Macros for Context-Free Grammars

Peter Thiemann Matthias Neubauer

Institut für Informatik, Universität Freiburg, Germany
{thiemann,neubauer}@informatik.uni-freiburg.de

ABSTRACT

Current parser generators are based on context-free grammars. Because such grammars lack abstraction facilities, the resulting specifications are often not easy to read. Fischer's macro grammars provide the equivalent of procedural abstraction by extending context-free grammars with macro-like productions. Unfortunately, macro grammars generate context-sensitive languages, in general, so they do not have efficient parsers.

To enable the specification of a language using a macro grammar, we define specialization for macro grammars. This specialization always yields context-free rules, but it does not always terminate. We present a sound and complete static analysis that decides whether specialization terminates for a given macro grammar and thus yields a (finite) context-free grammar. The analysis is based on an intuitive notion of self-embedding nonterminals, which is easy to check by hand and which gives the expected answer for all examples that we tried.

1. INTRODUCTION

Current parser generators are based on context-free grammars because their word problem is solvable in cubic time in the worst case or in linear time for deterministic grammars like LL- or LR-grammars [1]. While context-free grammars are a suitable and successful "assembly language" for specifying context-free languages, they are not really good for defining languages in a high-level way. In particular, they lack abstraction facilities so that grammars are neither modular nor easy to reuse.

While parser generators and related tools have evolved with respect to the modularization of parsing actions and integration of the specifications of parsing and scanning, the actual raw matter, the grammar, remains in its original form in the parser specification. As large grammars may well run into several hundred productions, grammar maintenance can become a tedious task. Hence, it is surprising that none of the parser generators has a facility for introducing abstractions over grammar rules.

- lists with separator (seven times)

```
DelimTSchemes    : /* empty */
                 | NEDelimTSchemes ;
NEDelimTSchemes : TScheme
                 | NEDelimTSchemes ',' TScheme ;
```

- plain lists (five times plus two non-empty lists)

```
TypeVars         : /* empty */
                 | NETypeVars ;
NETypeVars       : TypeVar
                 | NETypeVars TypeVar ;
```

- optional items (two times)

```
OptRenaming      : /* empty */
                 | Renaming ;
```

Figure 1: Patterns in existing grammars.

The main attempt to introduce some abbreviation mechanism into grammars is the consideration of regular right-hand sides for rules [5, 18]. Even though this facility falls short from providing flexible abstractions, only a few LR parser generators (*e.g.*, Eli [11]) and several LL parser generators (*e.g.*, ANTLR [18] and the formalization in Wilhelm and Maurer's textbook [27]) support regular right hand sides or extended BNF directly. Consequently, typical grammars for parser generators are full of rule groups that implement common grammatical patterns. Figure 1 contains some examples with the number of uses of the respective pattern in a randomly picked grammar. Often, even the semantic actions coincide or can be made to coincide easily.

Our proposal derives directly from these observations. Instead of relying on a fixed set of (regular) operators for use in the right-hand side of a grammar rule, we make available an arbitrary, user-definable set of operators in the form of *parameterizable nonterminal symbols*. These nonterminals behave like macros. They can be invoked on the right-hand side of a production with any string of terminals, nonterminals, and macro applications as actual parameters. This notion coincides exactly with Fischer's macro grammars [8]. The generative power of macro grammars is properly contained between the context-free languages and the context-sensitive languages. That is, the word problem is decid-

```
(SepList sep item)     : /* empty */
                       | (NESepList sep item) ;
(NESepList sep item)   : item
                       | (NESepList sep item) sep item ;

(List item)            : /* empty */
                       | (NEList item) ;
(NEList item)          : item
                       | (NEList item) item ;

(Option item)          : /* empty */
                       | item ;
```

Figure 2: Examples for parameterized rules.

```
DelimTypeSchemes : (SepList ',' TypeScheme) ;
TypeVars         : (List TypeVar) ;
OptRenaming      : (Option Renaming) ;
```

Figure 3: Uses of the parameterized rules.

able for general macro grammars, but not as efficient as for context-free grammars.

Hence, we impose an intuitive (and effectively checkable) restriction on macro grammars to ensure that they are specializable to context-free grammars. The specialization process creates a specialized nonterminal for each invocation of a macro-nonterminal with a particular arguments. The specialization is a straightforward adaption of standard techniques for program-point specialization [15, 12], the main challenge is to guarantee the termination of this specialization. If the specialization terminates, then it yields a context-free grammar equivalent to the original macro grammar.

With our approach, the author of a grammar can write parameterized productions corresponding to the patterns exhibited above once and for all. Sets of parameterized nonterminals may be collected in modules and reused between grammars. At the same time, the specialization process yields plain context-free grammars that have efficient parsers implemented using standard techniques.

The parameterized rules in Figure 2 capture the grammatical patterns identified in Figure 1. By convention, nonterminals have capitalized names whereas their parameters start with a lowercase letter. Terminal symbols are enclosed in single quotes. Figure 3 suggests uses of the parameterized rules that match the uses of the respective patterns in the original grammar (Figure 1). For the examples in Figure 2 it also makes sense to define generic semantic actions. The only requirement is that these actions are polymorphic with respect to the semantic values of the parameters. Hence, the `(SepList sep item)` and the `(List item)` might both return a value of type `List item` and the `(Option item)` might return a value of type `Maybe item`.

In earlier work [23], we have investigated the related notion of *parameterized LR parsing*. That work relies on a restricted formalism, the *simple macro grammars*, where each argument of a macro must be a single symbol, either a nonterminal or a parameter, and achieves specialization through an extended definition of the LR parsing framework. The single-symbol restriction trivially ensures terminating specialization by disallowing essentially nested macro invocations. The examples in Figures 2 and 3 only involve simple macro grammars. Section 2 contains examples which go beyong simple macro grammars. In direct comparison, the present work is neither restricted to LR parsing nor does it impose an a-priori syntactical restriction on the use of the macro facility. Thus, it can be combined and integrated with arbitrary parser generators and it does not impose artificial limits on the expressivity of specifications.

```
AdditiveExp       : MultiplicativeExp
                  | AdditiveExp '+' MultiplicativeExp

ShiftExp          : AdditiveExp
                  | ShiftExp '<<' AdditiveExp

RelationalExpNoIn : ShiftExp
                  | RelationalExpNoIn '<' ShiftExp
```

Figure 4: Excerpt from JavaScript infix expressions.

Overview

We start with further examples of the advanced use of macro grammars to specify commonly used grammatical pattern in Section 2. Then, in Section 3, we define macro grammars and macro languages formally. Section 4 defines the specialization process that creates context-free productions from macro productions. To determine that specialization terminates, Section 5 defines a *transition graph* which collects all necessary information about macro calls and its finite abstraction. Section 6 presents the main technical result of the work, a necessary and sufficient criterion for the finiteness of the transition graph. The paper closes with a few remarks on an implementation (Section 7), a discussion of related work (Section 8), and a conclusion in Section 10.

2. ADVANCED USES OF MACROS IN GRAMMAR SPECIFICATIONS

Our prior work [23] is restricted to simple macro grammars, as already mentioned. This notion can already express many patterns that occur in grammars describing real-world programming language, as demonstrated in the introduction. However, the restriction of arguments to single symbols turns out to be too limiting. Many natural abstractions do not obey this restriction and either require a cumbersome workaround (the introduction of an additional nonterminal) or are not expressible at all.

The following examples show how lifting the restriction further helps to shorten the syntactical description of real-world programming languages.

Encoding Operator Precedence. In JavaScript, as in many other programming languages, binary operators are equipped with different levels of precedences. A typical way to capture this fact on the syntactic level is to formulate several rules for expressions using several intermediate stages. Figure 4 contains a shortened version of three such stages from the JavaScript language description (the full grammar contains 14 stages). A parameterized rule `BinOp` conveniently factors the pattern of one stage. With this rule, we can express the `RelationalExpNoIn` by nested application of `BinOp` as shown in Figure 5. The resulting specification of precedence is much more readable than tracking levels

```
(BinOp op base)          : base
                         | (BinOp op base) op base

RelationalExpNoIn : (BinOp '<'
                     (BinOp '<<'
                      (BinOp '+' MultiplicativeExp)))
```

Figure 5: Example use of `BinOp`

```
RelationalExp     : ShiftExp
                  | RelationalExp '<' ShiftExp
                  | RelationalExp 'in' ShiftExp

RelationalExpNoIn : ShiftExp
                  | RelationalExpNoIn '<' ShiftExp

EqualityExp       : RelationalExp
                  | EqualityExp '==' RelationalExp

EqualityExpNoIn   : RelationalExpNoIn
                  | EqualityExpNoIn '==' RelationalExpNoIn

BitANDExp         : EqualityExp
                  | BitANDExp '&' EqualityExp

BitANDExpNoIn     : EqualityExpNoIn
                  | BitANDExpNoIn '&' EqualityExpNoIn
```

Figure 6: Expressions with `in` and without `in`.

of precedence through 14 different nonterminals. Moreover, it avoids the useless naming of these nonterminals. This use of `BinOp` does not fit the restriction of simple macro grammars because it contains nested macro invocations. Although the definition of `RelationalExpNoIn` is specializable to a context-free grammar.

Similar Exps in Different Contexts. Again in JavaScript, relational expressions occur in two slight variations depending on the context in which they appear. Most of the time, the `in` token is also a valid relational operator. However, in the header of a `for` statement the `in` is not allowed because it has a different use. The official JavaScript grammar literally duplicates the rules for relational expressions and for the ten subordinate stages to capture both variations. The rules in Figure 6 show three stages in both variations.

There are at least two ways to circumvent the duplication by making use of parameterized grammar rules. The first approach abstracts over the operators allowed in a relational expression, specifies two different nonterminals expressing the operation symbols allowed in either contexts, and use those to instantiate the parametric expression rule. Figure 7 shows this alternative.

Another alternative is to employ a more flexible rule for binary operators: a parametric rule for binary operators that abstracts not only over the operator symbols but also over an additional way to derive the operator expression.

```
(BinOpAlt op base alt) : base
                       | (BinOpAlt op base alt) op base
                       | alt
```

We use the new rule to specify the common part of expressions, add two nonterminals that specify the two additional ways to derive relational expressions in both contexts, and use those to instantiate the expression pattern.

```
(BitANDExp ops) : (BinOp '&'
                   (BinOp '=='
                    (BinOp ops ShiftExp)))

REOpsIn         : '<'
                | 'in'
REOpsNoIn       : '<'

BitANDExpIn     : (BitANDExp REOpsIn)
BitANDExpNoIn   : (BitANDExp REOpsNoIn)
```

Figure 7: Abstracting over a set of operators.

```
(BitANDExp x) : (BinOp '&'
                 (BinOp '=='
                  (BinOpAlt '<' ShiftExp x)))

REArgIn       : (BinOpAlt '<' ShiftExp REArgIn)
                'in' ShiftExp
REArgNoIn     : REArgNoIn

BitANDExpIn   : (BitANDExp REArgIn)
BitANDExpNoIn : (BitANDExp REArgNoIn)
```

Here, the nonterminal `REArgNoIn` is useless (no strings are derivable from it) that should be eliminated before generating a parser from the specialized grammar.

Permutation Phrases. The Java Language Specification [10] contains *permutation phrases*, another pattern commonly found in the syntax of real-world programming languages. A permutation phrase consists of a finite set of alternatives, which may occur in any order, but each of which may occur at most once. As a typical example, consider a fragment of the specification of field modifiers for field declarations:

```
FieldModifiers  : FieldModifier
                | FieldModifiers FieldModifier

FieldModifier   : 'static'
                | 'final'
                | 'public'
```

The additional condition stating that "A compile-time error occurs if the same modifier appears more than once in a field declaration" is only specified by an informal annotation to the grammar. Checking this side condition of a permutation phrase is usually left to the compiler's semantic analysis because it is cumbersome to express in a context-free grammar (it requires exponentially many productions).

We can encode the additional condition concisely in the grammar by using macro productions as shown in Figure 8. The nonterminal `Perm3` implements permutation phrases by taking three parameters that correspond to the three alternatives. The first production indicates the end of the phrase. The remaining three productions correspond to taking the first (second, third) alternative and they continue by disabling the taken alternative in the recursive call to `Perm3`. Disabling takes place by substituting the nonterminal `NUL` (which has no productions) for the chosen alternative.

Specialization generates exponentially many context-free productions from these productions. It is also expected that useless rules and unreachable nonterminals are removed from the specialized grammar.

```
(PermP3 m1 m2 m3) : /* empty */
                  | m1 (PermP3 NUL m2 m3)
                  | m2 (PermP3 m1 NUL m3)
                  | m3 (PermP3 m1 m2 NUL)

/* no production for NUL */

FieldModifiers    : (PermP3 'static' 'final' 'public')
```

Figure 8: A fixed-length permutation phrase.

3. MACRO GRAMMARS

To define macro grammars properly, we need some standard definitions inspired by universal algebra. As we will frequently be speaking about indexed lists of syntactic entities, we write $\overline{x}_n$ as a shorthand for $x_0, \ldots, x_{n-1}$ and omit the index n if it is not important.

DEFINITION 3.1. *A* signature *is a pair $\Gamma = (N, a)$ of a finite set N and an arity function $a : N \to \mathcal{A}$, where the set $\mathcal{A}$ is defined inductively by: If $n \in \mathbf{N}$ and, for all $0 \leq i < n$, $\alpha_i \in \mathcal{A}$, then $\langle \overline{\alpha}_n \rangle \in \mathcal{A}$.*

The set $\mathcal{A}$ generalizes numeric arities as follows. A constant has arity $\langle\rangle$. An n-argument macro has as arity the n-place vector $\langle \overline{\alpha}_n \rangle$ where α_{i-1} is the arity of the ith argument. If we are only interested in the number of arguments, we consider a as a function from $N \to \mathbf{N}$.

Arities serve to categorize different kinds of nonterminals. They are best compared with function types where the return type is fixed and left implicit. This choice is suitable for nonterminals because they will always turn into strings in the end.

In this work, the signature Γ always contains a binary operator $\cdot$ (concatenation) and a constant ε (empty string) where $a(\cdot) = \langle\langle\rangle, \langle\rangle\rangle$ and $a(\varepsilon) = \langle\rangle$.

DEFINITION 3.2. *Let $\Gamma = (N, a)$ be a signature. The set $T^{\alpha}_{\Gamma}(X)$ of Γ-terms of arity α with variables $X = (X^{\alpha})$ (a set disjoint from N indexed by arities α) is defined inductively by*

- $X^{\alpha} \subseteq T^{\alpha}_{\Gamma}(X)$,
- $\forall A \in N, A \in T^{a(A)}_{\Gamma}(X)$,
- $\forall n \in \mathbf{N}, (\forall 0 \leq i < n, t_i \in T^{\alpha_i}_{\Gamma}(X)), \forall t \in T^{\langle \overline{\alpha}_n \rangle}_{\Gamma}(X)$
$$t(t_0, \ldots, t_{n-1}) \in T^{\langle\rangle}_{\Gamma}(X).$$

That is, each variable is a term, each nonterminal is a term with its respective arity, and terms can be built from a "function/macro term" and an argument list, provided their arities match the argument arities of the function term. However, the facilities for actually constructing a function term are limited to symbols taken from the signature.

EXAMPLE 3.3. *A signature providing for the lists and optional items from the introduction allows only constants as arguments. It is defined by a table of its arity function.*

$$\begin{array}{lcl} a(\texttt{SepList}) & = & \langle\langle\rangle, \langle\rangle\rangle \\ a(\texttt{NESepList}) & = & \langle\langle\rangle, \langle\rangle\rangle \\ a(\texttt{List}) & = & \langle\langle\rangle\rangle \\ a(\texttt{NEList}) & = & \langle\langle\rangle\rangle \\ a(\texttt{Option}) & = & \langle\langle\rangle\rangle \end{array}$$

DEFINITION 3.4. *A* macro grammar *is a tuple (Γ, Σ, P, S) where $\Gamma = (N, a)$ is a signature with N the* nonterminal symbols, *Σ is a finite set of* terminal symbols *of arity $\langle\rangle$, $P \subseteq \{(A, w) \mid A \in N, w \in T^{\langle\rangle}_{\Gamma(A)}(\Sigma)\}$ is a finite set of macro productions, and $S \in N$ with $a(S) = \langle\rangle$ is the start symbol.*

The productions are subject to the following restriction which is already indicated in the type above. If $A \to w \in P$ with $a(A) = \langle \overline{\alpha}_n \rangle$, then $w \in T^{\langle\rangle}_{\Gamma(A)}(\Sigma)$ where $\Gamma(A) = \Gamma \cup \{a(0) = \alpha_0, \ldots, a(n-1) = \alpha_{n-1}\}$.

To increase readability of the examples, we take the liberty of naming the parameters of the nonterminals as in the introduction instead of using the numbering scheme. In addition, we drop the parentheses after nullary nonterminals and parameters. In formal statements, we will stick to the positional notation and the parameters.

A macro grammar generates words over the set of terminal symbols using the following derivation relation $\Rightarrow$ on $T^{\langle\rangle}_{\Gamma}(\Sigma)$. The definition uses the notation $w[i \mapsto t]$ to denote the term w with all occurrences of parameter i replaced by term t.

- If $A \to w \in P$, $a(A) = \langle \overline{\alpha}_n \rangle$, $(\forall 0 \leq i < n)\ t_i \in T^{\alpha_i}_{\Gamma}(\Sigma)$, then $A(t_0, \ldots, t_{n-1}) \Rightarrow w[0 \mapsto t_0, \ldots, n-1 \mapsto t_{n-1}]$, and
- if $f \in \Gamma$, $a(f) = \langle \overline{\alpha}_n \rangle$, $(\forall 0 \leq i < n)\ t_i \in T^{\alpha_i}_{\Gamma}(\Sigma)$, $t_j \Rightarrow t'_j$, then $f(t_0, \ldots, t_j, \ldots, t_{n-1}) \Rightarrow f(t_0, \ldots, t'_j, \ldots, t_{n-1})$.

That is, the relation comprises all pairs of terms which are instances of a production and it is closed under compatibility with operators from Γ. As usual, $\overset{*}{\Rightarrow}$ denotes the reflexive transitive closure of the derivation relation.

A term w is in the language generated by the grammar if $S \overset{*}{\Rightarrow} w$ and $w \in T^{\langle\rangle}_{\cdot,\varepsilon}(\Sigma)$, which can be considered as an element of Σ^* in the obvious way.

Often the derivation relation is restricted to either substitute nonterminals inside-out (IO) or outside-in (OI).

IO reduction $A(t_0, \ldots, t_{n-1}) \Rightarrow_{IO} w[0 \mapsto t_0, \ldots, n-1 \mapsto t_{n-1}]$ only if $t_0, \ldots, t_{n-1} \in T^{\langle\rangle}_{\cdot,\varepsilon}(\Sigma)$ do not contain nonterminals and the relation is closed under compatibility as before.

OI reduction the reduction rule for $\Rightarrow_{OI}$ is the same as for $\Rightarrow$, but compatibility is restricted to $f \in \{\cdot, \varepsilon\}$ so that reduction does **not** proceed into argument positions.

$$
\begin{array}{lll}
G_{abc}: & S \to F(\varepsilon,\varepsilon,\varepsilon) \\
& F \to 012 \\
& F \to F(a0,b1,c2)
\end{array}
$$

$$
\begin{array}{lll}
G_{list}: & S \to L(a) \\
& S \to L(b) \\
& L \to \varepsilon \\
& L \to N(0) \\
& N \to 0L(0)
\end{array}
$$

Figure 9: Example macro grammars. G_{abc} generates a context-sensitive language. G_{list} is a variation of the parameterized list-generating grammar from the introduction. It generates the language $\{a^n \mid n \in \mathbf{N}\} \cup \{b^n \mid n \in \mathbf{N}\}$.

The respective languages are called IO- and OI-macro languages. They have been investigated in detail [8] and we recall some of their properties below.[1]

1. In general, the language generated from a grammar under IO reduction is different from the language generated under OI reduction. (IO corresponds roughly to call-by-value and OI to call-by-name.)
2. The classes of IO- and OI-macro languages are incomparable.
3. The IO- and OI-macro languages form a strict hierarchy of languages between context-free languages and context-sensitive languages [6].

As an example for a macro grammar defining a language that is not context-free consider the grammar G_{abc} in Figure 9. This grammar generates the language $\{a^n b^n c^n \mid n \in \mathbf{N}\}$ which is context-sensitive but not context-free. The extra generative power comes from the possibility to pass arbitrary terms as parameters, in particular nontrivial terms that contain parameters themselves. The language of this grammar is indepedent of the reduction order because each sentential form contains at most one invocation of a macro.

4. CONTEXT-FREE GRAMMARS FROM MACRO GRAMMARS

The macro grammar for generating the context-sensitive language $\{a^n b^n c^n \mid n \in \mathbf{N}\}$ is not typical for the grammars that we are interested in in this paper. We are interested in macro grammars like G_{list} in Figure 9 that merely abbreviate a context-free grammar and where this context-free grammar can be obtained by specialization. Some notation is needed to define this specialization and to prove that a specialized grammar is equivalent to its underlying original macro grammar.

[1]The definition we are giving above is not the one that has been used to obtain the cited results. The original definition considers strings as trees build from monadic operators (the characters) so that standard nonterminals in a context-free grammar are also monadic operators serving as placeholders for trees. In a macro grammar, nonterminals receive *additional* parameters that range over monadic operators. Adding further parameter sets leads to higher levels in the mentioned hierarchy. Our definition achieves higher-orderness through the arity system.

DEFINITION 4.1. *A* parameter instance *for arity $\langle\overline{\alpha}_m\rangle$ is a tuple of terms $\overline{r}_m$ without free parameter variables, that is $r_j \in T_\Gamma^{\alpha_j}(\Sigma)$. We write $\overline{r}_m : \langle\overline{\alpha}_m\rangle$ to indicate this case.*

A parameter instantiation *from arity $\langle\overline{\alpha}_m\rangle$ to arity $\langle\overline{\beta}_l\rangle$ is an l-tuple of terms $\overline{s}_l$ where $s_j \in T^{\beta_j}_{\Gamma\cup\{a(i)\mapsto\alpha_i \mid 0\le i<l\}}(\Sigma)$. Let PI be the set of all parameter instantiations.*

The application *of a parameter instantiation $\overline{s}$ (from arity $\langle\overline{\alpha}_m\rangle$ to arity $\langle\overline{\beta}_l\rangle$) to parameter instance $\overline{r}$ (of arity $\langle\overline{\alpha}_m\rangle$) is defined by $\overline{r}'_l := \overline{r}_m \cdot \overline{s}_l$ where*

$$r'_j = s_j[i \mapsto r_i \mid 0 \le i < m].$$

The composition *of parameter instantiations $\overline{s}_l : \langle\overline{\alpha}_m\rangle \to \langle\overline{\beta}_l\rangle$ and $\overline{s}'_k : \langle\overline{\beta}_l\rangle \to \langle\overline{\gamma}_k\rangle$ is defined by $\overline{s}''_k := \overline{s}_l;\overline{s}'_k : \langle\overline{\alpha}_m\rangle \to \langle\overline{\gamma}_k\rangle$ where*

$$s''_j = s'_j[i \mapsto s_i \mid 0 \le j < k].$$

LEMMA 4.2. *1. Composition of parameter instantiations is associative and there is a unit element for each arity, namely $(0,\ldots,k-1) : \langle\alpha\rangle_k \to \langle\alpha\rangle_k$.*

2. Composition of parameter instantiations is compatible with application in the sense that $(\overline{r}\cdot\overline{s})\cdot\overline{s}' = \overline{r}\cdot(\overline{s};\overline{s}')$ (assuming that the arities fit).

In the following we make use of a slightly nonstandard definition of a (labeled) directed graph. In our definition, there may be more than one directed edge between a given pair of nodes. This setting is mostly useful for labeled graphs, where these edges may have different labels.

DEFINITION 4.3. *A directed graph is a tuple (V, E, src, trg) where V is the set of vertices and E is the set of edges. The two components src and trg are both mappings $E \to V$ that determine the source vertex and the target vertex of an edge. Often the mappings and E are given implicitly. Stating that $v_1 \longrightarrow v_2$ is an edge means that there is an element $e \in E$ such that $src(e) = v_1$ and $trg(e) = v_2$.*

A Λ-labeled directed graph has a fifth component $lab : E \to \Lambda$ that maps each edge to a label. Here the notation $v_1 \overset{l}{\longrightarrow} v_2$ means that there is some $e \in E$ such that $src(e) = v_1$, $trg(e) = v_2$, and $lab(e) = l$.

Often we just state the sets of vertices and edges as in (V, E) and add the labeling informally.

The *instantiation graph* is the registry of all calls (with parameters) to a nonterminal during the transformation of a macro grammar to a context-free grammar. It also encompasses all calls that may ever occur in a derivation of the macro grammar. A node of this graph is a pair of a nonterminal and a suitable parameter instance. The edges of the

graph indicate the caller-callee relationship and their labels indicate the parameter instantiation taking place.

DEFINITION 4.4. *The* instantiation graph $\mathcal{IG}(\mathcal{M})$ *of a macro grammar* $\mathcal{M} = ((N, a), \Sigma, P, S)$ *is the smallest* PI*-labeled graph* $G = (V, E)$ *such that the following holds.*

- $V \subseteq \{(A, \overline{r}_m) \mid A \in N, \overline{r}_m : a(A)\}$.
- $(S, ()) \in V$ *is a node of* G *if* S *is the start symbol of* $\mathcal{M}$.
- *If* $X = (A, \overline{r}) \in V$ *is a node,* $A \to w$ *a production of* $\mathcal{M}$*, and* $B(\overline{s})$ *is a subterm of* w*, then* $Y = (B, \overline{r} \cdot \overline{s}) \in V$ *is also a node and* $X \xrightarrow{\overline{s}} Y \in E$ *is a labeled edge.*

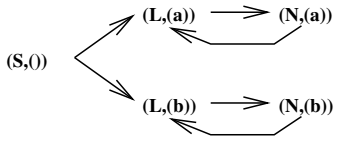
EXAMPLE 4.5. *As an example, consider the instantiation graphs for the grammars from Figure 9.*

$\mathcal{IG}(G_{abc})$:

$(S, ()) \to (F, (\varepsilon, \varepsilon, \varepsilon)) \to (F, (a, b, c)) \to (F, (aa, bb, cc)) \to \ldots$

This graph has infinitely many vertices, so that the specialization would not terminate.

$\mathcal{IG}(G_{list})$:



This graph has finitely many vertices and its grammar's specialization terminates.

The instantiation graph contains all the information needed to extract a context-free grammar equivalent to the underlying macro grammar $\mathcal{M}$. To state this equivalence, we need to make the extraction of the macro invocations ($B(\overline{s})$ in Definition 4.4) more formal using the following definition.

DEFINITION 4.6. *The following function is defined inductively on terms (for* $x \in \Sigma$, $C \in N$, $n \in \mathbf{N}$, $0 \le j < n$*).*

$$\begin{array}{lcl} |x|_{\overline{r}} & = & x \\ |C|_{\overline{r}} & = & C \\ |j|_{\overline{r}} & = & r_j \\ |s(s_0, \ldots, s_{l-1})|_{\overline{r}} & = & (|s|_{\overline{r}}, |s_0|^*_{\overline{r}} \ldots |s_{l-1}|^*_{\overline{r}}) \end{array}$$

We write $|w|^*_{\overline{r}}$ *for the obvious homomorphic extension to words and we write* $|a|$ *if* $\overline{r}$ *does not matter.*

LEMMA 4.7. *Let* $\mathcal{M} = (\Gamma, \Sigma, P, S)$ *be a macro grammar.*

If $\mathcal{IG}(\mathcal{M})$ *is finite, then there exists a context-free grammar* $\mathcal{G} = (N', \Sigma, P', S')$ *such that* $S() \overset{*}{\Rightarrow}_{\mathcal{M},OI} w$ *iff* $S' \overset{*}{\Rightarrow}_{\mathcal{G}} w$.

PROOF. Let $\mathcal{IG}(\mathcal{M}) = (V, E)$ and define $\mathcal{G}$ by $N' = V$, $S' = (S, \varepsilon)$, and

$$P' = \{(A, \overline{r}) \to |w|^*_{\overline{r}} \mid (A, \overline{r}) \in V, A \to w \in P\}.$$

$\mathcal{G}$ is a well-defined context-free grammar because V is finite. It remains to show that the two grammars are equivalent.

To this end, we need to generalize the claim as follows: for all v and w, $v \overset{*}{\Rightarrow}_{\mathcal{M},OI} w$ iff $|v|^* \overset{*}{\Rightarrow}_{\mathcal{G}} |w|^*$. In this case, $\overline{r}$ does not matter because neither v nor w contain any j.

The proof is by induction on n, the number of derivation steps. For $n = 0$ the result is trivial. If $n > 0$ then the derivation for $\mathcal{M}$ splits as follows:

$$v = u_1 A(\overline{r}) u_2 \Rightarrow_{\mathcal{M},OI} u_1 t[j \mapsto r_j] u_2 \overset{(n-1)}{\Rightarrow}_{\mathcal{M},OI} w$$

if $A \to t$ is a production. Now it holds that

$$|v|^* = |u_1 A(\overline{r}) u_2|^* = |u_1|^* |A(\overline{r})| |u_2|^* \Rightarrow_{\mathcal{G}} |u_1|^* |t|^*_{\overline{r}} |u_2|^*$$

by definition of $\mathcal{G}$. Because $|t|^*_{\overline{r}} = |t[j \mapsto r_j]|^*$ and the inductive hypothesis is applicable to the $n-1$ step derivation, it follows that $|u_1 t[j \mapsto r_j] u_2|^* \overset{(n-1)}{\Rightarrow}_{\mathcal{G}} |w|^*$ as desired. □

The above development yields a necessary and sufficient criterion when a macro grammar generates a context-free language. All we need to do is to test the instantiation graph for finiteness. To perform this test effectively, we define another graph—the transition graph—and consider a finite abstraction of this graph. This abstraction will help us decide finiteness of the instantiation graph.

5. THE TRANSITION GRAPH AND ITS ABSTRACTION

The transition graph contains the same information as the instantiation graph, but presents it in a different packaging. While it is guaranteed to have finitely many nodes, it may have infinitely many edges.

DEFINITION 5.1. *Let* $\mathcal{M} = ((N, a), \Sigma, P, S)$ *be a macro grammar. The* transition graph $\mathcal{TG}(\mathcal{M})$ *is a labeled directed graph with* N *as the set of nodes and edges labeled with parameter instantiations. The set of edges characterizes the* primitive transitions *with correspond directly to macro calls from one production to another:*

$$E = \{A \xrightarrow{\overline{s}} B \mid A \to w \in P, B(\overline{s}) \in w\} \subseteq N \times PI \times N.$$

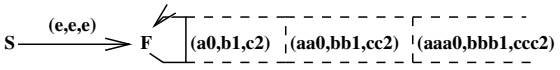
The closed transition graph $\mathcal{CTG}(\mathcal{M})$ *is the closure of* $\mathcal{TG}(\mathcal{M})$ *under transitivity. That is, its set of nodes is also* N *but its set of edges is the smallest set* E^* *such that* $E \subseteq E^*$ *and if* $A \xrightarrow{\overline{s}} B \in E^*$ *and* $B \xrightarrow{\overline{s}'} C \in E^*$ *then* $A \xrightarrow{\overline{s};\overline{s}'} C \in E^*$.

By construction, each path in the transition graph corresponds to an edge in the closed transition graph. The transition graph of a macro grammar is always finite whereas the closed transition graph is finite if and only if the instantiation graph is finite.

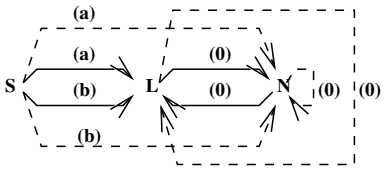
LEMMA 5.2. *There is a path in* $\mathcal{IG}(\mathcal{M})$ *that visits infinitely many different nodes of* $\mathcal{IG}(\mathcal{M})$ *iff there is an infinite number of edges in* $\mathcal{CTG}(\mathcal{M})$.

EXAMPLE 5.3. *Let's have a look at the (closed) transition graphs of the two examples from Figure 9. The solid arrows correspond to $\mathcal{TG}()$ whereas the dashed arrows are the additional edges in $\mathcal{CTG}()$.*

G_{abc}: This graph has infinitely many edges and is cut off on the right.



G_{list}: This graph has finitely many edges.



Coming back to finiteness of $\mathcal{IG}(\mathcal{M})$, we wish to check for the criterion specified in the following definition.

DEFINITION 5.4. *An edge $A \xrightarrow{\overline{s}} A$ in $\mathcal{CTG}(\mathcal{M})$ is* self-embedding *if there is some j such that s_j contains j but $s_j \neq j$.*

While it is straightforward to prove that the existence of a self-embedding transition is sufficient to construct an infinite $\mathcal{IG}(\mathcal{M})$, it is surprisingly hard to prove that the non-existence of a self-embedding guarantees finiteness of the instantiation graph. In addition, there is the issue of finding a suitable, finite abstraction of the transitions $\overline{s}$ without loosing the ability to detect self-embeddings.

To define this abstraction, we need to recall some basic definitions about multisets. A multiset M is a function $A \to \mathbf{N}$, for some underlying set A, which contains each element $a \in A$ with some multiplicity $M(a) \in \mathbf{N}$. The union $M_1 \cup M_2$ of multisets is defined by taking the *maximum* of the elements' multiplicities. In contrast, the join operation $M_1 \uplus M_2$ takes the *sum* of the the multiplicities. The cardinality $|M|$ of a multiset is the sum of its elements' multiplicities.

To proceed, we first construct a suitable abstraction for transitions. To create this abstraction in a compositional way requires to work with an instrumented instantiation graph. The instrumentation keeps track of the parameter positions of primitive transitions through which a value has been propagated already and if it has been augmented on that path. It does so by augmenting each component with a multiset of pairs $(\overline{p}, i)$ of a primitive transition and an argument number.

Hence, the instrumentation of a primitive transition $A \xrightarrow{\overline{p}} B$ is a vector of multisets $\overline{M}$ where, for $0 \leq j < a(B)$,

$$M_j = \begin{cases} \{\,\} & \text{if } p_j = \varepsilon \vee p_j \in \{0, \ldots, a(A)-1\} \\ \{(\overline{p}, j)\} & \text{otherwise.} \end{cases}$$

The composition of instrumented parameter instantiations is defined in the obvious way. The composition of $\overline{s}_l; \overline{s}'_k$ is $\overline{s}''_k$ where the components of the instantiation are defined as before in Definition 4.1. If $\overline{M}_l$ and $\overline{M}'_k$ are the multisets associated with $\overline{s}_l$ and $\overline{s}'_k$, then the components of $\overline{M}''_k$ are defined by

$$M''_i = M'_i \uplus \bigcup_{j \in var(s'_i)} M_i$$

where $\cup$ is the union of multisets and $\uplus$ is the join operation. While the union collects contributions to the result from different sources, the join operation models the sequential composition.

To build an abstraction of an instrumented parameter instantiation requires to abstract the term part as well as the multiset. We abstract multisets to 2-*bounded multisets*. They limit the multiplicity function by abstracting the multiplicities $2, \ldots$ to ∞. This cutoff yields a finite abstraction, provided the underlying set of elements is finite.

More formally, a 2-bounded multiset with elements from set A is a mapping M from A to $B = \{0, 1, \infty\}$. The operations $+$ and max on B are the obvious abstractions of addition and maximum of positive integers. If $M(a) = 0$ then a is absent from the multiset, $M(a) = 1$ denotes a's presence with multipicity 1, and $M(a) = \infty$ indicates that a is present more than once.

For a 2-bounded multiset over a finite set, the cardinality is ∞ if there is at least one element with multiplicity ∞.

While a concrete transition consists of a vector of terms (the parameter instantiation) and a vector of multisets, an abstract transition is a vector of abstract terms and a vector of 2-bounded multisets. (Where convenient we view either kind of transition as a vector of pairs, too.)

An abstract term abstracts from a term by capturing the variables contained in the term and by identifying whether a transition constructs a nontrivial term. Here, a term is nontrivial if it is neither ε nor i (just a parameter). Hence, an abstract term, v, is one of

E,	the abstraction of the empty word ε,
$J(i)$,	which denotes a value copied from argument position $i \in \mathbf{N}$,
$C(K)$,	which denotes a nontrivial term built from values from the argument positions mentioned in $K \subseteq \mathbf{N}$.

Abstract terms support a commutative operation $\oplus$. This operation abstracts the concatenation operation $\cdot$ on concrete terms. E is the unit of $\oplus$.

$$\begin{array}{lcl} J(i) \oplus J(k) & = & C(\{i,k\}) \\ J(i) \oplus C(K) & = & C(K \cup \{i\}) \\ C(K) \oplus C(L) & = & C(K \cup L) \end{array}$$

The intuition behind this operation is as follows. The abstraction function is a monoid homomorphism from concrete terms $T_{\Gamma \cup I}$ with ε and concatenation to abstract terms with E and $\oplus$. Hence, ε must be mapped to the unit E. If the

original word refers twice to arguments i and k (not necessarily different), it does not copy the arguments anymore, but builds a term containing both arguments (or one argument twice). Joining a term building operation with a copy operations (or another term building operation) yields a term building operation containing the union of the argument positions.

DEFINITION 5.5. *The abstraction of a concrete transition,* an abstract parameter instantiation, *is defined by*

$$\alpha(A \xrightarrow{\overline{s}} B) = A \xrightarrow{\overline{c}} B$$

where $c_j = (\alpha(s_j), \alpha(M_j))$ and M_j are the multisets associated with $\overline{s}$. Let API be the set of all abstract parameter instantiations.

The abstraction of a term is defined by

$$\begin{array}{lcll} \alpha(\varepsilon) & = & E & \\ \alpha(x) & = & C(\emptyset) & x \in \Sigma \\ \alpha(i) & = & J(i) & \\ \alpha(D(w_1, \ldots, w_n)) & = & C(\emptyset) \oplus \alpha(w_1) \oplus \cdots \oplus \alpha(w_n) & \\ & & D \in N, a(D) = n & \\ \alpha(v \cdot w) & = & \alpha(v) \oplus \alpha(w). & \end{array}$$

The abstraction of a multiset M is defined by

$$\alpha(M)(a) = \begin{cases} 0 & M(a) = 0 \\ 1 & M(a) = 1 \\ \infty & M(a) > 1. \end{cases}$$

DEFINITION 5.6. *Define abstract composition $A \xrightarrow{\overline{e}} C = A \xrightarrow{\overline{c}} B; B \xrightarrow{\overline{d}} C$ by*

$$e_j = \begin{cases} d_j & \text{if } d_j = (E, M') \\ (v, M \uplus M') & \text{if } d_j = (J(i), M'), c_i = (v, M) \\ (v, M \uplus M') & \text{if } d_j = (C(\{k_1, \ldots, k_m\}), M') \\ & \text{and } (v, M) = (C(\emptyset), \{\ \}) \oplus \bigoplus_{i=1}^{m} c_{k_i} \end{cases}$$

Clearly, abstraction is compatible with composition.

LEMMA 5.7. *If $\alpha(A \xrightarrow{\overline{s}} B) = A \xrightarrow{\overline{c}} B$ and $\alpha(B \xrightarrow{\overline{t}} C) = B \xrightarrow{\overline{d}} C$ then $\alpha(A \xrightarrow{\overline{s};\overline{t}} C) = A \xrightarrow{\overline{c};\overline{d}} C$.*

We can also go back from abstract transitions to concrete ones. However, some care must be taken to only obtain realizable concretizations, *i.e.*, transitions which are actually composable from primitive ones.

DEFINITION 5.8. *The* concretization function γ *is*

$$\begin{array}{l} \gamma(A \xrightarrow{\overline{c}} B) = \{\ A_0 \xrightarrow{\overline{p}^{(1)};\ldots;\overline{p}^{(m)}} A_m \mid A_0 = A, \\ \qquad\qquad A_{i-1} \xrightarrow{\overline{p}^{(i)}} A_i \in E(\mathcal{TG}(\mathcal{M})), A_m = B, \\ \qquad\qquad A \xrightarrow{\overline{c}} B = \alpha(A_0 \xrightarrow{\overline{p}^{(1)};\ldots;\overline{p}^{(m)}} A_m)\} \end{array}$$

where the $\overline{p}^{(i)}$ are drawn from the primitive transitions.

DEFINITION 5.9. *Let $\mathcal{M} = ((N, a), \Sigma, P, S)$ be a macro grammar. The* abstract transition graph $\mathcal{ATG}(\mathcal{M})$ *is a labeled directed graph with set of nodes N and edges $E \subseteq N \times API \times N$ labeled with abstract parameter instantiations:*

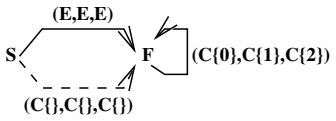
$$E = \{A \xrightarrow{\overline{c}} B \mid A \to w \in P, B(\overline{s}) \in w, \overline{c} = \alpha(\overline{s})\}.$$

The closed abstract transition graph $\mathcal{CATG}(\mathcal{M})$ *is the closure of $\mathcal{ATG}(\mathcal{M})$ under transitivity. Its set of nodes is N and its set of edges is the smallest set E^* such that $E \subseteq E^*$ and if $A \xrightarrow{\overline{c}} B \in E^*$ and $B \xrightarrow{\overline{d}} C \in E^*$ then $A \xrightarrow{\overline{c};\overline{d}} C \in E^*$.*

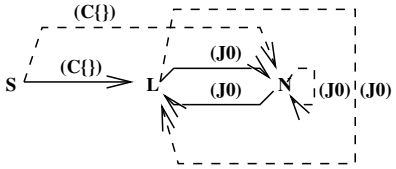
Both $\mathcal{ATG}(\mathcal{M})$ and $\mathcal{CATG}(\mathcal{M})$ are finite because there are only finitely many pairs of nonterminals A and B and, for each pair A and B, there are finitely many abstract transitions $A \xrightarrow{\overline{c}} B$: Each of the finitely many components of $\overline{c}$ contains one of the finitely many possible term abstractions and a 2-bounded multiset over a finite set, of which there are also finitely many.

EXAMPLE 5.10. *The term abstraction parts of the transition graphs for the grammars from Figure 9 look as follows.*

G_{abc}:



G_{list}:



6. FINITENESS OF THE ABSTRACT TRANSITION GRAPH

Having established a finite abstraction of the transition graph in the previous section, this section is devoted to deciding if an abstract transition has infinitely many concretizations. To this end, we first define a size function for abstract transitions, which counts the number of elements in all multisets associated to a term-building position. Then, an abstract transition is infinite if its size is infinite.

DEFINITION 6.1. *Let $A \xrightarrow{\overline{c}} B$ be an abstract transition. The* size of $\overline{c}$ *is defined as*

$$|\overline{c}| = |\bigcup_j \{M_j \mid c_j = (v_j, M_j), \exists K. v_j = C(K)\}|.$$

The $\overline{c}$ is finite *if its size is finite, that is, if $|\overline{c}| < \infty$.*

THEOREM 6.2. *Suppose that $A \xrightarrow{\overline{c}} B$ is* finite. *Then the associated set of concrete transitions $\gamma(A \xrightarrow{\overline{c}} B)$ is finite.*

This theorem can be proved by induction on the (finite) size of the abstract transition. However, getting the induction through requires a strengthened assumption as stated in the following lemma.

LEMMA 6.3. *Let $A \xrightarrow{\overline{c}} B$ be an abstract transition and $N \subseteq \{0,\ldots,a(B)-1\}$, a subset of B's argument positions. Let further $|\overline{c}|_N = |\bigcup_{j\in N}\{M_j \mid c_j = (v_j, M_j), \exists K.v_j = C(K)\}|$ be the size of the N-components of $\overline{c}$.*

If $|\overline{c}|_N$ is finite then the set $\gamma(A \xrightarrow{\overline{c}} B)$ projected on its N-components is finite.

Theorem 6.2 follows from the lemma by choosing

$$N = \{0,\ldots,a(B)-1\}.$$

The use of N is required because the size of an abstract transition is only subadditive: The size of a composed transition may be smaller than the size of its constituends because a subsequent constituend may ignore argument positions and thus cancel intermediate growth. As an example consider the composition of two one-component abstract transitions where the second cancels the effort of the former:

$$(C(\{0\}), \{(p,0)\}); (E, \{\,\}) = (E, \{\,\}).$$

The set N indicates the argument positions which are not canceled by subsequent constituends, which makes the N-indexed size monotonic (subject to the correct choice of N).

Proof. The proof of Lemma 6.3 is by course-of-value induction on the size $|\overline{c}|_N$ where $\overline{c}$ has components $c_j = (v_j, M_j)$.

If $|\overline{c}|_N = 0$ then, for each $j \in N$, $v_j \in \{E\} \cup \{J(i) \mid 0 \le i < a(A)\}$. Hence, for each $\overline{s} \in \gamma(\overline{c})$ and for each $j \in N$, $s_j = \varepsilon$ or $s_j = i$, for some $0 \le i < a(A)$.

If $|\overline{c}|_N > 0$ then let $\mathcal{N} = \bigcup_{j\in N, \exists K.v_j = C(K)} M_j$ and split $\gamma(\overline{c}) = \bigcup_{\mathcal{I}\subseteq\mathcal{N}} \mathcal{S}_\mathcal{I}$ into disjoint subsets (some of which may be empty) indexed by the subsets $\mathcal{I}$ of $\mathcal{N}$ as follows.

A transition $\overline{s} = \overline{p}^{(1)};\cdots;\overline{p}^{(m)}$ (considered as composition of primitive transitions) belongs to subset $\mathcal{S}_\mathcal{I}$ where $\mathcal{I} = \{(\overline{p}^{(k)}, i_1), \ldots, (\overline{p}^{(k)}, i_q)\} \subseteq \mathcal{N}$ if $k \in \{1,\ldots,m\}$ is maximal such that $|\alpha(\overline{p}^{(k+1)});\cdots;\alpha(\overline{p}^{(m)})|_N = 0$, but for $\overline{c}^{(2)} = \alpha(\overline{p}^{(k)});\cdots;\alpha(\overline{p}^{(m)})$ it holds that $\mathcal{I} = \bigcup\{M_j \mid j \in N, c_j^{(2)} = (v_j, M_j), \exists K.v_j = C(K)\}$ and thus $|\overline{c}^{(2)}|_N = q > 0$. Such a k must exist, otherwise there is a contradiction to $|\overline{c}|_N > 0$. Essentially, $\overline{p}^{(k)}$ is the last primitive transition that contributes to the growth in the N-components of $\overline{s}$. The subsequent transitions only pass on values, cancel values, or perform computation outside of N.

The point is that for each sequence in $\mathcal{S}_\mathcal{I}$ there is such a k and the abstraction of the sequence can be split into

- $\overline{c}^{(1)} = \alpha(\overline{p}^{(1)});\cdots;\alpha(\overline{p}^{(k-1)})$,
- $\overline{c}^{(2)} = \alpha(\overline{p}^{(k)});\cdots;\alpha(\overline{p}^{(m)})$,

where $D' \xrightarrow{\overline{p}^{(k)}} D$ such that

- $0 < |\overline{c}^{(2)}|_N = q \le |\overline{c}|_N$, that is, while its size is not necessarily less than the original size, the size q can be attributed to a single primitive transition $\overline{p}^{(k)}$, which we analyse next;
- there exists some $N_1 \subseteq \{0,\ldots,a(D')-1\}$ such that $|\overline{c}^{(1)}|_{N_1} = |\overline{c}|_N - q < |\overline{c}|_N$. N_1 consists of the positions not ignored by $\overline{c}^{(2)}$. These positions are exactly the indices that occur in the abstract terms of the N-components of $\overline{c}^{(2)}$.

The inductive hypothesis is applicable to $\overline{c}^{(1)}$ so that the N_1-components of the composition $\overline{p}^{(1)};\cdots;\overline{p}^{(k-1)}$ assume only finitely many different values. Next, the primitive transition $\overline{p}^{(k)}$ is fully determined by $\mathcal{I}$.[2] Clearly, it only produces finitely many results in the components that only depend on N_1. Now, by construction, the N-components of $\overline{p}^{(k+1)};\cdots;\overline{p}^{(m)}$ must be drawn from ε and $\{0,\ldots,a(D)-1\}$ so that this transition can only perform one of finitely many possible choices from the output components of $\overline{p}^{(k)}$. Hence, the N-components only assume finitely many values.

The desired result follows from the observation that there are finitely many subsets $\mathcal{I} \subseteq \mathcal{N}$ and each of them yields finitely many values as just explained. Some of the subsets $\mathcal{S}_\mathcal{I}$ may be empty, for example, the one indexed with $\mathcal{I} = \{\,\}$ or any index which mentions different primitive transitions. □

LEMMA 6.4. *The following statements are equivalent.*

1. *There exists an infinite transition $A \xrightarrow{\overline{c}} B$.*
2. *There exists an infinite transition $D \xrightarrow{\overline{d}} D$.*
3. *There exists an infinite, idempotent transition $D \xrightarrow{\overline{e}} D$.*
4. *There exists an idempotent, self-embedding transition $D \xrightarrow{\overline{e}} D$.*

Proof. Case (1) ⇒ (2)**.** Suppose that there exists an infinite transition $A \xrightarrow{\overline{c}} B$. Take any concretization $\overline{s}$ of $\overline{c}$. Because $\overline{c}$ is infinite, there is some j such that $c_j = (v_j, M_j)$ with $v_j = C(K)$ and, for some $\overline{p}$ and i, the multiplicity of $(\overline{p}, i)$ in M_j is greater than one, and $D' \xrightarrow{\overline{p}} D$. But that means that $\overline{s}$ splits up into

$$A \longrightarrow D' \xrightarrow{\overline{p}} D \xrightarrow{\overline{t}} D' \xrightarrow{\overline{p}} D \longrightarrow B$$

where $\alpha(\overline{t};\overline{p})_i = (C(K'), M')$ with $(\overline{p}, i) \in M'$ and $i \in K'$. Thus, $D \xrightarrow{\overline{t};\overline{p};\overline{t};\overline{p}} D$ abstracts to an infinite transition $D \xrightarrow{\overline{d}} D$ where $\overline{d} = \alpha(\overline{t};\overline{p};\overline{t};\overline{p})$.

Case (1) ⇐ (2)**.** Trivial.

Case (2) ⇒ (3)**.** If $D \xrightarrow{\overline{d}} D$ is infinite, then its concretizations can be split as in the previous case to find some concrete $D' \xrightarrow{\overline{t}\overline{p}} D'$. Wlog, let $D' = D$ and $\overline{d} = \alpha(\overline{t}\overline{p};\overline{t}\overline{p})$.

[2] There may be infinitely many lengths m of sequences and infinitely many positions k; the point is that each such sequence may be split into parts $\overline{c}^{(1)}$ and $\overline{c}^{(2)}$ with well-defined behavior.

Because the set of abstractions is finite, the sequence $(\overline{d})$, $(\overline{d};\overline{d})$, $(\overline{d};\overline{d};\overline{d})$, and so on must become stationary at some $n > 1$. Choosing $\overline{e} = \overline{d}^{(n)}$ yields an infinite, idempotent transition.

Case (2) $\Leftarrow$ (3)**.** Trivial.

Case (1) $\Rightarrow$ (4)**.** Select $\overline{t}$ and $\overline{p}$ as in the proof for (1) $\Rightarrow$ (2). Because of the choice of $\overline{t}$ and $\overline{p}$, $\overline{d} = \alpha(\overline{t};\overline{p})$ is self-embedding in position i. By finiteness of the abstraction, there is a finite power of $\overline{d}$ which is idempotent (each power is also self-embedding).

Case (4) $\Rightarrow$ (3)**.** Suppose that $D \xrightarrow{\overline{e}} D$ is idempotent and self-embedding, say, $e_i = (C(K'), M')$ with $i \in K'$. In any concretization $\overline{s} = \overline{p}^{(1)};\ldots;\overline{p}^{(m)}$ there must be some k and j so that $p_j^{(k)}$ is responsible for the C in e_i which means that $(p^{(k)}, j) \in M'$. Thus, for $\overline{d} = \overline{e};\overline{e}$ it holds that $d_i = (C(K'), M'')$ with $(p^{(k)}, j) \in M''$ with multiplicity > 1 so that $\overline{d}$ is infinite and idempotent. □

Theorem 6.2 gives rise to a sufficient criterion for ensuring quasi-termination.

THEOREM 6.5. *The concrete transition graph $\mathcal{CTG}(\mathcal{M})$ is finite iff all idempotent transitions in $\mathcal{CATG}(\mathcal{M})$ are finite.*

Proof. If all idempotent $\overline{c} \in \mathcal{CATG}(\mathcal{M})$ are finite, then Lemma 6.4 ensures that all transitions in $\mathcal{CATG}(\mathcal{M})$ are finite. Hence, Theorem 6.2 ensures that there are only finitely many concretizations for each abstract transition, so that $\mathcal{CTG}(\mathcal{M})$ is finite.

If any idempotent $\overline{c}$ is infinite, then Lemma 6.4 tells us that there is also some self-embedding transition $\overline{d}$. From the concretizations of $\overline{d}$, we can construct infinitely many configurations. □

An analysis based on Theorem 6.5 is still very expensive. It amounts to constructing $\mathcal{CATG}(\mathcal{M})$ and then checking all idempotent transitions for finiteness. However, Lemma 6.4 yields a simplification. By item 4 of the lemma, the existence of an infinite abstract transition is equivalent to the existence of a self-embedding idempotent transition. Hence, an implementation of the analysis need not keep track of the 2-bounded multisets but that it is sufficient to work with just the term abstraction and detect self-embedding.

7. IMPLEMENTATION

We have implemented grammar specialization as a standalone tool[3] geared at specializing input grammars for bison [7] or yacc [14]. It extends the syntax of rules slightly by allowing a macro definition on the left-hand side of a rule and macro invocations on the right-hand side. Figure 10 contains an example excerpted from a yacc grammar extended with parameterized rules and Figure 11 contains the

[3]The code is available at `http://www.informatik.uni-freiburg.de/~thiemann/haskell/YSPEC`.

```
%{
/* API for generic list construction */
List makeSingleton (void * elem);
List addLast (List, void * elem);
%}

/* parameterized rule */
commalist.1 (item)
: item
        { $$ = makeSingleton ((void *)$1); }
| commalist.1 (item) ',' item
        { $$ = addLast ($1, (void *)$3); }
;

/* two uses, both returning results of type List */
patternlist:    commalist.1(pattern) ;
explist1:       commalist.1(expr) ;
```

Figure 10: Excerpt of a parameterized Yacc grammar.

```
/* parameterized rule */
commalist.1__nexpr__1_0
: expr_
        { $$ = makeSingleton ((void *)$1); }
| commalist.1__nexpr__1_0 ',' expr_
        { $$ = addLast ($1, (void *)$3); }
;

/* parameterized rule */
commalist.1__npattern__1_0
: pattern_
        { $$ = makeSingleton ((void *)$1); }
| commalist.1__npattern__1_0 ',' pattern_
        { $$ = addLast ($1, (void *)$3); }
;
```

Figure 11: Specialized fragment.

corresponding fragment of the specialized grammar (which is suitable for processing with yacc).

The implementation consists of roughly 1000 lines of Haskell code [19]. About half of the code deals with parsing and printing bison grammars. The code is written in a framework style which abstracts over the representation of grammars, so that specialization backends for other kinds of parser generators can be written easily. The main effort is in writing the parser for grammar files.

One extension that we have contemplated, but not implemented is code parameters. Right now, a grammar author must resort to design generic datastructures like the `List` type in Figure 10 with parameterized rules. While this choice is pragmatic, it does have a number of drawbacks. First, type safety is not guaranteed in a language like C which does not have generics. The users of the rules have to insert the correct casts to extract values out of the generic datastructures. Second, inside of one program, the use of the rules is restricted to one particular implementation of the generic datastructure (`List` in the example).

These drawbacks could be addressed with code parameters. A code parameter is an additional parameter to a nonterminal that takes an action, *i.e.*, a code fragment surrounded by curly braces, as an argument. The specialization process then not only expands grammatical parameters (as demonstrated in this paper), but also actions. For instance, a parameterized nonterminal like `commalist.1` might take two action parameters to be used in place of the generic actions. This way, each call to `commalist.1` could provide its own implementation for `makeSingleton` and `addLast`, thus solving the two problems outlined in the previous paragraph. This extension would fit in nicely with the theory developed for detecting termination.

However, we have refrained from implementing this extension because a realistic implemention would have to support code splices, where a code parameter is inserted into an action skeleton specified with the parameterized rule. A good implementation of code splices, in turn, requires dealing with the delicate issues of hygiene and name capture and thus parsing of C code. Such a project would go well beyond the proof-of-purpose implementation that we provide.

8. RELATED WORK

Parser combinators [22, 13] are a highly flexible way of specifying parsers in functional programming languages. In particular, the use of polymorphic functions and parameterized parsers is a natural way of structuring code. In contrast to the present work, parser combinators are restricted to perform predictive or top-down parsing. Recent advances [21] have widened their scope considerably with respect to earlier, inefficient proof-of-concept implementations. The present work makes some of the polymorphism and flexibility that make parser combinators attractive available to all parser generators.

Cameron introduced a syntactic extension for context-free grammars to specifiy permutation phrases [3] and presents imperative pseudo-code demonstrating how to extend a predictive parser with the new construct without just simply expanding the grammar. Baars et al. [2] show how to add permutation phrases to a functional parser combinator library. Such extensions come for free with our grammar specialization technology.

The syntax definition formalism SDF [25] supports arbitrary context-free grammars and creates GLR parsers [16, 24, 20] for them. For convenience, right-hand sides may contain an extended set of regular operators. An SDF specification also defines a lexical syntax. SDF includes an abbreviation mechanism which works by expansion. However, the mechanism is much weaker because SDF neither specializes the grammars nor does it analyse the termination of the specialization.

Extensions of LR parsing with regular operators on the right-hand sides of productions have been explored by Chapman [4]. He extends the standard item set construction with new cases for these operators. However, the attached semantic actions are fixed to *e.g.* list construction.

The compiler construction toolkit Eli [11] also constructs bottom-up parsers from grammars with regular right-hand sides. The regular operators are expanded in a preceding grammar transformation. Extended BNF productions are more often supported by LL parser generators [18]. Our work makes such an expansion mechanism accessible to the programmer.

Van Wijngaarden (or W-) grammars [26] are a Turing-complete parameterized grammar formalism used in the definition of ALGOL 68. Conceptually, W-grammars consist of two-levels. The first level defines context-free languages of interpretations of grammar symbols. These interpretations are used to generate the actual grammar productions by substitution into rule templates. However, W-grammars are a conceptual modeling tool and are not geared at generating efficient recognizers. Rather, they have been designed for describing context-sensitive aspects of programming languages. They lack the conciseness and ease of use of direct parameterization, which is a familiar concept from programming practice. This two-level mechanism could be encoded with macro grammars (indeed, the expression of semantic conditions was one of Fischer's motivations for inventing them). However, the thus constructed macro grammars generate context-sensitive languages and are thus not amenable to our specialization framework.

The method for analyzing termination of the specialization are inspired by work on the termination of program specialization [9] and size-change termination [17]. The termination of program specialization is a much broader topic than the termination of grammar specialization. The latter is a special case of specialization for a first-order functional programming language which has some peculiar restrictions. First, the language does not have a conditional, that is, the invocation of a macro rule always invokes all macros in all right-hand sides of the rule. Second, the language has only increasing operators. A rule can either ignore a parameter, pass it along unchanged, or pass it on as part of a larger term. However, it cannot decompose or otherwise decrease a parameter. Thus, while the termination of grammar specialization is simpler than that of program specialization, it

```
(Perm x rest acc)   : x (acc rest)
                    | (rest (Pcons x acc))

(Pend acc)          : /* empty */

(Pcons x rest tail) : rest (Perm x tail)
(Pnil tail)         : (tail Pnil)

FieldModifiers : (Perm 'static'
                  (Perm 'final'
                   (Perm 'public' Pend))
                  Pnil)
```

Figure 12: A variable-length permutation phrase.

is sufficiently different to require its only analysis, as presented in the present paper.

9. FUTURE WORK

The main drawback of the present work is its restriction to first-order macros. For some problems, a higher-order formulation is the most appropriate. As an example, let's reconsider the encoding of permutation phrases from Section 2. It can be adjusted to permutation phrases of any number of items. However, each number n of items requires the definition of a separate nonterminal `PermPn` with the associated productions.

If we admit arbitrary arities (other than $\langle\rangle$) as parameters and also allow partial macro applications, as customary in functional programming, then we can state rules for encoding permutation phrases with a variable numbers of tokens to permute. Figure 12 shows support macros for such phrases.

The main workhorse is the `Perm` nonterminal, which takes three parameters. The `m` parameter contains the current parsing alternative. The `rest` parameter contains the remaining alternatives. The `acc` parameter accumulates unused parsing alternatives in the form of a list composed of `Pcons` and `Pnil`.

The idea is that `(Perm x rest acc)` tries to parse a phrase that either starts with `x` (first production) or with one of the remaining alternatives (second production). If parsing `x` succeeds, then `(acc rest)` resurrects the unused parsing alternatives by prepending them to the unused alternatives. This prepending is performed by the productions for `Pcons` and `Pnil`, they reactivate their elements at the same time by changing `Pcons` back to `Perm`. If parsing `x` does not succeed, then the unused alternative is `Pcons`ed on top of the accumulator `acc`.

If `Pend` is reached, then all alternatives have been tried and they are dismissed because there is no way to parse the phrase, anymore.

The specialization procedure would also work for this kind of grammar, but our analysis would require a significant extension.

10. CONCLUSION

Macro grammars extend context-free grammars with macro-like productions. Each nonterminal symbol may have parameters which can be arbitrarily instantiated at every invocation of the nonterminal. This extension enhances context-free grammars with procedural abstraction.

In general, macro grammars recognize context-sensitive grammars, which are inefficient to parse. We have defined grammar specialization to transform the productions of a macro grammar into a set of context-free productions. In general, this set is infinite, but we have developed a static analysis which gives sufficient and necessary conditions as to when the resulting set of context-free productions is finite (thus giving rise to a context-free grammar).

We have implemented grammar specialization in the `YSpec` tool, which is available from `http://www.informatik.uni-freiburg.de/~thiemann/haskell/YSPEC`.

11. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. I. Baars, A. Löh, and S. D. Swierstra. Functional pearl: Parsing permutation phrases. *J. Functional Programming*, 14(6):635–646, Nov. 2004.

[3] R. D. Cameron. Extending context-free grammars with permutation phrases. *Letters on Programming Languages and Systems*, 2(1-4):85–94, 1993.

[4] N. P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[5] K. Culik and R. Cohen. LR-regular grammars—an extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973.

[6] W. Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20(2):95–207, May 1982.

[7] C. Donnelly and R. Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, Nov. 1995. Part of the Bison distribution.

[8] M. J. Fischer. Grammars with macro-like productions. In *IEEE Conference Record of 9th Annual Symposium on Switching and Automata Theory*, pages 131–142, 1968.

[9] A. J. Glenstrup and N. D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Prog. Lang. and Systems*, 27(6):1147–1215, 2005.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[11] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, Feb. 1992.

[12] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in LNCS, pages 20–82. Springer, Copenhagen, Denmark, 1999.

[13] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.

[14] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[15] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[16] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *ICALP1974*, pages 255–269, 1974.

[17] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In H. R. Nielson, editor, *Proc. 2001 ACM Symp. POPL*, pages 81–92, London, England, Jan. 2001. ACM Press.

[18] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice & Experience*, 25(7):789–810, July 1995.

[19] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[20] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[21] S. D. Swierstra. Fast, error repairing parsing combinators. `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/`, Aug. 2003.

[22] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[23] P. Thiemann and M. Neubauer. Parameterized LR parsing. In G. Hedin and E. van Wyk, editors, *Fourth Workshop on Language Descriptions, Tools and Applications, LDTA 2004*, volume 110 of *ENTCS*, pages 115–132, Barcelona, Spain, Apr. 2004. Elsevier Science.

[24] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[25] M. van den Brand and P. Klint. ASF+SDF meta-environment user manual. `http://www.cwi.nl/projects/MetaEnv/meta/doc/manual/user-manual.html`, July 2002.

[26] A. e. van Wijngaarden. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14(2):79–218, 1969.

[27] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.