient compilation strategies are all part of on-going efforts to improve the performance of functional languages. The new HASKELL compiler being developed by Yale's Lisp and Functional Programming Research Group is based on the current ALFL compiler and is expected to be a considerable improvement over the current design. Hopefully this will lead to the more widespread acceptance and use of functional programming languages.

Acknowledgements

Support for this project has come from many sources and in many forms over the past six years. Foremost we would like to thank the National Science Foundation (under grants DCR-8403304, DCR-8302018, and DCR-8451415) and the Department of Energy (under grant FG02-86ER25012). In addition IBM, through a Faculty Development Award, and Burroughs/SDC (now Unisys) and MCC, through NSF PYI matching funds, have contributed to the cause.

In addition to these funding sources we wish to

acknowledge the support of the entire T Project, past ALFL hackers (in particular Fred Douglas and Ian Taylor), and current ones (in particular Maria Guzman). And as always, we thank the 'grapplers' at Yale for their never-ending support.

REFERENCES

1. S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*. Ellis Horwood (1987).
2. A. Bloss, Path analysis: using order-of-evaluation information to optimize lazy functional languages. Ph.D. thesis, Yale University, Department of Computer Science (1989).
3. P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pp. 238–252. ACM (1977).
4. P. Cousot and R. Cousot, Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages*, pp. 269–282. ACM (1979).
5. Jon Fairbairn and Stuart C. Wray, Code generation techniques for functional languages. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, pp. 94–104. ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts (August, 1986).
6. R. P. Gabriel, *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, Mass. (1985).
7. B. Goldberg, Detecting sharing of partial applications in functional programs. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference*, pp. 408–425. Springer Verlag, LNCS 274 (September, 1987).
8. B. Goldberg, Multiprocessor execution of functional programs. Ph.D. thesis, Yale University, Department of Computer Science, 1988. Available as technical report. YALEU/DCS/RR-618.
9. P. Hudak, *ALFL Reference Manual and Programmer's Guide*. Research Report YALEU/DCS/RR-322, 2nd edn. Yale University (October, 1984).
10. P. Hudak, Denotational semantics of a para-functional programming language. *International Journal of Parallel Programming*, **15** (2), 103–125 (1986).
11. P. Hudak, Para-functional programming. *Computer* **19** (8), 60–71 (1986).
12. P. Hudak, A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, pp. 351–363. ACM (August, 1986).
13. P. Hudak, A. Bloss and J. Young, Code optimizations for lazy evaluation. *Lisp and Symbolic Computation: An International Journal* **1** (2), 147–164 (1988).
14. P. Hudak and J. Young, A collecting interpretation of expressions (without power domains). In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 107–118 (January, 1988).
15. P. Hudak and J. Young, Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Symposium on Principles of Programming Languages*, pp. 97–109 (January, 1986).
16. R. J. M. Hughes, Super-combinators: a new implementation method for applicative languages. In *Proceedings 1982 ACM Conference on LISP and Functional Programming*, pp. 1–10. ACM (August, 1982).
17. D. Kranz, ORBIT: an optimizing compiler for scheme. Ph.D. thesis, Yale University, Department of Computer Science (1988). Available as technical report YALEU/DCS/RR-632.
18. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, Orbit: an optimizing compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, pp. 219–233. ACM (June, 1986). Published as *SIGPLAN Notices* vol. 21, no. 7 (July, 1986).
19. A. Mycroft, Abstract interpretation and optimizing transformations for applicative programs. Ph.D. thesis, University of Edinburgh (1981).
20. A. Mycroft, The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of International Symposium on Programming*, pp. 269–281. LNCS Springer-Verlag, vol. 83 (1980).
21. F. Nielson, abstract interpretation using domain theory. Ph.D. thesis, University of Edinburgh (October, 1984).
22. F. Nielson, A denotational framework for data flow analysis. *Acta Informatica* **18**, 265–287 (1982).
23. Simon Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, NJ (1987).
24. J. A. Rees and N. I. Adams, T: a dialect of lisp or, lambda: the ultimate software tool. In *Proceedings 1982 ACM Conference on LISP and Functional Programming*, pp. 114–122. ACM (August, 1982).
25. P. Wadler and Q. Miller, *An Introduction to Orwell*. Technical Report, Programming Research Group, Oxford University (1985).
26. J. Young, The semantic analysis of functional programs: theory and practice. Ph.D. thesis, Yale University, Department of Computer Science (1989).