STATIC EVALUATION OF FUNCTIONAL PROGRAMS

Gary Lindstrom
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
U.S.A.

ABSTRACT

Static evaluation underlies essentially all techniques for *a priori* semantic program manipulation, i.e. those that stop short of fully general execution. Included are such activities as type checking, partial evaluation, and, ultimately, optimized compilation.

This paper describes a novel approach to static evaluation of programs in functional languages involving infinite data objects, i.e. those using normal order or "lazy" evaluation. Its principal features are abstract interpretation on a domain of demand patterns, and a notion of function "reversal". The latter associates with each function f a derived function f' mapping demand patterns on f to demand patterns on its formal parameter. This is used for a comprehensive form of strictness analysis, aiding in efficient compilation.

This analysis leads to a revised notion of *basic block*, appropriate as an intermediate representation for a normal order functional language. An implementation of the analysis technique in Prolog is sketched, as well as an effort currently underway to apply the technique to the generation of optimized G-machine code.

This material is based upon work supported by the National Science Foundation under Grant No. DCR-8506000. Initial results were obtained while the author was a visiting scientist at INRIA, Sophia-Antipolis, France.

1. Static Analysis and Functional Programming

Many modern "software engineered" languages (e.g. Ada) reflect a trend toward stronger static program structure in the interest of enhanced comprehensibility, run-time security, and execution efficiency. However, an alternative approach to better engineered languages lies in functional (and logic) programming languages, where such features as infinite data types (e.g. streams) and higher-order functions are central concepts. These features seem to rely in a fundamental sense on more dynamic concepts of program organization.

Advocates of functional programming point out that such advanced features greatly aid program modularity, reusability, conciseness, and suitability for formal reasoning, as well as facilitating general purpose parallel computing. However, these advantages will be unrealized in practice until efficient implementations of such languages become available, first on today's sequential machines, and later on tomorrow's parallel architectures.

Efficient implementation of conventional (i.e. imperative) languages is based largely on extensive static analysis of programs. Until recently, the dynamic nature of modern functional languages has impeded effective static analysis upon them, with a resulting cost in target code quality. For example, a technological gap has existed between optimizing Lisp compilers and the best implementations of normal order functional languages.

Static analysis can be applied to functional programs to realize benefits similar in many respects to those for imperative programs, including:

a. *Code rearrangement:* Predictably constant common subexpressions (CSE's) can be located, and moved forward in evaluation order. The motivation for such "code hoisting" in imperative languages is suppression of repeated evaluation. The absence of side effects (i.e., "referential transparency") makes this point moot in functional languages. However, code rearrangement is important nevertheless in the optimization of evaluation order (see section 2.2).

b. *Pre-evaluation:* The advantages of static data type analysis and partial evaluation (e.g. "constant folding" at compile time) are clearly evident in both language paradigms.

c. *Intermediate representation:* Functional programming languages have greatly liberalized notions of evaluation sequencing. Nevertheless, any technique for their efficient compilation must be based on some notion akin to that of *basic blocks* in imperative languages, i.e. groups of expressions which can unconditionally be evaluated together.

However, from the perspective of conventional static analysis on imperative programs, functional programs present several special characteristics.

a. Control flow through a functional program is very loosely constrained. Hence its prediction, a relatively simple matter for imperative languages, becomes a major goal of static analysis.

b. Given the dominant program structuring role played by functions, static analysis across function calls is a crucial matter, rather than an esoteric specialty, as is generally the case in imperative languages.

c. While referential transparency makes detection of constant CSE's easy, thorough analysis of their usage patterns under various evaluation orders becomes a central issue, because:

    i. almost all formal parameters are CSE's, so inter-functional analysis depends on CSE analysis, and

    ii. the dominant program structuring role played by conditional expressions means CSE's shared across sibling conditional arms must be carefully considered.

d. Finally, although we are doing *static* analysis, functional programming leads us to take a *dynamic* or "liberal" attitude toward potential run time errors. That is, if the value (e.g. data type) of a CSE appears to be erroneous in one usage, that should not compromise the legitimacy of other, possibly valid usages of the same CSE. This is at variance to more static treatments of errors in imperative languages (e.g. data type "balancing" across conditional arms).

2. Strictness Analysis

The primary technique for predicting control flow in a functional language is *strictness analysis*. That notion is now receiving considerable attention in the functional programming community.

2.1. What Is Strictness Analysis?

Strictness analysis involves determining at compile time sets of expressions which can always evaluated together. In Mycroft's seminal paper [Mycroft 80], attention was focused on detecting "safe" situations in which call-by-need parameters could be compiled under call-by-value. Thus a function $f(x_1, \ldots, x_k) = \ldots$ is judged to be *strict* on formal parameter x_i if whenever x_i is undefined (i.e. equals $\perp$), then the value of f is undefined.

This definition involves some subtlety. Notice that we are not saying that f "always evaluates x_i", simply that "pre-evaluating x_i is *safe* in the sense that at worst it can cause an application of f to diverge because x_i diverged, rather than for some other, inescapable reason."

2.2. Benefits

Strictness analysis can facilitate generation of functional language object code that is optimized in several respects.

* On *parallel architectures*, strictness information can be used for more aggressive compile-time operator scheduling resulting in accelerated development of concurrency at run time [Clark, Peyton-Jones 85, Tanaka 84].

* On *sequential machines* such as the G-machine [Johnsson 84], strictness analysis can lead to larger basic blocks, thereby permitting better optimized linear code, e.g. with respect to stack management and register allocation.

* On *any variety of machine*, actual parameter pre-evaluation, whenever semantically sound, can attenuate the need for run-time context switching and reduce environment retention requirements. This can lead to important effects such as *in situ* activation record recycling for tail recursion.

2.3. Non-Flat Domains

Experience with modern functional languages has demonstrated that the notion of infinite data objects is vital to effective general purpose programming. For example:

* a functional treatment of I/O can thereby be achieved, through the representation of channels as infinite streams;

* parallelism can be enhanced, through the overlapped production and consumption of data, and

* mutual recurrences, represented as cyclic networks of functions (e.g. feedback systems [Keller, Lindstrom 81]) can be directly executed.

A number of labels have been applied to implementation techniques dealing with infinite data objects, including "lazy", "normal order", and "demand-driven" evaluation, and "suspension", "early completion", "lenient", and "future" data structures. From a denotational semantics standpoint, the crucial notion is that the domain of computation is *non-flat*, i.e. involves values that are not produced full-formed by discrete computational events.

Strictness analysis becomes both *more complex* and *more valuable* when applied to languages computing over non-flat domains. It is:

* *more complex* because:

    - The analysis must tell us not only "when do we evaluate?", but also "how far and in what directions do we evaluate?".

- Moreover, non-flat domains make functions *polystrict* in the sense that their degree of parameter strictness depends in part how their results are to be used, i.e. the "pattern of demand" imposed on each application.

* *more valuable* because:

- Accelerating the evaluation of data structure components can dramatically lessen storage requirements, since environment retention effects are decreased.

- Access of components can be simplified by avoiding tests of their evaluation status (at least), and avoiding a costly context switch to evaluate the components (at best).

3. Outline of the Approach

We present here an approach to strictness analysis based on a *demand sensitive* notion of abstract interpretation [Cousot, Cousot 77, Mishra, Keller 84].

a. The starting point lies in [Lindstrom 85], which presents a demand sensitive denotational semantics for a graph reduction language. The semantic domain D employed here includes *two* demand indicators (simple and exhaustive), as well as generic atom and error indicators, and recursively formed pairs of values from D.

b. Using this model, we define a form of static evaluation, in the following sense. Associated with each source program function f we define a "reversed" function f' computing on D. The collection of such reversed functions derived from a program can be statically analyzed to determine lower bounds on demand propagation effects throughout the program at run time.

c. Special techniques to strengthen the analysis of these reversed functions include:

   i. demand pattern pooling using:

      1. *least upper bounds* in D for unconditionally shared CSE's, and

      2. the *greatest lower bounds* in D at CSE's shared across sibling arms of conditionals;

   ii. determination of appropriate fixpoint solutions of recursive equations arising from function reversal (the *least* solution is not always the preferred one!).

d. We show how to exploit *coherence* within conditional expressions (i.e. observing that in cond(p, t, e), we know p holds within t, and ~p within e) to produce stronger strictness results.

e. An implementation in Prolog is outlined for our strictness analysis method. Logical variables play a central role in this method. That is, the accumulation of demand propagation constraints on a value in D is modeled by successive unifications on a Prolog term representing approximations to that value.

f. We demonstrate how this analysis leads to a new intermediate representation for functional programs, derived from an applicative version of *basic blocks*. Finally, we estimate how this representation can be used to produce optimized code for the G-machine implementation of Lazy ML.

Our technique is applied to the class of function graphs defined in [Keller 80], except that explicit **fork** operators are used to indicate CSE's. Hence each node has one outgoing (result) arc, except for **fork**, which has two. Functions take single arguments, and use tuples to "bundle" multiple parameters. The details of our sample language will become clear through examples.

4. The Semantic Domain

We define our domain D in fig. 4-1. A pictorial rendering of the ordering relationships in a subset of D is given in fig. 4-2.

The scalar elements of D may be interpreted as follows.

* $\perp$ denotes a total lack compile-time information about the value denoted by an expression.

* d represents a compile-time hypothesis or inference that an expression will be subjected to *at least* one level of evaluation. In other words, an attempt will be made at run-time to determine if the expression denotes an atom or a (possibly suspended) tuple. No expectation is indicated as to which of those two results will arise (if, indeed, either does -- divergence may occur). Thus d denotes *simple, indiscriminate demand.*

* d^e conveys the information that an expression will be subjected to an *exhaustive* evaluation attempt, i.e. to an atom, or a finite or infinite composition of tuples of atoms or errors (but with no *a priori* expectation of which case, if any, will result). Thus "call-by-value", "applicative order", or "fully eager" evaluation is indicated; this arises from fully eager pseudo-operators such as print, and call-by-value pragmas.

Domain:

$D = \{\perp, d, d^e, a, T\} \cup D \times D$

Auxiliary domain:

$D^e = \{d^e, a, T\} \cup D^e \times D^e$

Ordering:

$\perp < d$

$d < d^e$

$d < [\alpha,\beta], \forall \alpha, \beta \in D$

$d^e \leq \alpha, \forall \alpha \in D^e$

$[\alpha,\beta] \leq [\alpha',\beta']$ iff $\alpha \leq \alpha' \wedge \beta \leq \beta'$

$\alpha \leq T, \forall \alpha \in D$

Figure 4-1: The domain D.

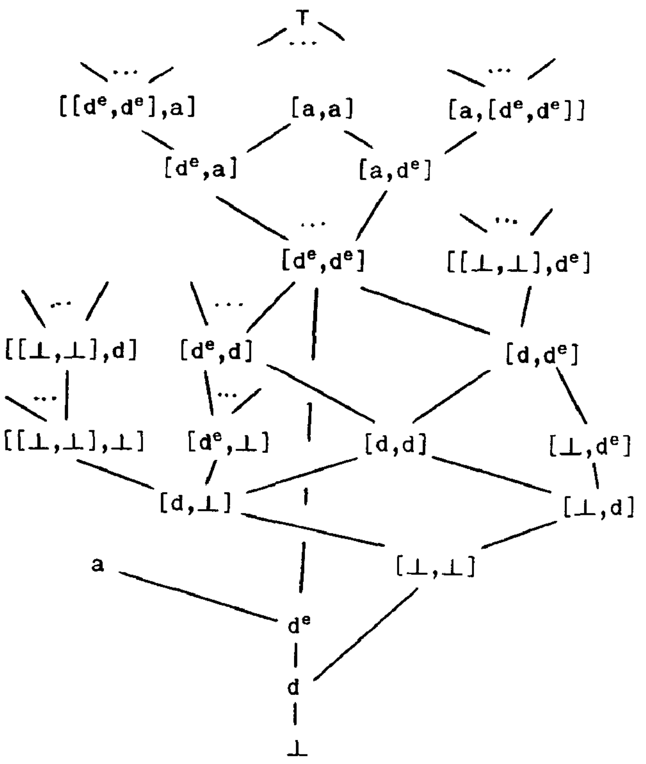

Figure 4-2: Pictorial rendering of D.

* a generically designates all atomic values, including functions. However, since d < a, it can also be interpreted as "demand with atomic result anticipated".

* T indicates conflicting information on the value of an expression, i.e. values which are constrained simultaneously to be atomic and a tuple. This indicates a rudimentary type error.

D is reflexively closed with pairs, i.e. if $\alpha \in$ D and $\beta \in$ D, then $[\alpha, \beta] \in$ D. Such pairs model tuple values in D, which for simplicity we limit here to length two. If $\delta = [\alpha, \beta]$, then we refer to α as the car of δ and β as the cdr of δ.

The interpretation of pairs in D is as follows. [⊥, ⊥] can be construed as "demand with pair result anticipated", [d, ⊥] as "(same) plus simple demand on the car of the pair", etc. In short, values in D may be paraphrased as being "undefined, demanded (simply or exhaustively), demanded and anticipated to be atomic, demanded and determined to be erroneous, and demanded and anticipated to be recursively defined pairs thereof".

Note that if $\alpha \in D^e$, then α cannot include any occurrences of ⊥ or d. This reflects our desired interpretation of d^e as exhaustive demand, since the presence of ⊥ and d would indicate a weakening of that commitment to relentless evaluation.

D is similar to the domain used in [Lindstrom 85], but:

a. the two "strengths" of demand included are d and d^e, rather than the "assertive" and "non-assertive" demand required to represent a "narrowing" style of evaluation, and

b. T can occur within pairs. This is the customary treatment of T in non-flat domains; its previous limitation to being the top element of the domain was a special requirement for properly handling unification failures in [Lindstrom 85].

5. Abstract Interpretation on D

We now explain how D can be used for the static evaluation of functional programs.

5.1. Function Reversal

Since we are primarily concerned with strictness predictions, our technique will focus on demand propagation effects, i.e. control flow from function *outputs* to *inputs*. These demand propagation effects will be modeled by constraints expressing the abstract behavior of individually "reversed" operations.

5.2. Simple Function Graphs

Initially, we consider programs using only *unconditional, acyclic,* and *non-recursive* functions. Let F be such a program, with ρ(F) its result arc. If ν is an arc in the graph of F, denote by $\chi(\nu)$ the value associated with ν by our method. Given a hypothesized demand constraint $\xi \leq \chi(\rho(F))$, we can directly compute F'(ξ) by the rules given in figs. 5-1 and 5-2 for propagating constraints through the graph of F in the anti-dataflow (demand) direction.

The rule for fork reflects the observation in [Lindstrom 85] that the annotation associated with a CSE generally must observe the *intersected constraints* of both usages, i.e. their least upper bound, or sup in D. Henceforth, we will denote sup(x,y) as ⊔{x,y}. A simple algorithm exists for computing ⊔{x,y} in all cases; see fig. 5-3. Examples of the effect of using ⊔ on CSE's are given in fig. 5-4, and the overall analysis result on the simple example in fig. 5-5 is given in fig. 5-6.

6. Conditional CSE's

When a CSE serves as an operand to a pair of paths emanating from alternative arms of a given conditional, the statically determined constraint on the shared operand should be the *maximum common constraint* the two usages present. Moreover, the maximum common constraint can be inferred to be *independent* of that conditional. To capture this, we must generalize our analysis to account for alternative demand propagation paths through program subgraphs, splitting at cond nodes and rejoining at fork nodes.

6.1. Path Contexts

Toward this end, we introduce the notion of a *path context*. An arc annotation is now a pair $<\xi, \pi>$, where ξ is a value in domain D, as before, and π is a path context equal to a list of *conditional branch records* $\beta_k \hat{} \beta_{k-1} \hat{} \ldots \hat{} \beta_1$. A conditional branch record is a pair $\langle\alpha, b\rangle$, where α is a reference to a conditional node in the graph, and *b* is

$\nu = \mathtt{c}$: *(atomic constant)*

Precondition:
$\mathrm{d} \le \chi(\nu)$
Postcondition:
$\mathrm{a} \le \chi(\nu)$

$\nu_0 = \mathtt{plus}(\nu_1, \nu_2)$: *(representative strict operator)*

Precondition:
$\mathrm{d} \le \chi(\nu_0)$
Postcondition:
$\mathrm{a} \le \chi(\nu_0) \wedge \mathrm{a} \le \chi(\nu_1) \wedge \mathrm{a} \le \chi(\nu_2)$

$\nu = \mathtt{f(p)}$: ($\mathtt{f}$ *invoked by name*)

Precondition:
$\mathrm{d} \le \chi(\nu)$
Postcondition:
$\mathtt{f}'(\chi(\nu)) \le \chi(\mathtt{p})$

$\nu = \mathtt{apply(f,\ p)}$: ($\mathtt{f}$ *computed*)

Precondition:
$\mathrm{d} \le \chi(\nu)$
Postcondition:
$\mathrm{a} \le \chi(\mathtt{f})$ (*no strictness on* $\mathtt{p}$)

$(\nu_0, \nu_1) = \mathtt{fork}(\nu_2)$:

Precondition:
(none)
Postcondition:
$\sqcup\{\chi(\nu_0), \chi(\nu_1)\} \le \chi(\nu_2)$

Figure 5-1: Strictness propagation: non-pair operators.

$\nu_0 = \mathtt{cons}(\nu_1, \nu_2)$:

Precondition 1:
$\mathrm{d} \le \chi(\nu_0)$
Postcondition 1:
$[\perp, \perp] \le \chi(\nu_0)$ *(laziness)*

Precondition 2:
$\mathrm{d}^e \le \chi(\nu_0)$
Postcondition 2:
$\mathrm{d}^e \le \chi(\nu_1) \wedge \mathrm{d}^e \le \chi(\nu_2)$ *(eagerness)*

Precondition 3:
$[\mathtt{x}, \mathtt{y}] \le \chi(\nu_0)$
Postcondition 3:
$\mathtt{x} \le \chi(\nu_1) \wedge \mathtt{y} \le \chi(\nu_2)$

$\nu_0 = \mathtt{car}(\nu_1)$:

Precondition:
$\mathrm{d} \le \chi(\nu_0)$
Postcondition:
$[\chi(\nu_0), \perp] \le \chi(\nu_1)$

$\nu_0 = \mathtt{cdr}(\nu_1)$:

Precondition:
$\mathrm{d} \le \chi(\nu_0)$
Postcondition:
$[\perp, \chi(\nu_0)] \le \chi(\nu_1)$

Figure 5-2: Strictness propagation: pair operators.

$\sqcup$	$\perp$	d	d^e	a	T	$[\delta,\gamma]$
$\perp$	$\perp$	d	d^e	a	T	$[\delta,\gamma]$
d		d	d^e	a	T	$[\delta,\gamma]$
d^e			d^e	a	T	$[\sqcup\{\mathrm{d}^e,\delta\},\sqcup\{\mathrm{d}^e,\gamma\}]$
a				a	T	T
T					T	T
$[\alpha,\beta]$						$[\sqcup\{\alpha,\delta\},\sqcup\{\beta,\gamma\}]$

Figure 5-3: sup algorithm.

α	β	$\sqcup\{\alpha,\beta\}$
d	$[\perp,\perp]$	$[\perp,\perp]$
$[\mathrm{d},\perp]$	$[\perp,\mathrm{a}]$	$[\mathrm{d},\mathrm{a}]$
a	$[\perp,\perp]$	T
$[[\mathrm{d},\mathrm{a}],\perp]$	$[[\mathrm{a},[\perp,\perp]],\mathrm{d}]$	$[[\mathrm{a},\mathrm{T}],\mathrm{d}]$
d^e	$[\perp,\mathrm{d}]$	$[\mathrm{d}^e,\mathrm{d}^e]$
$[\mathrm{a},\mathrm{d}^e]$	$[[\mathrm{a},\perp],[\mathrm{d},\mathrm{d}]]$	$[\mathrm{T},[\mathrm{d}^e,\mathrm{d}^e]]$

Figure 5-4: Examples of $\sqcup$ usage.

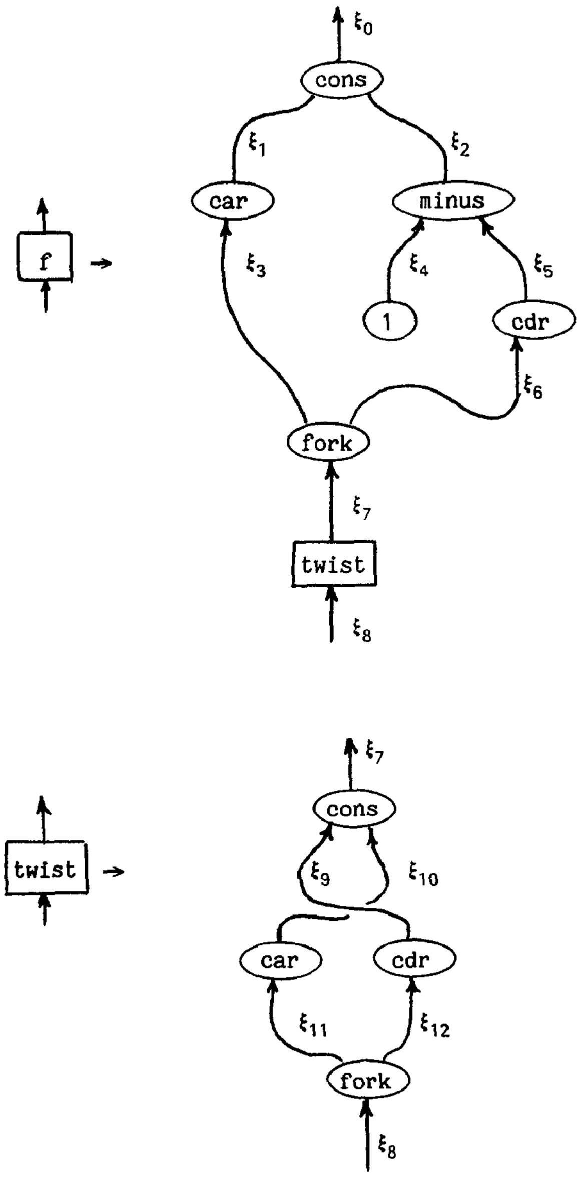

Figure 5-5: Simple program.

either t or e, denoting *then arm* or *else arm*. A conditional branch list can of course be empty (denoted ϕ), as will always be the case for hypothesized annotations.

D extends in a natural way to deal with these pairs. The new ordering is $<\xi_0, \pi_0> \leq <\xi_1, \pi_1>$ iff $\xi_0 \leq \xi_1$ and $\pi_0 = \pi_1$. Most of the operators defined in the previous section retain their definitions, modified to preserve path contexts in pairs. However, fork, cond and function invocation have revised definitions, as shown in fig. 6-1.

The net effect is that demand patterns are propagated through both the then and the else arms of cond nodes. If they rejoin, they combine to produce their greatest lower bound or inf in D, i.e. the most general constraint dominated by both. This reflects the "liberal" attitude

Under ξ_0 = [[d,⊥],d]:	Under ξ_0 = d^e:
ξ_1 = [d,⊥]	ξ_1 = d^e
ξ_2 = a	ξ_2 = a
ξ_3 = [[d,⊥],⊥]	ξ_3 = [d^e,⊥]
ξ_4 = a	ξ_4 = a
ξ_5 = a	ξ_5 = a
ξ_6 = [⊥,a]	ξ_6 = [⊥,a]
ξ_7 = [[d,⊥],a]	ξ_7 = [d^e,a]
ξ_8 = [a,[d,⊥]]	ξ_8 = [a,d^e]
ξ_9 = [d,⊥]	ξ_9 = d^e
ξ_{10} = a	ξ_{10} = a
ξ_{11} = [a,⊥]	ξ_{11} = [a,⊥]
ξ_{12} = [⊥,[d,⊥]]	ξ_{12} = [⊥,d^e]

Figure 5-6: Simple example of analysis.

ν_0 = cond(ν_1, ν_2, ν_3), where
    $x(\nu_0) = <\xi_0,\pi_0>$:

Precondition:
    $d \leq \xi_0$
Postcondition:
    $<a, \pi_0> \leq x(\nu_1)$
    $<\xi_0, (\alpha,t)\hat{}\pi_0> \leq x(\nu_2)$
    $<\xi_0, (\alpha,e)\hat{}\pi_0> \leq x(\nu_3)$
    where α refers to this conditional node.

(ν_0, ν_1) = fork(ν_2), where
    $x(\nu_0) = <\xi_0, \pi_0> \wedge x(\nu_1) = <\xi_1, \pi_1>$:

Precondition 1:
    $\pi_0 = \pi_1$
Postcondition 1:
    $<\sqcup\{\xi_0, \xi_1\}, \pi_0> \leq x(\nu_2)$

Precondition 2:
    $\pi_0 = (\alpha,t)\hat{}\pi \wedge \pi_1 = (\alpha,e)\hat{}\pi$
    (*or symmetrically*)
Postcondition 2:
    $<\sqcap\{\xi_0, \xi_1\}, \pi> \leq x(\nu_2)$

ν_0 = f(ν_1), where
    $\nu_0 = <\xi_0,\pi_0>$:

Precondition:
    $d \leq \xi_0 \wedge f'(<\xi_0,\phi>) = <\xi_1,\phi>$
Postcondition:
    $<\xi_1,\pi_0> \leq x(\nu_1)$

Figure 6-1: Conditional constraint propagation.

toward data type errors, mentioned in section 1. We henceforth denote inf(x,y) as $\sqcap\{x,y\}$. Examples of the effect of using $\sqcap$ on values in D are given in fig. 6-3.

$\sqcap$	$\bot$	d	d^e	a	T	$[\delta,\gamma]$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
d		d	d	d	d	d
d^e			d^e	d^e	d^e	*note 1*
a				a	a	*note 2*
T					T	$[\delta,\gamma]$
$[\alpha,\beta]$						$[\sqcap\{\alpha,\delta\},\sqcap\{\beta,\gamma\}]$

note 1: $\sqcap\{d^e,[\delta,\gamma]\} =$ if $[\delta,\gamma] \in D^e$ then d^e else d.

note 2: $\sqcap\{a,[\delta,\gamma]\} =$ if $[\delta,\gamma] \in D^e$ then d^e else a.

Figure 6-2: inf algorithm.

α	β	$\sqcap\{\alpha,\beta\}$
$[\bot,d]$	$[a,\bot]$	$[\bot,\bot]$
$[\bot,\bot]$	a	d
a	$[a,a]$	d^e
$[a,d^e]$	$[[d^e,d^e],a]$	$[d^e,d^e]$
$[T,T]$	$[[d,\bot],T]$	$[[d,\bot],T]$
$[d,[d^e,d]]$	$[[\bot,T],[d,d^e]]$	$[d,[d,d]]$

Figure 6-3: Examples of $\sqcap$ usage.

6.2. Arc Value Sets

There is, however, a slight oversimplification in this approach, in that it requires cond and fork nodes to "bracket" each other syntactically. This is an unreasonable requirement in a functional language where CSE recognition is aggressively performed, as is almost always the case. To eliminate this difficulty, we once again elevate our value denotations, this time to a set level. Static values associated with arcs are now sets of designations of the previous form $<\xi,\pi>$. The ordering induced is:

$$S_1 \leq S_2 \text{ iff } x_1 \in S_1 \rightarrow \exists x_2 \in S_2 : x_1 \leq x_2.$$

Our operators (except for fork) are defined as before, but now apply element-wise. Thus, for example, if ν_0 = car(ν_1), and $\chi(\nu_0)$ = S, then $\chi(\nu_1) = \{<[\xi_0,\bot], \pi> \mid <\xi_0,\pi> \in S\}$. fork is simplified to *set union*, but with a closure operator applied to its result set S, as shown in fig. 6-4.

$$<\xi_0, \pi> \in S \wedge <\xi_1, \pi> \in S \;\; -> \;\; <\sqcup\{\xi_0, \xi_1\}, \pi> \in S$$

$$<\xi_0, (\alpha,t)\hat{}\pi> \in S \wedge <\xi_1, (\alpha,e)\hat{}\pi> \in S \;\; -> \;\; <\sqcap\{\xi_0, \xi_1\}, \pi> \in S$$

Figure 6-4: Closure condition on static value sets.

7. Coherence

It is relatively simple to extend our method to exploit the coherence information arising from predicate tests in conditionals, as long as that test has an abstract interpretation over D. For example, the familiar pattern cond(atom(ν_1),ν_2,ν_3), where ν_2 and ν_3 depend on ν_1 can readily exploit coherence, since we can add to the denotation set for ν_1 the constraints $<a,(\alpha,t)\hat{}\ldots>$, and $<[\bot,\bot],(\alpha,e)\hat{}\ldots>$. The static analysis on ν_2 and ν_3 is thereby strengthened through the added strictness information obtained by their CSE sharing of ν_1. Fig. 7-1 states the rule in general terms.

ν_0 = cond(atom(ν_1),ν_2,ν_3)>, where
$<\xi_0,\pi_0> \leq \chi(\nu_0)$:

Precondition:
$d \leq \xi_0$
Postconditions:
$\{<d,\pi_0>, <a,(\alpha,t)\hat{}\pi_0>, <[\bot,\bot],(\alpha,e)\hat{}\pi_0>\} \leq \chi(\nu_1)$
$\{<\xi_0,(\alpha,t)\hat{}\pi_0>\} \leq \chi(\nu_2)$
$\{<\xi_0,(\alpha,e)\hat{}\pi_0>\} \leq \chi(\nu_3)$

Figure 7-1: Coherence in conditionals.

8. Direct Execution on D

For many recursive functional programs, the direct execution method described above terminates without further ado because of the demand dissipation effect of propagation through the non-strictness of cons. This can be illustrated by the direct static evaluation of the standard append function (fig. 8-1), under the hypothesized demand [d,d] (fig. 8-2).

9. Fixpoint Solutions

However, in other cases there is no escape from doing a fixpoint solution of the reversed function equations.

9.1. Least Fixpoints

Since all our operators are monotonic and continuous over D, the least fixpoints of these equations are unique and algorithmically determinable to any desired degree of accuracy. An example of a least fixpoint solution to the append example under hypothesized demand d^e is given in fig. 9-1.

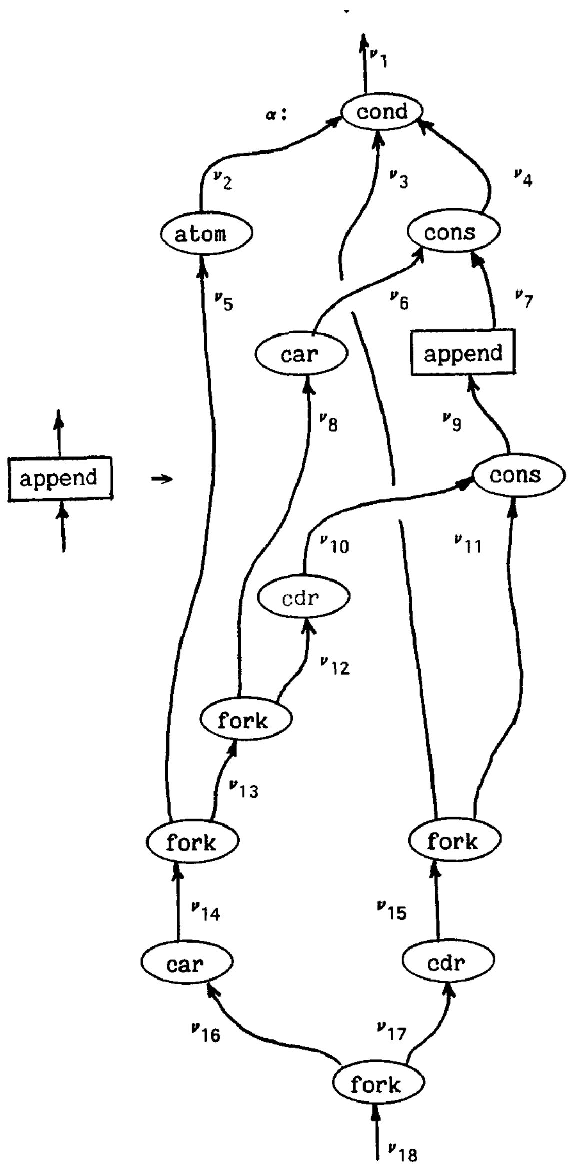

Figure 8-1: append example.

Given $x(\nu_1) = \{\langle[d,d],\phi\rangle\}$:

$x(\nu_2) = \{\langle a,\phi\rangle\}$
$x(\nu_3) = \{\langle[d,d],(\alpha,t)\rangle\}$
$x(\nu_4) = \{\langle[d,d],(\alpha,e)\rangle\}$
$x(\nu_5) = \{\langle d,\phi\rangle, \langle a,(\alpha,t)\rangle, \langle[\perp,\perp],(\alpha,e)\rangle\}$
$x(\nu_6) = \{\langle d,(\alpha,e)\rangle\}$
$x(\nu_7) = \{\langle d,(\alpha,e)\rangle\}$
$x(\nu_8) = \{\langle[d,\perp],(\alpha,e)\rangle\}$
$x(\nu_9) = \{\langle[d,\perp],(\alpha,e)\rangle\}$
$x(\nu_{10}) = \{\langle d,(\alpha,e)\rangle\}$
$x(\nu_{11}) = \{\}$
$x(\nu_{12}) = \{\langle[\perp,d],(\alpha,e)\rangle\}$
$x(\nu_{13}) = \{\langle[d,d],(\alpha,e)\rangle\}$
$x(\nu_{14}) = \{\langle d,\phi\rangle, \langle a,(\alpha,t)\rangle, \langle[d,d],(\alpha,e)\rangle\}$
$x(\nu_{15}) = \{\langle[d,d],(\alpha,t)\rangle\}$
$x(\nu_{16}) = \{\langle[d,\perp],\phi\rangle, \langle[a,\perp],(\alpha,t)\rangle,$
$\langle[[d,d],\perp],(\alpha,e)\rangle\}$
$x(\nu_{17}) = \{\langle[\perp,[d,d]],(\alpha,t)\rangle\}$
$x(\nu_{18}) = \{\langle[d,\perp],\phi\rangle, \langle[a,[d,d]],(\alpha,t)\rangle,$
$\langle[[d,d],\perp],(\alpha,e)\rangle\}$

Figure 8-2: Direct static evaluation of append.

Given $x(\nu_1) = \{\langle d^e,\phi\rangle\}$:

$x(\nu_2) = \{\langle a,\phi\rangle\}$
$x(\nu_3) = \{\langle d^e,(\alpha,t)\rangle\}$
$x(\nu_4) = \{\langle d^e,(\alpha,e)\rangle\}$
$x(\nu_5) = \{\langle d,\phi\rangle, \langle a,(\alpha,t)\rangle, \langle[\perp,\perp],(\alpha,e)\rangle\}$
$x(\nu_6) = \{\langle d^e,(\alpha,e)\rangle\}$
$x(\nu_7) = \{\langle d^e,(\alpha,e)\rangle\}$
$x(\nu_8) = \{\langle[d^e,\perp],(\alpha,e)\rangle\}$
$x(\nu_9) = \{\langle[d,\perp],(\alpha,e)\rangle\}$
$x(\nu_{10}) = \{\langle d,(\alpha,e)\rangle\}$
$x(\nu_{11}) = \{\}$
$x(\nu_{12}) = \{\langle[\perp,d],(\alpha,e)\rangle\}$
$x(\nu_{13}) = \{\langle[d^e,d],(\alpha,e)\rangle\}$
$x(\nu_{14}) = \{\langle d,\phi\rangle, \langle a,(\alpha,t)\rangle, \langle[d^e,d],(\alpha,e)\rangle\}$
$x(\nu_{15}) = \{\langle d^e,(\alpha,t)\rangle\}$
$x(\nu_{16}) = \{\langle[d,\perp],\phi\rangle, \langle[a,\perp],(\alpha,t)\rangle,$
$\langle[[d^e,d],\perp],(\alpha,e)\rangle\}$
$x(\nu_{17}) = \{\langle[\perp,d^e],(\alpha,t)\rangle\}$
$x(\nu_{18}) = \{\langle[d,\perp],\phi\rangle, \langle[a,d^e],(\alpha,t)\rangle,$
$\langle[[d^e,d],\perp],(\alpha,e)\rangle\}$

Figure 9-1: Least fixpoint static evaluation of append.

9.2. Greater Fixpoints

Taking least fixpoints during static evaluation has the advantage that all approximations are *conservative*, in the sense that all predicted evaluations are sure to transpire at run time. Thus it is safe use any information gathered even if the static evaluation must be aborted due to compile time overruns (which can occur, since D is infinite).

There are situations, however, in which the least fixpoint solution is *not* the optimal solution. An alternative solution strategy, more "optimistic" in viewpoint, lies in assuming all functions are *fully strict*, and collecting evidence to the contrary. Mechanically, this is done by assuming d^e rather than $\perp$ as the base of our solution chains.

The advantage of this approach is evident when append under eager demand is reconsidered. As fig. 9-2 shows, this method concludes that append under eager evaluation demands a pair of fully eagerly evaluated arguments. In other words, it can be compiled in this case using standard Lisp call-by-value methods.

Given $x(\nu_1) = \{<d^e,\phi>\}$:

$x(\nu_2) = \{<a,\phi>\}$

$x(\nu_3) = \{<d^e,(\alpha,t)>\}$

$x(\nu_4) = \{<d^e,(\alpha,e)>\}$

$x(\nu_5) = \{<d,\phi>, <a,(\alpha,t)>, <[\perp,\perp],(\alpha,e)>\}$

$x(\nu_6) = \{<d^e,(\alpha,e)>\}$

$x(\nu_7) = \{<d^e,(\alpha,e)>\}$

$x(\nu_8) = \{<[d^e,\perp],(\alpha,e)>\}$

$x(\nu_9) = \{<[d^e,d^e],(\alpha,e)>\}$

$x(\nu_{10}) = \{<d^e,(\alpha,e)>\}$

$x(\nu_{11}) = \{<d^e,(\alpha,e)>\}$

$x(\nu_{12}) = \{<[\perp,d^e],(\alpha,e)>\}$

$x(\nu_{13}) = \{<[d^e,d^e],(\alpha,e)>\}$

$x(\nu_{14}) = \{<d^e,\phi>, <a,(\alpha,t)>, <[d^e,d^e],(\alpha,e)>\}$

$x(\nu_{15}) = \{<d^e,\phi>\}$

$x(\nu_{16}) = \{<[d^e,\perp],\phi>, <[a,\perp],(\alpha,t)>, <[[d^e,d^e],\perp],(\alpha,e)>\}$

$x(\nu_{17}) = \{<[\perp,d^e],\phi>\}$

$x(\nu_{18}) = \{<[d^e,d^e],\phi>, <[a,\perp],(\alpha,t)>, <[[d^e,d^e],\perp],(\alpha,e)>\}$

Figure 9-2: Greater fixpoint static evaluation of append.

9.3. Termination

Unfortunately, using this stronger form of fixpoint analysis does not guarantee termination, either. Moreover, taking d^e as the first approximation to the strictness result of all functions means that the method is no longer safe, either. Given the recent result in [Kieburtz 86], it appears this dilemma is theoretically inescapable. From a practical standpoint, it appears that least fixpoints must be used, along with programmer pragmas to strengthen the analysis where appropriate.

10. Implementation in Prolog

Static analysis is very often represented via attribute grammars, by which compile time "semantic" aspects of program expressions are associated with their syntactic subtrees. Evaluation is then accomplished by traversal of the "decorated" parse tree in a manner guaranteeing complete execution of all attribute functions.

While attribute grammars could be used to express and implement the method reported here, we have chosen to use Prolog instead. Our technique involves associating a *logical variable* with each syntactic subtree, and constraining that variable to assume monotonically increasing values in D, largely by simple unification. As each value becomes stronger, we propagate its effect on neighboring nodes through their shared constraints, in a relaxation manner.

Underlying this method is a representation for values in D using Prolog terms. That is, values in D can be represented in schematic form, with unbound logical variables representing value attributes as yet unconstrained. Of particular interest is the effect whereby unifying two terms transforms them both to equal their sup. If the unification fails, they have T as their sup. Fig. 10-1 summarizes the representation.

constraint on x	*rep(*x*)*
$\perp \le x$	X *(logical variable)*
$d \le x$	`d(X,Y)`
$d^e \le x$	`d(X,de(Y))`
$a \le x$	`d(X,de(a))`
$[y,z] \le x$	`d(X,de(pair(`*rep(y)*`,`*rep(z)*`)))`
$T \le x$	`d(top,_)`

Figure 10-1: Encoding of D as Prolog terms.

11. Applications to Optimized Compilation

The utility of our method in optimized compilation of functional languages is now considered.

11.1. Group Compilation of Functions

The method as described is based on consideration of individual functions. More realistically, groups of functions will be compiled together, presenting greater opportunities for optimization. The following steps could be observed:

a. All functions are individually compiled under the hypothesis of simple demand.

b. All possible invocations of each function f are then collected in order to determine the inf of all possible demands that have been predicted to be applied to f at run time. Thus, for example, if every invocation of f is surrounded by a car, then we can compile f under the assumption that its minimal demand is $[d, \perp]$, and perhaps obtain stronger strictness results on actual parameters at *all* places of call on f.

c. After f is recompiled under this more aggressive strictness assumption, the stronger strictness results emerging on *its* actual parameters are then incorporated into its places of call. This process is iterated until convergence occurs or diminishing returns ensue.

11.2. Basic Blocks in Functional Languages

The focus of this paper on strictness of user-defined functions is important, but is not sufficient for thorough exploitation in a real compiler. Put more concretely, what are we to do during compilation with operators that have a static denotation of $\perp$? Two options appear:

a. Compile these operators under a simple demand assumption, reasoning that if they ever get executed at all, they will do so under at least simple demand. (This, after all, is what current functional language compilers do!)

b. Reassert demand whenever it dissipates in the program graph. This establishes new strict operator groupings, logically separated from those relating to function entry demand, but equally important from a code generation standpoint.

The latter approach is clearly more sensible, and we are currently developing it formally. Its essential characteristics are as follows:

* Instead of initializing the static denotation of each program arc to $\perp$, we initialize it to d_i, where i is a distinct integer tag appended to the simple demand indicator d.

* The strictness constraints expressed previously are adapted to construct equivalence classes among the integer tags appended to levels of evaluation in $\mathbb{D}$. For example, given ν_0 = cons(ν_1, ν_2), initially all three of ν_0, ν_1, and ν_2 would be assigned distinct demand "origins" d_0, d_1, and d_2. However, if ν_0 were to be constrained by a demand pattern $[d_3,d_4]_3$, then the strictness equivalence classes {0, 1, 3} and {2, 4} would result.

* After the overall analysis is complete, operators tagged with evaluation levels in the same equivalence class would be combined into a basic block, and compiled together unconditionally.

* Obvious dominance relationships would be exploited to generate code reliably assuming pre-evaluation of operands shared with other basic blocks. Such relationships include:

  a. the basic block in which a tuple is constructed dominates any basic block in which a component of that tuple is evaluated;

  b. the basic block of a conditional (and its predicate) dominates the basic blocks of its *then* and *else* subexpressions;

  c. the basic block of a function call dominates the basic block of its an actual parameter, and so on.

11.3. A Case Study: the G-machine

These ideas are being applied to the construction of and optimizing compiler for the G-machine. We anticipate the principal benefit will lie in the suppression of eval opcodes on expressions determined to be previously evaluated, by the basic block technique just described. Also, the prospect of fully evaluating tuple components whenever possible should greatly economize on stack manipulations and storage consumption (e.g. saving and restoring evaluation contexts on the "dump").

12. Extensions and Future Work

Clearly, much further work remains to be done before the practicality of this method is established. Specific areas needing attention include the following:

a. Cyclic function graphs can be both sensible and highly useful [Keller, Lindstrom 81]. We thus desire to accommodate them in our static analysis method. The method described *does* handle them correctly, but their presence complicates (further) termination in some cases.

b. More realistic data structures must be considered, including tuples of length greater than two, and vectors with run-time indexing.

c. Other sustained patterns of demand should be considered, including perhaps *stream* demand, which is lazy on cdr's, but eagerly evaluates car's whenever a pair is constructed.

d. Application of strictness to logic programming is very appealing, particularly given the closeness of this abstract model to that in [Lindstrom 85]. One hoped-for result would be a scientific basis for deciding how much open coding of unification should be done in the compilation of, say, each clause in a Prolog program.

e. Extension of the domain to represent more detailed data type information is certainly possible and desirable. The constraint basis employed here seems quite compatible with that used in polymorphic type checking [Milner 78]. Indeed, recent work [Mishra, Stark 85] indicates that the integration of strong typing and strictness analysis may be the best way to statically analyze programs with functional data objects and higher order functions.

13. Related Work

Strictness analysis of functional programs over non-flat domains such as this is a very active area. Recent work has included [Wadler 85], in which a very skillfully designed finite domain is used; [Hughes 85], which is most similar to the work reported here, and [Keiburtz, Napierala 85], a formulation in the untyped lambda calculus.

Companion efforts to bring strictness analysis methods to bear on higher order functions include [Burn, Hankin, Abramsky 85, Hudak, Young 85, Wray 85].

Acknowledgements

The author is indebted to Gilles Kahn for providing a productive environment in which to do this work. The critical comments of John Hughes, Richard Kieburtz, and Robin Milner are also greatly appreciated. Lal George and Dowming Yeh have provided valuable commentary and implementation support.

References

[Burn, Hankin, Abramsky 85]
Burn, G. L., C. L. Hankin, and S. Abramsky.
Theory and practice of strictness analysis for higher order functions.
April 1985.
Dept. of Computing, Imperial College of Science and Technology.

[Clark, Peyton-Jones 85]
Clark, C., and S. L. Peyton-Jones.
Generating parallelism from strictness analysis.
In *Prof. Conf. on Func. Prog. Lang. and Comp. Arch.*, IFIP, Nancy, France, September, 1985.

[Cousot, Cousot 77] Cousot, P., and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*. ACM, Los Angeles, 1977.

[Hudak, Young 85] Hudak, P., and J. Young. A set-theoretic characterization of function strictness in the lambda calculus. In *Proc. Workshop on Implementations of Functional Languages*. Chalmers Univ., Aspenas, Sweden, February, 1985.

[Hughes 85] Hughes, J. Strictness detection in non-flat domains. Programming Research Group, Oxford.

[Johnsson 84] Johnsson, T. Efficient compilation of lazy evaluation. In *Proc. Symp. on Compiler Const.* ACM SIGPLAN, Montreal, 1984.

[Keiburtz, Napierala 85] Kieburtz, R. B., and M. Napierala. A studied laziness -- strictness analysis with structured data types. 1985. Extended abstract, Oregon Graduate Center.

[Keller 80] Keller, R.M. *Semantics and applications of function graphs.* Technical Report UUCS-80-112, Univ. of Utah, 1980.

[Keller, Lindstrom 81] Keller, R.M., and G. Lindstrom. Applications of feedback in functional programming. In *Conf. on Func. Lang. and Comp. Arch.*, pages 123-130. ACM, Portsmouth, NH, October, 1981.

[Kieburtz 86] Kieburtz, R. B. Abstract interpretations over infinite domains cannot terminate uniformly. February 17, 1986. Unpublished note, Dept. of Computer Science, Oregon Graduate Center.

[Lindstrom 85] Lindstrom, G. Functional programming and the logical variable. In *Proc. Symp. on Princ. of Pgmming. Lang.*, pages 266-280. ACM, New Orleans, January, 1985.

[Milner 78] Milner, R. A theory of type polymorphism. *J. of Comp. and Sys. Sci.* 17(3):348-375, 1978.

[Mishra, Keller 84] Mishra, P., and R.M. Keller. Static inference of properties of functional programs. In *Proc. Symp. on Princ. of Pgmming. Lang.*, pages 7-21. ACM, January, 1984.

[Mishra, Stark 85] Mishra, Prateek, and Eugene W. Stark. Strictness and polymorphic type inference. September 18, 1985. Unpublished note, Dept. of Computer Science, SUNY Stony Brook.

[Mycroft 80] Mycroft, A. The theory and practice of transforming call-by-need into call-by-value. In *Int. Symp. on Prgmming.* Springer, April, 1980. Lecture Notes in C. S., vol. 83.

[Tanaka 84] Tanaka, J. *Optimized execution of an applicative language.* PhD thesis, Univ. of Utah, 1984.

[Wadler 85] Wadler, Phil. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). November 10, 1985. Unpublished note, Programming Research Group, Oxford Univ.

[Wray 85] Wray, S. C. A new strictness detection algorithm. In *Proc. Workshop on Implementations of Functional Languages.* Chalmers Univ., Aspenas, Sweden, February, 1985.