Fast Strictness Analysis Based on Demand Propagation

R. SEKAR
Bellcore
and
I.V. RAMAKRISHNAN
SUNY at Stony Brook

Strictness analysis is a well-known technique used in compilers for optimization of sequential and parallel evaluation of lazy functional programming languages. Ever since Mycroft's pioneering research in strictness analysis, there have been substantial research efforts in advancing the basic technique in several directions such as higher-order functions, nonflat domains, etc. While almost all of these methods define strictness based on denotational semantics, in 1990 we proposed an operational method called *ee*-analysis. Operational methods directly address the primary aspect of the strictness analysis problem, namely nontermination of evaluation, and can yield simplicity both in formulation and computational algorithms. Moreover, the use of operational approach has led us to a simple and natural treatment of polymorphism. While *ee*-analysis reasoned about *normal-form* evaluation herein we extend its power substantially so as to deal with other degrees of evaluation that are intermediate between normal and head-normal form. An interesting aspect of our approach is our formulation of a strictness property as a constraint that relates the demand placed on the output of a function to the demands placed on its arguments. Strictness properties are then computed using symbolic constraint-solving techniques. An important advantage of constraint-driven analysis is that it captures interargument dependencies accurately. Moreover, the analysis performance is relatively insensitive to domain size. Based on our implementation of this method, we show that our analysis techniques are efficient, as well as effective.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Program analysis, program optimization, strictness analysis

1. INTRODUCTION

Use of strictness analysis in parallel evaluation and optimization of lazy functional languages is well known. The first formal treatment of strictness analysis appeared in Mycroft's seminal research [Mycroft 1980]. His work dealt only with *flat domains* wherein no structures (such as lists and tuples) are permitted. Since then considerable research has been done on extending it to deal with data types over nonflat domains. (See for example the works of Burn [1987], Hall and Wise [1987], Hughes [1985], Lindstrom [1986], Hughes and Wadler [1987], and Wadler [1987].)

Work partially supported by NSF grants CDA-9303181, CCR-9404921 and INT-9314412. A preliminary version of some parts of this article was presented in POPL '90.
Authors' addresses: R. Sekar, Bellcore, Room 1J–226R, 445 South Street, Morristown, NJ 07960; email: sekar@bellcore.com; I.V. Ramakrishnan, Department of Computer Science, SUNY at Stony Brook, NY 11794; email: ram@cs.sunysb.edu.

All of these methods define strictness based on denotational semantics. In contrast to these approaches where the strictness analysis problem is defined indirectly on a semantic domain, orthogonal approaches based on operational semantics have been proposed by Kuo and Mishra [1989] and Sekar et al. [1990]. As pointed out by Kuo and Mishra [1989], operational methods directly address the primary aspect of strictness analysis problem, namely nontermination of evaluation, while making minimal requirements about other aspects of the language semantics that have no bearing on this aspect (e.g., the type system). This factor can lead to simplicity of formulations, as well as algorithms. While Kuo and Mishra focused on higher-order functions over flat data types, our work in Sekar et al. [1990] focused on the complementary problem of analysis of first-order functions over nonflat domains. Our formulation of strictness analysis, called *ee*-analysis, reasoned about *normal-form* evaluation, was quite intuitive, and was presented together with efficient algorithms for computing strictness. This article presents and thoroughly extends the power of *ee*-analysis to deal with other forms of strictness (such as *head-* and *tail-strictness* on lists), so as to reason about degrees of evaluation that are intermediate between normal and head-normal form.

1.1 Overview of Approach

We formulate strictness analysis in a first-order language using *demands*, which capture the degree of evaluation of a term. A demand D is formalized as membership in a subset $\mathcal{T}_D$ of the universe of terms. Strictness analysis is then formulated in terms of *demand analysis,* which describes how a function transforms a demand on its output into a demand on its arguments. This formulation corresponds to *backward analysis*, similar to that of Hughes and Wadler [1987] and Dybjer [1991], and is quite different from formulations based on abstract interpretation [Cousot and Cousot 1977] that typically use *forward analysis.* (Forward analyses yield the definedness of output of a function, given the definedness of its arguments.) However, previous approaches based on backward analysis (including the *reversing* abstract interpretation approach described in Hughes and Launchbury [1992b]) typically reason about evaluation of each function argument in isolation, thereby losing information about interargument dependencies. The most-dramatic manifestation of this loss arises in the context of the conditional *if*, where backward analyses lose the information that *if* needs to evaluate one of the *then* or *else* clauses. Since this will greatly impact the accuracy of analysis, backward analyses typically give special treatment to *if* (see, for example, Hughes and Wadler [1987]). The principal difficulty in this context has been the development of *conceptually simple* and *computationally efficient* representations that capture interargument dependencies. Our approach proposes a solution to this problem by using *symbolic constraints* that relate the demands placed on all the arguments to the demand placed on the output of a function. This enables us to reason about interargument dependencies, as in the case of *if*. In fact, our analysis can go far beyond this simple case of handling *conditional strictness*, i.e., strictness in one argument that arises when certain other arguments possess a particular strictness property. For instance, in the *matmult* example shown in Figure 1, our analysis detects that *matmult* is strict in its second argument *whenever its first argument is non-nil.* Previously known approaches would conclude that *matmult* is nonstrict in its second argument. More

$$
\begin{aligned}
matmult(x\!:\!xs, y) &\rightarrow rowmult(x,y)\!:\!matmult(xs,y)\\
matmult(nil, y) &\rightarrow nil\\
rowmult(x, y\!:\!ys) &\rightarrow dotmult(x,y)\!:\!rowmult(x,ys)\\
rowmult(x, nil) &\rightarrow nil\\
dotmult(x\!:\!xs, y\!:\!ys) &\rightarrow x * y + dotmult(xs,ys)\\
dotmult(nil, nil) &\rightarrow nil
\end{aligned}
$$

Fig. 1. Matrix multiplication example.

importantly, under normal-form demand, we can conclude that *matmult* needs both arguments in normal form when neither is *nil*. In contrast, traditional approaches would conclude that neither of the arguments is needed in normal form, thereby losing significant opportunities for optimization. These losses are particularly important in parallel implementations, where strictness information is often used to determine which expressions are evaluated in parallel.

Another important aspect of our formulation is that it is based on operational semantics, thereby directly addressing the fundamental aspect of strictness analysis in nonflat domains, namely, required degree of evaluation. This makes it easy to relate the analysis formulation with the optimization enabled by the analysis. Moreover, wherever necessary, we can easily incorporate assumptions that affect evaluation (e.g., the way pattern matching is performed), while excluding many aspects of language semantics that have no bearing on evaluation (e.g., whether all looping terms are considered as equal). These factors simplify our analysis and proofs.

Another important outcome of the operational approach is our treatment of polymorphism — we use demand variables (*a la* type variables) in the demand formulas to capture polymorphism directly. For instance, one of the strictness properties of *append* is given by

$$append^{\alpha\ list} \vdash x_1^{\alpha\ list} \wedge x_2^{\alpha\ list}$$

which says that whenever every element in the output of *append* needs to be evaluated to some extent α, then the same demand is placed on both arguments of α. In contrast, previous approaches based on denotational semantics employ an indirect approach: they specify strictness at a *monomorphic instance* of a polymorphic function and then use *polymorphic invariance proofs* [Abramsky 1985] to establish that the same property is valid at all instances. Such polymorphic invariance proofs can be quite difficult to obtain (see Hughes and Launchbury [1992a] for example).

Having defined our notion of strictness based on *demand constraints*, we then develop algorithms for computing these constraints. The analysis algorithms treat the demand constraints as *syntactic entities*, manipulated using *symbolic constraint-solving* techniques. The use of such techniques has made our analysis quite efficient, so that it can process several thousand lines of source code per second. More importantly, use of such symbolic techniques makes our analysis relatively insensitive to domain size. While efficiency can often be achieved by specializing the constraint-solving algorithms to particular constraints, this will have a negative impact on extending the analysis to deal with different kinds of domains. However, our approach does not sacrifice extensibility for efficiency. In particular, the analysis algorithms are partitioned into two independent modules: the first module is parameterized with respect to a few elementary operations on demands and supplies operations to manipulate demand constraints. The second module is specific to the domain and supplies the domain-dependent operations on demands needed by the first component. The domain component itself is structured so as to permit reuse across different domains by specifying the domain structure as data.

1.2 Summary of Results

We begin in Section 2 with a simple instance of demand analysis called *ee*-analysis. This analysis deals with *e*-demand or *normal-form* demand. (Lindstrom [1986] refers to it as d^e.) The *ee*-strictness property of a function f specifies those arguments of f that are needed in normal form in order to normalize an application of f. *ee*-strictness is an important form of strictness that is intuitive to understand, conceptually simple, and practically very useful. *ee*-analysis lays the groundwork to simplify exposition of the more-general demand analysis in subsequent sections. The results presented in this section include:

—We define *ee*-strictness and describe a method to compute it. The analysis operates on a two-point domain, which enables us to use particularly efficient representations for demand constraints, leading to fast implementations. Our implementation of the analysis can process about 15,000 lines of source code per second on a Sun Sparc LX. To put this in the context of the total compilation time, the analysis is about three orders of magnitude faster than compilation time using a modern functional language compiler such as SML/NJ (version 0.93).

—We show that *ee*-analysis is quite effective in practice. On a collection of benchmark programs, we have found that it identifies about 80% of the arguments to all functions as *ee*-strict arguments. (See Section 4.3.1 for details.)

—We present an effective linear-time heuristic (in the size of the program) for detecting functions that are *ee*-strict on all their arguments.

ee-analysis has formed the basis of parallelism detection and other optimizations in the EQUALS system by Kaser et al [1992; 1994] for parallel evaluation of lazy functional programs. The use of *ee*-analysis has resulted in high scalability, even up to 20 processors. It has also improved sequential performance by a factor of 1.5 to 3, making it comparable to that of a strict language such as SML/NJ. (See Section 4.3.2 for details.)

We then proceed to describe our analysis method in full generality in Section 3. The strictness of a function f under demand D is a constraint that specifies the demand on arguments $e_1, ..., e_n$ whenever the output of $f(e_1, ..., e_n)$ is required under demand D. With this formulation, we present the following results in this section:

—We present our method for strictness analysis with arbitrary demands. To make the analysis modular and reusable, it is structured into two components — the *demand constraint component* and the *domain component*. The former is parameterized by a few elementary operations on demands. The latter is specific to a particular analysis domain and supplies the domain specific operations. An important feature of this organization is that the analysis algorithms, as well as the correctness proofs, are modularized, thereby permitting the two components to be extended or modified independently.

—We show that demand analysis is quite expressive and effective using several examples. It can handle a wide variety of data structure strictness including head- and tail-strictness. Moreover, it can treat polymorphism quite naturally by representing variables in demand constraints and defining the domain-dependent functions to deal with these variables.

—Demand analysis can capture interargument dependencies accurately and thereby identify conditional strictness.

—We have implemented the general demand analysis and found that it is quite efficient. The analysis can process two to three thousand lines per second on a Sun Sparc LX, which is about two orders of magnitude faster than the SML/NJ compiler running on the same machine. Our implementation is also modular and exhibits significant degree of reuse from one instantiation to another. Moreover, the use of symbolic representations has made our analysis relatively insensitive to domain sizes. (See Section 4.3.3.)

Following the presentation of demand analysis, we discuss in Section 4 various issues in using demand analysis in practice. We outline code generation using the results of demand analysis and illustrate it with an example. It follows naturally from our analysis formulation that the degree of evaluation of arguments would depend upon the demand on the output of a function. (In this sense, our approach to code generation is similar to the evaluation-transformer-model-based approach [Burn 1987; Finne and Burn 1993].) We then outline how practically useful demands can be automatically synthesized, based on output types of functions. Our ideas are similar to previous work by Hughes and Launchbury [1992a].

In Section 4.3 we discuss the results of implementing *ee*-analysis and the more-general demand analysis. We present performance figures on efficiency and effectiveness of these analyses. For *ee*-analysis, we also present our experience in using it for detection of parallelism and other optimizations in the EQUALS system.

Traditionally, strictness analysis has been used for optimization but does not impact the semantics of programs in any other way. In Section 5 we show how demand analysis can be used to go beyond such optimizations. In particular, most lazy functional languages introduce an operational component in the semantics so as to define the meaning of pattern matching. To avoid introducing such an operational component, one must make use of purely lazy pattern-matching techniques [Huet and Levy 1979; O'Donnell 1985; Puel and Suarez 1990; Sekar and Ramakrishnan 1993; Sekar et al. 1992]. Unfortunately, such techniques do not handle all programs, and it is cumbersome to enforce restrictions to enable lazy pattern matching. To overcome this problem, we need to extend lazy pattern-matching algorithms to deal with a larger class of programs. We present an approach in this direction, based on an orthogonal generalization of demand analysis that reasons about head-normal forms of term-prefixes.

We present a detailed comparison of our work with previous works on strictness analysis in Section 6. Finally, concluding remarks appear in Section 7.

1.3 Preliminaries

We use a first-order language based on constructor-based term-rewriting system (TRS). In such a system, a function is defined by a set of oriented equations (called *rewrite rules*) of the form $l \rightarrow r$ where the left-hand side (lhs) l and right-hand side (rhs) r are terms that may contain variables. Almost all first-order programs written in functional languages such as LML [Augustsson 1984], Miranda [Turner 1985], HOPE [Burstall et al. 1980], and O'Donnell's equational logic [O'Donnell 1985] constitute such a TRS. The operational semantics of this language is based on *reductions* or *rewriting*. In particular, we repeatedly replace *redexes* (a redex is an instance of a left-hand side) in the input term by the corresponding right-hand side until the *normal form* of the term (which contains no redexes) is obtained. Such an evaluation algorithm is said to be *complete* if it finds the normal form of the input term whenever the normal form exists. It is said to be *lazy* if it reduces only *needed redexes*, i.e., those redexes that must be reduced in every sequence of reductions that normalize the given input term. (See Huet and Levy [1979] for a complete treatment of lazy evaluation strategies based on needed redexes.)

In a constructor-based TRS all the symbols are partitioned into two disjoint sets $\mathcal{F}$ and $\mathcal{C}$ such that only the outermost symbol of every lhs is in $\mathcal{F}$ while the rest are in $\mathcal{C}$. For a term t, $\mathcal{V}(t)$ is the set of variables in it. We assume that $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ for every rule $l \rightarrow r$. A *substitution* β maps variables to terms. We use the notation $\beta(x)$ to denote the term mapped to by x under β. If the substitution maps every variable into a variable-free term, then it is said to be a *ground* substitution. By $t[u \leftarrow v]$ we refer to the term obtained from t by replacing the subterm u in t by v. An *instance* $t\beta$ of a term t is obtained by replacing x by $\beta(x)$ for each variable x in t. We say that t is a *prefix* of another term s iff s is an instance of t. A *redex* in term u is an occurrence of an instance of an lhs. A term t is in *normal form* (NF) if it contains no redexes. By $t \rightarrow u$ we mean that u is obtained by replacing a redex $l\beta$ in t by $r\beta$, where $[l \rightarrow r]$ is a rewrite rule. The reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$. A term t is in *head-normal form* (HNF) if it has a constructor at the root. We remark here that the standard notion of head-normal form is weaker: it simply requires that t not be reducible to any t' such that t' contains a redex at the root. We use the stronger notion for two reasons. First, lazy evaluation is driven by the need to reduce subterms to terms with constructor roots — not merely to head-normalize them. Therefore reasoning about constructor-root forms is more appropriate. Second, in first-order languages that enforce a strong type discipline, it is considered a type error for a term to possess a head-normal form without a constructor at the root. Thus the two notions coincide in (the first-order fragment of) most functional languages.

In the rest of this article, we use f, g, h to refer to function symbols, c to constructor symbols, x, y, z, u, v to variables, s, t, e to arbitrary terms, and σ, β to substitutions.

2. *EE*-ANALYSIS

In this section, we present a conceptually simple (yet effective) analysis, called *ee*-analysis. As in Lindstrom [1986] we associate strictness with the notion of *demand propagation*, i.e., how a demand on the output of a function is transformed into demands on its arguments. A *demand* specifies the extent to which a term t needs to be evaluated. In this section, we deal with only one demand, namely, the exhaustive (e-) demand that requires a term to be evaluated to normal form. In subsequent sections, we deal with other kinds of demands such as head-normal-form demand or demands that correspond to various intermediate degrees of evaluation.

Our *ee*-analysis method determines which arguments of a function need to be normalized in order to yield the normal form of the function application. In order to capture interargument dependencies, we view functions as transforming a demand on their output into a *boolean constraint* on the demand on their arguments. The analysis operates by symbolic manipulation of these constraints. Specifically, strictness of a term t is given by a boolean constraint on the variables in t. Each conjunction in (the disjunctive normal form (DNF) of) this constraint corresponds to one possible sequence of reductions that normalizes an instance of t and specifies those variables in t that are e-demanded in that sequence. More precisely:

Definition (*ee-Strictness*). Let t be a term, $\varphi = (x_{11} \wedge \cdots \wedge x_{1r_1}) \vee (x_{21} \wedge \cdots \wedge x_{2r_2}) \vee \cdots \vee (x_{n1} \wedge \cdots \wedge x_{nr_n})$, with $x_{ij} \in \mathcal{V}(t)$. Then φ is an *ee*-strictness property of t iff, in every reduction sequence that normalizes any instance $t\beta$ of t, all the terms in one of the sets $\{\beta(x_{11}), ..., \beta(x_{1r_1})\}, ..., \{\beta(x_{n1}), ..., \beta(x_{nr_n})\}$ are normalized.

For instance, the *ee*-strictness of *if* is $(x_1 \wedge x_2) \vee (x_1 \wedge x_3)$, which means that in order to normalize $if(t_1, t_2, t_3)$, we have to normalize either t_1 and t_2 or t_1 and t_3.

2.1 Analysis Method

We use the notation t^{ee} to denote the *ee*-strictness of a term t. The strictness of arbitrary terms is computed by *composing* the strictness of constructors and function symbols using the following rule.

Definition (*Rule of Substitution*). Let t be a term, $x_1, ..., x_n$ be the variables in t and β be a substitution for these variables. Then

$$[t\beta]^{ee} = t^{ee}[x_i \leftarrow [\beta(x_i)]^{ee}].$$

To illustrate this rule, consider the term $t = f(x', x'')$ and the substitution $\beta(x') = x, \beta(x'') = g(y, z)$. The rhs of the above rule becomes $f^{ee}[x' \leftarrow x, x'' \leftarrow g^{ee}(y, z)]$. Suppose that $f^{ee}(x', x'') = x' \vee x''$, and $g^{ee}(y, z) = y \vee z$. Then the rhs would simplify to yield the strictness of $t\beta = f(x, g(y, z))$ as $x \vee (y \vee z)$.

Now we proceed to the equations to compute x^{ee}, c^{ee}, and f^{ee}, so that, together with the above rule of substitution, we can compute strictness of arbitrary terms. For the variable case, its substitution is to be normalized, and hence $x^{ee} = x$. For the case $c(x_1, ..., x_n)$, the NF of its instance $c(s_1, ..., s_n)$ is obtained only by normalizing each of $s_1, ..., s_n$. Hence c^{ee} is the conjunction of the x_i. Note that, by this definition, if the arity of c is zero then $c^{ee} = true$. This discussion yields the following two equations:

$$x^{ee} = x \tag{1}$$

$$[c(x_1, ..., x_n)]^{ee} = x_1 \wedge \cdots \wedge x_n \tag{2}$$

For the case when the root symbol is a function symbol f, assume that f is defined by the rules:

$$\begin{array}{rcl} f(t_1^1, ..., t_1^n) & \rightarrow & e_1 \\ \vdots & \vdots & \vdots \\ f(t_k^1, ..., t_k^n) & \rightarrow & e_k \end{array}$$

Observe that any instance $f(s_1, ..., s_n)$ (of $f(x_1, ..., x_n)$) can be rewritten into only one of $e_1, ..., e_k$. Therefore f^{ee} is a disjunction of e_j^{ee}, for $j = 1, ..., k$. However, f^{ee} is a formula on $x_1, ..., x_n$, and these variables differ from those in e_j (which are actually variables in $t_j^1, ..., t_j^n$). Hence we need a function g_j that projects the demand on the variables in $t_j^1, ..., t_j^n$ into a demand on $x_1, ..., x_n$. This leads us to the following equation for defining f^{ee}.

$$[f(x_1, ..., x_n)]^{ee} = \bigvee_{j=1}^{k} g_j(e_j^{ee}) \tag{3}$$

The equations to define g_j are as follows. These equations are to be applied in the order in which they are given. In this definition, φ and ψ stand for arbitrary boolean formulae in disjunctive normal form (DNF), and φ with a subscript stands for a conjunction.

$$g_j(\psi \vee \varphi) = g_j(\psi) \vee g_j(\varphi) \tag{4}$$

$$g_j(\varphi_1) = \left(\bigwedge_{i \mid t_j^i \text{ contains no variable}} x_i \right) \wedge g_j(\varphi_1) \tag{5}$$

$$g_j(x_{i1} \wedge \cdots \wedge x_{ir_i} \wedge \varphi_1) = x_i \wedge g_j(\varphi_1) \qquad \text{if } \mathcal{V}(t_j^i) = \{x_{i1}, ..., x_{ir_i}\} \tag{6}$$

$$g_j(\varphi_1) = \mathit{true}, \text{ otherwise.} \tag{7}$$

The justification for the definition of g_j is as follows. Eq. (4) simply "pushes" the g_j operations into individual conjunctions. For Eqs. (5) through (7), note that if $f(s_1, ..., s_n)$ is rewritten by the jth equation then it must be the case that $s_i = t_j^i\beta$, for some β. If t_j^i contains variables $x_{i1}, ..., x_{ir_i}$, and all these are e-demanded to normalize e_j, then s_i must be normalized. This is because all nonvariable symbols in t_j^i are constructors, and so s_i will be in NF if each of $\beta(x_{i1}), ..., \beta(x_{ir_i})$ are in NF. Hence x_i appears in $g_j(e_j^{ee})$ if $x_{i1} \wedge \cdots \wedge x_{ir_i}$ appears in e_j^{ee} (and hence (6)). By the same argument, if t_j^i consists entirely of constructors (no variables), s_i must be normalized to match it, and hence x_i is e-demanded (and hence (5)). Otherwise we cannot conclude an e-demand on x_i, and thus it does not appear in f^{ee} (and hence (7)).

The above equations directly compute f^{ee} if f is nonrecursive. In case of recursive functions we use the following fixpoint iteration procedure. Let $f_1, ..., f_m$ denote all functions in the program. For $i = 1, ..., m$, define

$$f_i^{ee,0} = \bigwedge_{j=1}^{arity(f_i)} x_j$$

$$f_i^{ee,r+1} = \bigvee_{j=1}^{k_i} g_j\left(e_j^{ee}[f_l^{ee} \leftarrow f_l^{ee,r}]\right), \text{ for } l = 1, ..., m$$

where k_i is the number of rules defining f_i. The limit of the sequence $f^{ee,0}, f^{ee,1}, ...$ is defined to be f_i^{ee}.

We now illustrate how the above equations are used to compute strictness of $reverse$ defined below:

$$\begin{aligned} reverse(x_{11}:x_{12}) &\rightarrow append(reverse(x_{12}), x_{11}:nil) \\ reverse(nil) &\rightarrow nil \end{aligned}$$

We assume that ap (abbreviation for $append$) has already been analyzed and that $ap^{ee}(x,y) = x \wedge y$. Since rv (abbreviation for $reverse$) is recursive, we use fixpoint iteration, starting with $rv^{ee,0}(x_1) = x_1$.

$$\begin{aligned} rv^{ee,1}(x_1) &= g_1([ap(rv(x_{12}), x_{11}:nil))]^{ee}) \vee g_2(nil^{ee}) && \text{by (3)} \\ &= g_1(ap^{ee}([rv(x_{12})]^{ee}, [x_{11}:nil]^{ee}) \vee g_2(true) && \text{by subst. rule, (2)} \\ &= g_1([rv(x_{12})]^{ee} \wedge [x_{11}:nil]^{ee}) \vee g_2(true) && \text{by strictness of } ap \\ &= g_1((rv^{ee,0}([x_{12}]^{ee})) \wedge x_{11}) \vee g_2(true) && \text{by substitution rule, (2)} \\ &= g_1(x_{12} \wedge x_{11}) \vee g_2(true) && \text{by simplification} \\ &= x_1 \vee x_1 && \text{by (6) and (5)} \\ &= x_1 \end{aligned}$$

Since $rv^{ee,1} = rv^{ee,0}$, a fixpoint has been reached, showing that $rv^{ee}(x_1) = x_1$.

We skip correctness proofs here, since correctness of ee-analysis follows from the correctness of the more-general analysis presented in the next section. (Direct proofs of correctness of ee-analysis can also be found in Sekar et al. [1990].)

2.2 Underlying Intuition: Nonerasing Functions and ee-Strictness

In this section, we provide the intuition behind ee-analysis. This intuition is quite helpful in understanding the method, as well as its power. In addition, it plays an important role in the linear-time heuristic algorithm presented in the next section.

A function is nonerasing in its ith argument if it does not discard portions of this argument without even examining them. More precisely, for every reduction sequence that normalizes $f(t_1, ..., t_n)$, the ith argument t_i can be reduced to a term $t'_i\sigma_i$ such that the following holds: in this reduction sequence, every constructor symbol in t'_i participates in a match against a nonvariable symbol of an lhs, and the substitutions σ_i appear unchanged in the normal form of $f(t_1, ..., t_n)$. We refer to t'_i as a *skeleton* and σ_i as a *body* of the term t_i. The function f can be thought of as *consuming* the skeleton and *conserving* the body. For example, in the case of $append(1:1+1:3:nil, 1:2*1:nil) = 1:2:3:1:2:nil$, we have $t'_1 = x:y:z:nil$, $\sigma_1 = \{x{=}1, y{=}1+1, z{=}3\}$, $t'_2 = u$, and $\sigma_2 = \{u{=}1:2*1:nil\}$.

Suppose that a term $t = f(t_1, ..., t_n)$ is to be normalized and that we know that f is nonerasing in its ith argument. Since each of the terms in the substitution σ_i appears in the normal form of t, each of them needs to be normalized. Note that in order to evaluate f, t_i has to be evaluated at least to the extent of yielding its skeleton. When the skeleton of t_i is evaluated, and each term in its body is in normal form, t_i will be in normal form. Therefore t_i can be safely normalized before evaluating f. Thus a nonerasing function transforms an e-demand on its output to an e-demand on its arguments.

Constructors such as $cons$ serve as "tuple" operators, and their effect is just to "enclose" data items. We need to be able to utilize this information if our strictness analysis is to be effectively applied for algebraic data types. The concept of nonerasing (or *ee*-strictness) precisely captures this effect of "enclosing." Just as functions on flat domains are strict in most of their arguments, functions on nonflat domains appear to be *ee*-strict on a majority of their arguments, and the effectiveness of our methods is largely because of this fact.

2.3 Linear-Time Heuristics for *ee*-Analysis

The worst-case time complexity of the analysis method presented earlier is exponential in the number of arguments to a function [Hudak and Young 1986]. However, we can employ efficient heuristics to test if a function is *ee*-strict in all its arguments. In this section we describe such a heuristic that requires time proportional to the size of the program. In those cases when this heuristic fails, we resort to the method in Section 2.1. This heuristic also computes Mycroft's $f^{\#}$ within the same time bounds.[1] In the following, we assume that if f and h are not mutually recursive, and f calls h, then the analysis of h is done prior to that of f. We also assume that the word size (typically 32 or 64 bits) is larger than the number of variables in an lhs (typically less than 10).

Fixpoint iteration is used to compute f^{ee} for recursive functions, starting with $f^{ee,0}$ being *ee*-strict in all its arguments. If f^{ee} is indeed *ee*-strict in all its arguments, then $f^{ee} = f^{ee,1}$, and there needs to be just a single iteration. If this iteration (as well as checking that $f^{ee,1} = x_1 \wedge \cdots \wedge x_n$) can be done in time proportional to the sizes of the rhs involved, we have an algorithm whose time complexity is linear in the size of the program.

Recall $f^{ee}(x_1, ..., x_n) = \bigvee_{j=1}^{k} g_j(e_j^{ee})$, and so $f^{ee,1} = \bigwedge_{i=1}^{n} x_i$ iff $g_j(e_j^{ee}) = \bigwedge_{i=1}^{n} x_i$ for every j. This in turn implies that if $x_{11}, x_{12}, ..., x_{1r_1}, x_{21}, ...x_{2r_2}, ..., x_{n1}, ...x_{nr_n}$ are all the variables in the lhs of the jth equation defining f then e_j^{ee} must be $\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{r_i} x_{ij}$. We need to test if this is the case for each e_j in time proportional to the size of e_j. To do this, we compute an approximation to e_j^{ee} by substituting an approximation of h^{ee} for each function or constructor h that appears in e_j. This approximation, denoted by h^{ne}, is the set of arguments that are **n**ot **e**rased by h, i.e., those arguments in which h is *ee*-strict. For the occurrences of f in e_j, $f^{ne} = \{x_1, ..., x_n\}$. The approximation e_j^{ne} is computed as follows.

$$
\begin{aligned}
x^{ne} &= \{x\} \\
[c(s_1, ..., s_n)]^{ne} &= s_1^{ne} \cup \cdots \cup s_n^{ne} \\
[h(s_1, ..., s_n)]^{ne} &= \bigcup_{h \text{ is } ee \text{ strict in } x_i} s_i^{ne} \qquad \text{if } h \neq if \\
&= s_1^{ne} \cup (s_2^{ne} \cap s_3^{ne}) \qquad \text{if } h = if
\end{aligned}
$$

If the sets s_i^{ne} are represented by bitvectors then union and intersection operations will take constant time. It is immediate from the above definition that the number of such operations is bounded by the number of edges in the tree representation of e_j and thus e_j^{ne} can be computed in time proportional to the size of e_j.[2] Since checking

[1] *ee*-analysis coincides with Mycroft's analysis on flat domains. See Sekar et al. [1991] for proofs.
[2] This algorithm is linear even when a dag (directed acyclic graph) representation of e_j is considered, since, for each subterm s that is shared, we can share the computation of s^{ne}.

if $e_j^{ne} = \{x_{11}, x_{12}, ..., x_{1r_1}, x_{21}, ...x_{2r_2}, ..., x_{n1}, ...x_{nr_n}\}$ can be done in constant time with the bitvector representation, we have the desired linear algorithm.

If $f_1, ..., f_l$ are mutually recursive then they have to be analyzed simultaneously using the above procedure. If there is at least one f_i such that $f_i^{ne} \neq \{x_1, ..., x_{arity(f_i)}\}$, then we cannot conclude anything about the ee-strictness of any of $f_1, ..., f_l$. We say that the above heuristic fails in such cases.

3. DEMAND ANALYSIS

In this section we present demand analysis in its full generality. In this context, we reason about not only demand e but also about demands such as those needing the head or all of the outer structure of a list. First, we formalize the notion of a demand and its representation. (Note the similarity between our notion of a demand and the notion of a *strictness type* in Kuo and Mishra [1989].)

Definition (*Demand*). A demand D denotes any set of terms that is closed under the equality relation $=_{TRS}$ defined by the TRS, i.e.,

$$\forall t \forall s\, [(t \in D) \wedge (t \rightarrow^* s)] \Leftrightarrow (s \in D).$$

The following closure properties of demands play a role in the construction of *demand domains* that will be used for analysis.

LEMMA (CLOSURE PROPERTY OF DEMANDS). *The union, as well as the intersection, of two demands are themselves demands.*

Of the many possible demands, the following three are particularly useful:

—u-demand (standing for unknown demand), which denotes the universal set T_Σ of terms,

—d-demand (standing for head-normal-form demand), which is the smallest set containing all terms with constructor roots and is closed under $=_{TRS}$, and

—e-demand (standing for normal-form demand), which is the smallest set containing all terms that consist entirely of constructors (and thus are in normal form) and is closed under $=_{TRS}$.

Based on the notion of a demand, we now define the notion of demand constraint.

Definition (*Demand Constraint and Satisfaction*). An *atomic demand constraint* C is of the form t^D, where t is a term and D is a demand. It is said to be satisfied under a ground substitution σ (denoted $C\sigma = true$ or simply as $C\sigma$) iff $t\sigma \in D$. An arbitrary demand constraint φ is obtained by combining atomic demand constraints using conjunction and disjunction operations. The notion of satisfaction is extended to arbitrary constraints in the natural way:

$$(\varphi_1 \wedge \varphi_2)\sigma \text{ iff } \varphi_1\sigma \text{ and } \varphi_2\sigma$$
$$(\varphi_1 \vee \varphi_2)\sigma \text{ iff } \varphi_1\sigma \text{ or } \varphi_2\sigma$$

We typically omit literals of the form t^u in demand formulas, since they are satisfied by any demand substitution and hence are equivalent to the formula *true*.

Of particular interest are demand constraints over a set of variables $\mathcal{V}$, i.e., demand constraints that contain only (variable) terms drawn from $\mathcal{V}$. We now define strictness of arbitrary terms based on such constraints.

Definition (*Strictness*). Let t be any term, D a demand, φ a demand constraint on variables in t, and σ a ground substitution for variables in t. Then we say that t has φ under D, denoted $t^D \vdash \varphi$, iff the following condition holds:

$$\forall\sigma\ (t\sigma \in D) \Rightarrow \varphi\sigma$$

To illustrate the concept of strictness properties, we list some of the strictness properties of *append* and *if* below:

$$append(x_1, x_2)^e \ \vdash \ x_1^e \wedge x_2^e \tag{8}$$
$$if(x_1, x_2, x_3)^d \ \vdash \ x_1^e \wedge (x_2^d \vee x_3^d) \tag{9}$$
$$if(x_1, x_2, x_3)^u \ \vdash \ true \tag{10}$$

Having defined the notion of strictness, we now proceed to describe a method for computing strictness properties. In order to make our analysis modular and extensible, we organize it into two components. The first component, called the demand constraint component, does not go into the actual structure of the demands used in the analysis but is parameterized by three operations on demands:

—the subset operation (denoted *sub*) on demands,

—the intersection operation (denoted *int*) on demands, and

—a projection operation (denoted *Pr*) that yields the demand satisfied by a term $c(t_1, ..., t_n)$ given the demands satisfied by $t_1, ..., t_n$.

Our correctness proofs will need to make appropriate assumptions about the correctness of these three operations.

The second component of our analysis, called the domain component, will provide these operations needed by the demand constraint component. Whereas the demand constraint component is generic and is reused for all strictness analyses, the domain component differs depending upon the choice of the domain of analysis. In an implementation, the domain component itself can be structured to be data driven, so that various domain-dependent functions can be derived from a minimal amount of information that describes the structure of the domain. (We will sketch this in a later section dealing with implementations.) We now proceed to describe each of these components in detail. In order to simplify the description, for the rest of this article we will be dealing only with demands that are subsets of d. This restriction allows us to avoid certain complications that correspond to type errors or run-time pattern match exceptions but does not otherwise reduce the power of the analysis.[3]

3.1 Demand Constraint Component

The definition of strictness properties was based on the notions of substitution and satisfaction. In contrast, our computation procedure will not make use of these

[3]In *ee*-analysis, we did not impose this restriction and thus permitted normal forms that contain a function symbol at the root. In reaching such normal forms, we would encounter a pattern match failure at the root. Therefore, the normal form would have to be obtained by reducing all arguments of the offending function symbol to normal form. For this reason, the fixpoint iteration was started at $x_1 \wedge \cdots \wedge x_n$. Starting at *false*, as we would do for the general demand analysis, would correspond to ruling out pattern match failures.

concepts but will perform the analysis using symbolic manipulation of demand formulas. In particular, we treat demand constraints as syntactic entities. For the purposes of symbolic manipulation, the formulas are maintained in a disjunctive normal form (DNF) using a simplification function S.

By definition, the strictness property of a function is not unique. However, the computational procedure will compute just a single strictness property for any term t under demand D. We will overload the notation t^D in the context of the computational procedure to refer to this single property. This property corresponds to the strongest strictness property that can be inferred by the analysis method.

There are two basic steps in the computation of strictness properties. First, we need to be able to compute strictness properties of terms, based on given strictness properties of functions and constructor symbols appearing in the term. This is accomplished through the composition step. Second, we need to be able to compute strictness of a function, based on its definition. We accomplish this through the function definition step. These two steps are described below.

3.1.1 *Composition Step.* The strictness of arbitrary terms is computed using composition, based on the strictness of constructors and function symbols using the following rule and a simplification function S defined later on.

Definition (*Rule of Substitution*). Let t be a term containing variables $x_1, ..., x_n$, D a demand, σ be any substitution for $x_1, ..., x_n$, and φ be a strictness property of t. Then

$$(t\sigma)^D \vdash S((t^D)\sigma).$$

Note the close similarity between this rule and the substitution rule for *ee*-analysis. The rule states that we compute the strictness of a compound term $t\sigma$ based on the strictness t^D of its component t as follows. We first replace atomic demand constraints in t^D of the form $x^{D'}$ by the formula $(\sigma(x))^{D'}$ and then use the simplification function S to obtain a DNF. The simplification function S uses the following identities to reduce demand constraints to their DNF:

$$(\varphi \vee \psi) \wedge \varphi' = (\varphi \wedge \varphi') \vee (\psi \wedge \varphi') \qquad \text{(Distributive Law)}$$

$$\left(\bigwedge_{i=1}^{n} x_i^{D_i}\right) \vee \left(\bigwedge_{i=1}^{n} x_i^{D'_i}\right) = \bigwedge_{i=1}^{n} x_i^{D_i}, \text{ if } sub(D'_i, D_i) \qquad \text{(Absorption Law)}$$

$$\left(x^D \wedge x^{D'}\right) = x^{int(D,D')} \qquad \text{(Demand Intersection)}$$

A computational procedure for implementing S is readily derived from these identities and is not discussed in further detail here.

To illustrate the substitution rule, let us compute the strictness of the term $append(append(x_1, x_2), x_3)$ under e-demand, using the strictness of append as given by Eq. (8). Using the substitution rule we get the following:

$$\begin{aligned}
append(append(x_1, x_2), x_3)^e &\vdash (append(x_4, x_3)[x_4 \leftarrow append(x_1, x_2)])^e \\
&\vdash S(append(x_4, x_3)^e[x_4 \leftarrow append(x_1, x_2)]) \\
&\vdash S((x_4^e \wedge x_3^e)[x_4 \leftarrow append(x_1, x_2)]) \\
&\vdash S((append(x_1, x_2)^e \wedge x_3^e)) \\
&\vdash x_1^e \wedge x_2^e \wedge x_3^e
\end{aligned}$$

3.1.2 *Function Definition Step.* Now we proceed to describe the method for computing strictness of functions from their definitions. Let f be defined by the following rules:

$$\begin{array}{rcl} f(t_1^1, ..., t_1^n) & \rightarrow & e_1 \\ \vdots & \vdots & \vdots \\ f(t_k^1, ..., t_k^n) & \rightarrow & e_k \end{array}$$

Then we define the strictness of f based on strictness of $e_1, ..., e_k$ as follows. If $D = u$, then we can immediately see that $f(x_1, ..., x_n)^D = true$. Otherwise, we have:

$$f(x_1, ..., x_n)^D \vdash S\left(\bigvee_{j=1}^{k} Proj\ (e_j^D, \{t_j^1, ..., t_j^n\})\right) \tag{11}$$

This definition makes use of the simplification function defined before and a projection function $Proj$. The latter function is used to transform a demand on the variables in an lhs pattern $f(t_j^1, ..., t_j^n)$ into a demand on the arguments $x_1, ..., x_n$ of f. It takes two arguments: the demand constraint e_j^D that needs to be projected and a list specifying the substitutions $t_1, ..., t_n$ for the arguments $x_1, ..., x_n$. This projection function is defined as follows:

$$Proj\left(\bigvee_{i=1}^{r} \psi_i, \{t_1, ..., t_n\}\right) \vdash S\left(\bigvee_{i=1}^{r} \bigwedge_{j=1}^{n} x_j^{getDem(\psi_i, t_j)}\right) \tag{12}$$

The projection operation is first "pushed" into individual conjunctions. Within a conjunction, it operates by computing the demand D_j on each argument position x_j using an auxiliary function $getDem$ and then takes the conjunction over $j = 1, ..., n$. The $getDem$ function itself takes two arguments: the second argument is a term s that consists only of variables and constructor symbols, and the first argument is a demand conjunct ψ that specifies the demand on variables in s. Based on the demands specified by ψ, $getDem$ computes the demand D' that is satisfied at the root of s. This computation is in turn based on a domain operation Pr, which, for each constructor symbol c, specifies the demand satisfied by $c(x_1, ..., x_n)$, given the demand on the arguments $x_1, ..., x_n$. Based on the Pr function, $getDem$ computes the demand D' as follows:

$$getDem(\psi, x) = \begin{cases} D', & \text{if } x^{D'} \text{ appears in } \psi \\ u, & \text{otherwise.} \end{cases} \tag{13}$$

$$getDem(\psi, c(t_1, ..., t_n)) = Pr(c(getDem(\psi, t_1), getDem(\psi, t_2), ..., getDem(\psi, t_n))) \tag{14}$$

This completes the function definition step.

For recursive functions, the definition of f^D is itself recursive. We can use the following fixpoint iteration procedure to compute solutions of such equations. To describe this procedure in more detail, let $f_1, ..., f_m$ denote all the functions in the program. Then f_i^D is the limit of the sequence $f_i^{D,0}, f_i^{D,1}, ...$ defined as follows:

$$f_i^{D,0} \vdash false$$

$$f_i^{D,r+1} \vdash S\left(\bigvee_{j=1}^{k} Proj\left(e_j^D[f_l^{D'} \leftarrow f_l^{D',r}], \{t_j^1, ..., t_j^n\}\right)\right), \text{ for any } D' \text{ and } l$$

3.1.3 *An Illustrative Example.* We now illustrate how the substitution and function definition steps are used to compute strictness. For this purpose, we make use of the following example function,

$$f(x, \langle y, z\rangle) \rightarrow if\ x\ then\ y + z\ else\ y - z,$$

under d-demand. We assume strictness of if as before (see Eq. (9)), strictness of $+$ and $-$ under d-demand as $x_1^e \wedge x_2^e$, and that $Pr(\langle e, e\rangle) = e$. By Eq. (11), we get (ignoring the outer disjunction, since there is only one rule defining f):

$$\begin{array}{lr}
f(x_1, x_2)^d \vdash & \\
\quad \vdash Proj((if\ x\ then\ y + z\ else\ y - z)^d, \{x, \langle y, z\rangle\}) & \text{by Eq. (11)} \\
\quad \vdash Proj(S(x^e \wedge ((y + z)^d \vee (y - z)^d)), \{x, \langle y, z\rangle\}) & \text{by (9), composition} \\
\quad \vdash Proj(x^e \wedge y^e \wedge z^e, \{x, \langle y, z\rangle\}) & \text{by strictness of } +, -, \text{ simplification} \\
\quad \vdash S(x_1^{getDem(x^e \wedge y^e \wedge z^e, x)} \wedge x_2^{getDem(x^e \wedge y^e \wedge z^e, \langle y, z\rangle)}) & \text{by Eq. (12)} \\
\quad \vdash S(x_1^e \wedge x_2^{Pr(\langle getDem(x^e \wedge y^e \wedge z^e, y), getDem(x^e \wedge y^e \wedge z^e, z)\rangle)}) & \text{by (14) and (13)} \\
\quad \vdash S(x_1^e \wedge x_2^{Pr(\langle e, e\rangle)}) & \text{by Eq. (13)} \\
\quad \vdash x_1^e \wedge x_2^e & \text{by defn of } Pr \text{ for } \langle\rangle
\end{array}$$

We now proceed to establish the correctness of the demand constraint component.

3.2 Proof of Soundness

Soundness of the analysis is based on the soundness of the substitution and function definition steps. These proofs rely on the correctness of the functions provided by the domain component:

Requirement 1. The sub, int, and Pr operations are sound. Specifically,

—for any two demands D and D', $sub(D, D') \Rightarrow D \subseteq D'$,
—for any two demands D and D', $int(D, D') \supseteq D \cap D'$, and
—for any $t_1, ..., t_n$ and $D_1, ..., D_n$ such that $t_i \in D_i$, $c(t_1, ..., t_n) \in Pr(c(D_1, ..., D_n))$.

Also, the strictness of constructors and predefined functions is sound.

With this requirement, we now proceed to establish the soundness of our analysis.

3.2.1 *Soundness of Composition Step*

LEMMA 3.2.1.1. *The simplification function S is sound, i.e., $\varphi \Rightarrow S(\varphi)$.*

PROOF. The soundness of distributive law defining S is obvious. The correctness of absorption and demand intersection rules follow readily from the soundness of sub and int, hence this lemma. $\square$

Based on the soundness of S, we can prove the soundness of composition:

THEOREM 3.2.1.2. *The rule of substitution is sound.*

PROOF. Let φ be of the form $\bigvee_{i=1}^{k} \bigwedge_{j=1}^{n} x_j^{D_{ij}}$. Also let ψ_{ij} be a strictness property of $\sigma(x_j)$ under demand D_{ij}, i.e., for $1 \leq i \leq k, 1 \leq j \leq n$, $(\sigma(x_j))^{D_{ij}} \vdash \psi_{ij}$. Then we need to show that $(t\sigma)^D \vdash \varphi'$, where

$$\varphi' = S(t^D\sigma) = S\left(\bigvee_{i=1}^{k} \bigwedge_{j=1}^{n} \psi_{ij}\right).$$

To prove this, by definition of strictness, we need to show that for any ground substitution β such that $t\sigma\beta \in D$, β satisfies φ'. Consider the ground substitution $\gamma = \sigma\beta$. Since $t^D \vdash \varphi$ and $t\gamma \in D$, by definition of strictness, we have that γ satisfies φ. By the structure of φ, this means that there must exist some i such that for $j = 1, ..., n$, $[\gamma(x_j) = (\sigma(x_j))\beta] \in D_{ij}$. This fact, in conjunction with the fact that $(\sigma(x_j))^{D_{ij}} \vdash \psi_{ij}$ and the definition of strictness, yields that ψ_{ij} is satisfied by β. This means that the ith conjunct $\bigwedge_{j=1}^{n} \psi_{ij}$ is satisfied by β. Given soundness of S, this implies that β satisfies φ'. $\square$

3.2.2 *Soundness of Function Definition Step.* To establish the soundness of this step, we show that Eq. (11) is sound for computing $f(x_1, ..., x_n)^D$. First, we show:

LEMMA 3.2.2.1. *Suppose that e_j^D is a strictness property of e_j under demand D. Then e_j^D is a strictness property of the corresponding lhs $l_j \equiv f(t_j^1, ..., t_j^n)$ under demand D, i.e., $\forall\sigma\ l_j\sigma \in D \Rightarrow e_j^D\sigma$.*

PROOF. We need to show that for any ground instance $l_j\sigma$, whenever $l_j\sigma \in D$, $e_j^D\sigma$ is true. Since $l_j\sigma$ can be rewritten into $e_j\sigma$, by definition of a demand, we have $(l_j\sigma \in D) \Rightarrow (e_j\sigma \in D)$. Moreover, since e_j^D is a strictness property of e_j, we also have $(e_j\sigma \in D) \Rightarrow e_j^D\sigma$, thus completing the proof. $\square$

We now proceed to show that $Proj$ is sound. We start by establishing soundness of $getDem$. Informally, this means that given the demand constraint ψ satisfied by the variables in a term t, $getDem(\psi, t)$ is a demand satisfied by t. More formally:

LEMMA 3.2.2.2. *Let ψ be a conjunction of atomic demand constraints, σ any ground substitution for variables in ψ, and t a term consisting of a subset of these variables. Then $\psi\sigma \Rightarrow (t\sigma \in getDem(\psi, t))$.*

PROOF. We establish this lemma by induction on height of t. For the base case, $getDem$ is defined by Eq. (13), whose soundness follows immediately from the definition of demand satisfaction. For the induction step, let $t = c(t_1, ..., t_n)$, and assume that for $1 \leq i \leq n$, $getDem(\psi, t_i) = D_i$ is satisfied by t_i. This means that

$$\forall\sigma\ \psi\sigma \Rightarrow [(t_1\sigma \in D_1) \wedge \cdots \wedge (t_n\sigma \in D_n)].$$

Putting this together with the condition that Pr is sound, we get

$$c(t_1, ..., t_n)\sigma = t\sigma \in getDem(\psi, t) = Pr(c(D_1, ..., D_n))],$$

thus completing the induction step. $\square$

With this lemma, we are ready to establish soundness of $Proj$:

LEMMA 3.2.2.3. *Let $x_1, ..., x_n$ be variables and σ be a substitution that substitutes $t_1, ..., t_n$ for these variables. Also let φ be a demand constraint on variables in $t_1, ..., t_n$ and β be any ground substitution for these variables. Then*

$$\varphi\beta \Rightarrow Proj(\varphi, \{t_1, ..., t_n\})\sigma\beta.$$

PROOF. Let φ be of the form $\bigvee_{i=1}^{r} \psi_i$. Based on the definition of $Proj$ from Eq. (12) and the fact that $\psi \Rightarrow S(\psi)$, it suffices to show that

$$\varphi\beta \Rightarrow \left(\bigvee_{i=1}^{r} \bigwedge_{j=1}^{n} \left[x_j^{getDem(\psi_i, t_j)} \right] \right) \sigma\beta.$$

Consider any β that satisfies the lhs of the above implication. By structure of φ, such a β must satisfy some ψ_i. By soundness of $getDem$, we have $\bigwedge_{j=1}^{n}[t_j\beta \in getDem(\psi_i, t_j)]$. Rewriting this in terms of σ, we have $\bigwedge_{j=1}^{n}[x_j\sigma\beta \in getDem(\psi_i, t_j)]$, which in turn is equivalent to asserting $\bigwedge_{j=1}^{n}[x_j^{getDem(\psi_i, t_j)}]\sigma\beta$. Taking disjunction over different possible values of i, we get the consequent of the above implication. □

THEOREM 3.2.2.4. *The function definition step in sound. Specifically, if e_j^D is a strictness property of e_j under D then $S(\bigvee_{j=1}^{k} Proj(e_j^D, \{t_j^1, ..., t_j^n\}))$ is a strictness property of $f(x_1, ..., x_n)$ under D.*

PROOF. We consider all ground substitutions γ of $f(x_1, ..., x_n)$ in $k+1$ cases and prove the soundness of the derived strictness property in each case. The jth case, for $1 \leq j \leq k$, corresponds to γ being compatible with the jth lhs, and the $k+1$th case corresponds to when γ is incompatible with any of these lhs. For the jth case, $\gamma = \sigma\beta$ such that $f(x_1, ..., x_n)\sigma = l_j$. Then, from Lemma 3.2.2.1, we get

$$[f(x_1, ..., x_n)\gamma \in D] \Rightarrow e_j^D\beta$$

which, in conjunction with Lemma 3.2.2.3, yields that

$$[f(x_1, ..., x_n)\gamma \in D] \Rightarrow Proj\left(e_j^D, \{t_j^1, ..., t_j^n\}\right)\gamma \Rightarrow \bigvee_{m=1}^{k} Proj\left(e_m^D, \{t_m^1, ..., t_m^n\}\right)\gamma$$

thus establishing soundness in the first k cases. For the $(k+1)$th case, $f(x_1, ..., x_n)\gamma$ may or may not be reducible to another term $l_j\delta$ that is an instance of some lhs. In the former case, soundness follows from the facts that (a) the strictness property is sound for $l_j\delta$ and (b) $f(x_1, ..., x_n)\gamma \in D \Rightarrow l_j\delta \in D$ (by definition of demand). In the latter case, $f(x_1, ..., x_n)\gamma$ cannot be reduced to a term with a constructor root, and hence this term does not belong to d. Since we have made the assumption that all demands D (other than u) are subsets of d, $f(x_1, ..., x_n)^D\gamma$ is $false$ for this case, and hence the required implication holds for this case as well. □

3.2.3 *Soundness of Fixpoint Iteration.* To prove soundness of fixpoint iteration, we use the standard technique of constructing successive approximations to the functions defined in the program, and show that the result of the ith fixpoint iteration is a sound strictness property for the ith approximation of the function. Specifically, for each function $f_1, ..., f_m$ in the program, we define their zeroth approximation as the nonterminating computation described by

$$f_i^0(x_1, ..., x_k) \rightarrow f_i^0(x_1, ..., x_k).$$

The rth approximation f_i^r is obtained by starting with the equations defining f_i, replacing f_i in the lhs by f_i^r and the occurrences of $f_1, ..., f_m$ in the rhs by $f_1^{r-1}, ..., f_m^{r-1}$, respectively. Observe that as r tends to infinity, f_i^r computes the same function as f_i. With this, we can prove:

LEMMA 3.2.3.1. *For $1 \leq i \leq m$, $f_i^{D,r}$ (the result of rth fixpoint iteration) is a sound strictness property of f_i^r under demand D.*

PROOF. Proof is by induction on r. For the base case of $r = 0$, the function f_i^0 does not reduce to any term with constructor root. So, for any $D \subseteq d$, $f_i^{0,D}$ is $false$, and hence the soundness in base case. For the induction step, assuming the soundness of $f_1^{r-1}, ..., f_m^{r-1}$, the soundness of f_i^r follows immediately from the soundness of the substitution and function definition steps. □

COROLLARY 3.2.3.2. *The limit of the chain $f_i^{D,0}, f_i^{D,1}, ...$, if it exists, is a sound strictness property of f.*

The existence and computability of this limit is established in the following section.

3.3 Termination Proof

The termination of our analysis is proved by showing that the sequence $f_i^{D,0}$, $f_i^{D,1}, ...$ has a finite limit and that we can effectively recognize this limit. The former is proved by showing that the definition of strictness properties is monotonic in some order $\sqsupseteq$ such that there is no infinite ascending chain of demand constraints that respect this order. The latter is proved by showing the demand constraints that are equivalent under the order $\sqsupseteq$ have identical canonical representations under S.

Typically, the order $\sqsupseteq$ coincides with the implication order. While we could do the same under certain restrictions on the domain of demands, such a choice would complicate our proofs. Hence, we choose the order $\sqsupseteq$ defined below. This order is *consistent* with the implication ordering but is "weaker" in the sense that it may order (i.e., consider to be distinct) certain formulas that are in fact logically equivalent. On conjunctions, $\sqsupseteq$ coincides with the definition of absorption law, i.e.,

$$\bigwedge_{i=1}^{n} x_i^{D_i} \sqsupseteq \bigwedge_{i=1}^{n} x_i^{D_i'} \text{ iff } sub(D_i, D_i'), \text{ for } 1 \leq i \leq n.$$

For disjunctions, let φ be of the form $\bigvee_{i=1}^{n} \psi_i$ and φ' of the form $\bigvee_{j=1}^{m} \psi_j'$. Then

$$\varphi \sqsupseteq \varphi' \text{ iff } \forall i \, \exists j \;\; \psi_i \sqsupseteq \psi_j'.$$

Since $\sqsupseteq$ is based on the absorption law, it follows that two demand formulas are equivalent under $\sqsupseteq$ iff they possess identical simplified representations (under S). Moreover, S is monotonic with respect to $\sqsupseteq$. Also note that for a finite demand domain, there are only finitely many conjunctions in a $\sqsupseteq$ chain, and consequently, only finitely many demand formulas in any ascending chain. Thus, we need only establish the monotonicity under $\sqsupseteq$ to prove the termination of the analysis.

3.3.1 *Proof of Monotonicity.* Note that our computation of strictness property makes use of projection, composition, conjunction, and disjunction. We first establish the monotonicity of conjunctions and disjunctions based on the following requirement on the domain functions:

Requirement 2. The sub, int, and Pr operations are monotonic with respect to the order given by sub. Specifically:

—for any demands D_1, D_2, D_3 $sub(D_1, D_2) \wedge sub(D_2, D_3) \Rightarrow sub(D_1, D_3)$,
—for any demands D_1, D_2, D_1', D_2'
$sub(D_1, D_1') \wedge sub(D_2, D_2') \Rightarrow sub(int(D_1, D_2), int(D_1', D_2'))$, and
—for any $D_1, ..., D_n$ and $D_1', ..., D_n'$
$\bigwedge_{i=1}^{n} sub(D_i, D_i') \Rightarrow sub(Pr(c(D_1, ..., D_n)), Pr(c(D_1', ..., D_n')))$.

Based on these conditions, we can show the following:

LEMMA 3.3.1.1. *Conjunction and disjunction are monotonic, i.e., for all demand constraints* $\varphi_1, \varphi_2, \varphi_1', \varphi_2'$,

$$(\varphi_1 \sqsupseteq \varphi_1') \wedge (\varphi_2 \sqsupseteq \varphi_2') \Rightarrow (\varphi_1 \vee \varphi_2) \sqsupseteq (\varphi_1' \vee \varphi_2')$$

$$(\varphi_1 \sqsupseteq \varphi_1') \wedge (\varphi_2 \sqsupseteq \varphi_2') \Rightarrow (\varphi_1 \wedge \varphi_2) \sqsupseteq (\varphi_1' \wedge \varphi_2').$$

The proof is straightforward, based on the three laws defining S and based on the above requirements on monotonicity of domain operations.

Given that conjunction, disjunction and simplification are all monotonic, the monotonicity of the function definition step will follow, if we can establish that $Proj$ and the rule of substitution are monotonic. We proceed to do this below.

LEMMA 3.3.1.2. *$Proj$ function is monotonic.*

PROOF. Since Pr is monotonic and $getDem$ is defined by repeated compositions of Pr, it is easy to see that $getDem$ is monotonic in the order given by sub. Based on the definition of $\sqsupseteq$ in terms of sub, this means that

$$(\psi \sqsupseteq \psi') \Rightarrow sub(getDem(\psi, t), getDem(\psi', t)).$$

From this implication, based on the definition of $\sqsupseteq$, we can conclude that

$$(\psi \sqsupseteq \psi') \Rightarrow \left(\bigwedge_{j=1}^{n} x_j^{getDem(\psi, t_j)} \sqsupseteq \bigwedge_{j=1}^{n} x_j^{getDem(\psi', t_j)} \right). \tag{15}$$

To extend this to the case of DNFs, let φ and φ' be of the forms $\bigvee_{i=1}^{r} \psi_i$ and $\bigvee_{k=1}^{s} \psi_k'$, respectively. Now, from the definition of $\sqsupseteq$ for DNFs we have

$$(\varphi \sqsupseteq \varphi') \Rightarrow \forall 1 \le i \le r \ \exists 1 \le k \le s \ (\psi_i \sqsupseteq \psi_k').$$

Substituting the consequence of (15) into this, we get,

$$(\varphi \sqsupseteq \varphi') \Rightarrow \forall 1 \le i \le r \ \exists 1 \le k \le s \ \left(\bigwedge_{j=1}^{n} x_j^{getDem(\psi_i, t_j)} \sqsupseteq \bigwedge_{j=1}^{n} x_j^{getDem(\psi_k', t_j)} \right).$$

By definition of $Proj$ and $\sqsupseteq$, the consequence of this implication is equivalent to $Proj(\varphi) \sqsupseteq Proj(\varphi')$, thus yielding the desired monotonicity property for $Proj$. ☐

LEMMA 3.3.1.3. *The substitution step is monotonic.*

Proof. In order to prove this lemma, we (1) need to isolate the computations of the demand constraints on t and the terms in the substitution and (2) formulate the monotonicity property by viewing the substitution rule as a function over demand formulas. To do this, let

$$\left(\varphi \equiv \bigvee_{j=1}^{r} \bigwedge_{i=1}^{n} x_i^{D_{ij}}\right) \sqsupseteq \left(\varphi' \equiv \bigvee_{k=1}^{s} \bigwedge_{i=1}^{n} x_i^{D'_{ik}}\right).$$

Also let σ be a *demand substitution* that substitutes a demand formula σ_{iD} in place of each atomic demand constraint x_i^D in φ. Similarly, let β be another demand substitution that substitutes demand formula β_{iD} for atomic demand constraint x_i^D in φ'. In addition let $\sigma_{iD} \sqsupseteq \beta_{iD}$ for all i and D. Furthermore assume that these substitutions are monotonic with respect to the demand, i.e., $\sigma_{iD} \sqsupseteq \sigma_{iD'}$ and $\beta_{iD} \sqsupseteq \beta_{iD'}$ whenever $sub(D, D')$. To establish the monotonicity of substitution, we need to show that $\varphi\sigma \sqsupseteq \varphi'\beta$.

By definition of $\sqsupseteq$, $\varphi \sqsupseteq \varphi'$ means that

$$\forall 1 \le j \le r \; \exists 1 \le k \le s \; \bigwedge_{i=1}^{n} sub\,(D_{ij}, D'_{ik}). \tag{16}$$

Given $\sigma_{iD_{ij}} \sqsupseteq \beta_{iD_{ij}}$ for all i and j, from the monotonicity of σ and β with respect to demands, the condition within the parenthesis above implies that for $1 \le i \le n$, $\sigma_{iD_{ij}} \sqsupseteq \sigma_{iD'_{ik}} \sqsupseteq \beta_{iD'_{ik}}$). Since conjunction has been shown to be monotonic, we can interchange the nesting of conjunction and $\sqsupseteq$ in this formula. Thus, the formula (16) reduces to

$$\forall 1 \le j \le r \; \exists 1 \le k \le s \; \left(\bigwedge_{i=1}^{n} \sigma_{iD_{ij}} \sqsupseteq \bigwedge_{i=1}^{n} \beta_{iD'_{ik}}\right).$$

By definition of $\sqsupseteq$, this condition is equivalent to

$$\bigvee_{j=1}^{r} \bigwedge_{i=1}^{n} \sigma_{iD_{ij}} \sqsupseteq \bigvee_{k=1}^{s} \bigwedge_{i=1}^{n} \beta_{iD'_{ik}}.$$

Noting that the antecedent of this formula is $\varphi\sigma$ and the consequence is $\varphi'\beta$, we have established the desired monotonicity property. □

3.4 An Example Domain

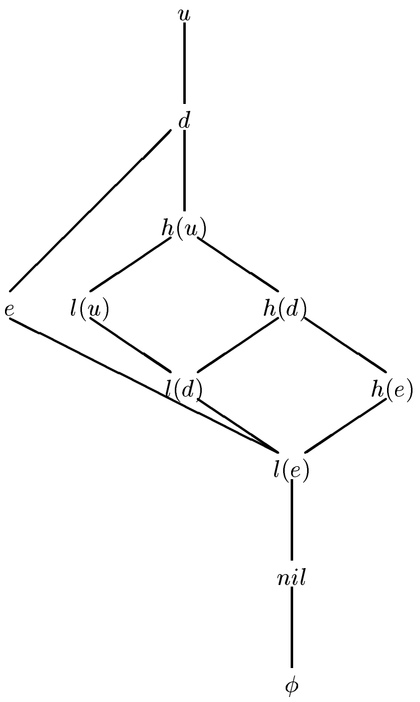
In this section, we illustrate our analysis by selecting the domain of demands and specifying the sub, int, and Pr functions needed by the demand constraint component. In general, a domain consists of *demand terms* built up from

—*demand constants*, e.g., the demands d, e and u,

—*demand variables* of the form X, Y, Z, and

—*demand constructors* that construct complex demands in terms of demand constants and variables.

Some examples of demand constructors include the *list* and *head* constructors with the following semantics. For any demand D, $list(D)$ denotes a demand that each



$$
\begin{aligned}
t^{\phi} &= \mathit{false} \\
(x\!:\!xs)^{d} &= \mathit{true} \\
(x\!:\!xs)^{head(t_D)} &= x^{t_D} \\
(x\!:\!xs)^{list(t_D)} &= x^{t_D} \wedge xs^{list(t_D)} \\
(x\!:\!xs)^{e} &= x^{e} \wedge xs^{e} \\
(x\!:\!xs)^{nil} &= \mathit{false} \\
nil^{t_D} &= \mathit{true}
\end{aligned}
$$

Fig. 2. An example demand domain, together with strictness of base functions/constructors.

element in a list be evaluated to the extent specified by D. Thus, the demand $list(u)$ denotes tail strictness, whereas $list(e)$ denotes that the list be completely evaluated. The demand constructor *head* is used to capture head-strictness, defined as follows. For any demand D, $head(D)$ denotes a demand that requires the argument to be either an empty list, or if the list is nonempty, it requires the first element to be evaluated to the extent demanded by D. Note that our definition of head-strictness coincides with the notion of "simple head-strictness" defined by Burn [1990a] but differs from the definition of Hughes and Wadler [1987]. We discuss the implications of the difference between these two notions in Section 6.

Unrestricted application of the demand constructors can lead to infinite domains. In many cases, this may not pose a problem since there may be no infinite ascending chains. However, for simplicity, in this section we will confine ourselves to a finite domain by restricting the demand constructors to at most one application in any demand term. Thus, our domain (for purposes of the illustration in this section) consists of the following demands (see Figure 2):

—demand constants d, e, u, nil, and ϕ,

—demand variables, and

—demand constructors *list* and *head*, with the restriction that the demand constructors cannot be applied more than once.

$$
\begin{array}{lrcl}
1. & sub(t_D, t_D) & \rightarrow & true \\
2a. & sub(list(t_{D_1}), d) & \rightarrow & true \\
2b. & sub(list(t_{D_1}), list(t_{D_2})) & \rightarrow & sub(t_{D_1}, t_{D_2}) \\
2c. & sub(list(t_{D_1}), head(t_{D_2})) & \rightarrow & sub(t_{D_1}, t_{D_2}) \\
3a. & sub(head(t_{D_1}), d) & \rightarrow & true \\
3b. & sub(head(t_{D_1}), head(t_{D_2})) & \rightarrow & sub(t_{D_1}, t_{D_2}) \\
4a. & sub(nil, head(t_D)) & \rightarrow & true \\
4b. & sub(nil, list(t_D)) & \rightarrow & true
\end{array}
$$

Fig. 3. Rules defining the *sub* operation.

$$
\begin{array}{lrcl}
1. & int(t_{D_1}, t_{D_2}) & \rightarrow & t_{D_1}, \text{ if } sub(t_{D_1}, t_{D_2}) \\
2. & int(X, t_D) & \rightarrow & t_D \\
3a. & int(list(t_{D_1}), list(t_{D_2})) & \rightarrow & list(int(t_{D_1}, t_{D_2})) \\
3b. & int(list(t_{D_1}), e) & \rightarrow & list(int(t_{D_1}, e)) \\
4a. & int(head(t_{D_1}), head(t_{D_2})) & \rightarrow & head(int(t_{D_1}, t_{D_2})) \\
4b. & int(head(t_{D_1}), e) & \rightarrow & e
\end{array}
$$

Fig. 4. Rules defining the *int* operation.

The demand ϕ denotes the empty set of terms. The demand nil consists of a single term, namely the empty list. In practice, it is very useful to distinguish between the empty list and nonempty list, since several multiargument functions are nonstrict when one of its list arguments is nil. This fact can be captured precisely when the distinction between empty and nonempty list is made. Otherwise, we will end up concluding that the function is always nonstrict in these arguments.

The structure of the domain is shown in Figure 2, where a line between two demands indicates that a demand D_1 that is textually above another demand D_2 is a superset of D_2. We now specify the strictness of base functions and constructors. In our language, the base functions consist mainly of arithmetic operations and are characterized by the strictness property that e-demands all arguments whenever any demand is placed on the output of the operation. (Other functions such as if can be defined in the language. Analysis of this definition would yield the property $if^X = x_1^e \wedge (x_2^X \vee x_3^X)$.) For constructors, the strictness of the two list constructors (namely, : and nil) are shown in Figure 2. In these rules, X, Y stand for demand variables, t_D for arbitrary demand terms, and D for any demand shown in Figure 2. The rules are to be applied in textual order, i.e., a rule lower down in the text is applied only when none of the rules above it are applicable.

The domain structure yields the sub and int functions for demands that contain no variables. When they do contain variables, we specify these relations using the equations in Figures 3 and 4. We remark that for the sub relation, if none of the rules is applicable, then the relation does not hold. Also note that int is commutative.

The rules for the Pr operation are shown in Figure 5. Since the definition of Pr uses the $union$ operation, the latter operation is defined in Figure 6. Note again that for base demands, the union operation is given by the domain structure in Figure 2, and also that the $union$ operation is commutative.

$$
\begin{array}{lrcl}
1. & Pr(c(...,\phi,...)) & \rightarrow & \phi \\
2. & Pr(nil) & \rightarrow & nil \\
3a. & Pr(t_D : nil) & \rightarrow & list(t_D) \\
3b. & Pr(t_{D_1} : list(t_{D_2})) & \rightarrow & list(union(t_{D_1}, t_{D_2})) \\
3c. & Pr(t_D : u)) & \rightarrow & head(t_D) \\
4. & Pr(c(e,...,e)) & \rightarrow & e \\
5. & Pr(c(u,...,u)) & \rightarrow & d
\end{array}
$$

Fig. 5. Rules defining the Pr operation.

$$
\begin{array}{lrcl}
1. & union(t_{D_1}, t_{D_2}) & \rightarrow & t_{D_1},\ \text{if } sub(t_{D_2}, t_{D_1}) \\
2. & union(X, t_D) & \rightarrow & u \\
3a. & union(list(t_{D_1}), list(t_{D_2})) & \rightarrow & list(union(t_{D_1}, t_{D_2})) \\
3b. & union(list(t_{D_1}), e) & \rightarrow & list(union(t_{D_1}, e)) \\
4a. & union(head(t_{D_1}), head(t_{D_2})) & \rightarrow & head(union(t_{D_1}, t_{D_2})) \\
4b. & union(head(t_{D_1}), e) & \rightarrow & head(t_{D_1})
\end{array}
$$

Fig. 6. Rules defining the $union$ operation.

We need to establish the soundness and monotonicity of the various domain operations given in Requirements 1 and 2. These proofs follow immediate from the equations defining sub, int, Pr, and $union$ in Figures 3 through 6.

We remark that the notion of demand variables was introduced in the domain component but did not affect the demand constraint module at all. This was because the demand constraint component does not depend on the structure of the demand but only requires that various domain operations be sound and monotonic. Another way to understand this is as follows. Both the demand constraint component as well as the domain component treat demand variables like an unknown domain constant. Since the proofs hold without making any assumptions on such constants, they hold for all possible demand constants.

3.5 Examples

In this section, we present some examples of the strictness properties derived by our method in order to give a better idea of its power. We consider the following examples, many of which are taken from Hughes and Wadler [1987].

$$
\begin{array}{rcl}
append(x : xs, y) & \rightarrow & x : append(xs, y) \\
append(nil, y) & \rightarrow & y \\
length(x : xs) & \rightarrow & 1 + length(xs) \\
length(nil) & \rightarrow & 0 \\
before(x : xs) & \rightarrow & if\ (x = 0)\ then\ nil\ else\ x : before(xs) \\
before(nil) & \rightarrow & nil \\
doubles(x : xs) & \rightarrow & 2 * x : doubles(xs) \\
doubles(nil) & \rightarrow & nil \\
reverse(x : xs) & \rightarrow & append(reverse(xs), x : nil) \\
reverse(nil) & \rightarrow & nil
\end{array}
$$

First, we analyze the function *append* under demand $list(X)$, where X is a demand variable. Below, we use the abbreviation *app* for *append* and l for $list$.

$$
\begin{array}{lll}
app(x_1, x_2)^{l(X),1} \vdash & & \\
& \vdash\ Proj((x\!:\!app(xs,y))^{l(X)}, \{x\!:\!xs, y\}) \vee Proj(y^{l(X)}, \{nil, y\}) & \text{by (11)} \\
& \vdash\ Proj(y^{l(X)}, \{nil, y\}) & \text{since } app^{l(X),0} \vdash false \\
& \vdash\ (x_1^{getDem(y^{l(X)}, nil)} \wedge x_2^{getDem(y^{l(X)}, y)}) & \text{by (12)} \\
& \vdash\ x_1^{Pr(nil)} \wedge x_2^{l(X)} & \text{by Eqs. (14) and (13)} \\
& \vdash\ x_1^{nil} \wedge x_2^{l(X)} &
\end{array}
$$

$$
\begin{array}{lll}
app(x_1, x_2)^{l(X),2} \vdash & & \\
& \vdash\ Proj((x\!:\!app(xs,y))^{l(X)}, \{x\!:\!xs, y\}) \vee Proj(y^{l(X)}, \{nil, y\}) & \text{by (11)} \\
& \vdash\ [Proj(x^X \wedge app(xs,y)^{l(X)}, \{x\!:\!xs, y\})] \vee [x_1^{nil} \wedge x_2^{l(X)}] & \text{by strictness of ":"} \\
& \vdash\ [Proj(x^X \wedge xs^{nil} \wedge y^{l(X)}, \{x\!:\!xs, y\})] \vee [x_1^{nil} \wedge x_2^{l(X)}] & \text{from prev. iteration} \\
& \vdash\ [x_1^{getDem(x^X \wedge xs^{nil} \wedge y^{l(X)}, x\!:\!xs)} \wedge x_2^{getDem(x^X \wedge xs^{nil} \wedge y^{l(X)}, y)}] & \\
& \qquad \vee [x_1^{nil} \wedge x_2^{l(X)}] & \text{by (12)} \\
& \vdash\ [x_1^{Pr(X:nil)} \wedge x_2^{l(X)}] \vee [x_1^{nil} \wedge x_2^{l(X)}] & \text{by Eqs. (13) and (14)} \\
& \vdash\ [x_1^{l(X)} \wedge x_2^{l(X)}] \vee [x_1^{nil} \wedge x_2^{l(X)}] & \text{by definition of } Pr \\
& \vdash\ x_1^{l(X)} \wedge x_2^{l(X)} & \text{by absorption law}
\end{array}
$$

Another iteration yields the same result; thus we reach the fixpoint of $x_1^{l(X)} \wedge x_2^{l(X)}$.

In a similar manner, we can analyze the other examples given above under various demands. Selected results of these analyses are shown below:

$$
\begin{array}{rcl}
append^{list(X)} & \vdash & x_1^{list(X)} \wedge x_2^{list(X)} \\
append^{head(X)} & \vdash & x_1^{head(X)} \\
append^{e} & \vdash & x_1^{e} \wedge x_2^{e} \\
append^{d} & \vdash & x_1^{d} \\
length^{d} & \vdash & x_1^{list(u)} \\
before^{d} & \vdash & x_1^{head(e)} \\
doubles^{list(u)} & \vdash & x_1^{list(u)} \\
doubles^{list(d)} & \vdash & x_1^{list(e)} \\
doubles^{head(d)} & \vdash & x_1^{head(e)} \\
reverse^{d} & \vdash & x_1^{list(u)} \\
reverse^{list(X)} & \vdash & x_1^{list(X)} \\
reverse^{head(X)} & \vdash & x_1^{list(u)}
\end{array}
$$

The properties derived are similar to those derived in Hughes and Wadler [1987], and the accuracies differ primarily because of the difference in strictness properties expressible in the two works. For instance, the notion of head-strictness in our work and in Hughes and Wadler [1987] are related but not the same.

The strictness of *length* is the same under all demands $D \subseteq d$ and is given by $x_1^{list(u)}$ as shown above. The function $before$ also has the same strictness for all $D \subseteq d$ and requires the first element in its argument to be evaluated. *Doubles* is tail-strict in a tail-strict context and is head- and tail-strict in a head- and tail-strict context. We can also derive that it is head-strict in a head-strict context. The function *reverse* behaves differently from others, in that it propagates *list* demands onto its argument unchanged but does not propagate *head* demand. In reality, the $head(X)$ demand on its output gets translated into an X demand on the last element of the list but the particular domain that we have chosen for analysis cannot capture this.

We now give two additional examples that give a better feel for the kind of strictness information that can be computed by the analysis.

$$\begin{aligned} positives(x:xs) &\rightarrow if\ (x > 0)\ then\ x:positives(xs)\ else\ positives(xs) \\ positives(nil) &\rightarrow nil \end{aligned}$$

$$\begin{aligned} rowmult(x, y:ys) &\rightarrow dotmult(x, y):rowmult(x, ys) \\ rowmult(x, nil) &\rightarrow nil \end{aligned}$$

The *positives* function, which filters out the negative numbers in a list, has the following properties:

$$\begin{aligned} positives^{d} &\vdash x_1^{head(e)} \\ positives^{list(X)} &\vdash x_1^{list(e)} \end{aligned}$$

Thus, under head-normal-form demand, *positives* requires the first element in its argument list to possess a normal form. In a tail-strict context, *positives* actually requires its argument in full normal form (noting that $list(e)$ is actually an instance of e). This implies that to evaluate the term $length(positives(xs))$, xs would have to be normalized.

The *rowmult* function illustrates the ability of our analysis to capture *conditional strictness*, i.e., the property that the function is strict in an argument under certain conditions on the value of the other arguments. In particular, given the strictness of *dotmult* as

$$dotmult^{list(d)} \vdash x_1^{list(e)} \wedge x_2^{list(e)}$$

we get the property

$$rowmult^{list(list(d))} \vdash x_2^{nil} \vee \left(x_1^{list(e)} \wedge x_2^{list(list(e))}\right)$$

which captures the fact that *rowmult* needs to evaluate its first argument completely, whenever its second argument is non-nil. We explain in the following section how this information can be very valuable and how it can be exploited.

4. PUTTING DEMAND ANALYSIS INTO PRACTICE

In this section, we discuss various issues that arise in putting demand analysis into practice in a real compiler.

4.1 Code Generation Based on Demand Propagation

Our analysis infers the extent of evaluation of arguments to a function f based on the extent of evaluation demanded of the function application. The result of the analysis is a table relating output demand D to a boolean formula φ on input demands. First, we compute a conjunction $\psi = conj(\varphi)$ as follows:

$$conj\left(\bigvee_{i=1}^{n}\bigwedge_{j=1}^{k} x_j^{D_{ij}}\right) = \bigwedge_{j=1}^{n} x_j^{D_j}, \text{ where } D_j = union(D_{1j}, D_{2j}, ..., D_{nj})$$

This conjunction specifies the minimum demand D_j placed on the jth argument when the output is D-demanded. Therefore, we can evaluate the jth argument (for $1 \leq j \leq k$) to the extent D_j before calling f. (Note that this approach to code generation is similar to the evaluation transformer model [Burn 1987].) To exploit conditional strictness, we need be a bit more careful. For instance, when generating code for $rowmult$ under $list(list(d))$, we retain both disjunctions in the strictness of $rowmult$. However, note that the first of these conjunctions (namely, x_2^{nil}) applies only when the second argument to $rowmult$ is nil. Thus, whenever we know that the second argument to $rowmult$ is of the form $x\!:\!xs$, we can eliminate the first conjunction to end up with the strictness property $x_1^{list(e)} \wedge x_2^{list(list(e))}$. This allows both arguments of $rowmult$ to be evaluated to normal form before $rowmult$ is called. Note that the savings due to this optimization can be substantial, since all calls to $rowmult$, with the exception of calls that correspond to the base case, will be with a non-nil second argument.

Note that, in general, the demand placed on each argument may differ in different contexts, since the demand on the function's output may differ. There are two approaches to deal with this in code generation:

—Specialize each function based on the demand placed on its output, and generate one version of the code corresponding to each demand under which the function could be evaluated.

—Propagate demands at run-time: make the demand as an explicit parameter to the function, and in the body of the functions' code, use this parameter to determine evaluation extents at run-time.

The second approach is more general, since one can use straightforward partial evaluation and specialization on the generated code to achieve the effect of the first approach. Therefore we will focus on this approach below and illustrate it with the example of *append* function, for which we will show the code generated using a C-like syntax. The target language definition of *append* will be denoted *appendT* and the source definition denoted *appendS*. Terms in the source language, as well as demand terms, are represented by a tree data structure called $Node$ in the target language. The code generated for *append* may look as follows. In this code, we make use of a two-argument function *eval* to perform evaluation of closures. The first argument to *eval* is the closure (i.e., a term-graph in a graph-rewriting-based implementation) to be evaluated, and the second argument is the extent of evaluation of this term, specified as a demand.

```
Node appendT(l1, l2, dem) {
    /* No need to head-normalize l1 before pattern match,
       since the caller would have done this based on strictness */
    switch l1 {
        case nil:
            switch dem {
                case d: return eval(l2, dem);
                case Node(head, dem1): return eval(l2, dem);
                case Node(list, dem1): return l2;
                    /* Note: l2 already evaluated by caller */
            }
        case Node(x, xs):
            switch dem {
                case d: return buildTerm(:, x, buildTerm(appendS, xs, l2));
                case Node(head, dem1):
                    /* Again, x would have been evaluated before calling appendT */
                    return buildTerm(:, x, buildTerm(appendS, xs, l2));
                case Node(list, dem1):
                    /* l1 and l2 are already evaluated to the extent required by dem */
                    return buildTerm(:, x, appendT(xs, l2, dem));
            }
    }
}
```

At a site that calls $appendS(l1, l2)$ and needs it to be evaluated to an extent dem, the following code will be generated.

```
eval(l1, dem);
switch (dem) {
    case d:
        appendReturnVal = appendT(l1, l2, d);
    case Node(head, dem1):
        appendReturnVal = appendT(l1, l2, dem);
    case Node(list, dem1):
        eval(l2, dem);
        appendReturnVal = appendT(l1, l2, dem);
}
```

Several improvements can be made to the code generated above to eliminate redundant calls to *eval*. One of the obvious ways is to specialize the code for different calling contexts corresponding to different demands on the output of a function. For instance, we could specialize the code for *append* into three versions that are called with $d, head$, and *tail* demands, respectively. Of course, generation of multiple versions can lead to a large increase in code size but this problem is not unique to our analysis — it is a problem faced by any method that deals with different contexts of evaluation, e.g., the methods proposed by Hall and Wise [1987], Burn [1987], and Finne and Burn [1993]. Techniques to deal with the problem have been suggested by Hall and Wise [1987], Kaser et al. [1992], and Finne and Burn [1993] and discussed in detail based on insights gained by actual implementation. Here, we present a summary of techniques that we propose to deal with this problem in the context of demand analysis.

There are several ways to control the number of versions generated. First, it is possible to estimate the time savings (in terms of the percentage reduction in the number of instructions in the function body or within each basic block within the body) and compare it with the increase in code size. We can avoid specializing whenever the ratio of time savings to code size increase is small. Second, there is often no need to specialize for each context of calling, since the code may be identical (or very similar) for a set of demands. In this case, we can share a single version for all these demands. For instance, in the case of the *append* function above, the code generated for evaluation under demand d and *head* are identical, so we can generate just one version corresponding to both these extents.

The above scheme for code generation has been implemented in the EQUALS compiler. EQUALS is based on *ee*-analysis and hence its compilation rules [Kaser et al. 1992] deal with only two contexts of evaluation, namely, normal form and head-normal form. However, the structure of the compilation rules is not affected by the number of different contexts, so the scheme would not be changed significantly to accommodate the results of demand analysis.

4.2 Automatic Synthesis of Demand Domains

It will be desirable to synthesize practically useful demand domains automatically from the program, rather than relying on user-specified annotations for this purpose. The synthesis is accomplished in a manner similar to that proposed by Hughes and Launchbury [1992a]. In this section, we present a overview of a possible approach, skipping a rigorous treatment due to space considerations.

Our main task is to synthesize an analysis domain for a given type. We do this by "truncating" the subterms in the rhs of the type definition at different points. In this context, truncation amounts to replacing a subterm by the demand u. For base types like int and $float$, there are two contexts for analysis: u and e. For enumerated types, the only truncation possible removes every thing in the definition of the type, once again leaving us with two contexts u and e. For the following recursive type definition of list (given in SML-syntax)

$$\textbf{datatype}\ \ \alpha\, list = nil\ \mid\ :\ \textbf{of}\ \alpha *\ \alpha list$$

the possible contexts are as follows. Herein, we use the notation T^i for the ith context derived for a type T.

$$list^1(\alpha)\ =\ nil\ \mid\ :\ \ \textbf{of}\ \ \alpha * u \qquad (17)$$

$$list^2(\alpha)\ =\ nil\ \mid\ :\ \ \textbf{of}\ \ \alpha * list^2(\alpha) \qquad (18)$$

We have deliberately blurred the distinction here between contexts and types to simplify presentation. In the first context derived for list, we truncated the second child of ":" to get the $head(\alpha)$ demand described earlier. In the second case, nothing is truncated, so we get the $list(\alpha)$ demand. Note that there are ways to generate additional contexts by replacing the type variable α by u but this is not useful since u is an instance of the type variable; hence the context generated by such replacement would be subsumed by a context that retains the type variables.

Note that additional contexts could be generated, corresponding to "base cases," such as *nil* in the above definition. Thus, we can have another domain point:

$$list^3 = nil.$$

Having synthesized the domain points, we can get the domain relations *sub*, *int*, *union*, and *Pr* as well as the strictness of constructors based on the definition of the demands. For instance, we can get from Eq. (18) that $Pr(X, list^2(X)) = list^2(X)$, $Pr(X, nil) = list^2(X)$, etc. Our implementation does not synthesize demand domains automatically at this time. Instead, it relies on programmer annotations to describe the demands using a notation similar to Eqs.(17) and (18). The domain relations are then computed based on these definitions.

4.3 Implementation Experience

In this section we discuss our experience in implementing and using demand analysis and *ee*-analysis. We provide performance results showing that our analyses are very fast and take a very small fraction (less than 0.2% for *ee*-analysis and less than 1% for the general demand analysis) of the time taken to compile the program under a modern functional language compiler such as SML of NJ. The run-time performance gains are handsome — our preliminary experiments indicate speedups of between 1.5 to 3 using the results of *ee*-analysis. As a result, the performance of our EQUALS implementation is comparable to that of a strict language such as SML of NJ.

4.3.1 *ee-Analyzer Implementation and Performance.* The implementation of *ee*-analysis is structured into two modules: the boolean formula module and the analyzer module. The first module deals with boolean formula manipulation, providing the basic operations for conjunction, disjunction, composition, and implication check. Due to the simplicity of representation used in *ee*-analysis, we can represent a conjunction as a bitvector, with the ith bit set exactly when x_i occurs in a conjunction. This representation makes the analysis particularly efficient, since we can use very few machine instructions to perform the basic operations on whole conjunctions, namely, taking conjunctions of conjunctions (bitwise OR) or checking implications (complement, bitwise AND followed by test for zero) among them. A DNF is represented as a list of bitvectors. Since the analysis is already very fast, we have not attempted to use the linear-time heuristics described in Section 2.3.

The analyzer module uses the operations provided by the boolean formula module to compute *ee*-strictness. In order to minimize the number of fixpoint iterations, the analyzer first constructs a dependency graph of functions in the program and performs the analysis bottom-up. Fixpoint iteration is used on strongly connected components of this graph, thereby minimizing the number of functions that are involved together in fixpoint iteration.

The performance of the analyzer is shown in Table I. The programs considered were among the benchmarks for EQUALS. All of them contain some utility and test functions in addition to the main function described by the program name. Of these, the first is an implementation of naive $O(n^3)$ matrix multiplication algorithm, with a matrix represented as a list of lists. The second is an implementation

Table I. Implementation of *ee*-Analysis

Program	# of lines	Analysis time[4] (msec)	# of args	# of strict args	%age *ee*-strict
Matmult	70	3.6	12	9	75
Strassen's Matmult	93	8.6	13	13	100
Mergesort	65	4	14	12	86
Quicksort	64	3	15	13	87
Sieve	24	1.8	7	6	86
Nqueens	90	7.6	21	13	60
Euler	57	3.6	12	9	75
FFT	343	23	70	59	84
Aggregates	806	55.2	164	134	82

of Strassen's matrix multiplication algorithm, chosen for its particularly large and complex rewrite rules. The third and fourth are straightforward implementations of the mergesort and quicksort algorithms. The fifth is a program to find prime numbers using the sieve of Erotosthenes. Nqueens is a program that computes all the configurations in which N queens can be placed on an $N \times N$ chess board without attacking each other, whereas the Euler program computes the Euler totient function. Finally, FFT is an implementation of the fast fourier transform and power spectrum algorithms, complete with functions for manipulating complex numbers.

The table indicates the number of lines in the source file and the analysis time (in milliseconds). As can be seen from the table, the analysis is very fast, being able to process about 15,000 lines of source code per second. This represents a very small fraction (less than 0.2%) of the compile time for this many source lines in a typical functional language compiler. For instance, the compilation of each of these programs using the Standard ML of New Jersey compiler (version 0.93 running on the same machine) took 500 to 1200 times longer than the *ee*-analysis time.

The table also indicates the effectiveness of *ee*-analysis on these programs. It indicates the sum of the number of arguments, taken over all the functions defined in each program. It also shows the number of arguments identified to be *ee*-strict by the analysis. Note that on the average, more than 80% of the functions are *ee*-strict in these programs, thus making our analysis quite effective. With an exception of one or two cases (over all of the programs), the arguments identified to be nonstrict were indeed nonstrict, showing that there is little room for improving accuracy of the analysis without increasing the size of the demand domain. In many of these cases, the nonstrictness was due to the fact that certain arguments were discarded in the base case — precisely the situation where conditional strictness comes in handy. In the full-demand analyzer described later, this conditional strictness is exploited, leading to 94% of the arguments being identified as *ee*-strict.

4.3.2 *Optimizations in* EQUALS *Using ee-Analysis.* Our *ee*-analysis has been implemented in the EQUALS system for parallel evaluation of functional programs and has been in use for five years now. The EQUALS system extracts parallelism automatically from programs, *without any need for programmer annotations,* using the

[4]These timings were obtained on a SUN SPARC LX.

results of the analysis. Below, we describe the optimizations performed in EQUALS based on the results of *ee*-analysis:

—*Automatic Detection of Parallelism:* When the analysis reports that several arguments to a function are needed in normal form, these arguments are evaluated in parallel. Since the parallelism exploited here is "conservative," and due to the large granularity of the tasks for computing normal forms, the overheads in EQUALS are significantly reduced. As a result, EQUALS scales almost linearly with the number of processors, even up to 20 processors [Kaser et al. 1994].

—*Transform Call-by-Need into Call-by-Value:* *ee*-strict arguments of a function are fully evaluated before the function call. This avoids costs of repeated construction and evaluation of closures (or term graphs, in a reduction-based approach such as ours) associated with call-by-need evaluation. This optimization improved our speeds by a factor of 1.5 to 3 on most of our benchmarks. Some programs that manipulate integers, such as *fibonacci*, showed significantly higher speedups.

—*Unboxing:* A boxed representation can store either a fully evaluated value or a closure that can be evaluated when needed. Such a representation is associated with storage overheads for storing tags (that specify whether the value is evaluated) and time overheads for repeated checking of the tags. To reduce these overheads, we can use unboxed representation whenever a value is known to be fully evaluated. Note that a function may be nonstrict in a particular argument but may still get this argument fully evaluated in a particular context. Based on the results of *ee*-analysis, we can detect this situation and perform unboxing. This optimization can be significant for functions that "throw away" arguments, such as *length*. Often, such a function is called with the argument evaluated in some other context, thereby enabling unboxed representation. (Context-based unboxing has not been implemented in EQUALS yet.)

If there is just a single version of the code for each function, we would depend exclusively on run-time propagation of demands to get the benefits of the above optimizations. In particular, there will be several tests on the current calling context before the optimized branch of code is executed. As mentioned earlier, we can reduce this overhead by generating multiple versions of code corresponding to different calling contexts. In EQUALS, we currently generate two versions of the code, that correspond to *e*- and *d*-contexts. For each of the versions, tests on calling contexts can be eliminated. On the average, we observed about 20% speedup in execution times as a result of generating the two versions.

Our current implementation in EQUALS was simplified by the fact that there were only two contexts of evaluation. For the more-general analysis the compilation problem is rendered more complex since we need to deal with (a) generation of contexts based on types, (b) providing *eval* functions that evaluate to each of these contexts, and (c) managing the number of code versions.

4.3.3 *Implementation of Demand Propagation.* In this section we describe our implementation of demand analysis. The implementation results show that our analysis is quite efficient, in that the analysis of a program takes less than 1% of the time to compile the same program using a modern functional language compiler such as that of SML of NJ. It is also very effective and is able to detect most of the

Table II. Implementation of Demand Propagation

Program	# of lines	Analysis time[5] (msec)	# of args	# of *ee*-strict args	%age *ee*-strict
Matmult	70	25	12	12	100
Strassen's Matmult	93	62	13	13	100
Mergesort	65	27	14	12	86
Quicksort	64	21	15	13	87
Sieve	24	12	7	6	86
Nqueens	90	60	21	21	100
Euler	57	28	12	11	92
FFT	343	140	70	66	94
Aggregates	806	375	164	154	94

strictness information in our benchmark programs. An important feature of this implementation is that it is entirely independent of the analysis domain and thus can be reused across different forms of strictness analysis. All domain-dependent information is provided as data to the analyzer.

The implementation is organized into three modules. The first module deals with composition, function definition, and fixpoint iteration steps. It is parameterized with respect to operations on demand constraints, such as simplification, projection and equality checking, and the strictness of base functions and constructors. This module first constructs the dependency graph of functions in the program and identifies strongly connected components in this graph. It then analyzes functions bottom-up in this graph and uses fixpoint iteration to analyze recursive functions that belong to the same strongly connected component. The implementation of this module closely follows the rules described in Section 3.1.

The second module is concerned with operations on demand formulas and implements the functions S, $Proj$, and $\sqsupseteq$ operations on the demand constraints. The functions in this module are parameterized with respect to domain-specific operations, such as the Pr function (which are in turn provided by the third module). Once again, the implementation closely follows the definitions given in Section 3.1.

The third module provides the domain-specific functions. As mentioned earlier, this module does not currently synthesize demand domains automatically. Rather, the user hand-computes the domain that would be synthesized by an automatic method and supplies it as input to the analyzer. However, only minimal information, such as the definition of a particular demand, is specified in a format similar to that of Eqs. (17) and (18) in Section 4.2. The domain component synthesizes all the domain-dependent functions ($sub, int, union$, Pr, and the strictness of constructors) from this information.

The implementation performance in shown in Table II, with the same programs as for *ee*-analysis. In order to simplify the domain specification, the same domain was used to analyze all functions. Thus the domain had to contain all the demands that were useful to analyze all the functions, resulting in a large domain with about 30 points. Our experiments show that this size did not affect our analysis adversely, thereby providing empirical support to our claim that the method is

[5]These timings were obtained on a SUN SPARC LX.

relatively insensitive to domain size.

Observe that the general method is about 6 to 10 times slower than *ee*-analysis but is still quite fast — it can process more than 2000 lines of code per second and accounts for about 1% of the compile time of these programs using the SML of NJ compiler. The slowdown as compared to *ee*-analysis is due to two factors: first, the constraints involved are more complex and thus take a longer time to simplify. Second, each function is analyzed in several contexts. For instance, the matrix multiplication program is analyzed under six different contexts.[6] These two factors actually account for more than a factor-10 slow down. But the slowdown is offset by additional implementation tricks that we have employed — in particular, we cache and reuse results of fixpoint iteration in order to improve the speed.

The last two columns in the table show the number and percentage of arguments that were *ee*-strict.[7] The additional strictness information obtained by our generalization (over that of *ee*-analysis) is conditional, in the sense that strictness on the additional arguments depends upon the value of other arguments. This fact has significant impact on $Nqueens$ and FFT, where several functions are nonstrict when some of the arguments are zero. As a result of these improvements, the enhanced analysis is able to detect *ee*-strictness in 94% of the function arguments.

5. PREFIX-BASED DEMAND ANALYSIS

Traditionally, strictness analysis has been used as an optimizing technique that improves performance but it does not affect the language or semantics in any way. We describe an analysis method in this section, using which we can enlarge the class of programs for which we can construct language implementations that are completely faithful to equational semantics.

Complete faithfulness to equational semantics requires the use of purely lazy pattern-matching techniques. Such techniques are known only for the class of *strongly sequential programs* [Huet and Levy 1979]. This class is generally considered restrictive for the purposes of programming, since natural definitions of many functions are not strongly sequential. For instance, even the following simple program to merge two sorted lists violates strong sequentiality.

$$\begin{aligned} merge(x\!:\!xs, y\!:\!ys) &\rightarrow if(x > y, y\!:\!merge(x\!:\!xs, ys), x\!:\!merge(xs, y\!:\!ys)) \\ merge(x, nil) &\rightarrow x \\ merge(nil, y) &\rightarrow y \end{aligned}$$

As a result, most lazy functional languages permit programs that do not observe strong sequentiality. Preserving equational semantics in the face of such programs requires the use of parallel evaluation strategies. (Parallelism is needed since multiple branches of computation need to be explored in parallel in order to guarantee finding an answer — see Sekar and Ramakrishnan [1993] for details.) Since parallel strategies are much less efficient than sequential strategies, modern functional lan-

[6] The six contexts are: $d, list(u), list(list(X)), list(head(X)), head(list(X))$, and $head(head(X))$.

[7] We have shown *ee*-strictness, as opposed to head- or tail-strictness, to enable easy comparison with results of *ee*-analysis. However, the *ee*-strictness figures alone are not good indicators of the effectiveness of our analysis. The examples given in Section 3.5 complement this table in giving the reader a better idea of the effectiveness of demand analysis.

guages make use of only sequential evaluation strategies, based on pattern-matching algorithms that are not purely lazy. The efficiency of the sequential strategy comes at the cost of complicating the semantics, since the meaning of the program is now closely tied to the pattern-matching algorithm implemented in a compiler. A natural question therefore is whether we can retain the simplicity of purely equational semantics without necessarily compromising on expressive power and convenience of the language. In this section, we show how our analysis can be used to address this problem. Specifically, we modify our analysis techniques so that the results can be used to perform purely lazy pattern-matching for a class of programs that we refer to as sequentializable programs. Alternatively, our technique could be considered as an optimization of the parallel evaluation strategy described by Sekar and Ramakrishnan [1993], so as to further minimize needless reductions performed.

To accomplish our goal, the method presented in this section analyzes prefixes of terms, as opposed to just function symbols (which are, in fact, a special case of prefixes with exactly one nonvariable symbol). This approach enables us to augment the power of the analysis without enlarging the domain or changing any of the domain-related or demand-formula-related algorithms. In particular, using *dd*-analysis (which reasons about those arguments of a function that need to be *head-normalized* in order to find the *head-normal form* of a function application) of pattern prefixes, we can compute those arguments of a prefix t that need to be head-normalized in order to find the head-normal form of any instance $t\beta$. A lazy strategy would then head-normalize the subterms thus identified. After this step, the prefix would be enlarged to include the roots of the subterms head-normalized, and the process is repeated until a pattern match is found. At this point a reduction is performed, and a new prefix is identified at the root of the term. The whole process is repeated again. It is crucial in this context to exploit all the symbols in the prefix to identify which arguments are needed in head-normal form, as we often cannot complete the pattern match by reasoning about only the function symbols.

The approach outlined above for lazy, pattern-matching-driven evaluation is similar to previously known techniques, originally due to Huet and Levy [1979] and later extended by Laville [1987], Puel and Suarez [1990], and Sekar et al. [1992]. In particular, the steps of identifying prefixes, selecting subterms to head-normalize, etc. are exactly the same. The difference arises in the algorithm used for selecting the so-called *index* positions, which correspond to subterms that are selected for head-normalization. Previous approaches would identify the indexes based only on the patterns, whereas we use a more-powerful inference procedure that reasons about the rhs of the rewrite rules as well. This fact permits us to handle programs for which previous approaches would fail to compute a lazy strategy.

Below, we describe how our analysis can be extended to deal with prefixes. This extension is orthogonal to the one described in previous sections that enabled us to deal with different kinds of demands. As such, in order to simplify our presentation, we will present the extension in the context of *dd*-strictness defined below.

Definition (*dd-Strictness*). Let t be a term and $\varphi = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{r_i} x_{ij}$ be a boolean formula on variables in t. We say that φ is a *dd*-strictness property of t iff, in every reduction sequence that head-normalizes any instance $t\beta$, all the terms in one of the sets $\{\beta(x_{11}), ..., \beta(x_{1r_1})\}, ..., \{\beta(x_{n1}), ..., \beta(x_{nr_n})\}$ are head-normalized.

5.1 Extension of Analysis to Pattern Prefixes

Before proceeding with the analysis, we must first decide upon those prefixes that are going to be analyzed. This choice may depend on the intended use of the analysis. For instance, with regards to computing lazy strategies, a natural choice is to examine all prefixes of all the lhs.

Let $w_1, ..., w_n$ be the prefixes selected for analysis. If the term t to be head-normalized is a variable x, then t^{dd} is simply x. Thus

$$x^{dd} \vdash x.$$

A term $t = c(t_1, ..., t_k)$ is already in HNF. Hence

$$[c(t_1, ..., t_k)]^{dd} \vdash true.$$

For a term t with a function symbol root, let w be its largest prefix that is among $w_1, ..., w_n$. If we denote $w[x_i \leftarrow t_i]$ as $w(t_1, ..., t_n)$, then t is of the form $w(t_1, ..., t_n)$. For this case, we have

$$[w(t_1, ..., t_n)]^{dd} \vdash w^{dd}(t_1^{dd}, ..., t_n^{dd}).$$

We now specify $w^{dd}(x_1, ..., x_n)$. Denote by $\mathcal{L}_w$ the set of all lhs that unify with a term w. Note that a term $w(t_1, ..., t_n)$ can be rewritten only by a rule $l \to r$, where $l \in \mathcal{L}_w$. Suppose it is rewritten by a rule $l \to r$, and let σ_l and β_l be the most-general substitutions such that $w\sigma_l = l\beta_l$. If $\sigma_l(x_i)$ is a nonvariable term in l, then t_i needs to be at least head-normalized if it is to match l. Moreover, if a variable x_i is demanded in $r\beta_l$, then the same variable can be demanded in $l\beta_l = w\sigma_l$. In other words, if x_i is a variable in w such that $\sigma_l(x_i)$ is a variable demanded in $r\beta_l$, then x_i can be demanded in w. Therefore,

$$[w(x_1, ..., x_n)]^{dd} \vdash \bigvee_{l \in \mathcal{L}_w} \left([\bigwedge_{\sigma_l(x_i) \text{ is a nonvariable term}} x_i] \wedge g_l^w\left([r\beta_l]^{dd}\right) \right) \quad (19)$$

where g_l^w is once again a projection function that converts a boolean formula on variables in l to one on variables in w. It is defined by:

$$g_l^w(\varphi_1 \vee \varphi_2) \vdash g_l^w(\varphi_1) \vee g_l^w(\varphi_2) \quad (20)$$
$$g_l^w(x) \vdash x, \qquad \text{if } \sigma_l(x) = x \quad (21)$$
$$g_l^w(x_{ij}) \vdash true, \qquad \text{otherwise} \quad (22)$$

The first equation simply "pushes" g_l^w into individual conjunctions. Within each conjunction, g_l^w operates as follows. If x is a variable in w that is d-demanded in rhs, then it is clear that it is d-demanded in order to head-normalize any instance of $w\sigma_l$, and hence (21). Otherwise, the particular variable x_{ij} (in Eq. 22) is either not d-demanded in the rhs or does not even occur in the rhs. In the first case, we cannot conclude anything about the demand on x_{ij}, and hence it is dropped. In the latter case, it must be the case that x_{ij} is matched against a nonvariable term in l, and hence the d-demand on it would have already been inferred in the first part of (19). Hence, we can discard such x_{ij} in this case as well.

Based on this analysis, we can define the following class of *sequentializable* programs that admit efficient lazy evaluation as follows.

Definition (*Sequentializable System*). A TRS is *sequentializable* iff for every proper prefix w of any lhs, $w^{dd} \vdash x_i$ for some variable x_i in w.

This definition is quite similar to that of strong sequentiality, except that now we are permitted to use a sophisticated inference procedure to determine if a subterm needs to be head-normalized. In particular, the previous approaches correspond to replacing the second term in Eq. (19) by *true*, thereby ignoring all information on the rhs of the equations. Therefore sequentializable systems are a strict superset of the class of strongly sequential programs. An example of a sequentializable program that is *not* strongly sequential is the *merge* function given earlier in this section.

6. COMPARISON WITH RELATED WORK

In this section we compare our method to previous work on strictness analysis of nonflat domains. Some of the main differences between our approach and previous approaches have already been described in the introduction. In this section, we will refrain from repeating these differences and instead focus on some of the more-specific aspects. First we mention some of the earlier works that do not have much similarity with our work. Later on, we provide a detailed comparison with works that were based on projection analysis and abstract interpretation.

Hughes [1985] was the first to address strictness analysis for nonflat domains. His method was based on an analysis of the context in which an expression is evaluated. However, as mentioned by Hughes and Wadler [1987], his methods were not completely intuitive, and the analysis requires algebraic manipulation techniques that do not seem mechanizable.

The notion of demand propagation we use is similar to that of Lindstrom [1986]. His demand patterns include not only $\{e, d\}$ but also recursively formed pairs of them. His method is based on abstract interpretation of a class of function graphs [Keller 1980] over this domain of demand patterns. However, he uses abstract interpretation over infinite domains, which means that the conventional fixpoint iteration may not terminate. Therefore he inverts the ordering on the abstract domain in the sense that $D_1 < D_2$ whenever D_1 requires less evaluation than D_2 and uses fixpoint iteration on this domain. When we use this procedure, we can stop iteration before reaching a fixpoint since every approximation f^i to strictness of f requires less degree of evaluation than required by the actual strictness of f. Since the ordering on this domain is inverted as compared to usual ordering, if the fixpoint iteration stops then it would have computed the greatest (in the conventional ordering) solution to the recursive equation defining strictness of f. However, as Lindstrom himself observes [Lindstrom 1986], these fixpoints provide much weaker strictness properties than least fixpoints. For instance, he cannot conclude that in order to normalize *append*(x, y), both x and y can be normalized before evaluating *append*.

Hall and Wise [1987] use strictness patterns (which resemble demand patterns of Lindstrom) to annotate programs with strictness information. Their focus is on strictness of programs using streams (which are potentially infinite lists). Their work bears some resemblance to that of Hughes and Wadler [1987] discussed later.

Kuo and Mishra [1989] propose using type inference for strictness analysis of higher-order functions in lambda calculus. They treat strictness properties as types.

For instance, a dd-strict function possesses the type $\phi \rightarrow \phi$, where ϕ denotes the set of terms lacking HNF. Their approach is also operational, based on β-reductions in lambda calculus. Treating strictness as a type enables them to deal with higher-order polymorphic functions in a very satisfactory manner. Their analysis reasons about only HNF, and so Leung and Mishra [1991] have extended the analysis to reason about normal forms as well. However, the complexity of the inference system appears to increase significantly with the size of the analysis domain, and this can have adverse impact on the efficiency of the method.

Nocker [1993] has proposed the use of *abstract reductions*, wherein the reductions in the concrete domain are mimicked in the abstract domain. An advantage of this approach is the elimination of fixpoint iteration techniques. Instead the author proposes the use of a technique called *reduction path analysis* to deal with infinite abstract reduction sequences over potentially infinite domains. This technique is based on identifying certain patterns in reduction sequences but there appears to be no way to characterize all "useful patterns." Consequently, the technique has to often rely on terminating the reduction sequence at some depth, which can lead to loss of valuable information.

Burn's work on evaluation transformers [Burn 1987; Finne and Burn 1993] complements most of these works, by focusing on how the information provided by strictness analysis can be used to alter evaluation. It uses context information to better utilize the strictness information. It particular, he determines the degree of evaluation permitted on the input when a certain degree of evaluation is permitted on the output. In this manner he associates with each function a mapping that takes an evaluator on its output to another on its inputs. He refers to these mappings as evaluation transformers.

We now proceed to present detailed comparison of our work with the projection analysis and abstract interpretation approaches.

6.1 Projection Analysis

Hughes and Wadler [1987] base strictness analysis on the domain-theoretic concept of *projections*. Like demands, projections are also used to denote evaluation contexts. However, projections can capture the notion of a function by explicitly discarding certain arguments (corresponding to the ABS ("absence") projection), whereas we are unable to do this in our analysis. In particular, the membership constraints that we have used in our analysis cannot capture absence, since it cannot distinguish between an argument being thrown away, versus any possible value being acceptable in that context — both cases would correspond to membership in u. It is quite possible, however, to use more-expressive constraints to denote conditions on evaluation extents, in order to capture absence.

By combining the ABS projection with other projections, projection analysis can reason about "if a function evaluates its argument, to what degree it will evaluate it." For this reason, whenever projection analysis tells us that an argument can be evaluated to some extent D, it is safe to evaluate it to this extent whenever there is at least a d-demand on this argument. In this sense the results of projection analysis imply the results of strictness analysis using our method as well as abstract interpretation. (This relationship between projection analysis and abstract interpretation is formally established by Burn [1990a] for a class of projections and abstract interpretations.) Using the ABS projection, Hughes and Wadler are able to capture a notion of head-strictness (denoted H in their work) that captures the notion that "whenever a cons cell is evaluated, its head needs to be evaluated fully, and moreover, the tail of the cons cell is again in a H-context." In contrast, our notion of head-strictness coincides with that of "simple head-strictness" defined in Burn [1990a] and denoted H_B. We discuss the impact of this difference below. The discussion, in fact, applies to the broader question of how the information provided by ABS projection could be used. Using Hughes and Wadler notion of head-strictness, at the time of construction of a list argument to a head-strict function, information can be stored in the closure so that whenever it is evaluated its head is evaluated as well. In contrast, we do not have any information at the time of closure construction. The decision (as to whether the head is evaluated) is made at the time the closure is evaluated. In the terminology of Burn [1990b], the former corresponds to *compile-time* choice of an evaluator, whereas the latter corresponds to *run-time choice*. As noted by Finne and Burn [1993], run-time choice leads to at least as much evaluation as compile-time choice. Moreover, while run-time choice can have additional overheads, the experience of Finne and Burn [1993] indicates that these overheads are typically more than offset by the gains due to additional eager evaluation enabled. Finally, even when we use Hughes and Wadler's notion of head-strictness, certain kinds of optimizations are not possible without run-time propagation of contexts (or, equivalently, code specialization). For instance, the function *doubles* is head-strict only in a head-strict context. Unless this context information is propagated, we cannot evaluate the heads of the argument to *doubles*.

Our method can also capture strictness properties that are not captured by projection analysis. In particular, Hughes and Wadler's method does not capture interargument dependencies. To ensure acceptable accuracy in spite of this, they need to give special treatment to functions such as if and *case*. For the same reason, their analysis does not detect other instances of conditional strictness, as in the matrix multiplication example. The later work of Davis and Wadler [1989] developed the theoretical basis for dealing with interargument dependencies, but left open the complementary problem of designing efficient algorithms that lead to practical implementations.

Another important difference between the two analyses is that projections are denotational concepts. Projection analysis reasons about whether altering the denotational semantics of an argument changes the denotational semantics of the result but no formal proof that the change permitted in denotational semantics allows a change in evaluation order is given. In contrast, we work only with the operational semantics of reductions and reason about whether changing the order of evaluation changes the termination properties of the program.

Of particular interest in comparing projection and demand analyses is the treatment of polymorphism. Like most works based on denotational semantics, Hughes and Launchbury [1992a] use polymorphic invariance proofs to show that a property that is established for a single instance of a polymorphic function is indeed valid for all instances. In contrast, our operational approach deals directly with polymorphism using "polymorphic demands" expressed using demand variables. We believe that this factor has resulted in a simpler and more-natural treatment of polymorphism in our method.

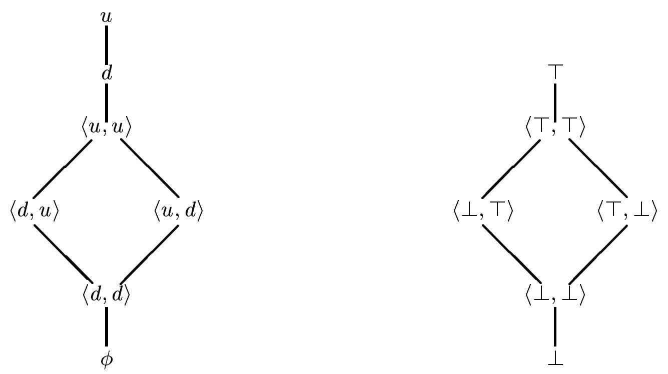


Fig. 7. Domains for analysis using (a) demand propagation and (b) abstract interpretation.

6.2 Abstract Interpretation versus Demand Propagation

Wadler [1987] uses abstract interpretation over a four-point domain for reasoning about strictness on lists. The four points correspond to undefined list (represented by value 0), infinite lists and lists with some tail undefined (value 1), lists with at least one head undefined (value 2), and all lists (value 3). Burn's work [Burn 1987] on evaluation transformers also uses abstract interpretation on the above-mentioned domain for strictness analysis. He defines four evaluators that correspond to the four points mentioned above in the sense that the ith evaluator will fail to produce an answer when given a list with the abstract value $i-1$.

As described before, abstract interpretation is a *forward analysis* approach, whereas our method uses *backward analysis*. By using boolean constraints, our analysis is able to capture interargument dependencies, which has traditionally been a strong point of abstract interpretation. In fact, our analysis can improve upon the accuracy of of abstract interpretation due to backward propagation of demand, as illustrated below. Consider the following example:

$$\begin{aligned} f(\langle x, y\rangle) &\rightarrow x + y \\ g(x, \langle y, z\rangle) &\rightarrow f(if(x, \langle y, z\rangle, \langle z, y\rangle)) \end{aligned}$$

When we use abstract interpretation (with the domain shown in Figure 7) to analyze f, we conclude that the output of f will be $\bot$ whenever either of the components of its tuple argument is $\bot$. Now let us compute the abstract interpretation $g^{\#}$ when given the arguments $\top$ and $\langle \bot, \top\rangle$. Observe that the *then* and *else* clauses of if possess the abstract interpretations $\langle \bot, \top\rangle$ and $\langle \top, \bot\rangle$, respectively. Therefore $[if(x, ...)]^{\#}$ is $\langle \top, \top\rangle$, and so abstract interpretation of the rhs of g's definition is $f(\langle \top, \top\rangle) = \top$. This means that abstract interpretation (using this domain) is unable to infer that g produces $\bot$ when its second argument is $\langle \bot, \top\rangle$. In contrast, our demand propagation based method can infer this, using the domain depicted in Figure 7. The crucial difference arises when we compute the strictness of the term *if* x *then* $\langle y, z\rangle$ *else* $\langle z, y\rangle$ under demand $\langle d, d\rangle$. In contrast to abstract interpretation, we do not take lubs and lose information, and hence we end up with the result $x^d \wedge z^d \wedge y^d$. (A similar example also appears in Hughes and Launchbury [1992b].) It is interesting to note that even ee-analysis, which reasons using only a two-point domain, infers that $g(t_1, t_2)$ has no normal form whenever one of t_1 or t_2 fails to possess a normal form.

Now we point out the relationship and differences between our work and that of Hughes and Launchbury [1992b] on reversing abstract interpretation. A reverse abstract (strictness) interpretation of a function specifies the definedness of each argument of the function in terms of definedness of the output. In particular, if the function has n arguments, there are n reverse interpretations, the ith interpretation specifying the definedness of the ith argument. In contrast, we use a *single constraint* on the definedness of the arguments in terms of the definedness of the output, which enables us to capture interargument dependencies.

7. CONCLUSIONS

In this article we presented *ee*-analysis as a new strictness analysis technique for reasoning about normal-form evaluation over nonflat domains. We then generalized it substantially to deal with degrees of evaluation other than normal form, so as to deal with data structure strictness such as head- and tail-strictness. There are two notable aspects to our analysis technique: namely, our formulation of strictness is operational, and we resort to the use of constraints to relate the demands placed on all the arguments to the demand placed on the output of a function. We showed that these two aspects yield quite a few advantages. First it has led to simplicity in the formulation of the strictness analysis problem (by eliminating extraneous issues) and in the corresponding computational algorithms. Second, the operational aspect has made it relatively easy to relate the analysis formulation directly to optimizations performed within the compiler. Moreover, it has given rise to a simple and natural treatment of polymorphism.

The orthogonal aspect of using constraints in our analysis lets us cope with interargument dependencies in ways beyond what was possible with previously known methods. Use of symbolic constraint-solving techniques has made the analysis relatively insensitive to domain sizes. Our implementation results show that our analysis is fast, as well as effective. By making the analysis method modular, we have achieved efficiency without compromising extensibility.

It appears feasible to incorporate higher-order functions in our analysis by allowing constraints to contain functions. In a typed framework (such as the one used by Burn et al. [1985]), a decidable procedure for solving such constraints can be based on enumerating all possible strictness properties for function symbols that appear in the constraint. Such approaches have traditionally been expensive in terms of analysis time. It appears possible to alleviate this problem through the use of *approximations* in our constraint-solving algorithms, along lines similar to that described in our recent work on analysis of logic programs [Ramakrishnan et al. 1995]. By doing so we can trade potential loss of accuracy for substantially reducing or even completely eliminating enumeration.

ACKNOWLEDGMENTS

The authors would like to thank all the referees for their comments and suggestions, which have resulted in significantly improving the content as well as the presentation of this article. Special thanks are due to C.R. Ramakrishnan, with whom we have had numerous, very valuable discussions on the subject of this article. We would also like to thank Owen Kaser and Uday Reddy for discussions on this subject.

REFERENCES

ABRAMSKY, S. 1985. Strictness analysis and polymorphic invariance. In *Workshop on Programs as Data Objects*, N. Jones and H. Ganzinger Eds. Lecture Notes in Computer Science, vol 217. Springer-Verlag, New York.

AUGUSTSSON, L. 1984. A compiler for lazy ML. In *ACM Conference on LISP and Functional Programming*. ACM, New York, 218–227.

BURN, G. 1987. Evaluation transformers — a model for parallel evaluation of functional languages. In *Functional Programming and Computer Architecture*. Lecture Notes in Computer Science, vol 27. Springer–Verlag, New York, 446–470.

BURN, G. 1990a. A relationship between abstract interpretation and projection analysis. In the *17th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 151–156.

BURN, G. 1990b. Using projection analysis in compiling lazy functional programs. In *ACM Conference on Lisp and Functional Programming*. ACM, New York.

BURN, G., HANKIN, C., AND ABRAMSKY, S. 1985. Theory and practice of strictness analysis for higher order functions. In *Workshop on Programs as Data Objects*, N. Jones and H. Ganzinger Eds. Lecture Notes in Computer Science, vol 217. Springer–Verlag, New York.

BURSTALL, R., MACQUEEN, D., AND SANELLA, D. 1980. HOPE: An experimental applicative language. In the *1st ACM Conference on LISP and Functional Programming*. ACM, New York, 136–143.

CLACK, C. AND JONES, S. P. 1985. Strictness analysis — a practical approach. In *Functional Programming and Computer Architecture*. Lecture Notes in Computer Science, vol 20. Springer–Verlag, New York, 35–49.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In the *4th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 238–252.

DAVIS, K. 1993. Higher-order binding-time analysis. In *Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, New York.

DAVIS, K. AND WADLER, P. 1989. Strictness analysis: Proved and improved. In *Functional Programming, Glasgow 1989*. K. Davis and J. Hughes Eds. Springer Workshops in Computing, Springer–Verlag, New York, 12–30.

DYBJER, P. 1991. Inverse image analysis generalizes strictness analysis. *Inf. Comput. 90,* 2, 194–216.

FINNE, S. AND BURN, G. L. 1993. Assessing the evaluation transformer model of reduction on the spineless g-machine. In *Functional Programming and Computer Architecture*. ACM Press, New York.

HALL, C. AND WISE, D. 1987. Compiling strictness into streams. In the *14th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 132–143.

HUDAK, P. AND YOUNG, J. 1986. Higher order strictness analysis for untyped lambda calculus. In the *12th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 97–109.

HUET, G. AND LEVY, J. 1979. Computations in nonambiguous linear term rewriting systems. Tech. Rep. 359, IRIA, Le Chesney, France.

HUGHES, R. 1985. Strictness detection in non-flat domains. In *Workshop on Programs as Data Objects*, N. Jones and H. Ganzinger Eds. Lecture Notes in Computer Science, vol 217. Springer–Verlag, New York.

HUGHES, R. AND LAUNCHBURY, J. 1992a. Projections for polymorphic first-order strictness analysis. *Math. Struct. Comput. Sci. 2*, 301–326.

HUGHES, R. AND LAUNCHBURY, J. 1992b. Reversing abstract interpretations. In *European Symposium on Programming*. Springer–Verlag, New York.

HUGHES, R. AND WADLER, P. 1987. Projections for strictness analysis. In *Functional Programming and Computer Architecture.* Lecture Notes in Computer Science, vol 27. Springer–Verlag, New York, 385–407.

KAMIN, S. 1992. Head-strictness is not a monotonic abstract property. *Inf. Process. Lett. 41,* 4 (Mar.), 195–198.

KASER, O., PAWAGI, S., RAMAKRISHNAN, C., RAMAKRISHNAN, I., AND SEKAR, R. 1992. Fast parallel implementation of functional languages — the EQUALS experience. In *ACM Conference on Lisp and Functional Programming.* ACM, New York, 335–344.

KASER, O., RAMAKRISHNAN, C., AND SEKAR, R. 1994. A high performance runtime system for parallel evaluation of lazy languages. In *Parallel and Symbolic Computation.* World Scientific, River Edge, N.J.

KELLER, R. 1980. Semantics and applications of function graphs. Tech. Rep. UUCS–80–112, Univ. of Utah, Salt Lake City, Utah.

KUO, T. AND MISHRA, P. 1989. Strictness analysis: A type inference perspective. In *Conference on Functional Programming and Computer Architecture.* ACM, New York.

LAVILLE, A. 1987. Lazy pattern matching in the ML language. In *Foundations of Software Technology and Theoretical Computer Science.* Lecture Notes in Computer Science, vol 287. Springer–Verlag, New York.

LEUNG, A. AND MISHRA, P. 1991. Reasoning about simple and exhaustive demand in higher-order lazy languages. In *Functional Programming and Computer Architecture.* Springer–Verlag, New York.

LINDSTROM, G. 1986. Static evaluation of functional programs. In the *SIGPLAN Symposium on Compiler Construction*, ACM, New York, 196–206.

MYCROFT, A. 1980. The theory and practice of transforming call-by-need into call-by-value. In the *International Symposium on Programming.* Lecture Notes in Computer Science, vol 8. Springer–Verlag, New York, 269–281.

NOCKER, E. 1993. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture.* ACM, New York.

O'DONNELL, M. 1985. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Mass.

PUEL, L. AND SUAREZ, A. 1990. Compiling pattern matching by term decomposition. In *ACM Conference on Lisp and Functional Programming.* ACM, New York, 273–281.

RAMAKRISHNAN, C., RAMAKRISHNAN, I., AND SEKAR, R. 1995. A symbolic constraint-solving framework for analysis of logic programs. In *Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, New York, 12–23.

SEKAR, R. AND RAMAKRISHNAN, I. 1993. Beyond strong sequentiality. *Inf. Comput. 104,* 1 (May), 78–109.

SEKAR, R., RAMESH, R., AND RAMAKRISHNAN, I. 1992. Adaptive pattern matching. In *International Conference on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol 62. Springer–Verlag, New York, 247–260.

SEKAR, R., MISHRA, P., AND RAMAKRISHNAN, I. 1991. On the power and limitation of strictness analysis based on abstract interpretation. In the *18th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 37–48.

SEKAR, R., PAWAGI, S., AND RAMAKRISHNAN, I. 1990. Small domains spell fast strictness analysis. In the *17th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 169–183.

TURNER, D. 1985. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming and Computer Architecture.* Lecture Notes in Computer Science, vol 201. Springer–Verlag, New York.

WADLER, P. 1987. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin Eds. Ellis Horwood, Chichester, U.K., 266–275.

Received September 1993; revised June 1995; accepted August 1995