Generating Efficient Code from Strictness Annotations

by

Gary Lindstrom
Lal George
Dowming Yeh

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

ABSTRACT

Normal order functional languages (NOFLs) offer conceptual simplicity, expressive power, and attractiveness for parallel execution. However, current implementations of NOFLs on conventional von Neumann machines are not competitive with those of imperative languages. The central reasons for this poor performance include the high control overhead (e.g. demand evaluation) and fine object code granularity (e.g. SKI combinators) used in most NOFL implementations. Strictness analysis gathers information that helps to overcome these inefficiencies through optimized compilation. We propose here a rule-based strategy for such compilation, working from a new textual representation for strictness analyzed source programs. This representation offers readability and ease of manipulation, while expressing all essential strictness information, including basic block structure and block dominance and disjunction relationships. The rules presented here show how to compile this intermediate form into optimized single processor G-machine code. In addition, this representation appears to be useful for a number of other execution methods, including interpretation, compilation into conventional Lisp with "promises", and mapping into "supercombinators" for parallel architectures.[1]

1. NOFLs and Strictness Analysis

1.1. Basic Concepts

Most modern functional languages are based on *normal order* semantics, where divergence of a program occurs only if the program's overall result directly depends on a divergent subexpression. Customary implementations of NOFLs ensure this property by individually operationalizing the normal order characteristics of *each operator*, e.g. through demand evaluation or combinator reduction.

NOFLs offer many advantages in programming practice, including clean treatment of I/O streams, overlapped production and consumption of large data objects, and the facile representation of feedback systems. For example, functional modeling of non-trivial hardware systems seems to require normal order evaluation in a fundamental sense [15].

Another appeal of NOFLs is their suitability for distributed evaluation on innovative architectures, e.g. via graph reduction. However, a more pressing need exists for efficient NOFL implementations on *today's* machines, to gain (i) experience with large-scale software engineering in NOFLs, and (ii) a better understanding of what is *familiar* about their implementation, as well as what is *exotic*. In comparison, one must be impressed with the rapid acceptance that has greeted Prolog, and acknowledge that this has been greatly aided by the early availability of efficient implementations on conventional computers [17].

One of the most promising avenues currently under investigation for improving the efficiency of NOFLs is *strictness analysis* [14]. Under this technique *strict subsets* of operators, i.e. groups unconditionally executable together, are determined at compile time. The major strictness analysis research areas at the moment include:

- "non-flat" domains [6, 8, 16], and
- higher-order functions [1, 4, 11].

While strictness analysis *theory* appears to be developing apace, its *application* to actual code

[1] This material is based upon work supported by the National Science Foundation under Grant No. DCR-8506000.

generation seems to be lagging. An exception is the recent paper [3], which uses a finite domain. We attempt here to fill this gap, by showing a method which relates well to existing compiler technology, and sheds light on the similarities and differences in compiling NOFLs for single- and multi-processor machines.

1.2. Overview of Method

Our method involves four processing steps, from source program to optimized object code:

1. The source program is converted to function graph form, after exhaustive common sub-expression (CSE) detection and elimination. CSEs are represented via binary output, single input **fork** operators, e.g. $(\mu, \nu) = \mathbf{fork}(\upsilon)$.
2. Strictness analysis is performed on the function graph representation, using abstract interpretation on a "non-flat" infinite domain of predicted patterns of evaluation.
3. The resulting annotated graph is converted to textual form, which may be viewed as a semantically attributed abstract syntax tree.
4. Finally, the textual form is translated to object code, optimized by observance of the block structure, predicted prior evaluation, and type information conveyed by the semantic attributes.

1.3. Analysis Method

The first two steps of our method are reported in [12] and [13]. We briefly summarize their essentials here, by way of background for our new results involving steps 3 and 4 above.

1.3.1. Simplified Domain

It is useful to describe our method first via a *simplified* abstract interpretation domain, and then an *augmented* domain. The former represents the effects of a *single source* of hypothesized demand, while the latter analyzes the effect of *all sources* of demand throughout the program, on a "wholesale" basis.

The primitive elements of our simplified domain are as follows:

- $\perp$ expresses a total lack of compile time information as to whether an expression will be evaluated, and if it is, what datatype wil result.
- **d** represents a compile-time hypothesis or inference that an expression will be subjected to *at least* one level of evaluation, ie. to an atom or tuple (possibly with suspended components).
- $\mathbf{d}^{\mathbf{e}}$ 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 error indicators, (but with no *a priori* expectation of which case, if any, will result).
- **a** generically designates all atomic values, including functions. However, **a** can also be interpreted as "demand with atomic result required.
- T indicates conflicting information on the value of an expression, i.e. values which are constrained simultaneously to be atomic and nonatomic. This indicates a rudimentary type error.

This primitive element set is closed with binary Cartesian cross products, representing *pairs* produced by the **cons** operator. The resulting set, constitutes the domain D used in [12]. The operators receiving non-$\perp$ annotations as a result of a non_$\perp$ hypothesis on a particular arc is termed a *strictness subset* of the graph.

1.3.2. Augmented Domain

Reference [13] presents an augmentation of this simplified domain, supporting "wholesale" strictness analysis on an entire function graph. In comparison to the former approach, the analysis method is extended in two respects:

1. $\perp$ has been removed from the domain. This reflects the fact that, although prediction of evaluation causality at compile time is imperfect,
    a. code must be compiled for *every* operator in a function, and
    b. when that code is executed, it will *certainly* be executed under at least simple demand.

Hence the role of $\perp$ in the domain is played by d.

2. However, the fact that a d is placed where a $\perp$ would previously have appeared on an arc must not be construed as necessarily *extending* a strictness subset. Rather, it must indicate the introduction of a *new* strictness subset, which might have arisen from an *independent* application of the previous method with d asserted instead of $\perp$ on the arc in question.

   To establish such a subset boundary, we augment D to include natural number *subscripts* on each denoted level of evaluation. In anticipation of their ultimate use for code generation, we term such subscripts *block numbers*.

   - Initially, all block numbers are considered to be distinct. As the analysis proceeds, some block numbers become *equivalenced*, and their associated strictness subsets are thereby merged.
   - There are also the important notions of block *dominance* and *disjunction*, discussed in section 1.3.3.

In referring to an element of D, it is important at certain times to refer to its outermost block number; other times, that number is irrelevant. To help make this distinction clear, we adopt the following notation:

- When the outermost block number of an element of D must be mentioned, that element will be denoted by a subscripted early alphabet Greek letter, e.g. α_i, β_j, γ_k.
- When the outermost block number of an element of D is irrelevant, that element will be denoted by a non-subscripted late alphabet roman letter, e.g. x, y, z.

Our domain D is as follows:

Domain:

$D = \{d_i,\ d^e_i,\ a_i,\ T_i\} \cup [u,\ v]_i \qquad i = 0,\ 1,\ \ldots,\ \text{and } u,\ v \in D$

Equality:

$\alpha_i = \alpha_j \quad (\alpha \text{ not a pair}) \qquad \Leftrightarrow i \equiv j$

$[u,\ v]_i = [x,\ y]_j \qquad \Leftrightarrow i \equiv j \wedge u = x \wedge v = y$

Partial Ordering:

$d_i < d^e_j \qquad \Leftrightarrow i \equiv j$

$d^e_i < a_j \qquad \Leftrightarrow i \equiv j$

$d_i < [x,\ y]_j \qquad \Leftrightarrow i \equiv j$

$d^e_i < [x,\ y]_j \qquad \Leftrightarrow i \equiv j \wedge d^e_i \leq x \wedge d^e_i \leq y$

$[u,\ v]_i \leq [x,\ y]_j \qquad \Leftrightarrow i \equiv j \wedge u \leq x \wedge v \leq y$

$\alpha_i \leq T_j \qquad \Leftrightarrow i \equiv j$

Note that these rules imply:

1. d^e_i is never a lower bound on any element of D in which an α_j appears for some j not equivalent to i.
2. Similarly, d^e_i is never a lower bound on any element of D in which d_k appears for any k.

Failure of either of these properties to hold would indicate a decrease in "commitment" to evaluate *fully* the denoted expression within block i, once the eager evaluation indicated by d^e_i has begun.

1.3.3. Dominance and Disjunction

A principal result of the strictness analysis method outlined in [13] is the final equivalence relation derived among block numbers. However, another important relationship exists among blocks, reflecting necessary evaluation order.

Definition. A block i *dominates* a block j, denoted $i \angle j$, if whenever block j is executed, that execution is a consequence of block i also being executed.

Dominance among blocks is an important concept that facilitates the generation of efficient object code. For example, if a CSE is shared between two blocks bearing a dominance relationship, then evaluation of that CSE can be moved into the dominating block.

Four axioms govern the dominance relationship:

a. $(\forall i,\ j)\ i \angle j \Rightarrow$
   $(\forall m,\ n)\ m \equiv i \wedge n \equiv j \Rightarrow m \angle n$ *(equivalence consistency)*

b. $(\forall i)\ i \angle i$ *(reflexivity)*

c. $(\forall i, j, k)\ i \angle j \wedge j \angle k \Rightarrow i \angle k$ *(transitivity)*

d. $(\forall i, j)\ i \angle j \wedge j \angle i \Rightarrow i \equiv j$ *(antisymmetry).*

Lastly, a disjunction relationship among block numbers, denoted $i = j \vee k$, is also determined by our analysis. This too is made consistent over equivalence classes, e.g. if $m \equiv i$, $n \equiv j$, and $p \equiv k$, then $m = n \vee p$ in the example above. Disjunctions are used to associate blocks that are related through `cond` operators. If, for example, a CSE is shared by both arms (`then/else` parts) of a conditional, then that expression can be "hoisted" to the surrounding unconditional block.

1.3.4. Sample rules

We denote the strictness annotations of an arc v by $\chi(v)$. The analysis method proceeds by the application of *strictness rules*, expressed in terms of *precondition - postcondition* pairs. These should be interpreted as "if the *precondition* is true, make the *postcondition* true". Notice from the partial ordering specified in section 1.3.2 that "making a postcondition true" can mandate block number equivalencing. For example, suppose $v = \mathtt{ident}(\mu)$, with $\chi(v) = \alpha_i$ and $\chi(\mu) = \beta_j$. Then the postcondition $\alpha_i \leq \chi(\mu)$ ensures that i is equivalenced to j. This captures the idea that `ident` is strict in its argument, and should not constitute a boundary between strictness subsets. Hence block i is merged with block j.

Our strictness analysis rules are designed to be *monotonic* in the sense that:

- If a value in D is changed, it is always to a *greater* value in the partial order.
- If the equivalence relation is changed, it is always through equivalencing two block numbers, thereby *coarsening* it.
- The dominance and disjunction relations are changed only through extension by equivalence consistency.

Our method terminates when no further annotation changes result from rule application, or when it is deemed that sufficient information has been obtained for compilation needs. There is evidence that uniform termination of our method, and others involving infinite domains, cannot be guaranteed [10].

A representative sample of our strictness analysis rules are enumerated in [13]. To suggest their flavor, we exhibit here the "backward" or "demand flow" rules for a (lazy) tuple constructor and selector.

$v_0 = \mathtt{cons}(v_1, v_2)$, where $\chi(v_0) = \alpha_i \wedge \chi(v_1) = \beta_j \wedge \chi(v_2) = \delta_k$

Precondition 1:
$[u, v]_i \leq \alpha_i$
Postcondition 1: *(laziness; j and k not made $\equiv i$)*
$u \leq \chi(v_1) \wedge v \leq \chi(v_2)$

Precondition 2:
$d^e_i \leq \alpha_i$
Postcondition 2: *(eagerness; j and k made $\equiv i$)*
$d^e_i \leq \chi(v_1) \wedge d^e_i \leq \chi(v_2)$

$v_0 = \mathtt{car}(v_1)$, where $\chi(v_0) = \alpha_i$

Precondition:
(none)
Postcondition:
$[\alpha_i, d_k]_i \leq \chi(v_1)$ *(new k, cf. $\perp$)*

2. Resulting Graph Structure

Given a function graph with final annotations and their associated equivalence relation, basic blocks can be formed. We collect together all operators $v = \mathbf{OP}(\ \ldots\ )$, where $\chi(v) = \alpha_i$ for equivalent i. Four varieties of basic blocks result:

- Function bodies.
- Conditional arms.
- Actual parameters.
- Tuple components.

The first two syntactic occurrences always cause block boundaries to be formed. The last two may or may not occur at block boundaries, depending on the strength of the strictness analysis results. To illustrate, we show an annotated graph taken from [13].

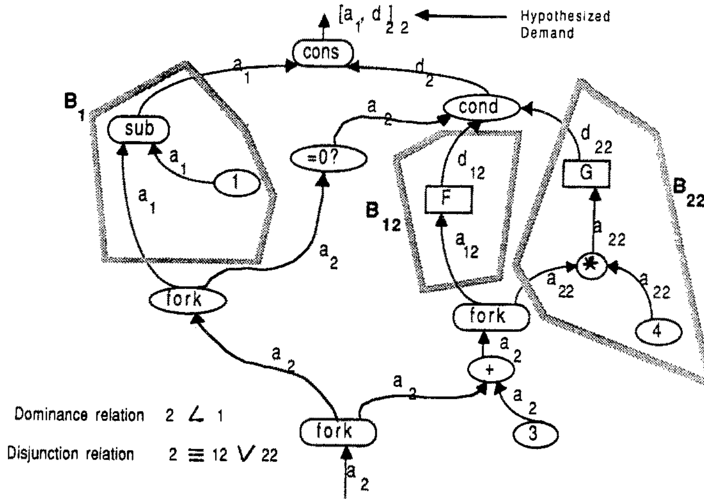

Figure 2-1: Sample annotated function graph.

2.1. Basic Block Structure

The basic blocks that result have several important properties:

1. They nest in a manner consistent with the dominance relation previously obtained.
2. Within a block, the arc annotations convey only datatype information (i.e. atomic vs. pair value predictions). Hence the subscripts used to accomplish block partitioning may now be discarded.
3. The arcs crossing block boundaries represent values produced in one block and consumed in another.

We categorize arcs crossing block boundaries into three kinds, depending on the transition in block level that each represents (see figure 2-2):

1. *Ascending:* arguments to nonstrict operators, e.g. **cons**, **cond** and **apply**.
2. *Descending:* a CSE usage, e.g. (v_0, v_1) = **fork**(v_2), where $\chi(v_0) = \alpha_i$, $\chi(v_1) = \beta_j$, and $\chi(v_2) = \gamma_k$, with $i \angle j$ or $k = i \vee j$.
3. *Lateral:* a CSE as above, but with i and j incomparable.

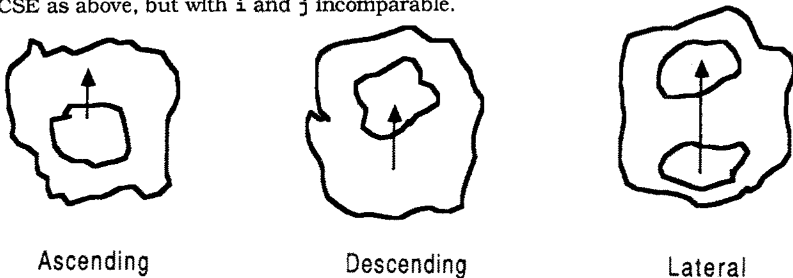

Figure 2-2: Arcs between basic blocks.

2.2. Applicative Order Interpretation of Basic Blocks

As remarked in section 2.1, the outcome of our strictness analysis method has two components: (i) determination of basic blocks, and (ii) atom/list type annotation of the result value denoted by each arc. This information is used in compilation for two distinct purposes: (i) conversion to applicative order evaluation, and (ii) generation of code in which redundant expression evaluation and type checking operations are omitted. We first consider conversion to applicative order, and return to redundant operation suppression in section 2.3.

In fact, once isolation of basic blocks in a NOFL program has been accomplished, the program has in a sense already been converted to applicative order form. This statement will be given sharper meaning in section 3, where a textual representation for a derived applicative order language is specified. However, intuitive support for this claim can be offered as follows:

1. We informally define *applicative order* evaluation on this "blocked" graph representation as follows:

   - Whenever the result of a block is needed at run time, all the operators local to the block are evaluated in bottom up manner. This can be done by unconditional code executing in any order that observes data dependencies, i.e. systematically left to right, in some other order optimizing register usage, or even in parallel.

   - The need for the result of an inner block is a run time event not predictable by the compile time analysis. As noted in section 2, there are four types of blocks, but only three can occur as inner blocks. We consider each in turn.

     - *Conditional arms:* The predicate and any CSE's shared across sibling conditional arms are evaluated in the surrounding block. Once the predicate is evaluated, the **then** or **else** part is selected as appropriate, and its block is evaluated in applicative order.

     - *Actual parameters:* Isolation of an actual parameter into a separate block indicates compile time uncertainty of the need for its evaluation. In our applicative order interpretation, we view such a block as evaluating at function call time to a *suspension*, encapsulating the block's code and values entering through descending arcs.

     - *Tuple components:* Tuple components appearing as separate blocks indicate similar evaluation uncertainty at compile time. These are also compiled into suspensions, for subsequent evaluation as needed by an appropriate selector.

2. This applicative interpretation has many advantages:

   - Evaluation order is outermost-first with respect to block nesting. This, together with our data dependency based evaluation order within blocks, means that all arcs descending into an inner block carry values which are necessarily pre-evaluated when accessed from within.

   - Since inner block nesting is static, these values will be located in known registers or stack locations when compiled code is used. Such values are used directly within conditional arms; for actual parameters and tuples, the values are speedily accessed at suspension creation time, as follows.

     - Formal parameter accesses may trigger the evaluation of an actual parameter, existing as a suspension. These can be represented as function applications, closed with a full parameter set conveying environment values (i.e. those on descending arcs entering the corresponding actual parameter block). Hence normal `apply` operator semantics and implementation techniques can be employed.

     - Selector accesses behave similarly.

2.3. Datatyping Information

The datatype information in arc annotations provides reliable datatype information, in the sense that if an arc is annotated with a datatype indicator `t`, we may be sure that (i) any value **v** conveyed by that arc at run time will be consistent with `t`, and (ii) all usages of **v** will be consistent with `t`, or will check the type more specifically.

This means that many operators, e.g. `car`, can be applied to values without run time type

testing if the value's type annotation, e.g. $[d_i, d_j]_k$, so indicates. If the type annotation is simply d_i, then we know the value will be prior evaluated, but to an unknown type. In this case the `car` can be compiled *with* type testing, but *without* an internal `eval` providing for the case when the value is a suspension.

Even more important are guarantees of prior evaluation as shown by equivalence of subscripts across type descriptor levels. For example, $[d_i, _j]_i$ indicates that when the pair is computed in block `i`, its first component will *also* be computed, to at least an atom or pair (possibly of suspensions). The exhaustive demand indicator d^e, signalling full applicative order, is particularly useful since call by value implementation may be used in thoroughly this case.

2.4. Soundness

Formal proof of the soundness of our applicative order interpretation is beyond the scope of this paper. However, it can be informally argued as follows:

- *Will we evaluate enough?* Yes, because all assumptions of prior evaluation are validated by our method observing outermost-first evaluation order among blocks (observing dominance) and bottom up evaluation order within blocks (observing data dependencies).
- *Will we evaluate too much?* No, because the only dangers are infinite data construction (protected by tuple suspensions), and non data producing runaway recursions (which, in keeping with [14], can only occur *earlier* in our method).

3. Textual Representation

We now turn to the new program representation developed since the preparation of our previous two papers.

3.1. Why a Textual Representation?

The graphical representation of strictness information is conceptually pleasing, but poses some practical difficulties. These include awkwardness of transmission through input and output devices, and unfamiliarity as a programming notation. As an alternative, we have developed a textual notation which captures all the essential information in an annotated graph, while facilitating subsequent processing, especially code generation.

3.2. Intermediate Representation

We require an intermediate representation that captures both the annotations placed on the arcs of the graph and the dominance relation between basic blocks. Typically function application will be represented by the intermediate form `(expr (f ℜ(e1)..ℜ(em)) sr)` where `f` is the function being applied to arguments also represented in our intermediate form. `ℜ(ei)` is the intermediate representation of the ith argument and `sr` is the strictness pattern expected from the function application. `sr` is expressed as a list structure representing values in our *simplified* domain (see sec. 1.3.1). Often, `sr` will be stronger than the pattern to which the function in the expression has been compiled to produce, so appropriate `eval` and type checking instructions will be compiled on the result. The dominance relation between blocks derived by our strictness analysis is used to place the `susp` form defined in section 3.3.

The individual varieties of expressions are represented as follows.

Constants: Unstructured literals are represented directly, since no strictness information is required, e.g. `1, 2.34, nil, true, false` etc.

CSEs: Common subexpressions are introduced via a `let` construct. Each newly introduced variable names a CSE represented in our intermediate form. The let is placed local to the *expr* forming the basic block ℜ(en) local to which the CSE appears.

```
(expr (let ((var1    ℜ(e1))
                     ...
                     (varm    ℜ(em)))
                ℜ(en))
      sr)
```

Local variables and formal parameters:

These are represented as (*var* x sr sx), where x is the formal parameter or local variable, and sr is as explained above. sx is the degree to which the formal parameter x has currently been evaluated. Again, sr may be stronger than sx, if for example a CSE is used in a conditional context that is stronger than its unconditional prior evaluation.

User defined function application:
When all arguments are present in a function application, the representation is as explained above, namely (*expr* (f ℜ(e1) .. ℜ(en)) sr).

Functional argument and higher order functions:
A full treatment regarding functions as first class objects would be beyond the space limits of this paper. However, our an intermediate representation and compilation techniques aim at *full laziness* as defined by Hughes [5], and thus avoid recomputation as much as possible.

Suspended Results: This is represented as (*susp* (f x_1..x_n) sr); see section 3.3.

3.3. Suspended results

In this paper only suspended results for function applications with all arguments will be considered. On the G-machine as presented by Johnsson [7], the creation of a graph for a function application is extremely costly, both in execution time and heap space. We shall see how to optimise this here. A suspended result for a function application (f e1..en) with all arguments present is represented through our analysis as a block boundary enclosing this application. All the arcs *into* the block represent *imports* required to build the suspension ie. the descending arcs as per figure 2-2. We then construct a suspension for a new function g, whose actual parameters are these very imports. Thus the new function g is defined as g x1..xm = f e1..en, where x1..xm correspond to the imported variables. Creating a suspension in this case now simply involves building a graph for a much simpler application. The sr in the susp expression represents the result produced when the suspension is demanded. This information is not utilized in this paper.

3.4. Example

We illustrate our representation on the familiar append function. In figure 3-1, we assume that the strictness signature of append is (d, d) → d, indicating that append is a function of two arguments, both of which will be prior evaluated to either an atom or a (possibly suspended) tuple. Similarly, the result of append is to be delivered already evaluated as an atom or tuple.

```
(fun append (x y)
   (expr (if (expr (null (var x d d)) a)
             (var y d d)
             (expr (cons
                      (expr (hd (var x [⊥, ⊥] [⊥, ⊥])) ⊥)
                      (susp (append1 x y) d))
                    [⊥, ⊥]))
          d))
```

Figure 3-1: Textual representation of annotated append.

Notes:

- The definition for append1 is append1 x y = append (tl x) y, and its strictness signature is append1: ([⊥,⊥] d) → d.
- With the strictness information available for x, it is not necessary to create a suspension for the expression (hd x).

4. Compilation Rules

In [13], hand generated code for figure 2-1 is given to suggest the quality of code that might be generated using the strictness information our method gathers. Our prototype compiler is being written in Prolog [2] mainly for the power of pattern matching provided by unification, and the ease of adding optimisations as new clauses to the compiler. We now formalize the G-code generation as a set of Prolog clauses.

4.1. Major Rule Groups

The major clauses are defined below.

`f(AST, G_CODE)` Generates `G_CODE` for a *function definition* from the intermediate form, `AST`. The clause assumes that actual parameters will have been pre-evaluated to the degree predicted by our analysis before a call to the function is made.

`t(AST, MAP, SDEPTH, G_CODE)`
This clause attempts to perform tail recursion optimisation, for the expression represented in `AST`. `MAP` and `SDEPTH` are the mapping of parameters and local variables to positions on the stack, and the current depth of the evaluation stack respectively.

`e(AST, MAP, SDEPTH, G_CODE)`
This clause will evaluate the expression represented by `AST` and leave a pointer to the result on top of the evaluation stack - the s_stack.

`b(AST, MAP, SDEPTH, G_CODE)`
Generates code for arithmetic operations. This scheme is an optimization to conserve on the utilization of heap space during the generation of intermediate results. The result of evaluating `AST` is left on top of an arithmetic stack - the *a-stack*.

`susp(AST, MAP, SDEPTH, G_CODE)`
Create a graph. `AST` represents an instance of our `susp` intermediate form.

`s(PAT1, PAT2, G_CODE)`
Generates code to raise the strictness pattern of a result from `PAT1` to be `PAT2`. This is typically required when the result produced by a function application is weaker than that required.

4.2. Language Subset

For expository reasons and to demonstrate the use of strictness information during code generation, we shall restrict ourselves to a very small language defined below.

D (*Definitions*) ::=	Fun $x_1..x_n$ = E
E (*Expressions*) ::=	Constants | IntOp E E | RelOp E E
	cons E E | ListOp E | Fun E..E | if E E E
Constants ::=	Integers | Booleans | nil
IntOp ::=	add | sub | mult | div
RelOp ::=	gt | ge | lt | le | eq | ne
ListOp ::=	hd | tl | null
Fun ::=	Identifier

Functions are applied with all arguments present. We now present each of the rule groups in turn. All the clauses have been extracted from our existing compiler. However they are presented here in a simplified form omitting parameters required for later phases of the code generation. Also, most of the error checking clauses and clauses to terminate recursion have been omitted for brevity. The clauses below are written in the DEC10 Prolog syntax [2]. $\perp$ is denoted by ? in the clauses.

4.3. F-scheme: Function Compilation

```
F-1. f([fun, Fname, Parm, Body], GCode) :-
        length(Parm, Lp), Lp1 is L + 1,     /* Compute stack depth */
        args_map(Parm, PMap, Lp1),          /* mapping of params onto stack */
        t(Body, PMap, Lp1, GCode).          /* try Tail Recursion Optimisation */
```

Given a function definition we attempt to perform tail recursion optimisation via the `t-scheme`. `args_map` returns the mapping of formal parameters to positions on the stack. The default is handled by rule T-3.

4.4. T-scheme: Tail Recursion Optimization

```
T-1. t([expr, [F|Args], Sr], Map, Depth, GCode) :-
        type(F, FormalPrm, Sf),                    /* database lookup */
        length(Args, L), length(FormalPrm, L),     /* sufficient args? */
        reverse(Args, RArgs),                      /* not relevant here */
        eArgs(RArgs, Map, Depth, EArgs),           /* evaluate args */
        s(Sf, Sr, Strict),                         /* refine result pattern*/
```

```
        (Strict == [] ->
            append(EArgs, [[move,M],[jfun,F]], GCode);
            flatten1([EArgs, [[move,M],[fcall,F]], Strict], GCode))

T-2. t([expr, [if, E1, E2, E3], Sr], Map, Depth, GCode) :-
        b(E1, Map, Depth, E1Code),              /* result on a-stack*/
        t(E2, Map, Depth, E2Code),              /* then part */
        t(E3, Map, Depth, E3Code),              /* else part */
        (flatten1([E1Code, [[jtrue,L1]], E3Code,
                           [[label,L1]], E2Code],
                GCode)).

T-3. t(E, Map, Depth, GCode) :-                 /* default case */
        e(E, Map, Depth, ECode),                /* evaluate and update */
        flaten1([ECode, [[update], [ret]]], GCode).

T-4. eArgs([A| Args], Map, Depth, GCode) :-
        e(A, Map, Depth, E1Code),               /* evaluate arg */
        Depth1 is Depth + 1,                    /* position for next arg*/
        eArgs(Args, Map, Depth1, ArgsCode),     /* rest of args */
        append(E1Code, ArgsCode, GCode).
```

When a user defined function culminates in a call to another user defined function, tail recursion optimisation is possible. In Rule T.1 we check that the function has been supplied with sufficient arguments, and evaluate them via the `eArgs` clause. The `eArgs` evaluates each of the arguments using the `e-scheme`. Note that the degree to which each of the arguments is to be evaluated is contained in our abstract representation. The `s-scheme` is used to raise the pattern produced by the function, namely `Sf`, to the level `Sr`. Tail recursion optimisation (via the ***jfun*** instruction) is only possible if this refinement code is absent. The newly created arguments are moved in place of the old via the [***move***, M] instruction. Rule T.2 propagates the recursion scheme into the branches of the conditional, with the default rule being T.3. The `t-scheme` is extended naturally to the `let` construct by evaluating the common subexpressions to the degree required, and propagating the `t-scheme` into the expression to be evaluated. The default rule evaluates the expression and updates the result application node.

4.5. E-scheme: Evaluate Expression

```
E-1. e(I, _, _, [[pushint, I]]) :- integer(I).  /* no evaluation required*/
                likewise for boolean constants and nil.

E-2. e([var, X, Sr, Sx], Map, Depth, [[push, OFFSET]| STRICT]) :-
        assoc(X, Map, [X, OFFSET]),
        s(Sx, Sr, STRICT).                          /* refinement possible*/

E-3. e([expr,[add,E1,E2],Sr], Map, Depth, GCode) :-   /* arithmetic */
        (Sr==d; Sr==a; Sr==de),
        b([expr,[add,E1,E2],a], Map, Depth, BCode),
        append(BCode, [[mkint]], GCode).
                likewise for sub, div, mult.

E-4. e([expr,[eq,E1,E2],Sr], Map, Depth, GCode) :-    /* relational */
        (Sr==d; Sr==a; Sr==de),
        b([expr,[eq,E1,E2],Sr], Map, Depth, BCode),
        append(BCode, [[mkbool]], GCode).

E-5. e([expr,[if,E1,E2,E3],Sr], Map, Depth, GCode) :- /* conditional */
        b(E1, Map, Depth, E1Code),
        e(E2, Map, Depth, E2Code),
        e(E3, Map, Depth, E3Code),
        flatten1([E1Code, [[jtrue,L1]], E3Code, [[jmp,L2]],
                          [[label,L1]], E2Code, [[label,L2]]],
                GCode).

E-6. e([expr,[null,E],Sr], Map, Depth, GCode) :-          /* null operator */
        e(E, Map, Depth, ECode),
        append(ECode, [[null]],  GCode).
```

```
E-7. e([expr,[hd,[var,X,_,[B,_]]],A], Map, Depth,           /* (hd x) */
                                   [[push,OFF],[hd]| STRICT]) :-
        assoc(X, M, [X, OFF]),
        s(B, A, STRICT).                              /* refinement possible */
              Likewise for tl function        /* after direct access */

E-8. e([expr,[hd,[var,X,_,d]],A], Map, Depth,
                                   [[push,OFF],[hd_check]| STRICT]) :-
        assoc(X, M, [X, OFF]),                        /* test required */
        s(?, A, STRICT).                              /* refinement possible */
              Likewise for tl

E-9. e([expr,[hd,[var,X,_,de]],A], Map, Depth,
                                   [[push,OFF],[hd_check]| STRICT]) :-
        assoc(X, M, [X, OFF]),                     /* test required */
        s(de, A, STRICT).                          /* further tests possible */
              Likewise for tl

E-10. e([expr,[hd,[expr,[F|Args],_]],A], Map, Depth, GCode) :-
        type(F, P, [B,_]),
        length(P, L), length(Args, L),                /* sufficient args */
        e([expr,[F|Args],[B,?]], Map, Depth, FCode),  /* call function */
        s(B, A, STRICT),                              /* refine result */
        flattenl([FCode, [hd], STRICT], GCode).
              Likewise for tl

E-11. Rule 8 is extended to handle function application in a fashion similar
    to rule 10.

E-12. Rule 9 extended to handle function application in a fashion similar to
    rule 10.

E-13. e([expr,[hd,E],_], Map, Depth, GCode) :-        /* default case*/
        e(E, Map, Depth, ECode),
        append(ECode, [[hd_check]], GCode).

E-14. e([expr,[cons,E1,E2],_], Map, Depth, GCode) :-
        e(E1, Map, Sd, Vd, E1Code),                   /* head part */
        Sd1 is Sd +1,
        e(E2, Map, Sd1, Vd, E2Code),                  /* tail part */
        flattenl([E1Code, E2Code, [[cons]]], Code).

E-15. e([susp|S], Map, Depth, GCode) :-               /* graph creation */
        susp([susp|S], Map, Depth, GCode).

E-16. e([expr,[F|Args],Sr], Map, Depth, GCode) :      /* function application */
        type(F, P, Sf),
        length(P, M), length(Args, M),                /* sufficient args */
        reverse(Args, RArgs),
        Depth1 is Depth + 1,
        eArgs(RArgs, Map, Depth1, EArgs),             /* evaluate args*/
        s(Sf, Sr, STRICT),                            /* refine result */
        flattenl([[[mkhole]], EArgs, [[fcall,F]], STRICT], GCode).

E-17. e(E, _, _, _, [[error]]) :-
        error("Cannot compile expression").
```

The **e-scheme** leaves a pointer to a result on top of the evaluation stack. Rule E-2 refines a formal parameter to the degree required. Rules E-3 and E-4 transfer the code generation task to the B-scheme which computes all temporaries on the arithmetic stack, until the result is to be finally transferred to the s-stack via the ***mkbool*** or ***mkint*** instructions. In Rule E-6, the instruction ***null*** leaves a boolean result on the a-stack. Rules E-7 to E-13 are optimisations on the head and tail functions, which take advantage of the specific situation to generate better code. The default rule is E-13. The instruction ***hd_check*** accesses the *head* component of the list after a type *check* has been performed. In Rule E-14 we handle function application. The degree of evaluation required by each of the arguments will be manifest in our intermediate representation. The ***mkhole*** instruction makes space for the result on the heap, and the ***fcall*** instruction performs the context switch. This rule

under fully strict conditions generates code that follow the *call-by-value* semantics of parameter passing.

4.6. B-scheme: Compute Basic Value

```
B-1. b(I, _, _, [[pushbasic,I]]) :- integer(I).
                Likewise for boolean constants.

B-2. b([var,X,Sr,a], Map, Depth, [[getv,OFF]]) :-
        (Sr==d; Sr==a),
        assoc(X, Map, [X, OFF]).

B-3. b([expr,[add,E1,E2],Sr], Map, Depth, GCode) :-
        b(E1, Map, Depth, B1Code),
        b(E2, Map, Depth, B2Code),
        flattenl([B1Code, B2Code,[[add]]], GCode).
                Likewise for sub, mult, div, eq, ne, gt, ge, lt, le

B-4. b([expr, [if, E1, E2, E3], Sr], Map, Depth, Code) :-
        (Sr == d ; Sr == a),
        b(E1, Map, Depth, E1Code),
        b(E2, Map, Depth, E2Code),
        b(E3, Map, Depth, E3Code),
        flattenl([E1Code, [[jtrue,L1]], E3Code, [[jmp, L2],
                        [label, L1]], E2Code, [[label, L2]]],
            Code).

B-5. b(E, Map, Depth, GCode) :-
        e(E, Map, Depth, Vd, ECode),
        append(ECode, [[get]], GCode).
```

The B-scheme rule does the standard bottom up evaluation of entirely strict expressions on the arithmetic stack. Rule B-5, needs to resort to the **e-scheme** to compute the value of the expression E, and get its result on the a-stack.

4.7. Susp-scheme: Create Graph

```
susp([susp,[F|Free],_], Map, Depth, [[pushfun,F]| FreeC]) :-
        Depth1 is Depth + 1,
        susp_param(Free, Map, Depth1, FreeC).

susp_param([X| Free], Map, Depth, [[push,OFF],[mkap]| FreeC]) :-
        assoc(X, Map, [X, OFF]),
        Depth1 is Depth + 1,
        susp_param(Free, Map, Depth1, FreeC).
```

This scheme constructs the graph for the susp intermediate form. We merely need to determine the offset of the free variables required in the graph and connect them together via the ***mkap*** instruction.

4.8. S-scheme: Strictness Pattern Refinement

```
S-1. s(X, X, []).

S-2. s(?, d, [[eval]]).

S-3. s(?, a, [[eval],[atomicp]]).

S-4. s(?, [A,B], [[eval],[listp]| GCode]) :- s([?,?], [A,B], GCode).

S-5. s(d, a, [[atomicp]]).

S-6. s(d, [A,B], [[listp]| Code]) :- s([?, ?], [A, B], Code).
```

```
S-7. s(a, [A, B], [[error]]) :- error("Atomic value where list expected").

S-8. s(de, a, [[atomicp]]).

S-9. s(de, [A,B], [[listp]| Code]) :-
        s([de,de], [A,B], Code).

S-10. s([A,B], [C,D], Code) :-
        s(A, C, Code1),
        s_head(Code1, Code1_head),
        s(B, D, Code2),
        s_tail(Code2, Code2_tail),
        append(Code1_head, Code2_tail, Code).

S-11. s(_, _, []).

S-12. s_head([], []).
S-13. s_head(Code, Code_head) :-
        flatten1([[[push_top],[hd]], Code, [[pop]]], Code_head).

S-14. s_tail([], []).
S-15. s_tail(Code, Code_tail) :-
        flatten1([[[push_top],[tl]], Code, [[pop]]], Code_tail).
```

`s(Pat1, Pat2, GCode)` generates code to refine `Pat1` to be `Pat2`.

4.9. Peephole Optimisations

Direct short cuts are made when updating the application node with the result. Instead of forming the result structure on top of the stack and then copying the result into the application node to be updated, we directly create the result on the application node. Thus the following optimisations result:

```
[cons][update] → [update_cons]
[mkint][update] → [update_int]
[mkbool][update] → [update_bool]
[mkap][update] → [update_appl]
```

In the same spirit there is no need to create a boolean value on the a-stack if it is going to be immediately tested and removed in the next instruction. Therefore we get the following optimisation:

```
[eq][jtrue,Label] → [jeq, Label]
[lt][jtrue,Label] → [jlt, Label]
[null][jtrue,Label] → [jnull, Label]     etc.
```

5. Sample Code Generated

Below we give the code generated for the `from` function defined below.

```
from x y = if x > y then nil else cons x (from (x + 1) y);
```

The intermediate representation assuming a strictness signature of `(a a)` → `de` is shown below. This was used to generate the first column in figure 5-1. The second column in figure 5-1 was generated assuming a strictness signature of $(\perp\ \perp) \rightarrow d$. This is a convenient example to hand test the rules given in this paper.

```
(fun from (x y) (expr (if E1 E2 E3) de))
where
      E1 = (expr (gt2 (var x a a) (var y a a)) a)
      E2 = nil
      E3 = (expr (cons (var x de a)
                       (expr (from (expr (add2 (var x a a) 1) a)
                                   (var y a a))
                             de)
                 (de de)))
```

Several differences should be noted when comparing the code generated in the two cases, referred to as the strict version (SV) and the non strict version (NSV).

```
1.  getv(3)          /* a_stack := x */
2.  getv(2)          /* a_stack := y */
3.  jgt2(g0001)      /* test */

4.  push(3)          /* s_stack := x */
5.  mkhole           /* result node */
6.  push(2)          /* s_stack := y */
7.  getv(3)          /* a_stack := x */
8.  pushbasic(1)     /* a_stack := 1 */
9.  add2
10. mkint            /* s_stack := x+1 */
11. fcall(from)     /* call from */
12. update_cons      /* make result */
13. ret

14. label(g0001)
15. pushnil          /* s_stack := nil */
16. update
17. ret
```

```
1.  push(3)          /* s_stack := x */
2.  eval             /* evaluate */
3.  atomicp          /* atom test */
4.  get              /* a_stack := x */
5.  push(2)          /* s_stack := y */
6.  eval             /* evaluate */
7.  atomicp          /* atom test */
8.  get              /* a_stack := y */
9.  jgt2(g0002)      /* test */
10. push(3)          /* param x */
11. pushfun(from1)/* new function*/
12. push(3)          /* s_stack:= x */
13. mkap             /* make graph */
14. push             /* s_stack:= y */
15. mkap             /* make graph */
16. update_cons      /* make result */
17. ret

18. label(g0002)
19. pushnil
20. update
21. ret
```

Figure 5-1: Sample Code For Function `from`

- In the SV, the parameters are directly accessed and moved to the arithmetic stack whereas in the NSV, evaluation and type checking is performed.
- The SV implements *call-by-value* parameter passing semantics whereas the NSV creates a suspended result to be later evaluated upon demand.

One would expect that in a fully strict version of a function, the G-code generated would be comparable to that produced by any LISP compiler. Indeed our preliminary timing tests seem to confirm this notion.

Our analysis method is currently under development. To be able to test our compiler we have developed an annotated user language, with type declarations, where the type information is propagated into the subexpressions of a function definition. The resulting intermediate form is not as rich as the one we expect from the analysis due to the simple nature of the pattern propagation. The resulting G-code is macro expanded to form a C program [9]. This has enabled us to perform some valuable comparisons, the results of which are summarized below. All timings were measured on a VAX 8600 running UNIX™.

	SV	NSV[2]	ML[3]	Miranda	PSL[4]	C	Pascal
fib 20	0.7	1.4	1	26.3	0.7	0.1	0.1
tak 18 12 6	2.1	7.2	11	87.0	1.4	0.3	0.8
sieve 2..500 (10 times)	3.2	7.0	13	43.0	3.2	-	-
insertion sort[5]	4.3	13.0	23	51.0[6]	2.8	-	-

References

[1] 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.

[2] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1984.
2nd Edition.

[2] A small amount of strictness information was used in defining some of the functions to avoid the tedium involved in our annotated source language

[3] Standard ML, Timing Resolution = 1 sec

[4] Compiled Portable Standard LISP without fast integers

[5] Sorted a list of 500 elements in reverse order

[6] Sorted a list of 250 elements in reverse order

[3] Fairbairn, Jon, and Stuart C. Wray.
Code generation techniques for functional languages.
In *Proc. Symp. on Lisp and Func. Pgmming.*, pages 94-104. ACM, 1986.

[4] 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.

[5] Hughes, R. J. M.
Super Combinators.
In *Lisp and Functional Programming Conference*, pages 1-10. ACM, 1982.

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

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

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

[9] Kernighan, B.W. and Ritchie, D.M.
Software Series: The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1978.

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

[11] Kuo, T.-M., and P. Mishra.
On Strictness and its Analysis.
In *Proc. Symp. on Princ. of Pgmming. Lang.*. ACM, Munich, West Germany, March, 1987.
To appear.

[12] Lindstrom, Gary.
Static evaluation of functional programs.
In *Proc. Symposium on Compiler Construction*, pages 196-206. ACM SIGPLAN, Palo Alto, CA, June, 1986.

[13] Lindstrom, Gary, Lal George and Dowming Yeh.
Optimized compilation of functional programs through strictness analysis.
August 4, 1986.
Technical summary.

[14] 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 Computer Science, vol. 83.

[15] Sheeran, Mary.
Designing regular array architectures using higher order functions.
In *Proc. Conf. on Functional Programming Languages and Computer Architectures*, pages 220-237. Springer Verlag, 1985.
Lecture Notes in Computer Science, vol. 201.

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

[17] Warren, David H. D.
Applied logic: its use and implementation as a programming tool.
Technical Report, SRI, Inc., 1983.
Note 290.