BRICS

Basic Research in Computer Science

BRICS RS-00-24 Brabrand & Schwartzbach: Growing Languages with Metamorphic Syntax Macros

Growing Languages with Metamorphic Syntax Macros

Claus Brabrand
Michael I. Schwartzbach

BRICS Report Series RS-00-24

ISSN 0909-0878 September 2000

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/24/`

Growing Languages with Metamorphic Syntax Macros

Claus Brabrand Michael I. Schwartzbach

BRICS,* Department of Computer Science
Ny Munkegade, building 540
8000 Aarhus C, Denmark
{brabrand,mis}@brics.dk

Abstract

"From now on, a main goal in designing a language should be to plan for growth." Guy Steele: Growing a Language, OOPSLA'98 invited talk.

We present our experiences with a syntax macro language augmented with a concept of *metamorphisms*. We claim this forms a general abstraction mechanism for growing (domain-specific) extensions of programming languages.

Our syntax macros are similar to previous work in that the compiler accepts collections of grammatical rules that extend the syntax in which a subsequent program may be written. We exhibit how almost arbitrary extensions can be defined in a purely *declarative* manner without resorting to compile-time programming. The macros are thus *terminating* in that parsing is guaranteed to terminate, *hygienic* since full α-conversion eliminates the risk of name clashes, and *transparent* such that subsequent phases in the compiler are unaware of them. Error messages from later phases in the compiler are tracked through all macro invocations to pinpoint their sources in the extended syntax.

A concept of *metamorphic rules* allows the arguments of a macro to be defined in an almost arbitrary *meta* level grammar and then to be *morphed* into the host language.

We show through examples how creative use of metamorphic syntax macros may be used not only to create convenient shorthand notation but also to introduce new language concepts and mechanisms. In fact, whole new languages can be created at surprisingly low cost. The resulting programs are significantly easier to understand and maintain.

This work is fully implemented as part of the `<bigwig>` system for defining interactive Web services, but could find use in many other languages.

---

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

1 Introduction

Syntax macros have long been advocated as a means for extending programming languages [23, 3, 12]. Recent interest in domain-specific and customizable languages poses the challenge of using macros to realize new language concepts and constructs or even to grow entire new languages [18, 1, 13].

Existing macro languages are either unsafe or not expressive enough to lift this challenge, since the syntax allowed for macro invocations is too restrictive. Also, many macro languages resort to compile-time meta programming, making them difficult to use safely.

In this paper we propose a new macro language that is at once sufficiently expressive and based entirely on simple declarative concepts like grammars and substitutions. Our contributions are:

- a *survey* of related work, identifying and classifying relevant properties;
- the design of a *declarative* and *expressive* macro language;
- a concept of *metamorphism* to capture almost arbitrary syntax of invocations;
- a full and *efficient implementation* for a syntactically rich host language; and
- *examples* of creative applications.

This work is carried out in the context of the `<bigwig>` project [7], but could find uses in many other host languages for which a top-down parser can be constructed. For a given application of our approach, knowledge of the host grammer is required. However, no special properties of such a grammar are used. In fact, we could easily build a generator that for a given host grammar automatically will provide a parser that supports our notion of syntax macros.

2 Related Work Survey

Figure 2 contains a detailed survey of the predominant macro languages that have previously been proposed. We have closely investigated the following eight macro languages and their individual semantic characteristics: C's preprocessor, `CPP` [9, 17]; the Unix macro preprocessor, `M4`; TEX's built-in macro mechanism; the macro mechanism of `Dylan` [16]; The `C++ templates` [19]; `Scheme`'s hygienic macros [8, 11]; the macro mechanism of the Jakarta Tool Suite, `JTS` [1]; and the Meta Syntactic Macro System, `MS`^{2} [23]. The survey has led us to identify and group 31 properties that characterize a macro language and that we think are relevant for comparing such work. Our own macro language is designed by explicitly considering exactly those properties; for comparison, it is included in the last column of the survey table in Figure 2.

2.1 General Properties

The paramount characteristic of a macro language is whether it operates at the *lexical* or *syntactical* level. Lexical macro languages allow tokens to be substituted by arbitrary sequences of characters or tokens. These definitions may be parameterized so that the substitution sequence contains placeholders for the actual parameters that

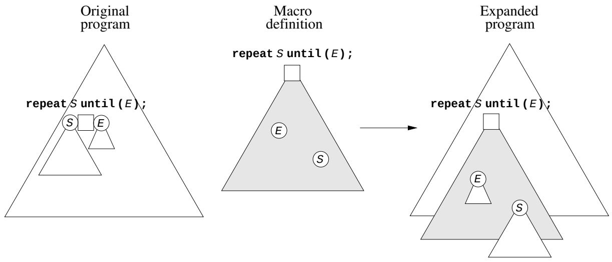


Figure 1: Syntax macros—operators on parse trees.

are themselves just arbitrary character sequences. `CPP`, `M4`, and TEX are well-known lexical macro languages. Conceptually, lexical macro processing precedes parsing and is thus ignorant of the syntax of the underlying host language. In fact, `CPP` and `M4` are language independent preprocessors for which there is no concept of host language. As a direct consequence of syntactic independence, all lexical macro languages share many dangers that can only be avoided by clever hacks and workarounds, which are by now folklore.

In contrast, syntactical languages operate on parse trees as depicted in Figure 1, which of course requires knowledge of the host language and its grammar. Syntactical macro languages include `C++ templates`, `Scheme`, `JTS`, and `MS`^{2}. The language `Dylan` is a hybrid that operates simutaneously on token streams and parse trees.

Some macro languages allow explicit *programming* on the parse trees that are being constructed, while others only use pattern matching and substitution. `CPP` only allows simple conditionals, `M4` offers simple arithmetic, `C++ templates` performs constant folding (which together with multiple definitions provide a Turing-complete compile-time programming language [21]), while `Scheme` and `MS`^{2} allow arbitrary computations.

2.2 Syntax Properties

The syntax for defining and invoking macros varies greatly. The main point of interest is how liberal an invocation syntax is allowed. At one end of the spectrum is `CPP` which requires parenthesized and comma separated actual arguments, while at the other end `Dylan` allows an almost arbitrary invocation syntax following an initial identifier.

2.3 Type Properties

There are two notions of *type* in conjunction with syntactical macro languages, namely *result types* and *argument types*, both ranging over the *nonterminals* of the host language grammar. These are often explicitly declared, by naming nonterminals of some standardized host language grammar. Using these, syntactical macro languages have the possibility of type checking definitions and invocations. Definitions may be checked to comply with the declared nonterminal return type of the macro, assuming that the placeholders have the types dictated by the arguments. Invocations may be checked to ensure that all arguments comply with their declared types. Often the argument type information is used to guide parsing, in which case this last check comes for free. If both checks are performed, no parse errors can occur as a direct consequence of macro expansion.

Only JTS and MS^2 take full advantage of this possibility. The others we have mentioned fall short in various ways, for example by not checking that the macro body conforms to the result nonterminal. The languages also differ in how many nonterminals from the host grammar can be used as such types.

2.4 Definition Properties

There are many relevant properties of macro definitions. The languages Dylan, CPP, and Scheme, allow more than one macro to be defined with the same name; a given invocation then selects the appropriate definition either by trying them out in the order listed or by using a notion of *specificity*.

Most macro languages have *one-pass* scope rules for macro definitions, meaning that a macro is visible from its lexical point of definition and onward. Only MS^2 employs a *two-pass* strategy, in which macro definitions are available even before their lexical point of definition. With one-pass scope rules, the order in which macros are defined is significant, whereas with two-pass scope rules the macro definitions may be viewed as a *set*. This has the nice property that the definition order can be rearranged without affecting the semantics. However, this is not completely true of MS^2 since its integrated compile-time programming language has one-pass scope rules. Some of the languages allow macros to be undefined or redefined which of course only makes sense in the presence of one pass scope rules. Many languages permit local macro definitions, but CPP, Dylan, and JTS have no such concept.

There are two kinds of macro recursion; *direct* and *indirect*. Direct recursion occurs when the body of a macro definition contains an invocation of itself. This always leads to infinite expansions. Indirect recursion occurs when a self-invocation is *created* during the expansion. This can be the result of a compile-time language *creating* a self-invocation or the result of the expansion being reparsed as in the *prescan* expansion strategy (see below). Without a compile-time programming language with side-effects to "break the recursion", indirect recursion also yields infinite expansions. The above generalizes straightforwardly to mutual recursion. Most of the languages tolerate some form of macro recursion, only CPP and JTS completely and explicitly avoid recursion.

An important issue is the argument structure that is allowed. Most languages require a fixed number of arguments for each macro. Scheme allows lists of argument, MS^2 allows lists, tuples, and optional arguments, while Dylan is the most flexible by allowing the argument syntax to be described by a user defined grammar.

	Property \ Language	CPP	M4	TeX	Dylan	C++ templates	Scheme	JTS	MS[2]	<bigwig>
Gen.	Level of operation	lexical	lexical	lexical	hybrid	syntactical	syntactical	syntactical	syntactical	**syntactical**
	Language dependent	no	no	yes	yes	yes	yes	yes	yes	**yes**
	Programmable	conditionals	arithmetic	yes	no	constant folding	yes	no	yes	**no**
Syntax	Definition keyword	`#define`	`define`	`\def`	`define macro`	`template`	`define-syntax`	`macro`	`syntax`	**`syntax`**
	Formal argument def	*id*	N/A	#1 to #9	?*id*:*id*, ?:*id*, ?<u>*id*</u>	<*nt id*>	*id*	*nt id*	\$\$*nt*::*id*, \$\$...::*id*	<*nt id*>, <<u>*id*</u>: *nt id*>
	Formal argument use	*id*	\$0 to \$9	#1 to #9	?*id*	*id*	*id*	*id*	\$*id*	<*id*>
	Invocation syntax	*id*(_, _, _)	*id*(_, _, _)	*id* ...	*id* ...	*id*<_, _, _>	(*id* _ _ _)	#*id*(_, _, _)	*id* ...	*id* **...**
Type	Arg. types declared	N/A	N/A	N/A	yes	yes	implicitly	yes	yes	**yes**
	Argument nonterminals	N/A	N/A	N/A	7+*token*	*id, type, const*	*s-exp*	6	15	**all 55**
	Argument types checked	N/A	N/A	N/A	yes	yes	yes	yes	yes	**yes**
	Result types declared	N/A	N/A	N/A	yes	no	implicitly	yes	yes	**yes**
	Result nonterminals	N/A	N/A	N/A	*stm, fcall, def*	*decl*	*s-exp*	5	15	**all 55**
	Result types checked	N/A	N/A	N/A	no	N/A	no	yes	yes	**yes**
Definition	Multiple definitions	no	no	no	yes	yes	yes	no	no	**yes**
	Definition selection	N/A	N/A	N/A	order listed	specificity	order listed	N/A	N/A	**specificity**
	Definition scope	one pass	one pass	one pass	one pass	one pass	one pass	one pass	two pass	**two pass**
	Undefine	yes	redefine	redefine	no	no	redefine	no	N/A	**N/A**
	Local macro definitions	no	yes	yes	no	yes	yes	yes	no	**yes**
	Direct recursion	no	yes	yes	yes	no	yes	no	no	**rejected**
	Indirect recursion	no	yes	yes	yes	yes	yes	N/A	yes	**N/A**
	Argument structure	fixed	fixed	fixed	grammar	fixed	list	fixed	option, list, tuple	**grammar**
Invocation	Body expansion	lazy	eager	lazy	lazy	lazy	lazy	eager	eager	**eager**
	Order of expansion	prescan	prescan	outer	prescan	N/A	outer	inner	outer	**inner**
	Parsing ambiguities	N/A	N/A	shortest	shortest	N/A	N/A	N/A	greedy	**greedy**
	Hygienic expansion	no	no	no	yes	no	yes	(yes)	no	**yes**
	Macros as results	no	yes	yes	no	no	yes	yes	yes	**no**
	Guaranteed termination	yes	no	no	no	no	no	yes	no	**yes**
Impl.	Transparent	yes	N/A	yes	yes	yes	yes	yes	yes	**yes**
	Error trailing	N/A	N/A	no	no	no	no	no	no	**yes**
	Pretty printing	no	no	no	no	no	no	no	no	**yes**
	Package Mechanism	no	no	no	no	no	no	no	no	**yes**

Figure 2: A macro language survey.

2.5 Invocation Properties

A macro body may contain further macro invocations. The languages are evenly split as to whether a macro body is expanded *eagerly* at its definition or *lazily* at each invocation.

Similarly, the actual arguments may contain macro invocations; here, the languages split on using an *inner* or *outer* expansion strategy. However, `CPP`, `M4`, and `Dylan` use a more complex strategy known as *argument prescan*. When a macro invocation is discovered, all arguments are parsed and any macros in them are invoked. These expanded arguments are then substituted for their placeholders in a copy of the macro body. Finally, the entire result is *rescanned*, processing any newly produced macro invocations. Note that this strategy only makes sense for lexical macro languages.

The languages that allow a liberal invocation syntax where the arguments are not properly delimitered sometimes face ambiguities in deciding how to match actual to formal macro arguments. The lexical languages, TeX and `Dylan`, resolve such ambiguities by chosing the *shortest* possible match; in contrast, the syntactical language `MS`^{2} employs a *greedy* strategy that for each formal argument parses as much as possible. None of the languages investigated employed *back-tracking* for matching invocations with definitions.

Most syntactical languages employ automatic α-conversion to obtain *hygienic* macros; `MS`^{2} requires explicit renamings to be performed by the programmer. Several languages allow new macro definitions to be generated by macro expansions. Only `CPP` and `JTS` guarantee termination of macro expansion; the others fail either by a naive treatment of recursive macros or by allowing arbitrary computations during expansion.

2.6 Implementation Properties

Macro languages are generally designed to be *transparent*, meaning that subsequent phases of the compilation need not be aware of macro expansions. However, none seem to allow *error trailing*, which would mean that errors from subsequent phases could be traced back to the unexpanded syntax. Similarly, no languages allow *pretty printing* of the unexpanded syntax. Finally, a *package* concept for macros is typically not supported.

2.7 Other Related Work

Our macro language shares some features of a previous work on extensible syntax [4], although that is not a macro language. Rather, it is a framework for defining new syntax that is represented as parse tree data structures in a *target* language, in which type checking and code generation is then performed. In contrast, our new syntax is directly translated into parse trees in a *host* language. Also, the host language syntax is always available on equal footing with the new syntax. However, the expressiveness of the extensible syntax that is permitted in [4] is very close to the argument syntax that we allow, although there are many technical differences, including definition selection, parsing ambiguities, expansion strategy, and error trailing. Also, we allow a more general translation scheme.

3 Designing a Macro Language

The ideal macro language would allow *all* nonterminals of the host language grammar to be extended with arbitrary new productions, defining new constructs that appear to the programmer as if they were part of the original language. The macro languages we have seen in the previous section all approximate this, some better than others.

In this section we aim to come as close to this ideal as practically possible. We will carefully consider the semantic aspects identified in Figure 2 in our design.

Later, we take a further step by allowing the programmer to define also new nonterminals.

Syntax

Our syntax macro language looks as follows:

macro	:	**`syntax`** <*nonterm*> *id* ⟨*param*⟩* **`::=`** `{` *body* `}`
param	:	*token*
	|	<*nonterm id*>

A syntax macro has four constituents: a nonterminal *result type*, an identifier *naming* the macro, a *parameter list* specifying the invocation syntax, and a *body* that must comply with the result type.

The result type declares the type of the body and thereby the syntactic contexts in which invocations of the macro are permitted. Adhering to Tennent's *Principle of Abstraction* [20], we allow *nonterm* to range over *all* nonterminals of the host language grammar. Of course, the nonterminals are from a particular standardized abstract grammar. In the case of the `<bigwig>` host language, 55 nonterminals are available.

As in `MS`^{2}, a macro must start with an identifier. It is technically possible to lift this restriction [14], but it serves to make macro invocations easier to recognize. The parameter list determines the rest of the invocation syntax. Here, we allow arbitrary tokens interspersed among arguments that are identifiers typed with nonterminals. The list ends with the "**`::=`**" token.

The macro body enclosed in braces conforms to the result type and references the arguments through identifiers in angled brackets.

Simple Examples

A simplest possible macro without arguments is:

```
syntax <floatconst> pi ::= {
  3.1415927
}
```

whose invocation `pi` is only allowed in places where a *floatconst* would be. The next macro takes an argument and executes it with 50% probability:

```
syntax <stm> maybe <stm S> ::= {
  if (random(2)==1) <S>
}
```

A more interesting invocation syntax is:

```
syntax <stm> repeat <stm S> until (<exp E>); ::= {
  {
    bool first = true;
    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

which extends the host language with a `repeat` construct that looks and feels exactly like the real thing. Identifiers such as `repeat` and `until` are even treated as keywords in the scope of the macro definition. The semantic correctness of course relies on α-conversion of `first`. Incidentally, this is the macro used in Figure 1.

An example with multiple definitions supplies a Francophile syntax for existing constructs:

```
syntax <stm> si (<exp E>) <stm S> ::= {
  if (<E>) <S>
}

syntax <stm> si (<exp E>) <stm S1> sinon <stm S2> ::= {
  if (<E>) <S1> else <S2>
}
```

The two definitions are both named `si` but have different parameters.

Macro Packages

Using macros to enrich the host language can potentially create a Babylonic confusion. To avoid this problem, we have created a simple mechanism for scoping and packaging macro definitions. A package containing macro definitions is viewed as a *set*, that is, we use two pass scope rules where all definitions are visible to each other and the order is insignificant. A dependency analysis intercepts and *rejects* recursive definitions.

A package may `require` or `extend` other packages. Consider a package P that contains a set of macro definitions M, requires a package R, and extends another package E. The definitions visible inside the bodies of macros in M are $M \cup R \cup E$ and those that are exported from P are $M \cup E$. Thus, `require` is used for obtaining *local* macros. The strict view that a package defines a set eliminates many potential problems and confusions.

Parsing Definitions

Macro definitions are parsed in two passes. First, the macro headers are collected into a structure that will later guide the parsing of invocations. The bodies are lexed to discover macro invocations from which a dependency graph is constructed. Second, the macro bodies are parsed in topological order. The result is for each body a parse tree that conforms to the result type and contains placeholder nodes for occurrences of arguments. It is checked that the body is always derivable from the result type nonterminal when the placeholders are assumed to be replaced with derivations from the corresponding argument type nonterminals.

Parsing Invocations

Macro invocations are detected by the occurrence of an identifier naming a macro. At this point, the parser determines if the nonterminal result type of the macro is reachable from the current point in parsing. If not, parsing is aborted. Otherwise, parsing is guided to this nonterminal and *invocation parsing* begins. The result is a parse tree that is inserted in place of the invocation.

Invocation parsing is conducted by interpreting the macro parameter list, matching required tokens and collecting actual argument parse trees. When the end of the parameter list is reached, the actual arguments are substituted into the placeholders in a copy of the macro body. This process is commonly referred to as *macro expansion*. It is *greedy* since an actual argument is parsed as far as possible in the usual top-down parsing style.

However, this basic mechanism is not powerful enough to handle *multiple* definitions of a macro. For that purpose, we must interpret a *set* of parameter lists. We base the definition selection on the concept of *specificity* which is independent of the macro definition order. This is done by gradually challenging each parameter list with the input tokens. There are three cases for a challenge:

- if a list is empty, then it always survives;
- if a list starts with a token, then it survives if it equals the input token; and
- if a list starts with an argument $<N\ a>$, then it survives if the input token belongs to $\text{first}(N)$ in the host grammar.

Several parameter lists may survive the challenge. Among those, we only keep the most *specific* ones. The empty list is always eliminated unless all lists are empty. Among a set of non-empty lists, the survivors are those whose first parameter is maximal in the ordering $p \sqsubset q$ defined as $\phi(q) \subset \phi(p)$, where $\phi(\mathit{token})$ is the singleton $\{\mathit{token}\}$ and $\phi(<N\ a>)$ is $\text{first}(N)$ in the host grammar. The tails of the surviving lists are then challenged with the next input token, and so on.

The intuition behind our notion of specificity can be summarized is a few rules of thumb: 1) always prefer longer parameter lists to shorter ones, 2) always prefer a token to a nonterminal, 3) always prefer a narrow nonterminal to a wider one. Rule 1) is the reason that the dangling `sinon` problem for our Francophile example is solved correctly. This strategy has a far reaching generality that also works for the metamorph rules introduced in Section 5.

For the *order of expansion* we have chosen the *inner* strategy. Since our macros are terminating, the expansion order is semantically transparent, apart from a subtle difference with respect to α-conversion that we will not get into here. The inner strategy is simpler since arguments are only parsed once.

Well-Formedness

A set of macros with the same name must be *well-formed*. This means that they must all have the same result type. Actually, this restriction could be relaxed to allow different return types for macros with the same name by using a contravariant specificity ordering to determine which one to invoke. Furthermore, to guarantee that the challenge rounds described above have a unique final winner, we impose two requirements. First, all parameter lists must be strictly ordered in the lexicographical generalization of the $\sqsubseteq$ order from *param* to *param**. Second, for all pairs of parameter lists of the form $\pi p_1 \pi_1$ and $\pi p_2 \pi_2$, if $\phi(p_1)$ equals $\phi(p_2)$ then p_1 must equal p_2.

Hygienic Macros

To achieve hygienic macros, we automatically α-convert the bound identifiers inside macro bodies during expansion. Unlike `Scheme` [10, 5], we also α-convert *free* identifiers, since they cannot be guaranteed to bind to anything sensible in the context of an invocation. As we thus α-convert *all* identifiers, the macro needs only recognize all parse tree nodes of nonterminal *id*; that is, no symbol table information is required. To communicate identifiers from the invocation context we encourage the macro programmer to supply those explicitly as arguments of type *id*. If an unsafe free variable is required, it must be *backpinged* to avoid α-conversion. It is often necessary to use computed identifiers, as seen in Figure 3. For that purpose, we introduce an injective and associative binary concatenation operator "~" on identifiers. The inductive predicate α determines if an identifier will be α-converted:

- $\alpha(\texttt{`}i) = \mathit{false}$;
- $\alpha(i\,\tilde{}\,j) = \alpha(i) \wedge \alpha(j)$;
- $\alpha(\texttt{<}i\texttt{>}) = \mathit{false}$, if $\texttt{<}i\texttt{>}$ is an argument of type *id*; and
- $\alpha(i) = \mathit{true}$, otherwise.

4 Growing Language Concepts

Our macro language allows the host language to grow, not simply with handy abbreviations but with new concepts and constructs. Our host language, `<bigwig>`, is designed for programming interactive Web services and has a very general mechanism for providing concurrency control between session threads [15, 2]. The programmer may declare labels in the code and use temporal logic to define the set of legal traces for the entire service. This is a bit harsh on the average programmer and consequently a good opportunity for using macros.

Figure 3 shows a whole stack of increasingly high-level concepts that are introduced on top of each other, profiting from the possibility to define macros for all nonterminals of the host language. Details of the `<bigwig>` syntax need not be understood. The `allow`, `forbid`, and `mutex` macros abbreviate common constructs in the temporal logic and produce results of type *formula*. The macro `region` of type *toplevel* is different; it introduces a new concept of *regions* that are declared on equal footing with other native concepts. The `exclusive` macro of type *stm* defines a new control structure that secures exclusive access to a previously declared region. The `resource` macro of type *toplevels* declares an instance of another novel concept that together with the macros `reader` and `writer` realizes the *reader/writer protocol* for specified resources. Finally, the `protected` macro seemingly provides a *modifier* that allows any declared variable to be subject to that protocol. The macros all build on top of each other and produce no less than six levels of abstraction as depicted in Figure 4. A similar development could have implemented other primitives, such

```
syntax <formula> allow <id L> when <formula F> ::= {
  all now: <L>(now) => restrict <F> by now;
}

syntax <formula> forbid <id L> when <formula F> ::= {
  allow <L> when !<F>
}

syntax <formula> mutex ( <id A> , <id B> ) ::= {
  forbid <A> when (is t: <A>(t) &&
    (all tt: t<tt => !<B>(tt)))
}

syntax <toplevel> region <id R> ; ::= {
  constraint {
    label <R>~A, <R>~B;
    mutex(<R>~A, <R>~B);
  }
}

syntax <stm> exclusive ( <id R> ) <stm S> ::= {
  {
    wait <R>~A;
    <S>
    wait <R>~B;
  }
}

syntax <toplevels> resource <id R> ; ::= {
  region <R>;
  constraint {
    label <R>~enterR, <R>~exitR, <R>~P;
    trigger <R>~RC when #<R>~enterR == #<R>~exitR;
    trigger <R>~PC when #<R>~P == #<R>~B;
    allow <R>~enterR when never(<R>~P) ||
      (is t: <R>~PC(t) && (all tt: t<tt => !<R>~P(tt)));
    allow <R>~A when never(<R>~enterR) ||
      (is t: <R>~RC(t) && (all tt: t<tt => !<R>~enterR(tt)));
  }
}

syntax <stm> reader ( <id R> ) <stm S> ::= {
  {
    wait <R>~enterR;
    <S>
    wait <R>~exitR;
  }
}

syntax <stm> writer ( <id R> ) <stm S> ::= {
  {
    wait <R>~P;
    exclusive (<R>) <S>
  }
}

syntax <toplevels> protected <type T> <id I> ; ::= {
  <T> <I>;
  resource <I>;
}
```

Figure 3: Concurrency control abstractions

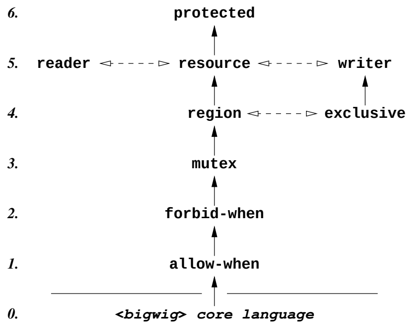


Figure 4: A stack of macro abstractions.

as semaphores, monitors, and fifo pipes. This demonstrates how the host language becomes highly tailorable with very simple means. The <bigwig> language employs an extensive collection of predefined macros to enrich the core language.

5 Metamorphisms

Macro definitions specify two important aspects: the *syntax definition* aspects characterizing the syntactic structure of invocations and the *syntax transformation* aspects specifying how "new syntax" is *morphed* into host language syntax.

So far, our macros can only have a finite invocation syntax, taking a fixed number of arguments each of which is described by a host grammar nonterminal. In the following we will move beyond this limitation, focusing initially on the syntax definition aspects.

The previously presented notion of *multiplicity* allows the definition of macros with varying arity. The following example defines an `enum` macro as known from C that takes one, two, or three identifier arguments:

```
syntax <decls> enum { <id X> } ; ::= {
  const int <X> = 0;
}

syntax <decls> enum { <id X> , <id Y> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
}

syntax <decls> enum { <id X> , <id Y> , <id Z> } ; ::= {
  const int <X> = 0;
  const int <Y> = 1;
  const int <Z> = 2;
}
```

Evidently, this can only emulate varying arity to a fixed and finite extent, in other words, it is not possible to define macros with arbitrary arity. Another point of criticism is that the syntactic transformation specification exhibits a high degree of redundance. In terms of syntax definition, the three `enum` definitions correspond to adding three *unrelated* right-hand side productions for the nonterminal *decls*:

$$
\begin{array}{rcl}
\textit{decls} & : & \textbf{enum}\ \texttt{\{}\ \textit{id}\ \texttt{\}}\ \textbf{;} \\
 & | & \textbf{enum}\ \texttt{\{}\ \textit{id}\ \textbf{,}\ \textit{id}\ \texttt{\}}\ \textbf{;} \\
 & | & \textbf{enum}\ \texttt{\{}\ \textit{id}\ \textbf{,}\ \textit{id}\ \textbf{,}\ \textit{id}\ \texttt{\}}\ \textbf{;}
\end{array}
$$

`Scheme` amends this by introducing a special ellipsis construction, "`...`" to specify lists of nonterminal s-expressions. `MS`^{2} moves one step further by permitting also tuples and optional arguments. The syntactic definition aspects of these extensions correspond to allowing the use of regular expressions over the terminals and nonterminals of the host grammar on the right-hand sides of productions. The ubiquitous EBNF syntax is available for designating options "`?`", lists "`*`" or "`+`", and tuples "`{...}`" (for grouping). In addition, `MS`^{2} provides a convenient variation of the Kleene star for specifying token-separated lists of nonterminals. Here, we use $N^{\oplus}$ as notation for one-or-more *comma separated* repetitions of the nonterminal N. An `enum` macro defined via this latter construction corresponds to extending the grammar as follows:

$$
\textit{decls} \quad : \quad \textbf{enum}\ \texttt{\{}\ \textit{id}^{\oplus}\ \texttt{\}}\ \textbf{;}
$$

The `Dylan` language has taken the full step by allowing the programmer to describe the macro invocation syntactic structure via a user defined *grammar*, permitting the introdution of new user defined nonterminals. This context-free language approach is clearly superior to that of the regular language approach, and this way options, lists, and tuples are just special cases. This approach can handle balanced tree structures that cannot be captured using the regular expression extensions. The `enum` invocation syntax could be described by the following grammar fragment that introduces a user defined nonterminal called <u>*enums*</u> (underlined for readability):

$$
\begin{array}{rcl}
\textit{decls} & : & \textbf{enum}\ \texttt{\{}\ \textit{id}\ \underline{\textit{enums}}\ \texttt{\}}\ \textbf{;} \\
\underline{\textit{enums}} & : & \textbf{,}\ \textit{id}\ \underline{\textit{enums}} \\
 & | & \varepsilon
\end{array}
$$

In `Dylan` the result of parsing a user defined nonterminal also yields a result that can be substituted into the macro body. However, this result is an *unparsed* chunk of tokens with all the associated lexical macro language pitfalls. We want to combine this great definition flexibility with static safety. Turning to the syntactic transformation specification aspects, we need some way of specifying the *type* of the result of parsing a user defined nonterminal. Clearly, user defined nonterminals cannot exist on an equal footing with that of the host language; our syntax macro must ultimately produce host language syntax trees and thus cannot return user defined ASTs. To this end, as for macros, we associate to every user defined nonterminal a host nonterminal result type and require that the body result of parsing a user defined nonterminal be a syntax tree of this type. The syntax defined by the user defined nonterminals is always morphed directly into host syntax. The specification of this morphing is inductively given for each production of the grammar. In contrast, `MS`^{2} relies on programming and computation for specifying and transforming their regular expressions of nonterminals into parse trees.

To distinguish clearly from the host grammar, we call the user defined nonterminal productions typed with host nonterminals for *metamorphisms.* A metamorphism is thus a user defined nonterminal (at *meta* grammar level) along with a rule specifying how the new (macro) syntax is *morphed* into host language syntax. The syntax for macro definitions is generalized as follows to accommodate the metamorphisms:

```
macro  : syntax <nonterm> id ⟨param⟩* ::= { body }
       | metamorph <nonterm> id --> ⟨param⟩* ::= { body }
param  : token
       | <nonterm id>
       | <id: nonterm id>
```

We have introduced two new constructs. A parameter may now also be of the form `<M: N a>`, meaning that it is named a, has an invocation syntax that is described by the metamorph nonterminal <u>*M*</u>, and that its result has type N when it occurs in the body. The metamorph syntax and the inductive translation into the host language is described by the `metamorph` rules. To the left of the "`-->`" token is the result type and name of the metamorph nonterminal, and to the right is a parameter list defining the invocation syntax and a body defining the translation into the host language. The metamorph rules may define an arbitrary grammar. In its full generality, a metamorph rule may produce multiple results each defined by a separate body.

We are now ready to define the general `enum` macro in our macro language. The three production rules above translates into the following three definitions:

```
syntax <decls> enum { <id I> <enums: decls Ds> } ; ::= {
  int e = 0;
  const int <I> = e++;
  <Ds>
}

metamorph <decls> enums --> , <id I> <enums: decls Ds> ::= {
  const int <I> = e++;
  <Ds>
}

metamorph <decls> enums --> ::= {}
```

The first rule becomes a macro called `enum` with the metamorph argument `<enums: decls Ds>` describing a piece of invocation syntax that is generated by the nonterminal *enums* in the metamorph grammar. However, *enums* parse trees are never materialized, since they are instantly morphed into parse trees of the nonterminal *decls* in the host grammar.

The body of our `enum` macro commences with the declaration of a variable `e` used for enumerating all the declared variables at runtime. This declaration is followed by the morphing of the (first) identifier `<I>` into a constant integer declaration with initialization expression `e++`. Hereafter, comes `<Ds>` which is the *decls* result of metamorphing the remaining identifiers to constant integer declarations.

The next two productions in the enum grammar translates into two metamorph definitions. The first will take a comma and an identifier followed by a metamorph argument and morph the identifier into a constant integer declaration as above and return this along with whatever is matched by another metamorph invocation. The second metamorph definition offers a termination condition by parsing nothing and returning the empty declarations.

The next example shows how the invocation syntax of a `switch` statement syntax is easily captured and desugared into nested `if` statements:

```
syntax <stm> switch (<exp E>) { <swbody: stm S> } ::= {
  {
    typeof(<E>) x = <E>;
    <S>
  }
}

metamorph <stm> swbody -->
    case <exp E>: <stms Ss> break; <swbody: stm S> ::= {
  if (x==<E>) { <Ss> } else <S>
}

metamorph <stm> swbody --> case <exp E>: <stms Ss>
    break; ::= {
  if (x==<E>) { <Ss> }
}
```

In its full generality, a metamorph production may morph the invocation syntax into *several* resulting parse trees in the host grammar. This can be seen as a generalization of the `divert` primitive from `M4`; however, our solution statically guarantees syntactic well-formedness of the combined result. The `metamorph` rules and metamorph formals are extended to cope with multiple returns and arguments:

$$\begin{array}{lcl} \textit{macro} & : & \texttt{\textbf{metamorph}}\ \texttt{<}\langle \textit{nonterm}\rangle^{\oplus}\texttt{>}\ \underline{\textit{id}}\ \texttt{-->}\ \langle \textit{param}\rangle^{*}\ \texttt{::=} \\ & & \quad \langle \texttt{\{}\ \textit{body}\ \texttt{\}}\rangle^{+} \\ \textit{param} & : & \texttt{<}\underline{\textit{id}}\texttt{:}\ \langle \textit{nonterm id}\rangle^{\oplus}\texttt{>} \end{array}$$

The following example illustrates in a simple way how multiple metamorph results add expressive power to our macro language. We extend a simple version of statement blocks with optional initializations of declared variables (this is of course already possible in `<bigwig>`). To build the expansion, we need to obtain from a variable declaration *both* a *decl* declaring it and a *stm* initializing it. This base case is seen in the rules for the metamorph nonterminal *init*; the nonterminal *inits* generalizes this inductively for lists; and the two results are finally used in the `block` macro:

```
syntax <stm> block { <inits: decls Ds, stms Ss> <stms S> }
    ::= {
  <Ds> <Ss> <S>
}

metamorph <decls,stms> inits --> <init: decl D, stm S>
    <inits: decls Ds,stms Ss> ::= {
  <D><Ds>
}{
  <S><Ss>
}

metamorph <decls,stms> inits --> ::= {}{}
```

```
metamorph <decl,stm> init --> <type T> <id I> ; ::= {
  <T> <I> ;
}{}

metamorph <decl,stm> init --> <type T> <id I> = <exp E> ;
    ::= {
  <T> <I> ;
}{
  <I> = <E> ;
}
```

With these definitions, the macro expands as follows:

```
block {                              int i;
  int i = 42;                        float f;
  float f;               ==>         int j;
  int j = i+10;                      i = 42;
  ...                                j = i+10;
}                                    ...
```

Without multiple results, some transformations are impossible or require contorted encodings.

Parsing Invocations

The strategy for parsing invocations is unchanged. The $\sqsubset$ order is generalized appropriately by defining $\phi(<\underline{M}\texttt{:}\ N\ a>)$ to be first($\underline{M}$) in the metamorph grammar. Note that it is always possible to abbreviate part of the invocation syntax by introducing a new metamorph nonterminal while preserving the semantics.

Well-Formedness

As for syntax macros, the set of productions for a given metamorph nonterminal must be well-formed. Furthermore, to ensure termination of our greedy strategy, we prohibit left-recursion in the metamorph grammar. Finally, we include the sanity check that each metamorph nonterminal must derive some string.

Hygienic Macros

Metamorph productions do not initiate α-conversion. This is only done on the entire body of a syntax macro, conceptually *after* its metamorphic arguments have been substituted. This is seen in the `enum` example, where the expansion of "`enum {d,e};`" is:

```
int e~42 = 0;
const int d = e~42++;
const int e = e~42++;
```

In this resulting parse tree, the local occurrence of `e` is everywhere α-converted to the same `e~42`, which is necessary to yield the proper semantics.

6 Growing New Languages

Section 4 contains examples that use macros to enrich the host language with new concepts and constructs. A more radical use of particularly metamorphisms is to design and implement a completely new language at very little cost.

Our host language `<bigwig>` is itself a domain-specific language designed to facilitate the implementation of interactive Web services. To program a family of highly specialized services it can be advantageous to first define what we shall call a *very* domain-specific language, or VDSL.

We consider a concrete example. At the University of Aarhus, undergraduate Computer Science students must complete a Bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide students towards this goal, they must maintain a so-called "Bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service that for each student iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight `<bigwig>` application, but quickly became annoying to maintain. Thus it was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in pseudo natural language style. This makes it possible for a secretary—or even the responsible faculty member—to maintain and update the service. Figure 5 shows an example of the input. There is only a single macro, `studies`, which accepts as argument an entire specification in the VDSL syntax, defined using 27 metamorph rules. Its result is a corresponding `<bigwig>` service. Apart from the keyword `require`, none of the syntax shown is native to `<bigwig>`. The file `bach.wigmac` is only 400 lines and yet contains a complete implementation of the new language, including "parser" and "code generator". Thus, our macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with arbitrary syntax. Error trailing and unexpanded pretty printing supports the illusion that a genuinely new language is provided.

7 Implementation

The work presented is fully implemented in the `<bigwig>` compiler. The implementation is in `C` with extensive support from `CPP` and is available from the `<bigwig>` project homepage [7] in an Open Source distribution. In the following we present two important aspects from the implementation that achieve transparency for all other phases of the compiler. These are the *transparent representation* of macros and the *generic pretty printer* responsible for communicating macro-conscious information. These aspects support the illusion that the host language is really extended.

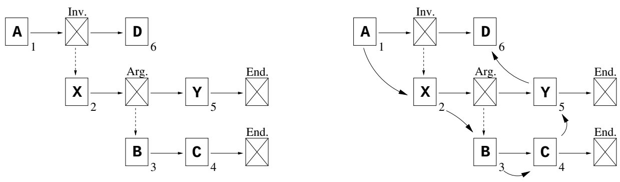
Transparent Representation

Consider the following macro definition:

```
syntax <ids> xIDy ( <ids Is> ) ::= {
  X,<Is>,Y
}
```

```
require "bach.wigmac"

studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Lab304
    title "Lab Work 304"
    1 point fall term

  exclusions
    Math101 <> MathA
    Math102 <> MathB

  prerequisites
    Math101,Math102 < Math201,Math202,Math203,Math204
    CS101,CS102 < CS201,CS203
    Math101,CS101 < CS202
    Math101 < Stat101
    CS202,CS203 < CS301,CS302,CS303,CS304
    Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
    Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
    Lab101,Lab102 < Lab201,Lab202
    Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304

  field "CS-Mathematics"
    field courses
      Math101,Math102,Math201,Math202,Stat101,CS101,
      CS102,CS201,CS202,CS203,CS204,CS301,CS302,CS303,
      CS304,Project
    other courses
      MathA,MathB,Math203,Math204,Phys101,Phys102,
      Phys201,Phys202
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      at least one of Math201,Math202
      at least 2 courses among Stat101,Math202,Math203
      has 4 points among Project,CS303,CS304
      in total between 36 and 40 points

  field "CS-Physics"
    field courses
      MathA,MathB,Stat101,CS101,CS102,CS201,CS202,
      CS203,CS204,CS301,CS302,CS303,CS304,Project,
      Phys101,Phys102,Phys201,Lab101,Lab102,Lab201,
      Lab202
    other courses
      Phys202,Phys301,Phys302,Phys303,Phys304,Lab301,
      Lab302,Lab303,Lab304,Math202,Math203,Math204
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      has passed Phys101,Phys102
      has 4 points among MathA,MathB,Math101,Math102
      has 6 points among Phys201,Phys202,Lab101,Lab102,
                         Lab201,Lab202
      in total between 38 and 40 points
```

Figure 5: A VDSL for Bachelor's contracts.



(a) Ordinary (b) Weaved

Figure 6: Macro representations.

The representation of the parse tree for the identifier list "`A,xIDy(B,C),D`" is seen in Figure 6(a). All node kinds of the parse tree are capable of holding three explicit macro nodes: `Inv`, `Arg`, and `End`.

This representation yields a perfectly balanced structure with complete knowledge of the scope of all macro invocations and arguments. It is, however, clearly not transparent for subsequent phases in the compiler. Transparency is achieved through a *weaving phase* in which new pointers are after parsing short-circuited around the macro nodes giving two ways of traversing the parse tree. Macro conscious phases follow the paths in Figure 6(a), while macro ignorant phases only see the new short-circuited paths of Figure 6(b). A naive data structure would double the size of the parse tree, but this can be avoided by storing the macro nodes at negative offsets in only those parse tree nodes where they are required. Desugaring is not fully compatible with preserving macro information [22] and this is the only sense in which transparency is not completely achieved. However, explicit desugaring is not really necessary in a compiler that supports metamorphic syntax macros.

Generic Pretty Printing

Four *indent directives* control the pretty printing of macros:

$$\textit{param} \quad : \quad \langle \textit{whitespace} \rangle^* \;\mid\; \texttt{\textbackslash n} \;\mid\; \texttt{\textbackslash +} \;\mid\; \texttt{\textbackslash -}$$

The macro header is augmented with *whitespace*, *newline*, *indent*, and *unindent* directives. The pretty printer can be instructed to print the `si-sinon` statement without spaces around the conditional expression and with a newline before the alternate branch:

```
syntax <stm> si (<exp E>) <stm S1> \n sinon <stm S2> ...
```

A more sophisticated indention correctly renders the `switch` control structure:

```
syntax <stm> switch (<exp E>) {\+\n<swbody: stm S>\-\n} ...
```

```
field "CS-Math"
  field courses
    Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS201,CS202,
    CS203,CS204,CS301,CS302,CS303,CS304,Project
  other courses
    MathA,MathB,Math203,Math204,Phys101,Phys102,Phys201,Phys202
  constraints
    has passed CS501,CS102
    at least 2 courses among CS201,CS202,CS203
    at least one of Math201,Math202
    at least 2 courses among Stat101,Math202,Math203
    has 4 points among Project,CS303,CS304
    in total between 36 and 40 points
```

Figure 7: HTML pretty print with an error message.

These extensions are purely cosmetic; they have no semantics attached and are ignored in the invocation challenge rounds.

Our implementation supports a *generic nonterminal pretty printer* that together with a *specific terminal pretty printer* will *unparse* the code with or without macro expansion. This only depends on the choice of arrows in Figure 6(b).

Our implementation currently has three *terminal pretty printers* for printing `ascii`, `LaTeX`, and `HTML/JavaScript` of which the last is by far the most sophisticated. It inserts use-def hyperlinks, visualizes expression types, highlights errors, and expands individual macros at the click of a button.

Error Reporting

With our generic pretty printing strategy, error reporting is a special case of pretty printing using a special kind of terminal printer that only print nodes with a non-empty error string. Consequently, error messages can be viewed with or without macro expansion. Figure 7 shows how a simple error is pinpointed in the unexpanded syntax. The compiler can be instructed to dump the error trail as follows:

```
*** symbol errors:
*** bach.wig:175:
    Identifier 'CS501' not declared
      in macro argument 'I'
      in macro invocation 'course_ids' (bach.wig:175) defined in [bach.wigmac:60]
      in macro argument 'C'
      in macro invocation 'cons' (bach.wig:175) defined in [bach.wigmac:112]
      in macro argument 'C'
      in macro invocation 'cons_list' (bach.wig:175) defined in [bach.wigmac:126]
      in macro argument 'CN'
      in macro invocation 'fields' (bach.wig:168) defined in [bach.wigmac:134]
      in macro argument 'A'
      in macro invocation 'studies' (bach.wig:3) defined in [bach.wigmac:158]
```

which is useful when debugging macro definitions.

8 Conclusion and Future Work

We have designed and implemented a safe and efficient macro language that is sufficiently powerful to grow domain-specific extensions of host languages or even entire new languages.

There are several avenues for future work. First, we will take this approach even further, by defining a notion of *invocation constraints* that restrict the possible uses of macros. Such constraints capture some aspects of the static semantic analysis of the language extensions that are grown. The constraints work exclusively on the parse tree, similarly to [6], and thus preserve transparency. Second, we will build implementations for other host languages, in particular Java. Third, it is possible to create a parser generator that given a host grammar builds a parser that automatically supports metamorphic syntax macros. Most of the required techniques are already present in the implementation of metamorphisms.

References

[1] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Fifth International Conference on Software Reuse*, 1998.

[2] C. Brabrand. Synthesizing safety controllers for interactive Web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/`.

[3] W. R. Campbell. A compiler definition facility based on the syntactic macro. *Computer Journal*, 21(1):35–41, 1975.

[4] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, 1994.

[5] W. Clinger and J. Rees. Macros that work. In *Principles of Programming Languages (POPL)*, pages 155–162, 1991.

[6] N. Damgaard, N. Klarlund, and M. Schwartzbach. Yakyak: Parsing with logical side constraints. In *Developments in Language Theory (DLT)*, 1999.

[7] C. Brabrand *et al.* The `<bigwig>` project homepage. `http://www.brics.dk/bigwig/`.

[8] R. Kelsey, W. Clinger, and J. R. (Eds.). Revised(5) report on the algorithmic language scheme (r5rs), 1998.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1978.

[10] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Lisp and Functional Programming*, pages 151–161, 1986.

[11] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages (POPL)*, pages 77–84. ACM, 1987.

[12] B. M. Leavenworth. Syntax macros and extended translation. *CACM*, 1966.

[13] W. Maddox. Semantically-sensitive macroprocessing. Technical report, University of California, Berkeley, 1989. Technical Report UCB/CSD 89/545.

[14] D. Sandberg. Lithe: A language combining a flexible syntax and classes. In *Principles of Programming Languages (POPL)*, pages 142–145, 1982.

[15] A. Sandholm and M. I. Schwartzbach. Distributed safety controllers for Web services. In *Fundamental Approaches to Software Engineering, FASE'98*, pages 270–284, 1998.

[16] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley-Longman, 1996.

[17] R. M. Stallman. The C preprocessor online documentation. `http://gcc.gnu.org/onlinedocs/cpp_toc.html`.

[18] G. Steele. Growing a language. OOPSLA invited talk, 1998.

[19] B. Stroustrup. *The C++ Programming Language*, chapter 13. Addison Wesley, third edition, 1997.

[20] R. D. Tennent. *Principles of Programming Languages*. Prentice Hall, 1981.

[21] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999.

[22] O. Waddell and R. K. Dybvig. Visualizing partial evaluation. In *ACM Computing Surveys Symposium on Partial Evaluation*, volume 30(3es):24-es, September 1998.

[23] D. Weise and R. F. Crew. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, pages 156–165, 1993.

Recent BRICS Report Series Publications

RS-00-24 **Claus Brabrand and Michael I. Schwartzbach.** ***Growing Languages with Metamorphic Syntax Macros*****. September 2000. 22 pp.**

RS-00-23 **Luca Aceto, Anna Ingólfsdóttir, Mikkel Lykke Pedersen, and Jan Poulsen.** ***Characteristic Formulae for Timed Automata*****. September 2000. 23 pp.**

RS-00-22 **Thomas S. Hune and Anders B. Sandholm.** ***Using Automata in Control Synthesis — A Case Study*****. September 2000. 20 pp. Appears in Maibaum, editor,** ***Fundamental Approaches to Software Engineering: First International Conference*****, FASE '00 Proceedings, LNCS 1783, 2000, pages 349–362.**

RS-00-21 **M. Oliver Möller and Rajeev Alur.** ***Heuristics for Hierarchical Partitioning with Application to Model Checking*****. August 2000. 30 pp.**

RS-00-20 **Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***2-Nested Simulation is not Finitely Equationally Axiomatizable*****. August 2000. 13 pp.**

RS-00-19 **Vinodchandran N. Variyam.** ***A Note on*** $\mathrm{NP} \cap \mathrm{coNP}/\mathrm{poly}$**. August 2000. 7 pp.**

RS-00-18 **Federico Crazzolara and Glynn Winskel.** ***Language, Semantics, and Methods for Cryptographic Protocols*****. August 2000. ii+42 pp.**

RS-00-17 **Thomas S. Hune.** ***Modeling a Language for Embedded Systems in Timed Automata*****. August 2000. 26 pp. Earlier version entitled** ***Modelling a Real-Time Language*** **appeared in Gnesi and Latella, editors,** ***Fourth International ERCIM Workshop on Formal Methods for Industrial Critical Systems*****, FMICS '99 Proceedings of the FLoC Workshop, 1999, pages 259–282.**

RS-00-16 **Jiří Srba.** ***Complexity of Weak Bisimilarity and Regularity for BPA and BPP*****. June 2000. 20 pp. To appear in Aceto and Victor, editors,** ***Expressiveness in Concurrency: Fifth International Workshop EXPRESS '00 Proceedings*****, ENTCS, 2000.**