Flow Analysis of Lazy Higher Order Functional Programs

Neil D. Jones[a] Nils Andersen[a]

[a]*DIKU, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark*

Abstract

In recent years much interest has been shown in a class of functional languages including HASKELL, lazy ML, SASL/KRC/MIRANDA, ALFL, ORWELL, and PONDER. It has been seen that their expressive power is great, programs are compact, and program manipulation and transformation is much easier than with imperative languages or more traditional applicative ones. Common characteristics: they are purely applicative, manipulate trees as data objects, use pattern matching both to determine control flow and to decompose compound data structures, and use a "lazy" evaluation strategy.

In this paper we describe a technique for data flow analysis of programs in this class by safely approximating the behavior of a certain class of term rewriting systems. In particular we obtain "safe" descriptions of program inputs, outputs and intermediate results by regular sets of trees. Potential applications include optimization, strictness analysis and partial evaluation. The technique improves earlier work because of its applicability to programs with higher order functions, and with either eager or lazy evaluation. The technique addresses the call-by-name aspect of laziness, but not memoization.

This paper extends [19] (a chapter in the out-of-print [2]) with proofs that the algorithm terminates and yields safe (i.e., sound or correct) approximations to program behavior. Relevance to this festschrift: John Reynolds' 1969 and 1972 papers developed a related first-order program analysis framework [27], and inspired our method for handling higher-order functions [28].

1 Introduction

It has long been known that a great amount may be found out about the syntax of the input data to some programs by examining the source text of those programs.

For one example, the program structure of a recursive descent compiler directly reflects the context-free grammar of its input. For another, it is often relatively straightforward to deduce by hand the syntax of both the input to and the output yielded by a given LISP program, simply by examining its code to see which portions of the input are accessed, in relation to how the program results are constructed.

In this paper we will systematize these ideas, and present analysis methods which can automatically obtain finite descriptions of the input and output sets of lazy or eager functional programs that manipulate structured data. The same methods also work for higher order functional programs, by a simple device: the closures typically used to implement functional values are themselves regarded as data structures, to be analyzed by just the same methods as used for any other data structures.

One motivation for this work is its application in the MIX system, which uses partial evaluation to generate compilers from interpreters given as input [23]. MIX's algorithms perform partial evaluation by means of abstract interpretation (i.e., data flow analysis) of an input program over various domains. Further, compiling and compiler generation are accomplished (efficiently!) by applying the partial evaluator to itself. MIX is, however, limited when compared to more traditional semantics-directed compiler generators due to its restricted specification language: essentially first-order LISP. A natural goal has thus been to develop methods for extending MIX's techniques to encompass higher order functions.

Preview: analysis of higher order functions by tracing structured data values

Our approach "lambda lifts" [17,28] a given lambda expression to translate it into an equivalent term rewriting system, and then flow analyzes the result. Consider as first example the closed lambda expression:

$$[\lambda X.(\lambda F.F(F2))(\lambda Y.X * Y)]5$$

First account for the free X in $(\lambda Y.X * Y)$ by adding an explicit parameter X':

$$[\lambda X.(\lambda F.F(F2))\{(\lambda X'\lambda Y.X' * Y)X\}]5$$

Now make $(\lambda X'\lambda Y.X' * Y)$ into a function i, make $\lambda F.F(F2)$ into a function h, and $[\lambda X.(\cdots)\{\cdots\}]$ into a function g. The given lambda expression can thereby be transformed into an equivalent term rewriting system where X, X', Y, F are variables; @ is a defined operator representing function application; and g, h, i identify the subexpression's functions. (Technically g, h, i will be seen to be constants.)

$$\begin{aligned}
&v &&\rightarrow g @ 5\\
&g @ X &&\rightarrow h @ (i @ X)\\
&h @ F &&\rightarrow F @ (F @ 2)\\
&i @ X' @ Y &&\rightarrow X' * Y
\end{aligned}$$

The program's computation:

$$v \Rightarrow g @ 5 \Rightarrow h @ (i @ 5) \Rightarrow (i @ 5) @ (i @ 5 @ 2) \Rightarrow i @ 5 @ 10 \Rightarrow 50$$

Note that every function is treated as "curried". Function application is handled by pattern matching on terms containing the "apply operator" @ (written left associatively), so a function of order n is defined by rewrite rules of the form:

$$f @ X_1 @ \cdots @ X_n \rightarrow t$$

(whence this program notation is sometimes called "named combinators"). Curried functions applied to incomplete argument lists yield functional values, for example $(i@X')$ above. The conclusion of all this: higher order functions can be implemented using structured data values. Thus a flow analysis method able to handle structured data values in term rewriting systems can also analyze programs with higher order functions.

1.1 Outline

A simple language of the sort mentioned in the abstract is presented in Section 2, with programs containing constructors and selectors but for simplicity not atomic values such as integers or reals. Formally, a program is regarded as a term rewriting system of a certain restricted form[1]. The nondeterministic rewriting semantics naturally includes lazy evaluation, where function calls and data constructions are performed only as needed to continue execution. As a result programs may be written which (at least conceptually) manipulate infinite data structures, streams, etc, and their behavior can be analyzed by the methods to be developed. The framework allows recursively defined functions in "curried" form, where higher order ("functional") values are obtained by applying functions to incomplete argument lists.

[1] Term rewriting gives a convenient framework to express higher order functional programs' semantics and flow analysis, but we neither use nor prove nontrivial results from term rewriting.

Section 3 reviews an earlier and simpler construction of a regular tree grammar G from a given program, combining notions of Reynolds [27] and Jones and Muchnick [20]. Characteristic of G is that its nonterminals generate supersets of the computational states, function inputs and function outputs computed by the program on given data. Applications include program optimization, transformation, etc. The construction of Section 3 can be proven to give safe analyses of programs using call-by-value; however it can be unsafe when applied to lazy programs, and cannot deal with higher order functions.

In Section 4 a new algorithm is given to construct tree grammars from programs, able to safely approximate the behavior of lazy programs with higher-order functions. Proofs of correctness and termination of the new construction are given in Section 5. Section 6 contains conclusions, directions for further work and acknowledgements.

1.2 Relation to other work

Until 1987

Broad overviews of program flow analysis may be found in Hecht [12] and Muchnick and Jones [25].

The first paper with goals similar to ours was Reynolds [27], which analyzed LISP programs to obtain "data set definitions" in the form of (equationally defined) regular sets of trees approximating inputs to and the results of the functions in a given first order LISP program. The mathematically equivalent tree grammars were used in [20] in order to approximate the behavior of flow chart programs manipulating LISP-like data structures.

Turchin independently developed a method using "covering context free grammars" to analyze programs in the language REFAL (essentially Markov algorithms equipped with variables, see [33]). His techniques require some rather sophisticated machinery, in particular the use of so-called supercompilation and metasystem transition. Our results extend those of Reynolds and Turchin in two directions: we allow delayed evaluation and higher order functions; and the method described here is conceptually simpler than Turchin's.

Burn, Hughes, Wadler and coauthors [5,15,35] develop methods for strictness analysis of programs with structured data, i.e., ways to describe access patterns to subtrees of tree-structured data, for use in efficient compilation of a lazy language. Their type-based goals and methods differ from this paper's.

Other work concerning higher order functions

Our method is independent of the program's type structure and so can handle the type-free lambda calculus. In addition it can do analyses such as "constant propagation" which would be difficult by type-based methods due to the need for an infinitely wide approximation lattice.

A rather complicated way to flow analyze lambda expressions (and the first way, to our knowledge) was described in [18]. The present techniques extend and simplify those methods and appear to be at least as strong.

This paper's theoretical basis is similar to that of the minimal function graphs in [21], but applies to higher order programs with structured data. In a different paper by the same authors [26], a simple and general framework for abstract interpretation of lambda expressions is described, and an important distinction is drawn: between questions that concern "global" program behavior (e.g., type analysis) and those that concern "local" behavior (e.g., what is the range of values a given variable may assume at a particular point in the program?). The framework of [26] applies only to global questions and takes no account of structured data, while it will be seen that the current (quite different and more operational) techniques naturally yield local flow information, and can as well handle tree-structured data.

A standard trick for implementing functional values is to use *closures*, where a closure consists of a function name and the values of some of its parameters. The "lambda-lifting" transformation, i.e., the idea to make the apply function explicit, and to replace function parameters by explicit names for functions, does this at source code level. It has roots in Reynolds' defunctionalization [28].

Since 1987

Since [19] (this paper's predecessor) much research has been done on flow analysis of higher order programs, including the following. Untyped functional languages: Shivers' Ph.D. thesis [29] focused on control-flow analysis (CFA), and Heintze's Ph.D. thesis [13] focused on sets of data values (set-based analysis, at the start essentially a constraint-based version of regular tree grammars). Both of these have close connections with this paper's results, and each has led to several papers and much further research, e.g., Jagannathan and Weeks [16], and Stefanescu and Zhou [30]. A "minimal function graph" approach using higher order functions at analysis time may be seen in [22].

Typed functional languages: Xi's Ph.D. thesis focused on tracing value flow via data types for termination verification of higher order programming languages [36], and Abel has another approach to termination checking with higher order types and recursive data structures [1].

The term rewriting community has begun to study higher order term rewriting systems for their own sake, including flow analysis aspects. Research in this direction has been done by Giesl at.al. [11], Toyama [32] and several others.

Forward or backward analyses?

This somewhat technical subsection may be skipped on first reading.

The methods of [27,20,35] and this paper correspond to Cousot's "forward analyses" [6]. They begin with a description of program inputs and derive from these an approximation to the set of all program states reachable during computations on the given input data. In this context a "safe approximation" requires accounting for all reachable computational states.

On the other hand, several including Hughes [15] and Turchin [33] have used various forms of "backwards analysis" that extract from the program alone, without input description, the form of the input data that can drive it. In this context "safety" has a different flavor: recognition of all possible patterns of data access, including "must be evaluated" and "may be evaluated". This approach can be used for strictness analysis.

The precise relation between the two approaches is not yet entirely clear, though there are certainly connections with Dijkstra's strongest postconditions and weakest preconditions [8]. Forward analyses seem more relevant to this paper, for at least three reasons:

(1) We are interested in higher order functions. These are hard to handle in a backwards analysis; for example, assuming a function-valued argument to range over all possible functions will yield far too conservative results. In particular it seems hard to formulate natural preconditions on program inputs when they include functions.
(2) An intended application is partial evaluation. In this case one starts with the precise values of some (typically not all) program inputs. It is most natural to propagate this information forwards.
(3) The result of analysis by either approach is a description of the program's data in the form of a grammar (or something similar), which closely resembles a "data type" for structured data. If one accepts the motto that *well-typed programs can't go wrong* then forward analysis seems more relevant since its safety concept stresses accounting for all reachable computational states; if the analysis cannot reveal the possibility of an error such as `3+true` then that error cannot possibly occur. Forward methods thus seem to provide a more natural

framework for type checking.

2 A functional language

The syntax and operational semantics of the functional language discussed in the rest of the paper are defined using terminology from the theory of term rewriting systems. For simplicity of notation we assume only one data sort; the paper's results extend to the many-sorted case without difficulty.

2.1 Programs: syntax and operational semantics

A *signature* is a set Σ furnished with a function $\mathit{arity} : \Sigma \to I\!N_0$. $\Sigma_n = \mathit{arity}^{-1}(n)$ is the set of *n-ary operators.* Operators of arity 0 are called *constants.* We assume the signature is partitioned into $\Sigma = \Gamma + \Delta$ (where $+$ is disjoint union) and call the operators in Γ *constructors* and those in Δ, *defined operators* [2].

Let $T_\Sigma(V)$ be the set of terms t over operators in Σ and a denumerable set V of *variables* (disjoint from $\cup\Sigma_i$). Thus $T_\Sigma(V)$ is the smallest set containing V such that if $t_1, \ldots, t_n \in T_\Sigma(V)$ and $n = \mathit{arity}(op)$ then $op\, t_1 \cdots t_n \in T_\Sigma(V)$.

Terms in T_Σ (without variables) are also called *ground terms*, and terms in T_Γ, built from constructors only, are *ground constructor terms.* A *call* is a term of form $\delta t_1 \cdots t_n$ with $\delta \in \Delta$. We denote elements of $T_\Sigma(V)$ by t (possibly decorated); elements of T_Σ by g (for "ground term"); and elements of T_Γ by c (for "constructor term"). The set of variables occurring in a term t is denoted by $\mathit{Vars}(t)$.

Definition 2.1 A *term rewriting system over* Σ is a set of rewrite rules $\{p_i \to t_i\}_{i \in I}$ where I is an index set (usually finite), left and right sides $p_i, t_i \in T_\Sigma(V)$, and each t_i is a term using only variables from the left side p_i. A *program over* Σ is a term rewriting system such that each "pattern" p_i is a call in which no variable occurs more than once.

Operational semantics

Informally: a term rewriting system defines a relation $\Rightarrow$ over its set of ground terms, and computation may be thought of as the transitive closure of $\Rightarrow$ (note that it is not necessarily deterministic). The constraints on p_i, t_i in programs are computationally

[2] Defined operators correspond to the functions that are defined by a functional program.

motivated: The constraint that p_i be a call ensures that once a constructor appears outside the scope of any defined operator, then it cannot be further rewritten; the variable name constraints ensure that the rewritten term can be built from t_i using only the bindings got from matching with variables in p_i, and that equality must be tested for explicitly. More precisely:

A *context* $t[\,]$ is a "term with a hole", i.e., a term from $T_\Sigma(V \cup \{[\,]\})$ containing just one occurrence of $[\,]$ (a 0-ary operator). An occurrence of term t' in context $t[\,]$ is a pair $(t[\,], t')$. More compactly, we write this as $t[t']$, and also (ambiguously) use $t[t']$ to denote the term obtained by replacing the $[]$ in $t[]$ by t'. (In the examples, square brackets are also used in list denotations; hopefully, this doesn't lead to confusion.)

A *substitution* is a function $\theta : V \to T_\Sigma(V)$. We do not differentiate between a substitution and its natural extension to terms, $\theta : T_\Sigma(V) \to T_\Sigma(V)$. θ_{id} denotes the identity mapping on terms.

Definition 2.2 The *derivation relation* $\Rightarrow\; \subseteq\; T_\Sigma \times T_\Sigma$ is defined by:

$$t[\theta p_i] \Rightarrow t[\theta t_i]$$

for any ground context $t[\,]$, rule $p_i \to t_i$ and ground substitution θ. The relation $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$. A term in T_Γ is said to be *in normal form.*

Note that if c is in normal form there is no term c' such that $c \Rightarrow c'$. For programming languages it is desirable that the normal forms derived from initial calls are unique when they exist. A sufficient condition for uniqueness is the *Church-Rosser property*: if $t \Rightarrow t'$ and $t \Rightarrow t''$ then there exists t''' with $t' \Rightarrow^* t'''$ and $t'' \Rightarrow^* t'''$.

A *unifier* of two terms t, t' is a substitution θ such that $\theta t = \theta t'$. It is well known that if two terms are unifiable, then there exists a *most general unifier* $mgu(t, t')$, such that any unifier of t, t' is a refinement of $mgu(t, t')$. There exists a computable function $mgu : T \times T \to \mathit{Subst} \cup \{\mathit{fail}\}$ (where *Subst* is the set of all substitutions) such that:

$$mgu(t,t') = \text{ a most general unifier of } t, t' \textbf{ if } \text{they are unifiable, } \textbf{else } \mathit{fail}$$

We use only the special case of *matching*, where t' is a ground term. It is easy to see that if distinct rules in a program have nonunifiable, left sides, then the Church-Rosser property holds (weaker conditions will also suffice).

2.2 Examples and special cases

The following two list-manipulating programs use the convention of writing *nil* or [] for the empty list, "cons" as an infix list constructor written :, and $[t_1, t_2, \ldots, t_n]$ to stand for the list $t_1 : (t_2 : (\ldots : (t_n : \mathit{nil}) \ldots))$. Variable names begin with upper case letters to make it easier to distinguish them from constants or operators (an idea taken from Prolog).

The function *append* is a well-known example; Reynold's function *ss* from [27] yields the list of all subsets of $\{x_1, \ldots, x_n\}$ when given a list $[x_1, \ldots, x_n]$ of distinct atoms.

Signature Σ*:*

$$\Gamma_0 = \{\mathit{nil}, 0, 1, \ldots\}, \Gamma_1 = \{\}, \Gamma_2 = \{:\}, \Delta_1 = \{\mathit{ss}\}, \Delta_2 = \{\mathit{append}\}, \Delta_3 = \{\mathit{aux}\}$$

Rules:

$$\begin{array}{ll}
\mathit{append}(\mathit{nil}, \mathit{Xs}) & \rightarrow \mathit{Xs} \\
\mathit{append}(X : \mathit{Xs}, \mathit{Ys}) & \rightarrow X : \mathit{append}(\mathit{Xs}, \mathit{Ys}) \\
 & \\
\mathit{ss}(\mathit{nil}) & \rightarrow [\mathit{nil}] \\
\mathit{ss}(U : V) & \rightarrow \mathit{aux}(U, \mathit{ss}(V), \mathit{ss}(V)) \\
\mathit{aux}(W, \mathit{nil}, Z) & \rightarrow Z \\
\mathit{aux}(W, X : Y, Z) & \rightarrow (W : X) : \mathit{aux}(W, Y, Z)
\end{array}$$

Inside-out versus outside-in

Consider the following program. Given input n, it produces the first n elements of the infinite list: $[\mathit{nil}, [1], [1, 1], [1, 1, 1], \cdots]$. Here n is given as a list of n ones, so for example $g([1, 1, 1]) = [\mathit{nil}, [1], [1, 1]]$.

$$\begin{array}{ll}
g(N) & \rightarrow \mathit{first}(N, \mathit{sequence}(\mathit{nil})) \\
\mathit{first}(\mathit{nil}, \mathit{Xs}) & \rightarrow \mathit{nil} \\
\mathit{first}((1 : M), (X : \mathit{Xs})) & \rightarrow X : \mathit{first}(M, \mathit{Xs}) \\
\mathit{sequence}(Y) & \rightarrow Y : \mathit{sequence}(1 : Y)
\end{array}$$

A computation of $g([1,1])$:

$$\begin{aligned} g([1,1]) &\Rightarrow \mathit{first}([1,1], \mathit{sequence}(\mathit{nil})) \\ &\Rightarrow \mathit{first}([1,1], \mathit{nil} : \mathit{sequence}([1])) \\ &\Rightarrow \mathit{nil} : \mathit{first}([1], \mathit{sequence}([1])) \\ &\Rightarrow \mathit{nil} : \mathit{first}([1], [1] : \mathit{sequence}([1,1])) \\ &\Rightarrow \mathit{nil} : [1] : \mathit{first}(\mathit{nil}, \mathit{sequence}([1,1])) \\ &\Rightarrow \mathit{nil} : [1] : \mathit{nil} \\ &= [\mathit{nil}, [1]] \end{aligned}$$

The point of this example is that programs may be written which one thinks of as using infinite data structures, streams etc, even though no such concepts are explicitly present in the term rewriting framework. Note that this program's termination requires use of an "outside-in" evaluation strategy; it will loop infinitely if calls are done "by value", i.e., inside-out. These terms are now defined precisely:

Definition 2.3 A *redex* is an occurrence of a term θp_i in $t[\theta p_i]$ where $p_i \to t_i$ is a rule and θ is a substitution. It is called *innermost* if θp_i contains no other redexes, and *outermost* if the occurrence $[\theta p_i]$ is contained in no other redex.

The *innermost* and *outermost derivation relations* $\underset{io}{\Rightarrow}$, $\underset{oi}{\Rightarrow}$ $\subseteq T_\Sigma \times T_\Sigma$ are defined by: $t[\theta p_i] \underset{io}{\Rightarrow} t[\theta t_i]$, provided $[\theta p_i]$ is innermost, and $t[\theta p_i] \underset{oi}{\Rightarrow} t[\theta t_i]$, provided $[\theta p_i]$ is outermost.

Outside-in evaluation is obtained by using $\underset{oi}{\Rightarrow}$ instead of $\Rightarrow$, and similarly for inside-out. Clearly $\underset{io}{\Rightarrow}$ and $\underset{oi}{\Rightarrow}$ coincide with $\Rightarrow$ on programs without nested calls such as $g(\cdots, f(--), \cdots)$. Inside-out evaluation has been called: *call-by-value* or *applicative order*, and outside-in evaluation has been called: *call-by-name* or *normal order* evaluation.

We have declined to give a definition of "lazy evaluation" since there seem to be some disagreements as to just what it is (e.g., what is "maximally lazy?"), and it is not entirely clear how the memoization concept should be expressed in the context of term rewriting systems.

One natural suggestion: equate lazy evaluation with *leftmost outside-in* evaluation. This yields the computation above, and has the natural "pattern-directed" character often seen (e.g., [34]). On the other hand there are objections: the scheme is still nondeterministic; it can reduce expressions that were interior to a λ in the original lambda expression and so does not yield the usual "head normal form"; and there is a potential need to scan the entire term in order to discover the next redex (although schemes to reduce such overhead have been suggested, e.g., [14]).

Finally, an observation: it won't matter that we have no iron clad definition of "lazy", since the grammar to be constructed will safely approximate all reachable computational states, regardless of evaluation order. Further, an alternative will be proposed that traces just the outside-in derivations.

Higher order functions

In the popular "named combinator" style for writing functional programs (HASKELL, etc.), every function f is thought of as "curried", and a definition might be an equation of form: $f x_1 \ldots x_n = \mathit{expression}$. An incomplete function application (such as $f e_1 \ldots e_m$ with $m < n$) thus has a mathematical function as its denotation.

This program style is easily expressed in a term rewriting context by modelling a standard implementation technique. Traditionally, a functional value is represented by a *closure* of the form $(\lambda x.\mathit{exp}, v_1, \ldots, v_n)$ where $v_1, \ldots, v_n$ is a list of the values of the free variables in $\lambda x.\mathit{exp}$. Lambda-lifting [17] converts $\lambda x.\mathit{exp}$ to a function (say f) of those free variables, so a closure becomes an *incompletely applied function*: $(\ldots((f v_1) v_2) \ldots v_n)$.

From this viewpoint, a function name such as f is regarded not as a defined operator but as a constant, and closures will be formed using a binary *closure-forming operator* @.

Following is a traditional example from LISP with defined operator $\Delta_2 = \{@\}$ and constructors $\Gamma_0 = \{\mathit{map}, \mathit{double}, \mathit{cons}, f, \mathit{nil}, 0, 1, \ldots\}$ and $\Gamma_2 = \{:\}$.

The operator @ (left associating infix) is used for function application. Function *map* as usual applies a functional argument to a list, so $\mathit{map} \mathbin{@} g \mathbin{@} [x_1, \ldots, x_n]$ evaluates to $[g(x_1), \ldots, g(x_n)]$. Function f below thus transforms $[x_1, \ldots, x_n]$ into:

$$[(x_1 : x_1), \ldots, (x_n : x_n)] : [(5 : x_1), \ldots, (5 : x_n)]$$

Rules:

$$\begin{array}{ll}
f \mathbin{@} X & \to (\mathit{map} \mathbin{@} \mathit{double} \mathbin{@} X) : (\mathit{map} \mathbin{@} (\mathit{cons} \mathbin{@} 5) \mathbin{@} X) \\
 & \\
\mathit{double} \mathbin{@} X & \to X : X \\
\mathit{cons} \mathbin{@} X \mathbin{@} Y & \to X : Y \\
 & \\
\mathit{map} \mathbin{@} U \mathbin{@} \mathit{nil} & \to \mathit{nil} \\
\mathit{map} \mathbin{@} U \mathbin{@} (X : \mathit{Xs}) & \to (U \mathbin{@} X) : (\mathit{map} \mathbin{@} U \mathbin{@} \mathit{Xs})
\end{array}$$

If we write $t_1 t_2 \ldots t_n$ as syntactic sugar for $(\ldots((t_1 \mathbin{@} t_2) \mathbin{@} t_3) \ldots)$ then the equations above can be written without @, giving a more familiar appearance.

The effect of higher order functions is thus achieved without any change to the syntax or semantics of term rewriting systems. Further, it may be argued that such a representation is entirely relevant for the purpose of analyzing program behavior, since closures are the standard tool for implementing higher order functions on the computer.

2.3 Tree grammars

The technique to be presented constructs from a given program a tree grammar that safely approximates the collecting semantics of the next section and so can be used reliably to answer questions about program behavior. In contrast to programs intended for computational use, a tree grammar is nondeterministic and has no variables.

Definition 2.4 A *tree grammar* is a program, G, in which all defined operators are 0-ary. A set $A \subseteq T_\Gamma$ of ground constructor terms is *regular* iff there is a tree grammar G and a defined operator S such that

$$A = \{ \text{ ground constructor term } g \mid S \Rightarrow^* g \}$$

In terms of traditional formal language theory, the 0-ary defined operators (i.e., constants) correspond exactly to nonterminal symbols, and will be so called in the following development. Further, constructors correspond to terminal symbols.

Tree grammars are chosen partly because they are able directly to generate the terms that are of computational interest, and partly because of their similarity to the well-understood regular string grammars, with all their desirable mathematical properties. Numerous theorems, constructions, decision procedures and alternate characterizations of regular sets of terms are described in the literature, e.g., by Brainerd [4], Gecseg and Steinby [10], and by Thatcher [31].

Notation. We will follow the grammatical convention of writing

$$A \to \mathit{term}_1 \mid \mathit{term}_2 \mid \ldots \mid \mathit{term}_n$$

to stand for the several rules $A \to \mathit{term}_1, \ldots, A \to \mathit{term}_n$

A classical approach to program analysis begins with an exact *collecting semantics* [7] that (for flow charts) accumulates with each "program point" the set of program states attainable when control reaches that point during all possible program executions on given initial input data. The next step is to find a finite over-approximation to the collecting semantics.

For programs in the form of term rewriting systems, the natural program points are the rewrite rules. By analogy with flow charts, a natural candidate for a "program state" at a rule $p_i \to t_i$ is a *substitution* $\theta : V \to T_\Sigma$, namely the variable bindings got by matching with p_i at some time during a computation.

On the other hand we also need to account for program *results*, namely terms derived from t_i during computations on given input. Our solution is to associate with rule i a set of pairs (θ, g), where θ is a substitution resulting from a successful match with p_i, and g is a term derivable from θt_i. (This is analogous to the "minimal function graphs" of [21], extended to allow nondeterministic programs.)

Definition 2.5 Let *Input* $\subseteq T_\Sigma$ be a set of input terms (typically calls), and let $P = \{p_i \to t_i\}_{i \in \{1,\dots,n\}}$. The *collecting semantics* is the tuple

$$\mathit{Colsem}(\mathit{Input}) = (Z_0, Z_1, \dots, Z_n)$$

where for $i = 0, 1, \dots, n$, set $Z_i \subseteq \mathit{Subst} \times T_\Sigma$ is given by:

$$\begin{aligned} Z_0 &= \{(\theta_{id}, g) \mid \exists g_0 \in \mathit{Input}\ (g_0 \Rightarrow^* g)\} \\ Z_i &= \{(\theta, g) \mid \exists g_0 \in \mathit{Input}\ \exists g_1[\,]\ (g_0 \Rightarrow^* g_1[\theta p_i] \text{ and } \theta t_i \Rightarrow^* g)\} \end{aligned}$$

Note: terms g in $(\theta, g) \in Z_i$ include *all* terms derivable from θt_i, and not just those in normal form.

Condition 1 in the following lemma correponds to Z_0, and conditions 2 and 3 are closure properties relating the various Z_k. Condition 2 says that any rewriteable Z_j context may be rewritten and placed in Z_i. Condition 3 says that *any* such Z_j context seen in Z_i must have come from Z_j.

Lemma 2.6 The collecting semantics is the smallest tuple of sets of pairs (ordered componentwise) that satisfies the following conditions.

(1) If $g_0 \in \mathit{Input}$ then $(\theta_{id}, g_0) \in Z_0$
(2) If $1 \le i \le n$ and $0 \le j \le n$ satisfy $(\theta, g[\theta' p_i]) \in Z_j$, then $(\theta', \theta' t_i) \in Z_i$
(3) If $1 \le i \le n$ and $0 \le j \le n$ satisfy $(\theta, g[\theta' p_i]) \in Z_j$,
 and $(\theta', g'') \in Z_i$, then $(\theta, g[g'']) \in Z_j$

Proof of the lemma is straightforward.

2.5 Approximating the collecting semantics by regular tree grammars

The flow analysis methods of Sections 3 and 4 will build regular tree grammars G that approximate the collecting semantics. Grammar G will have a nonterminal X for each variable X, and some "result nonterminals" R_i. The tree grammars will *safely approximate* P's behavior in the following sense: for any $(\theta, g) \in Z_i$, grammar G has derivations $X \Rightarrow^* \theta X$[3] and $R_i \Rightarrow^* g$.

Informally, X generates all possible ground terms bound to X in computations on the given input, and R_i generates all possible results derivable from p_i during the computations. In case the same variable X is used in several rules, nonterminal X generates the union of all the values bound to variable X.

3 Program Flow Analysis by Tree Grammars: Earlier Results

In 1968 Reynolds developed a method for analysis of applicative LISP programs, using a system of set equations derived from the program when applied to a pre-specified set of input data [27]. The variables in the least fixpoint solution of these equations represent the sets of values that can be *assumed by program variables* or *returned by program functions* (note the analogy with our collecting semantics). The method works by first deriving an equation system containing variables, sets of atoms and the operations: set union, $car*$, $cdr*$ and $cons*$ (the extensions of LISP's car, cdr, $cons$ to sets of lists, e.g., $car*(X) = \{car(x) \mid x \in X\}$) and a conditional expressed in terms of sets. This equation system is then simplified and transformed into an equivalent system without $car*$, $cdr*$, etc by applying various set identities. The final result is a definition of a collection of regular sets of LISP lists.

In [20] the analogous problem for flow-charts with LISP-like data is solved by a method similar to Reynolds', using *extended tree grammars* instead of set equations. The effect of Reynolds' operations $car*$, $cdr*$, etc is achieved by use of production rules with *selectors*; for example an extended production rule may take the form:

$$A \rightarrow B.hd$$

[3] Note the two different meanings of X in the rule $X \Rightarrow^* \theta X$: On the left side, X is the new nonterminal; on the right side, X is the term rewriting system variable to which substitution θ can be applied.

where a rule $A \rightarrow B.hd$ is interpreted: if B derives a term $value_1 : value_2$, then A derives $value_1$, and similarly for *tl* (hd = head, tl = tail). It is then shown that extended rules involving selectors may be eliminated, yielding an ordinary tree grammar without selectors.

3.1 Inside-out Program Analysis by Tree Grammars

In the following we briefly illustrate a hybrid of the methods of [20,27], adapted to programs in the form of term rewriting systems. While sound for call-by-value (inside-out evaluation), the approach will be seen unsound for outside-in evaluation. Further, it is inadequate for higher order programs. Some readers may wish to skip ahead to Section 4, where a new and more powerful technique will be presented for deriving tree grammars.

Let $P = \{p_i \rightarrow t_i\}_{i \in \{1,2,\ldots,n\}}$ be a fixed finite program over signature $\Sigma = \Delta + \Gamma$ and let $Input \subseteq T_\Sigma$. The idea is to construct from program P a tree grammar G that approximates the collecting semantics on given initial input data.

Tree grammar G has signature $\Sigma' = \Delta' + \Gamma$. Constructors in Γ are used the same way in G and P, and every left side variable and every defined operator in P is a nonterminal of G (0-ary!). Special nonterminals Ω (perhaps together with others), will generate possible input terms, and R_0 will generate possible program results. ($R_1, \ldots, R_n$ are not used here, but will be seen in Section 4.) The defined operator (nonterminal) set Δ' thus satisfies:

$$\Delta' \supseteq \{R_0, \Omega\} \cup \Delta \cup \{X \mid X \in V \text{ occurs in some } p_i\}$$

In order to avoid confusion between P's and G's rewrite rules we will write the i-th rule of the former as: $p_i \underset{P}{\rightarrow} t_i$ and rules of the latter as: $A \underset{G}{\rightarrow} t$. The derivation relation $\Rightarrow^*$ for G and P are respectively written: $\underset{P}{\Longrightarrow}^*$ and $\underset{G}{\Longrightarrow}^*$.

An example from [27]. Consider the subset program given before:

$$\begin{array}{lll}
ss(nil) & \underset{P}{\rightarrow} & [nil] \\
ss(U : V) & \underset{P}{\rightarrow} & aux(U, ss(V), ss(V)) \\
aux(W, nil, Z) & \underset{P}{\rightarrow} & Z \\
aux(W, X : Y, Z) & \underset{P}{\rightarrow} & (W : X) : aux(W, Y, Z)
\end{array}$$

We begin by building an extended tree grammar G with nonterminal symbols: ss, aux, and all program variables, and rules generating (from R_0) initial calls to ss. G is then transformed to remove rules with selectors. The grammar "safely" approximates inside-out derivations in the following sense.

(1) Nonterminal *ss* generates a superset of all result values (ground constructor terms) computed by calls to the defined function *ss* on the given input; and similarly for nonterminal *aux* and function *aux*.
(2) W generates a superset of all the values assumed by *aux*'s first argument on the given data. Similarly Z approximates *aux*'s second argument, U approximates the set of heads of values assumed by *ss*'s argument, and similarly for V, X, Y.

Step 1: build a tree grammar (extended with selectors)

The initial call is $ss(\Omega)$ where Ω generates all possible lists of (LISP) atoms:

$$\begin{array}{lll} R_0 & \underset{G}{\rightarrow} & ss \\ \Omega & \underset{G}{\rightarrow} & nil \mid Atom : \Omega \\ Atom & \underset{G}{\rightarrow} & 0 \mid 1 \mid \ldots \end{array}$$

Rules generating sets of results are obtained by copying P's rules but without the defined functions' arguments:

$$\begin{array}{lll} ss & \underset{G}{\rightarrow} & [nil] \mid aux \\ aux & \underset{G}{\rightarrow} & Z \mid (W : X) : aux \end{array}$$

Rules generating sets of argument values are obtained from the calls $ss(V)$ by applying selectors $.hd$ and $.tl$ as needed for argument subterms U, V:

$$\begin{array}{lll} U & \underset{G}{\rightarrow} & V.hd \\ V & \underset{G}{\rightarrow} & V.tl \end{array}$$

The same is done for all calls to *ss* and *aux*, including the initial call $ss(\Omega)$:

$$\begin{array}{lll} U & \underset{G}{\rightarrow} & \Omega.hd \\ V & \underset{G}{\rightarrow} & \Omega.tl \\ W & \underset{G}{\rightarrow} & U \mid W \\ X & \underset{G}{\rightarrow} & ss.hd \mid Y.hd \\ Y & \underset{G}{\rightarrow} & ss.tl \mid Y.tl \\ Z & \underset{G}{\rightarrow} & ss \mid Z \end{array}$$

Step 2: eliminate rules containing selectors

Consider a selector rule such as $U \underset{G}{\rightarrow} \Omega.hd$. This can be replaced by the rule $U \underset{G}{\rightarrow} Atom$. The reasoning (described in [20]) is that the grammar has a production $\Omega \underset{G}{\rightarrow} Atom : \Omega$ that generates a value pair $Atom : \Omega$. This implies that $\Omega.hd$ can generate the first component of the pair, so production $U \underset{G}{\rightarrow} \Omega.hd$ can be "short circuited", replacing it by the selector-free rule $U \underset{G}{\rightarrow} Atom$.

The same reasoning can be used to eliminate selectors for V, X, Y, yielding a classical selector-free grammar.

Step 3: simplify the remaining rules

Note that *ss* and *aux* generate the same terms, since each yields all terms generated by the other. Further, $[nil] = nil : nil$ so the rules can be simplified a bit further, to yield a reasonably informative description of the program's argument and result values:

$$\begin{array}{lll} R_0 & \underset{G}{\rightarrow} & ss \\ ss & \underset{G}{\rightarrow} & [nil] \mid (Atom : X) : ss \\ X & \underset{G}{\rightarrow} & nil \mid Atom : X \end{array}$$

3.2 Limitations of the method

While sound for call-by-value (inside-out) evaluation, the method above gives unsafe approximations for the *outside-in* behavior of some programs. Consider the example with infinite sequences given earlier.

$$\begin{array}{lll} g(N) & \underset{P}{\rightarrow} & first(N, sequence(nil)) \\ first(nil, Xs) & \underset{P}{\rightarrow} & nil \\ first((1 : M), (Y : Ys)) & \underset{P}{\rightarrow} & Y : first(M, Ys) \\ sequence(Z) & \underset{P}{\rightarrow} & Z : sequence(1 : Z) \end{array}$$

Assuming initial call $g(\Omega)$, the resulting tree grammar contains result rules:

$$\begin{array}{lll} Start & \underset{G}{\rightarrow} & g \\ g & \underset{G}{\rightarrow} & first \\ first & \underset{G}{\rightarrow} & nil \mid Y : first \\ sequence & \underset{G}{\rightarrow} & Z : sequence \end{array}$$

and argument rules:

$$\begin{array}{lll}
\Omega & \xrightarrow[G]{} & \mathit{nil} \mid \mathit{Atom} : \Omega \\
N & \xrightarrow[G]{} & \Omega \\
\mathit{Xs} & \xrightarrow[G]{} & \mathit{sequence} \mid \mathit{Ys} \\
M & \xrightarrow[G]{} & N.\mathit{tl} \mid M.\mathit{tl} \\
Y & \xrightarrow[G]{} & \mathit{sequence}.\mathit{hd} \mid \mathit{Ys}.\mathit{hd} \\
\mathit{Ys} & \xrightarrow[G]{} & \mathit{sequence}.\mathit{tl} \mid \mathit{Ys}.\mathit{tl} \\
Z & \xrightarrow[G]{} & \mathit{nil} \mid 1 : Z
\end{array}$$

The only rule rewriting *sequence* is $\mathit{sequence} \xrightarrow[G]{} Z : \mathit{sequence}$, so clearly *sequence* and $\mathit{Xs}, Y, \mathit{Ys}$ all generate the empty set. Thus the tree grammar does not account for the program's possible computations. The underlying reason is that the method just given describes each program part by the set of *ground* constructor terms it can evaluate to, and $\mathit{sequence}(\mathit{nil})$ yields no such terms at all as values.

The method above also seems hard to extend to higher order functions (a "potential extension" in [27]).

4 Program flow analysis by tree grammars: a new algorithm

In this section an alternate tree grammar construction is given that yields safe approximations for all programs, even with outside-in semantics. Correctness and termination will be proven in Section 5.

The main difference is a more detailed tracking of the results of applying the rewrite rules. Further, the tricky "selector elimination" phase of Section 4 will be circumvented by better exploiting the pattern-matching nature of rewrite rules.

The idea is to approximate the collecting semantics by means of a tree grammar G constructed from program $P = \{p_i \xrightarrow[P]{} t_i\}_{i\in I}$, where $I = \{1, 2, \ldots, n\}$. As in Section 3, we regard P's program variables as 0-ary defined operators (nonterminals) in G. The set Δ' of nonterminals of G will satisfy

$$\Delta' \supseteq \{X \mid X \in \mathit{Vars}(p_i) \text{ and } 1 \le i \le n\} \cup \{R_0, R_1, \ldots, R_n\}$$

Following the structure of Lemma 2.6, R_i (read "the result of rule p_i") is intended to generate a superset of all the terms derived from left side p_i in computations on the given input. As before, R_0 generates those terms derived from initial calls to P (perhaps with the aid of other nonterminals in Δ').

If P has signature Σ then G has signature $\Sigma' = \Sigma + \Delta'$, where defined operators in P are regarded as constructors in G; this is needed so that G can generate the "result sets" R_i of the collecting semantics.[4]

4.1 Safety, and a preview of the algorithm

G's grammar rules may be divided into three groups:

Input rules of form $R_0 \underset{G}{\rightarrow} \ldots$ and auxiliary rules (such as $\Omega \underset{G}{\rightarrow} \ldots$) generate a set of initial calls to P.

Variable rules of form $X \underset{G}{\rightarrow} \ldots$ generate values bound to P-variable X on the given inputs. A value may now be *any* ground term, not just ground constructor terms as for the inside-out derivations of Section 3.

Result rules of form $R_i \underset{G}{\rightarrow} \ldots$ yield possible values derived from p_i during P's computations on the given initial calls.

Safety with respect to a given set of input terms is defined using the collecting semantics $\mathit{Colsem}(\mathit{Input}) = (Z_0, \ldots, Z_n)$ as follows.

Definition 4.1 G is *safe* with respect to P and $\mathit{Input} \subseteq T_\Sigma$ if for all $i = 0, \ldots, n$

(1) $R_i \underset{G}{\Rightarrow}^* g$ for all $(\theta, g) \in Z_i$
(2) $X \underset{G}{\Rightarrow}^* \theta X$ for all $(\theta, g) \in Z_i$

As a consequence R_0 generates all program results, including intermediate terms. Condition 2 implies a rather gross approximation, since G is an "independent attribute" method: it neglects all coordination among the variable bindings, and merges all bindings of the same variable.(Nonetheless it seems to work well in practice.)

G will be constructed iteratively. We begin with $G = G_0$ containing only initial rules $R_0 \underset{G}{\rightarrow} \ldots$ etc. We then progressively add rules to G until it becomes safe with respect to P. Concisely and formally, the construction is defined by the fixpoint process in Section 4.3. For intuition's sake a more algorithmic presentation is given first, with remarks. The reader may wish to review Lemma 2.6, since the algorithm closely models that construction, representing the sets R_i by grammar rules as in Definition 4.1 of safety.

[4] Actually, in the remaining part of this paper, the distinction between constructors and defined operators is not needed. For the construction to work, a program could be any set $P \subseteq T_\Sigma(V) \times T_\Sigma(V)$ of rules $p \to t$ such that $\mathit{Vars}(p) \supseteq \mathit{Vars}(t)$.

4.2 Informal version: construction of G from P

```
Start with $G = G_0$;
**while** $G$ can be enlarged **do**
**begin**
   choose a $G$ rule $A \underset{G}{\rightarrow} g[g']$ where $g'$ is a call, and a $P$ rule $p_i \underset{P}{\rightarrow} t_i$;
   **loop**
      unify $g'$ against $p_i$;
      **if** there is an operator clash between $g'$ and $p_i$
        **then** *fail* and escape the loop;
      **if** unification succeeds
        **then** escape the loop;
      **if** some nonterminal $N$ in call $g'$ matches an operator in left side $p_i$
        **then** rewrite that $N$ in $g'$ by some $G$-rule $N \underset{G}{\rightarrow} g''$ and repeat the loop
   **end loop**;

   **if** unification succeeded with substitution $\theta : Vars(p_i) \rightarrow T_{\Sigma'}$
   **then** add to $G$ the new rules:
      $A \underset{G}{\rightarrow} g[R_i]$
      $R_i \underset{G}{\rightarrow} t_i$
      $X \underset{G}{\rightarrow} \theta(X)$ for all $X \in Vars(t_i)$
**end**
```

Remarks

(1) The first idea is to determine for each right side P-call g' in a G-rule $A \underset{G}{\rightarrow} g[g']$ the set of P's rules with which g' may be matched[5]. The results of all possible calls are then used to construct G rules that model the corresponding bindings of P's variables to subterms of g'.

(2) Call applicability is determined by unifying P's left sides with g'. Note that g' is a term in $T_{\Sigma'}$, so that any X occurring in g' is a 0-ary defined operator. For unification purposes, g' is thus variable-free, unification is a one-sided "matching", and the outcome of unification will be either "*fail*" or a substitution $\theta : Vars(p_i) \rightarrow T_{\Sigma'}$.

(3) If unification succeeds, the bindings given by θ will be added to G as new rewrite rules for the variables[6] in the term rewriting rule $p_i \underset{P}{\rightarrow} t_i$. Further, the new rules: $A \underset{G}{\rightarrow} g[R_i]$ and $R_i \underset{G}{\rightarrow} t_i$ will be added to G to update the result sets it generates (note the close analogy with Lemma 2.6).

[5] and not *all* rules for the corresponding operator, as assumed in the method of Section 3.
[6] Since $Vars(p_i) \supseteq Vars(t_i)$ it is actually sufficient to add rules corresponding to the variables in t_i.

(4) Suppose g' cannot be matched with a P rule's left side due to the presence of an X or R_i. Then the X or R_i will be *rewritten by applying G's rules*, and the result will be checked for unifiability as above. This is appropriate by the intended interpretation of X and R_i (according to the definition of safety).
(5) Rewriting according to G is done *just enough* to allow unification. (This turns out to be essential to ensure termination, see Section 5.) The resulting rules are added to G (if new), and the process is repeated until G stabilizes.

4.3 Formal version: construction of G from P

$G =$ the smallest set of rules such that $G \supseteq G_0$ and

$$G \supseteq \mathit{Ext}_P(A \underset{G}{\rightarrow} g[g']) \text{ for every rule } A \underset{G}{\rightarrow} g[g'] \in G \text{ with } g' \text{ a call}$$

where

$$G_0 = \{R_0 \underset{G}{\rightarrow} \ldots \text{ (input rules of Section 4.1)}\}$$

and

$$\mathit{Ext}_P(A \underset{G}{\rightarrow} g[g']) = \left\{ \begin{array}{l|l} A \underset{G}{\rightarrow} g[R_i],\ R_i \underset{G}{\rightarrow} t_i, & p_i \underset{P}{\rightarrow} t_i,\ g' \underset{G}{\Longrightarrow}{}^{n}\ \theta p_i,\ \text{ and} \\ X \underset{G}{\rightarrow} \theta X \text{ for all } X \in \mathit{Vars}(t_i) & \neg\exists\theta'(g' \underset{G}{\Longrightarrow}{}^{n-1}\ \theta' p_i \underset{G}{\Rightarrow} \theta p_i) \end{array} \right\}$$

Remarks

(1) As in the informal algorithm, rules are added to G for each call $A \underset{G}{\rightarrow} g[g']$ already in G, and this process is iterated until G stabilizes.
(2) The definition of $\mathit{Ext}_P(A \underset{G}{\rightarrow} g[g'])$ formalizes the idea that g' is rewritten until a definite match with a left side of a P rule is obtained (the condition $g' \underset{G}{\Longrightarrow}{}^{n}\ \theta p_i$). The condition $\neg\exists\theta'(g' \underset{G}{\Longrightarrow}{}^{n-1}\ \theta' p_i \underset{G}{\Rightarrow} \theta p_i)$ expresses minimality: that rewriting is done only just enough for a definite match to occur.

4.4 Example: lazy evaluation

The given program is:

$$\begin{array}{lcl}
g(N) & \underset{P}{\rightarrow} & \mathit{first}(N, \mathit{sequence}(\mathit{nil})) \\
\mathit{first}(\mathit{nil}, \mathit{Xs}) & \underset{P}{\rightarrow} & \mathit{nil} \\
\mathit{first}((1 : M), (X : \mathit{Xs})) & \underset{P}{\rightarrow} & X : \mathit{first}(M, \mathit{Xs}) \\
\mathit{sequence}(Y) & \underset{P}{\rightarrow} & Y : \mathit{sequence}(1 : Y)
\end{array}$$

The initial rules in G are:

$$\begin{array}{lcll}
R_0 & \underset{G}{\rightarrow} & g(\Omega) & \text{Initial call} \\
\Omega & \underset{G}{\rightarrow} & \mathit{nil} \mid \mathit{Atom} : \Omega & \text{Initial data} \\
\mathit{Atom} & \underset{G}{\rightarrow} & 0 \mid 1 \mid \ldots &
\end{array}$$

Now apply Ext_P to $R_0 \underset{G}{\rightarrow} [g(\Omega)]$ (where $[\ldots]$ indicates an empty context around $\ldots$ and not a list). This yields (since $g(\Omega)$ can be matched with $p_1 \underset{P}{\rightarrow} t_1$):

$$\begin{array}{lcl}
R_0 & \underset{G}{\rightarrow} & R_1 \\
R_1 & \underset{G}{\rightarrow} & \mathit{first}(N, \mathit{sequence}(\mathit{nil})) \\
N & \underset{G}{\rightarrow} & \Omega
\end{array}$$

Ext_P applied to $R_1 \underset{G}{\rightarrow} \mathit{first}(N, [\mathit{sequence}(\mathit{nil})])$ adds rules:

$$\begin{array}{lcl}
R_1 & \underset{G}{\rightarrow} & \mathit{first}(N, R_4) \\
R_4 & \underset{G}{\rightarrow} & Y : \mathit{sequence}(1 : Y) \\
Y & \underset{G}{\rightarrow} & \mathit{nil}
\end{array}$$

and then Ext_P applied to $R_4 \underset{G}{\rightarrow} Y : [\mathit{sequence}(1 : Y)]$ adds:

$$\begin{array}{lcl}
R_4 & \underset{G}{\rightarrow} & Y : R_4 \\
Y & \underset{G}{\rightarrow} & 1 : Y
\end{array}$$

$\mathit{Ext}_P(R_1 \underset{G}{\rightarrow} [\mathit{first}(N, R_4)])$ now adds (after rewriting N to nil or $1 : \Omega$, and R_4 to $Y : R_4$ or $Y : \mathit{sequence}(1 : Y)$) the following new rules:

$$\begin{array}{lll}
R_1 & \underset{G}{\rightarrow} & R_2 \mid R_3 \\
R_2 & \underset{G}{\rightarrow} & \mathit{nil} \\
R_3 & \underset{G}{\rightarrow} & X : \mathit{first}(M, Xs) \\
M & \underset{G}{\rightarrow} & \Omega \\
X & \underset{G}{\rightarrow} & Y \\
Xs & \underset{G}{\rightarrow} & R_4 \mid \mathit{sequence}(1 : Y)
\end{array}$$

and finally $\mathit{Ext}_P(R_3 \underset{G}{\rightarrow} X : [\mathit{first}(M, Xs)])$ adds rules:

$$R_3 \underset{G}{\rightarrow} X : R_2 \mid X : R_3$$

G has now stabilized. If we now restrict G to those rules that yield terms containing the only ground constructor terms of P, the grammar can be simplified considerably:

$$\begin{array}{lll}
R_0 & \underset{G}{\rightarrow} & \mathit{nil} \mid R_3 \\
R_3 & \underset{G}{\rightarrow} & Y : \mathit{nil} \mid Y : R_3 \\
Y & \underset{G}{\rightarrow} & \mathit{nil} \mid 1 : Y
\end{array}$$

5 Termination and correctness

Let ∂_P denote our extension operation: For a term rewriting system P and a tree grammar G, define

$$\partial_P(G) = \bigcup_{A \rightarrow g[g'] \in G} \mathit{Ext}_P(A \rightarrow g[g'])$$

For a tree grammar G_0 (generating the input) we construct the new tree grammar G as the least set H of rules satisfying

$$H \supseteq G_0 \cup \partial_P(H)$$

5.1 Termination

Formally, this construction amounts to a denumerably infinite process: For $k = 0, 1, \ldots$ define $G_{k+1} = G_k \cup \partial_P(G_k)$. Then $G_0 \subseteq G_1 \subseteq \ldots$, and $G = \bigcup_{k=0}^{\infty} G_k$ where the limit satisfies the defining formula with equality: $G = G_0 \cup \partial_P(G)$.

Fortunately, this process is actually finite:

Theorem 5.1 If G_0 and P are finite, then so is G.

PROOF.

Let a *variant* of a term be the result of replacing zero or more of its subterms by nonterminals. We show that every G rule must be a variant of a subterm of the right side of a G_0 rule or a P rule.

The Ext_P-construction forms new rules of three kinds: $A \underset{G}{\to} g[R_i]$, $R_i \underset{G}{\to} t_i$ and $X \underset{G}{\to} \theta X$. Obviously, the right hand side $g[R_i]$ of the first kind of rules is a variant (formed by replacing exactly one subterm by a nonterminal) of the right hand side of the generating rule $A \underset{G}{\to} g[g']$.

The second kind of rule copies its right hand side from one of the rules of the original term rewriting system.

For the third kind of rule, the minimality requirement on the derivation $g' \underset{G}{\Longrightarrow}^* \theta p_i$ ensures that the new right hand side θX will actually be a *subterm* of the right hand side of an existing G-rule. This can be seen as follows:

Assume $g' \underset{G}{\Longrightarrow}^n \theta p_i$ and $X \in \mathit{Vars}(t_i) \subseteq \mathit{Vars}(p_i)$. This means that θp_i may be written $g_1[\theta X]$ for some context $g_1[\,]$.

Since θX is part of a term generated by A, if θX is not itself contained in the right hand side of some rule $A' \underset{G}{\to} w$ then it must contain such a right hand side. In other words, θX would have the form $g_2[w]$ for a non-empty context $g_2[\,]$, and the derivation $g' \underset{G}{\Longrightarrow}^n \theta p_i$ could be seen as $g' \underset{G}{\Longrightarrow}^{n-1} g_1[g_2[A']] \underset{G}{\Rightarrow} g_1[g_2[w]] = \theta p_i$. Since X only occurs once in p_i we could define θ' as the modification of θ mapping X to $g_2[A']$ so that $g' \underset{G}{\Longrightarrow}^{n-1} g_1[g_2[A']] = \theta' p_i$, thus violating the last requirement in the definition of Ext_P.

Note that the set of variants of subterms is closed under the formation of variants and the extraction of subterms. During the construction of G, the right hand sides of the added rules will therefore always be variants of subterms of right hand sides of G_0 and P rules.

If G_0 and P are finite, the set of nonterminals and of right hand sides are also finite, so that G becomes finite.

Lemma 5.2 For a tree grammar H subjected to the ∂_P-construction, if $A \underset{H}{\Longrightarrow}^* g[\theta p_i]$, then $A \underset{H\cup\partial_P(H)}{\Longrightarrow}^* g[R_i]$, $R_i \underset{\partial_P(H)}{\rightarrow} t_i$, and $X \underset{H\cup\partial_P(H)}{\Longrightarrow}^* \theta X$ for all $X \in \mathit{Vars}(t_i)$.

PROOF.

The derivation $A \underset{H}{\Longrightarrow}^* g[\theta p_i]$ must use a rule $A' \underset{H}{\rightarrow} h'[h'']$ where $h'[\,] \underset{H}{\Longrightarrow}^* g'[\,]$, $h'' \underset{H}{\Longrightarrow}^* \theta p_i$ and $g[\,] = h[g'[\,]]$.

In other words, the derivation may be reordered as

$$A \underset{H}{\Longrightarrow}^* h[A'] \underset{H}{\Longrightarrow} h[h'[h'']] \underset{H}{\Longrightarrow}^* h[g'[h'']] = g[h''] \underset{H}{\Longrightarrow}^* g[\theta p_i]$$

For the derivation $h'' \underset{H}{\Longrightarrow}^* \theta p_i$ there must be some n and θ' such that $h'' \underset{H}{\Longrightarrow}^n \theta' p_i \underset{H}{\Longrightarrow}^* \theta p_i$, but $\neg\exists\theta''(h'' \underset{H}{\Longrightarrow}^{n-1} \theta'' p_i \underset{H}{\Longrightarrow} \theta' p_i)$. $\mathit{Ext}_P(A' \underset{H}{\rightarrow} h'[h''])$ will therefore add

$$\begin{array}{lll} A' & \underset{\partial_P(H)}{\rightarrow} & h'[R_i]\ , \\ R_i & \underset{\partial_P(H)}{\rightarrow} & t_i\ \text{, and} \\ X & \underset{\partial_P(H)}{\rightarrow} & \theta' X \text{ for all } X \in \mathit{Vars}(t_i) \end{array}$$

and we obtain a derivation

$$A \underset{H}{\Longrightarrow}^* h[A'] \underset{\partial_P(H)}{\Longrightarrow} h[h'[R_i]] \underset{H}{\Longrightarrow}^* h[g'[R_i]] = g[R_i]$$

as desired.

The fact that $\theta' p_i \underset{H}{\Longrightarrow}^* \theta p_i$ means that there must exist derivations $\theta' X \underset{H}{\Longrightarrow}^* \theta X$ for all $X \in \mathit{Vars}(t_i)$, so that $X \underset{\partial_P(H)}{\rightarrow} \theta' X \underset{H}{\Longrightarrow}^* \theta X$ for all $X \in \mathit{Vars}(t_i) \subseteq \mathit{Vars}(p_i)$.

Lemma 5.3 If $A \underset{G_0}{\Longrightarrow}^* w$ and $w \underset{P}{\Longrightarrow}^* g[\theta p_i]$, then $A \underset{G}{\Longrightarrow}^* g[R_i]$, $R_i \underset{G}{\rightarrow} t_i$, and $X \underset{G}{\Longrightarrow}^* \theta X$ for all $X \in \mathit{Vars}(t_i)$.

PROOF.

Let $w \underset{P}{\Longrightarrow}^n g[\theta p_i]$; the proof is by induction on n. Recall that $G = G_0 \cup \partial_P(G)$.

For $n = 0$ employ Lemma 5.2.

For $n > 0$, assume the claim for $n-1$. By induction, from $A \underset{G_0}{\Longrightarrow}^* w \underset{P}{\Longrightarrow}^{n-1} g'[\theta' p_j] \underset{P}{\Rightarrow} g'[\theta' t_j] = g[\theta p_i]$ we therefore get $A \underset{G}{\Longrightarrow}^* g'[R_j]$, $R_j \underset{G}{\rightarrow} t_j$ and $X \underset{G}{\Longrightarrow}^*$

$\theta' X$ for all $X \in \mathit{Vars}(t_j)$ so that the derivation may be continued $g'[R_j] \underset{G}{\Longrightarrow} g'[t_j] \underset{G}{\Longrightarrow}^* g'[\theta' t_j] = g[\theta p_i]$. Applying Lemma 5.2 now proves the lemma.

Theorem 5.4 If $A \underset{G_0}{\Longrightarrow}^* w$ and $w \underset{P}{\Longrightarrow}^* g$, then $A \underset{G}{\Longrightarrow}^* g$.

PROOF.

Assume $w \underset{P}{\Longrightarrow}^n g$. For $n = 0$ the claim is obvious, using $G \supseteq G_0$.

Otherwise the last step in the derivation is $g'[\theta p_i] \underset{P}{\Rightarrow} g'[\theta t_i]$ for some context $g'[\,]$, index i and substitution θ.

The derivation $A \underset{G}{\Longrightarrow}^* g'[R_i]$ obtained from Lemma 5.3, which also ensures $R_i \underset{G}{\rightarrow} t_i$ and $X \underset{G}{\Longrightarrow}^* \theta X$ for all $X \in \mathit{Vars}(t_i)$, may therefore be continued $g'[R_i] \underset{G}{\Longrightarrow} g'[t_i] \underset{G}{\Longrightarrow}^* g'[\theta t_i] = g$, proving the theorem.

6 Conclusions and directions for future development

An algorithm for the flow analysis of applicative programs, using term rewriting systems as a program formalism, has been presented, and a proof was given that the method terminates with a finite grammar, and that it yields safe appoximations to program behaviour. It has been seen that the method is more powerful than earlier methods in that it gives safe approximations to lazy programs (outside-in evaluation order) and to programs using higher order functions.

Several extensions are natural and desirable for practical applications (such as optimizing compilers for applicative languages or the MIX system), including the following:

(1) *Outside-in and inside-out rewriting:* Languages as implemented mostly use either a strictly inside-out rewriting order (call-by-value, as in LISP), or strictly outside-in (lazy evaluation), while G as constructed above models all possible reduction sequences. This may be seen in the algorithm, which adds new variable bindings for all possible calls: $A \underset{G}{\rightarrow} g[g']$. An outside-in rewriting order is easily modeled by changing the algorithm to call $\mathit{Ext}_P(A \underset{G}{\rightarrow} g[g'])$ only when $[g']$ is outermost, and the analogous trick models inside-out rewriting: $\mathit{Ext}_P(A \underset{G}{\rightarrow} g[g'])$ is called just in case $[g']$ is innermost. The resulting grammars would then give more precise safe descriptions, with smaller sets of computational configurations.

(2) *Atomic values:* Grammar G in essence traces the control flow of the program being analyzed. It assumes that all data values are terms, and does not account for programs manipulating atomic values such as numbers or booleans. On the other hand, traditional flow analysis as used in optimizing compilers (e.g., constant propagation, see [12]) is mainly concerned with atomic values, and makes rather conservative assumptions about flow of control. This type of flow analysis uses a lattice or cpo of *abstract values* to describe sets of atomic values; for example **pos**, **neg** and **num** could describe nonempty sets containing only positive values, only negative values, and arbitrary values, respectively.

It seems clear that the analysis could be extended to handle atomic values as follows. Suppose that certain arguments of the defined operators or constructors are designated as being atomic, for example that the first argument to $cons(x, y)$ will be numeric. Then abstraction may be performed when adding rules to G, so that if G already contains a rule: $A \underset{G}{\rightarrow} cons(\mathbf{pos}, B)$, and a new rule: $A \underset{G}{\rightarrow} cons(\mathbf{neg}, B)$ is to be added by Ext_P, then the original rule is replaced by the least common abstraction of both: $A \underset{G}{\rightarrow} cons(\mathbf{num}, B)$.

(3) *More general result symbols:* Turchin's "basic configurations" resemble a more general version of the patterns used in Section 4.3, and are used to perform program transformation in [33]. The MIX system uses an essentially similar idea to classify function arguments for the purpose of partial evaluation. It would be desirable to put these various ideas in a more general framework.

(4) *Strictness analysis:* It would be interesting to see whether our algorithm could be modified to yield a grammar describing the way the program accesses the substructures of its data, and determine those substructures which are required to run the program. For example, "append" requires all tails of its first argument but no heads, and does not examine its second argument. This topic has been addressed in [15], [33] and [35] for the case of first order programs.

Acknowledgements

Many people read early drafts of this paper, and their comments helped considerably in clarifying its ideas and tightening it up: Olivier Danvy, Carsten Kehler-Holst, Thomas Johnsson,Torben Mogensen, Alberto Pettorossi, Jakob Grue Simonsen, Harald Søndergaard and Valentin Turchin. Special thanks are due to Klaus Grue, Phil Wadler and the editors of [2] where an earlier version of this paper appeared.

We would also like to thank the anonymous referees of this Festschrift for valuable comments.

References

[1] Abel, A.: 2004, 'Termination Checking with Types'. *RAIRO - Theoretical Informatics and Applications, Special Issue: Fixed Points in Computer Science (FICS'03)* **38**(4), 277–319.

[2] Abramsky, S. and C. Hankin: 1987, *Abstract Interpretation of Declarative Languages.* Ellis Horwood.

[3] Aho, A.: 1973, *Currents in the theory of computing.* Prentice-Hall.

[4] Brainerd, W. S.: 1969, 'Tree generating regular systems'. *Information and Control* **13**, 217–231.

[5] Burn, G. L., C. L. Hankin, and S. Abramsky: 1986, 'The theory of strictness analysis for higher order functions'. In: *[9].* pp. 42–62.

[6] Cousot, P.: 1981, *Semantic foundations of program analysis,* in [25], pp. 303–342.

[7] Cousot, P. and R. Cousot: 1977, 'Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints'. In: *POPL '77: 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* pp. 238–252.

[8] Dijkstra, E.: 1976, *A Discipline of Programming.* Prentice-Hall.

[9] Ganzinger, H. and N. Jones (eds.): 1986, 'Programs as Data Objects', Vol. 217 of *Lecture Notes in Computer Science.* Springer-Verlag.

[10] Gécseg, F. and M. Steinby: 1984, *Tree Automata.* Akademiai Kiado, Budapest.

[11] Giesl, J., R. Thiemann, and P. Schneider-Kamp: 2005, 'Proving and disproving termination of higher-order functions'. Technical report, RWTH Aachen.

[12] Hecht, M.: 1977, *Flow Analysis of Computer Programs.* North-Holland.

[13] Heintze, N.: 1992, 'Set Based Program Analysis'. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA.

[14] Huet, G. and J. J. Lévy: 1979, 'Call by need computations in nonambiguous linear term rewriting systems'. Technical report, INRIA, France.

[15] Hughes, J.: 1986, 'Strictness detection in non-flat domains'. In: *[9].* pp. 112–135.

[16] Jagannathan, S. and S. Weeks: 1995, 'A Unified Treatment of Flow Analysis in Higher-Order Languages'. In: *POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* pp. 393–407.

[17] Johnsson, T.: 1985, 'Lambda lifting: transforming programs to recursive equations'. In: *Proceedings IFIP Symposium on Functional Programming Languages and Computer Architecture*, Vol. 201 of *Lecture Notes in Computer Science.*

[18] Jones, N. D.: 1981, 'Flow analysis of lambda expressions'. In: *Proceedings of ICALP 1981*, Vol. 115 of *Lecture Notes in Computer Science.*

[19] Jones, N. D.: 1987, *Flow analysis of lazy higher-order functional programs,* in [2], pp. 103–122.

[20] Jones, N. D. and S. S. Muchnick: 1979, 'Flow analysis and optimisation of LISP-like structures'. In: *POPL '79: 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* pp. 244–256.

[21] Jones, N. D. and A. Mycroft: 1986, 'Data flow analysis of applicative programs using minimal function graphs'. In: *POPL '86: 13th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* pp. 296–306.

[22] Jones, N. D. and M. Rosendahl: 1997, 'Higher-Order Minimal Function Graphs'. *Journal of Functional and Logic Programming* **2**.

[23] Jones, N. D., P. Sestoft, and H. Sondergaard: 1985, 'An experiment in partial evaluation: the generation of a compiler generator'. In: J.-P. Jouannoud (ed.): *Rewriting Techniques and Applications*, Vol. 202 of *Lecture Notes in Computer Science.*

[24] Jones, S. P.: 2003, *Haskell 98 Language and Libraries.* Cambrdige University Press.

[25] Muchnick, S. S. and N. D. Jones (eds.): 1981, *Program Flow Analysis: Theory and Applications.* Prentice-Hall.

[26] Mycroft, A. and N. D. Jones: 1986, 'A relational framework for abstract interpretation'. In: *[9].* pp. 156–171.

[27] Reynolds, J.: 1969, 'Automatic computation of data set definitions'. In: *Information Processing* **68**. pp. 456–461.

[28] Reynolds, J.: 1972, 'Definitional interpreters for higher-order programming languages'. In: *Proceedings of the ACM annual conference.* pp. 717–740. ACM Press.

[29] Shivers, O.: 1991, 'Control-Flow Analysis of Higher-Order Languages'. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA, USA.

[30] Stefanescu, D. and Y. Zhou: 1994, 'An equational framework for the flow analysis of higher order functional programs'. In: *ACM Symposium on Lisp and Functional Programming.* pp. 318–327.

[31] Thatcher, J.: 1973, *Tree automata: an informal survey,* in [3].

[32] Toyama, Y.: 2004, 'Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms'. In: *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, Vol. 3091 of *Lecture Notes in Computer Science.* pp. 40–54.

[33] Turchin, V.: 1980, 'The language REFAL, the theory of compilation, and metasystem analysis'. Technical report, Courant Institute Report, New York.

[34] Wadler, P.: 1985, 'An introduction to Orwell'. Technical report, Programming Research Group, Oxford University.

[35] Wadler, P.: 1987, *Strictness analysis on non-flat domains (by Abstract interpretation over finite domains*, Chapt. 12, pp. 266–275.

[36] Xi, H.: 1998, 'Dependent Types in Practical Programming'. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA.