On Domain-Specific Macro Expansion and Its Use in Optimizing Code Generation

Eero Lassila
Helsinki University of Technology
Digital Systems Laboratory
Otakaari 1, FIN-02150 Espoo, Finland
eero.lassila@hut.fi

Submission to Workshop on Domain-Specific Languages

October 1996

Abstract

We identify two types of domain-specific features in macro expansion systems: with the expansion-time values of *domain-specific attributes* the user can control conditional expansion; *domain-specific actions* are performed by the system, and we divide them into *expansion-integrated* and *conversion-integrated* ones. The domain of our implemented prototype system is machine-level code generation, but we also formulate a model intended to capture domain-specific macro expansion in various domains. An important property of the model is that the expansion result is independent of the macro call expansion order.

1 Introduction

TEX by Knuth [18] is surely one of the most successful domain-specific languages. The TEX typesetting system is capable of transforming a manuscript prepared in TEX format into pages of finest typographic quality. For example, TEX autonomously positions the line and page breaks by applying sophisticated built-in algorithms. The TEX vocabulary includes 300 primitive *control sequences* and a macro definition facility for building composite control sequences. Accordingly, already the "plain TEX" by Knuth offers the user 600 additional macro control sequences, and the popular LATEX system by Lamport [19] is also a TEX macro package providing still higher-level easy-to-use tools. It seems that much of the power of TEX is in its support for macros. On the other hand, basic general-purpose macro expanders just perform simple domain-independent text string substitutions. What kind of domain-specificity is there in TEX macro expansion, and how does TEX compare in this respect with macro assemblers, the most traditional special-purpose macro expanders?

It is clear that the usefulness of both TEX and assemblers stems from *domain-specific actions*, which include the TEX line and page break control and the (much simpler) support for static memory allocation offered by the assemblers. But these domain-specific actions are *conversion-integrated*, i.e. separate from the macro expansion proper: the macro expansion can be seen as a preprocessing phase whose output is the input of the actual "conversion" phase. Nevertheless, domain-specificity can be found even within the macro expansion: both TEX and many macro assemblers (see [17] for an example) provide built-in *domain-specific attributes* whose expansion-time values the macro writer can use to steer *conditional expansion*. A TEX user can ask for the height of the text accumulated on the current page and use this value as a condition for a "hard" page break. In a somewhat similar fashion, with many macro assemblers the user can find out whether some text string denotes an object of a particular type in the underlying assembly program. In general, however, in both these cases the domain-specific attributes can be seen as only an additional feature that sometimes happens to be useful.

In this paper we suggest a novel domain-specific macro expansion scheme, in which conditional expansion based on domain-specific attributes is of utmost importance. In addition, the scheme features *expansion-integrated* domain-specific actions. These are built in the macro expansion facility itself: the expander is preprogrammed to perform routine but—from a human viewpoint—laborious domain-specific actions instead of the plain text string substitution. In particular, in Sec. 2 we present a rudimentary example of a domain-specific language complying with the scheme. Not fully accidentally, our language shares the domain with conventional macro assemblers. We hope, however, that these ideas could spread even outside the field of machine-level code generation, and in Sec. 3 we formulate a model that should make the principles of the scheme explicit and applicable in various domains. Indeed, finally in Sec. 4 we briefly touch the characterization of other "promising" domains.

The advantage of machine code generation as an exploratory domain is its long history and great importance: only the fittest techniques can survive in this field. One might argue that typical domain-specific languages are very "high-level", whereas assembly code is, of course, very "low-level". But we are not proposing a new machine-specific language; what we are proposing is a machine-independent domain-specific language that an expert can use for specifying code generators for various processor architectures. In all, the problem of generating optimum machine code is surely not fundamentally different from the versatile set of all other optimization problems.

2 Towards greater modularity in code generation

When an experienced system programmer is faced with a task in which efficiency does matter, he should possess tools by which he could conveniently optimize the machine-specific result of high-level language compilation precisely as far as necessary—and no further. To approach this ideal situation, one needs more flexible compiler back-ends. We are developing techniques for the construction of freely hierarchical libraries of optimized machine-level code. Such libraries would serve as the rule base of the compiler back-end, and our goal is that the optimization of the compiler output could be carried out by detecting and rewriting exactly those modules in the library hierarchy that are critical to the particular application at hand.

2.1 Background and motivation

First, in Sec. 2.1.1 we further examine the intended domain of our domain-specific macro system. In Sec. 2.1.2 we recall the basic properties of domain-independent macro expansion, and in Sec. 2.1.3 we discuss the need for extensions.

2.1.1 On embedded processor code generation

For *general-purpose* processor architectures, there are powerful code-generator generators available [1, 9, 13]. The input to such a machine-independent metatool is a machine-specific set of tree patterns with associated costs and semantic actions. The output is a machine-specific code generator that uses tree matching and dynamic programming: first, for each expression tree in the intermediate program representation produced by a compiler front-end, it selects a set of patterns that covers the tree with a minimum total cost; second, it produces assembly code by executing the semantic actions of the selected patterns.

In the case of embedded *special-purpose* processors, in contrast, code generation remains a most difficult problem [29, 25, 26, 27]. With many *digital signal processors (DSPs)* [24] used in real-time telecommunication systems, for instance, high-level language compilation is currently ruled out by efficiency requirements. In general, especially because of the irregularity of the specialized processor architecture, the case analysis needed in code generation eventually becomes too enormous for compilers to manage, as suggested by Wulf in 1981 [35]. Therefore, the compiled code quality may be rather low in comparison with the high efficiency requirements; moreover, as the applications are highly specialized, there may be little need to consider code portability; and finally, often the decisive reason favoring assembly language programming is the insignificance of the programming costs in the case of mass-produced articles. For instance, when a mobile phone is developed, even extensive code optimization pays if it results in hardware savings [14].

Accordingly, what should embedded processor compiler tools be like? First of all, the user should be capable of producing as optimized code as he wishes: the quality of the code should be a monotonously—and even rather steadily—increasing function of the time spent in optimization. Second, it should be straightforward to retarget the tool to new processors, as programmable special-purpose processor architectures are becoming more and more dedicated [31, 28]. Third, one is ready to sacrifice some of the compilation speed to attain the two preceding objectives.

2.1.2 On general-purpose macro expanders

By 'general-purpose' macro expansion, we mean the simple text string substitution. With general-purpose macro expanders [6, 5, 7], conditional expansion is domain-independent. It takes place against the expansion time values—which may be arbitrary text strings—associated with certain identifier tokens. How are the expansion-time values determined? The identifiers involved are divided into *macro parameters* and *macro variables*: the macro parameters inherit their values from the corresponding arguments of each call instance of the macro; the value of any macro variable can be changed with an explicit redefinition directive. As the source text is processed strictly sequentially, the effective definition of each macro variable is always unique, i.e. the most recent one encountered.

Brown [5, Ch. 1.7] demonstrates how the domain-independent conditional expansion facility can be utilized in optimizing code generation. Suppose that we have written a two-argument macro that copies the contents of one memory location into another on our single-accumulator machine. For instance, the macro call sequence below on the left might produce the result shown on the right.

```
Move M1 M2                                  load M1
                                            store M2
Move M2 M3                                  load M2
                                            store M3
```

The above result is, however, not an optimum one: the second load from memory is readily seen to be redundant. This deficiency can be eliminated by conditional expansion: we can introduce a macro variable, say `$Acc`, for representing our knowledge on the run-time contents of the accumulator. Now, if the first `Move` instance set `$Acc` to value '`M2`', then the second `Move` instance could examine the value of `$Acc` and consequently drop out the redundant `load` instruction. (This suggestion is not a very general one yet: it would not work if the second `Move` instance were replaced with an equivalent macro call reading as '`Move M1 M3`'.)

There are, of course, inherent problems with any simple-minded mechanism of the above kind. The strictly sequential top-down (or left-to-right) macro call processing order enables the macros to "look back", but they cannot "look ahead" to find out, say, in which register the next operation would like its operands to be located. Moreover, the above technique can be applied only to basic blocks, i.e. straight-line code segments, whereas control flow branches cannot be coped with. Nevertheless, we believe that overcoming these problems is not impossible, but some innovations are obviously needed in attacking the limitation expressed by Brown [5, p. 62]: "Macro processors simply do not have the facilities to look at the dynamic behaviour of a program and optimize on the basis of this."

2.1.3 Need for modularity

Macros provide a means to *hierarchical abstraction.* Macros were an early addition to assembly languages; somewhat later, considerable effort was spent in constructing elaborate macro systems that would more or less be alternatives to conventional high-level language compilers [10, 5]. Ideally, a macro assembler is a powerful implementation tool for compiler back-ends, but when we look at how macro assemblers are nowadays used [33], we find a limitation: there are few examples of significant multilayer hierarchy. This is due to the lack of *modularity.*

For example, if some assembler macro M_1 uses a register R as a temporary data storage, the programmer has to take care when calling M_1: the macro expander is not able to issue a warning that the value possibly already stored in R will be lost during the execution of

M_1 [20, pp. 15–17]. Additionally, if M_1, in turn, calls another macro, say M_2, then the original caller of M_1 must take into consideration even similar restrictions that perhaps concern M_2. In general, such implicit restrictions effectively prevent the introduction of macro hierarchy.

Modularity requires that the *module implementations* are independent of each other. The structure of the *module interface* should, first, be carefully defined to cover all possible intermodule dependencies and, second, be carefully respected when the modules are processed. Both support for hierarchy and rigorous modularity are needed for *composability*: on any abstraction level, it should be possible to both create and employ library components to make the code more easily readable, maintainable, and reusable. In the example above, modularity could be achieved if each call instance of M_1 somehow knew whether register R is *free* to be overwritten at the call point or, better yet, whether there is any free register that is interchangeable with R. Clearly, such knowledge about free storage cells is an essential ingredient in any feasible *interface definition for assembly code modules.* This knowledge requires sensitivity to the data and control flow of the underlying program—a storage cell is free if it is guaranteed not to contain data that may have been earlier written into it and may be later read from it.

We try to illustrate the idea of modularity further with Fig. 1. The figure shows the tree structure that results from the expansion of a single root macro call: each non-root node in the tree corresponds to a lower-level macro call generated during the expansion. If we now find out that the expansion result is not optimized enough, we may be able to locate within the tree a single macro whose rewriting is a sufficient improvement. We can freely carry out this local modification since modularity guarantees that global consistency of the expansion result is always preserved.

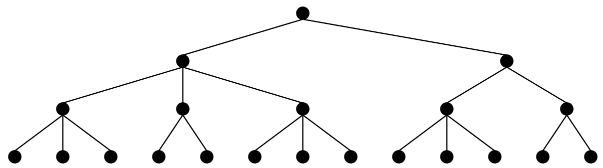

Figure 1: A macro expansion tree.

2.2 Basic implementation principles

We present our outline for the assembly code module interface definition in Sec. 2.2.1. The components of the interface can be examined through domain-specific attributes in the manner introduced in Sec. 2.2.2. Our storage allocation scheme relies heavily on expansion-integrated domain-specific actions, as described in Sec. 2.2.3.

2.2.1 Components of the module interface

We identify macros as assembly code modules. An important part of the macro interface is known already before the actual expansion time: the macro name and the number of macro arguments. The more challenging remaining part is specific to each macro call instance and is not revealed until expansion time. The main components of this part of the interface are:

- How is the physical location of each argument to be accessed?
- What is the available temporary storage, i.e. the set of free registers and memory locations?
- What possibly useful knowledge do we already have about the run-time contents of the registers at the point of the macro call?

The macro expansion must obviously take place conditionally; we offer the programmer a set of diagnostic primitives (see Sec. 2.2.2 below) with which he can examine the expander-maintained *expansion-time environment* of each macro call instance. This environment comprises all the information about the context of the macro call included in the macro interface definition.

Some constraints on the structure of the program code may enable the expander to make stronger decisions concerning the actual expansion-time environments. For example, each macro call instance should—explicitly or implicitly—identify each data item that it may read or write, and each point in the code to which it may hand over the run-time control; with this information, the expander is able to fix the free storage cell set as large as possible. We believe that meaningful constraints of this kind can be formulated in a way that does not complicate the programming task but, on the contrary, makes the program more easily readable even by humans.

We give some further comments on the interface components listed above:

- *Access to macro arguments.* With direct addressing, the expander is readily capable to tell the user the storage class and location of each argument (see Sec. 2.3 for examples). With indirect addressing, instead, a more elaborate scheme is needed: the user must be able to ask the expander about the type and parameters (e.g. the index of the address register employed) of the particular indirection mechanism.
- *Temporary storage.* Provided that the user sticks to direct addressing, the macro expander can determine the free registers by straightforward flow analysis (again, see Sec. 2.3 for an example). With indirect addressing, the task is understandably much more complicated: the more accurate *pointer analysis* [8, 34] we are capable of, the larger safe estimate for the free storage cell set can we obtain. (Note, however, that indirect addressing typically applies to memory locations, whereas the register file constitutes the most useful temporary storage.)
- *State of the registers.* Often it is extremely useful if the programmer knows for sure the value of some status register at the point of the macro call (see the discussion of *mode control* in [2, 26]). Therefore, one needs a device that can record the information about the run-time register states that becomes available already at expansion time. We suggest introduction of special expansion-time variables that can be associated with individual registers to keep track of their run-time contents. These variables are sensitive to the control flow of the program, and their range includes a special value that denotes the ordinary

case in which the associated register has different contents at different execution times of the particular code segment. The difference from the (global) macro variables discussed in Sec. 2.1.2 is that no macro call expansion result is allowed to modify the value of a "flow-sensitive" variable unless this modification is (explicitly or implicitly) specified already as a part of the macro call.

2.2.2 Expansion-time expression interpreter

In 1960, McIlroy [30] suggested that any "compiler should include within it an interpreter for its source language or somewhat equivalent". As opposed to most general-purpose high-level languages, our programming system offers no predefined run-time operators (all operations supported by the target must be explicitly declared by the user), but it does include a powerful expansion-time expression interpreter that provides the usual arithmetic operations as predefined primitives. The most important feature of the interpreter is, however, constituted by the built-in domain-specific attributes and the closely related flow analysis capability.

The operations recognized by the interpreter are called *forms*. There are some twenty predefined *primitive* forms: a set of *diagnostic* forms, a set of *arithmetic-logic* forms, and a single *selective form*. The diagnostic forms can be seen as the access interface of the domain-specific attributes. The current ReFlEx 1.0 implementation (see Sec. 2.3) provides the following diagnostic forms:

`?M(x)`	Is `x` a storage cell of class `M`? (`M` might consist of the memory locations, `R` might mean a set of auxiliary registers, and `A` might denote an accumulator—the storage classes are declared by the user.)
`=(x,y)`	Are storage cell designators `x` and `y` aliases of each other?
`!(x)`	Is storage cell `x` free at the point of the current macro call?
`#R()`	How many cells of storage class `R` are currently free?
`&(m,n)`	Do instruction label designators `m` and `n` refer to the same point in the code?

In addition, the user may freely define new *compound* forms. Here, our intention is to say that the cells in storage classes `R` and `A` contain relatively many bit positions:

```
Or(a,b) = _Nand(_Nand(a,a), _Nand(b,b));
Long(x) = Or(?R(x), ?A(x));
```

There are some things to be noticed. Of course, the `_Nand` arithmetic-logic primitive returns the logical *NAND* of its arguments. Each one of the above forms returns an integer value; by convention, nonzero values are interpreted as 'true'. Form arguments, then, may represent either integer values, storage cells, or instruction labels (the actual type of a particular argument cannot be fully resolved until expansion time).

The power of the interpreter is greatly extended by *lazy evaluation* [32] through the `_If` selective form. `_If` takes three arguments: it begins by evaluating the first one; if the result is nonzero, the result of `_If` is given by the evaluation of the second argument; otherwise, the result is given by the evaluation of the third argument. Either the second or third argument is thus always left unevaluated. For example, here is the definition for the factorial function $x!$:

```
Fact(x) = _If(_Lt(x,2), 1, _Mul(x,Fact(_Add(x,-1))));
```

Lazy evaluation means that the arguments of a compound form call are *not* evaluated as the first step of the evaluation of the call. Instead, the interpreter first textually replaces the call with the definition of the particular compound form. The laziness is realized if this definition happens to be a call of `_If`, as it is the case with `Fact` above.

2.2.3 Storage allocation

Many macros need temporary storage: M_1 of Sec. 2.1.3 is an example. Suppose that some data storage is free at the point of a macro call. How should this storage be allocated to the temporary variables of the macro definition? The natural division of labor is as follows. We assume that there is a set of *storage classes*, and that the *storage cells* in each single class are fully interchangeable. The macro definition writer chooses the optimum storage class for each temporary, and the macro expander tries to find a free member of the chosen class. Thus, the macro writer completely specifies the *interclass* storage allocation, whereas the expander autonomously performs the *intraclass* allocation. (See Sec. 2.3 for examples.) In all, this policy means that temporary storage allocation faithfully reflects the structure of the macro hierarchy determined by the user.

The intraclass storage allocation is our main expansion-integrated domain-specific action. It is clearly non-trivial, as it requires flow (and pointer) analysis capability. Still, it is fairly *transparent*. In general, if a domain-specific action is not transparent, much of the usefulness of powerful domain-specific attributes may effectively be lost. It is not reasonable to devote oneself to ultimate refinement of the conditional expansion if some forthcoming expansion-integrated actions may have unpredictable consequences.

The main limitations of our current storage allocator implementation are the following two: it should be possible that two storage classes are defined to be physically overlapping; especially with indirect addressing, the user should be able to specify that two variables used together should also be located near each other [27].

2.3 Tutorial on the use of ReFlEx 1.0

We have implemented and documented an experimental prototype [22, 21, 23] (which is available through the Internet) of the suggested macro expander. The present Sec. 2.3 consists of an example of its use. This ReFlEx 1.0 system is aimed at demonstrative purposes only: it can cope only with utterly simplified and fully hypothetical processor architectures. The principal restrictions on the target architecture include the following:

- It is not possible to declare data vectors—only scalar variables can be used.

- Indirect addressing is not supported—the only possible addressing modes are direct and immediate addressing (see Sec. 2.2.1).
- There is no support for keeping track of register values (see Sec. 2.2.1).
- Any two registers must be distinct, that is, they cannot overlap (see Sec. 2.2.3).
- VLIW-type fine-grained instruction-level parallelism [12] is not supported (for code compaction, see Sec. 3.3).

Currently, we are working to eliminate these deficiencies. The first real targets we are approaching are stripped-down special-purpose processors like AT&T DSP1610 [4], a 16-bit fixed-point DSP chip.

2.3.1 Declaring the run-time data storage

A rule file read by the expander should start with a target architecture specification. First, we have to introduce the data storage elements, i.e. CPU registers and memory locations. (For brevity we skip here the compound form definition section of the rule file; compound forms `Int`, `Lt`, `Gte`, `And`, and `Or` will be used below.) As shown in Fig. 2, there are three storage classes: 1024 memory locations, four auxiliary registers, and a single accumulator.

```
STORAGE {
  M[1024];
  R[4];
  A[1];
}
```

Figure 2: Declaration of storage classes.

2.3.2 Declaring the machine instruction set

Second, we have to present the machine instructions as system macros; see Fig. 3. The *system macros* constitute the base of the macro hierarchy and the expander leaves their calls intact (thus, they are more like placeholders than proper macros). So we have, in all, ten system macros, among which there are one unconditional branch (`JUMP`) and three conditional branches (`BRANCH`). Most of the system macros are provided with a test (`TEST`) that constrains their use. For instance, `load` can only copy the contents of a memory location into an auxiliary register, while `store` is able to perform the opposite data transfer. Note in particular that the accumulator cannot be loaded directly from memory.

2.3.3 Defining higher-level macros

A rule file without non-system macro definitions would not be very useful. In Fig. 4, you can see how a general data transfer (without storage class restrictions) is implemented. The `my_move` macro definition lists the alternative implementations in the priority order;

```
KERNEL {
  set(c > r)
    { TEST And(And(?R(r),Int(c)), And(Gte(c,-1024),Lt(c,1024))); }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) { TEST And(Or(?A(s),?R(s)), Or(?A(d),?R(d))); }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}
```

Figure 3: Declaration of system macros.

note that the last alternative, `temp_is_needed`, is a recursive one. Some alternative implementations or *versions* are provided with a *version-specific test*: for example, the `same` version cannot be selected unless the source and destination of the transfer are the same storage cell. (The short definition of the empty `my_null` non-system macro is omitted here.)

```
my_move(s > d) {
  same: TEST =(s,d); { my_null(s > d); }
  as_set: { set(s > d); }
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  temp_is_needed: TEST And(Not(?R(s)), Not(?R(d)));
    USE R[r]; { my_move(s > r); my_move(r > d); }
}
```

Figure 4: Definition of the `my_move` macro.

Our second macro interchanges the contents of two memory locations; see Fig. 5. The `my_mswap` macro requires two free storage cells, because direct memory-to-memory transfers are not supported by the system macro set. At least one of these two must be an auxiliary register, while the other one may alternatively be the accumulator. Note also that, like many of the system macros in Fig. 3, the `my_mswap` macro includes a *macro-specific test*. This test is evaluated already as a part of the expansion of some upper-level macro call: only if the test succeeds, a call of the current macro can be accepted as a component of the expansion result of the upper-level call.

We could also define macros that are conditional or unconditional branches (see [22] for examples); we state again that control flow branches do not disrupt the data flow analysis

```
my_mswap(m,n > n,m) {
  TEST ?M(m,n);
  two_aux_free: TEST Gte(#R(),2);
    USE R[r]; { my_move(m > r); my_move(n > m); my_move(r > n); }
  acc_and_aux_free:
    USE A[a]; { my_move(m > a); my_move(n > m); my_move(a > n); }
}
```

Figure 5: Definition of the `my_mswap` macro.

performed by the macro expander.

2.3.4 Expanding macro calls

We are now ready to generate code for a call of the `my_mswap` macro (defined in Fig. 5). We start the macro expander with the rule file created above, and interactively type in the following line:

> *> my_mswap(M[5],M[7] > M[7],M[5]); {R[1],R[3]}*

Thus, we specify that registers `R[1]` and `R[3]` are free at the call point. An explicit specification like this is needed here but not allowed inside a macro definition. At the lower-level macro calls that are generated during the expansion of the original call, the expander can autonomously determine the free registers.

The expansion comprises two phases: first, a tree structure is built up; second, this structure is linearized. Fig. 6 shows the results of both these phases. (You may want to see [22, Sec. 7] for a detailed description of the macro expansion mechanism.) Note especially that when the expander is expanding the `my_move(M[7]>M[5])` call (on the second level), it has to recognize that the originally free `R[3]` register is not free any more.

3 A model for domain-specific macro expansion

In Sec. 3.1 we formulate a domain-independent model that captures the operation principles of our macro expansion system. The model should clarify the differences between our system and more conventional macro systems. Perhaps there are also other domains than ours (i.e. machine-level code generation) in which the model could be adopted. In Sec. 3.2 we state a Church-Rosser property for the model, and in Sec. 3.3 we deal with a particular important consequence of this property.

```
my_mswap(M[5],M[7] > M[7],M[5]) {R[1],R[3]}

my_mswap(M[5],M[7] > M[7],M[5])
    my_move(M[5] > R[3])
        load(M[5] > R[3])
    my_move(M[7] > M[5])
        my_move(M[7] > R[1])
            load(M[7] > R[1])
        my_move(R[1] > M[5])
            store(R[1] > M[5])
    my_move(R[3] > M[7])
        store(R[3] > M[7])

load(M[5] > R[3])
load(M[7] > R[1])
store(R[1] > M[5])
store(R[3] > M[7])
```

Figure 6: Expansion result of a `my_mswap` macro call.

3.1 Formulation of the model

First we introduce some basic concepts. (Note that we usually denote a set by an uppercase letter and its member by the corresponding lowercase letter.)

Specification 1
(a) *There is a countably infinite set M of* ***macro names****; moreover, M contains a subset $\widehat{M}$, which is finite.*

Set $\widehat{M}$ identifies the base-level macros, whose calls need not be expanded.

(b) *There is a set C of* ***macro calls****, and a function $\mu_C : C \to M$. Furthermore, the set of finite macro call sequences is denoted by S.*

Function μ_C identifies the macro called.

(c) *There is a set V of* ***macro versions****.*

A macro version is a user-written implementation outline for the macro; the user may provide a single macro with several alternative versions.

(d) *There is a set E of* ***environments****.*

The environment of a particular macro call contains all the information about its context included in the macro interface definition.

(e) *There is a set U of* ***user settings****.*

The top-level environment must be explicitly specified by the user; the lower-level environments are determined by the macro expander during the expansion.

We represent the user-written macro definitions by *generator* functions. Thus, a generator function selects the most suitable macro version for each call of the particular macro. This selection is based on the environment of the call, which can be examined by finding out the values of the available domain-specific attributes.

Definition 2
*A **generator** function is any function of the form $\gamma : E \to V \cup \{\bot\}$, where $\bot$ is a special value.*

Special value $\bot$ simply denotes macro expansion failure.

There must be a *library* of user-written macros. It constitutes the rule base driving the expansion.

Definition 3
*A finite set Λ of pairs $\langle m, \gamma \rangle$, where m belongs to $M - \widehat{M}$ and γ is a generator function, is a **library** if for each two distinct $\langle m', \gamma' \rangle$ and $\langle m'', \gamma'' \rangle$ in Λ, it holds that $m' \neq m''$.*

The built-in domain-specific expansion-integrated actions (such as our intraclass cell allocation) are taken to be performed by an universal *transformer* function, which finishes the result of each individual expansion step by transforming the user-written version specification into a macro call sequence.

Specification 4
*There is a universal **transformer** function $\tau : E \times (V \cup \{\bot\}) \to S \cup \{\bot\}$. Specifically, for each e it holds that $\tau(e, \bot) = \bot$.*

The hierarchical structure that originates from a single macro call and is during the expansion augmented with new branches is, not surprisingly, called a *tree*. Each tree consists of a collection of *nodes*, some of which are *leaves*, and one of which is the *root*.

Definition 5
*The set of **trees** is the smallest set T of objects that meets both the following conditions:*
(a) *Each macro call c belongs to T. In this case, sets $N(c)$ and $L(c)$ are both defined as $\{c\}$, and $r(c)$ is simply defined as c.*

For each tree t, $N(t)$ is the set of nodes, $L(t)$ is the set of leaves, and $r(t)$ is the root.
(b) *Each pair $\langle c, z \rangle$ belongs to T if c is a macro call and z is a finite sequence whose elements $t_1, \ldots, t_m$ all belong to T, such that for each t in z it holds that $c \notin N(t)$, and for each two distinct i' and i'' in range $1, \ldots, m$ it holds that $N(t_{i'}) \cap N(t_{i''}) = \emptyset$. In this case, $N\langle c, z \rangle$ is defined as $\{c\} \cup \bigcup_{i=1}^{m} N(t_i)$, and $L\langle c, z \rangle$ is defined as $\bigcup_{i=1}^{m} L(t_i)$, whereas $r\langle c, z \rangle$ is again simply defined as c.*

Note that the above definition utilizes the concept of a node set only to ensure that all the macro calls in any given tree are unique. In practice, uniqueness is achieved by providing the macro calls with unique labels. (We assume that the set of labels is countably infinite. A set of "available" labels is a component of the environment of each macro call. At each new expansion step, the countably infinite label set of the old leaf is partitioned into a finite number of countably infinite subsets that are allocated to the new leaves.)

The form of the expansion-time tree structure depends on the initial environment specification written by the user. (For instance, a tree whose all leaves are base-level macro calls may be unachievable if there is no free temporary storage even to begin with.) A tree accompanied by this specification is called an *x-tree*.

Definition 6
*An **expansion tree** or, more shortly, an **x-tree** is a pair $\langle u, t \rangle$, where $u \in U$ and t is a tree.*

For each x-tree $x = \langle u, t\rangle$, we simply define $N(x) = N(t)$, $L(x) = L(t)$, and $r(x) = r(t)$.

The expander determines the expansion-time environment of each macro call instance. We assume that for this purpose the expander incorporates two built-in *extraction* functions.

Specification 7
(a) *There is a universal* ***initial extractor*** *function* $\widehat{\xi} : U \times C \to E$.
(b) *There is a universal* ***extractor*** *function* $\xi : E \times S \times C \to E$.

With the above extraction functions, the environment of a macro call can be given a recursive definition. Specifically, part (b) of Definition 8 determines the environment of a node in a tree by referring to the environment of the father of that node, whereas part (a) is the recursion base.

Definition 8
Each node c of an x-tree $x = \langle u, t\rangle$ has an unique environment $e(x, c)$.
(a) *If c is the root of x, then $e(x, c) = \widehat{\xi}(u, c)$.*
(b) *Otherwise, x contains such a subtree $\langle c', \langle t_1, \ldots, t_m\rangle\rangle$ that sequence $\langle r(t_1), \ldots, r(t_m)\rangle$ contains c; in this case, $e(x, c) = \xi(e(x, c'), \langle r(t_1), \ldots, r(t_m)\rangle, c)$.*

Now we are ready to define an expansion step as a relation between two x-trees.

Definition 9
An x-tree x' is an ***immediate derivative*** *of an x-tree x with respect to some library Λ, denoted as $x \Longrightarrow_\Lambda x'$, if there is a leaf c of x and $\langle m, \gamma\rangle \in \Lambda$ such that all the following conditions are met:*
(a) $\mu_C(c) = m$.
(b) *$s = \tau(e(x', c), \gamma(e(x', c)))$ is a macro call sequence (i.e. different from $\perp$).*
(c) *x' is obtained from x by replacing c with $\langle c, s\rangle$.*

Naturally, we define expansion as the reflexive-transitive closure of the one-step expansion.

Definition 10
An x-tree x' is a ***derivative*** *of an x-tree x with respect to some library Λ, denoted as $x \Longrightarrow_\Lambda^* x'$, if the pair $\langle x, x'\rangle$ belongs to the reflexive-transitive closure of the $\Longrightarrow_\Lambda$ relation.*

The final expansion result is represented by the leaf sequence of a *complete* x-tree.

Definition 11
An x-tree x is ***complete****, with respect to some library Λ, if $x \Longrightarrow_\Lambda^* x'$ implies $x' = x$.*

(You may notice that the above model is somewhat simplified with respect to the description of our macro expander in Sec. 2.3: the model lacks explicit macro-specific tests, and in it the generator function returns only one version instead of a prioritized list of versions. In principle, these same simplifications could be followed in ReFlEx 1.0 programming, but the resulting code would be quite a bit longer and more difficult to maintain.)

3.2 Church-Rosser property

We are able to state the following Church-Rosser-type property [15].

Proposition 12
Suppose that $x \Longrightarrow_\Lambda^* x'$ *and* $x \Longrightarrow_\Lambda^* x''$*. Then there exists such* x^* *that both* $x' \Longrightarrow_\Lambda^* x^*$ *and* $x'' \Longrightarrow_\Lambda^* x^*$*.*

The main point of the readily seen proof is that in part (b) of Definition 8, we have $e(x,c) = \xi(e(x,c'), \langle r(t_1), \ldots, r(t_m)\rangle, c)$ instead of the perhaps more general-looking alternative $e(x,c) = \xi(e(x,c'), \langle t_1, \ldots, t_m\rangle, c)$. In other words, the environment of a macro call c depends on no other parts of the possibly already existing subtrees t_i than their root macro calls (which exist whenever c exists). For example, in Fig. 7 only the white nodes contribute to the environment of node *N1*: this environment is determined by nodes *N1*, *N2*, and *N3*, and the environment of node *N0*. Consequently, the structure of each subtree in an x-tree is independent of the structure of the neighboring subtrees. This, in turn, implies that the macro expansion result is not affected by the expansion order of the leaf macro calls. If a complete x-tree exists, then it is unique and cannot be missed by a wrong choice of the leaf to be expanded next.

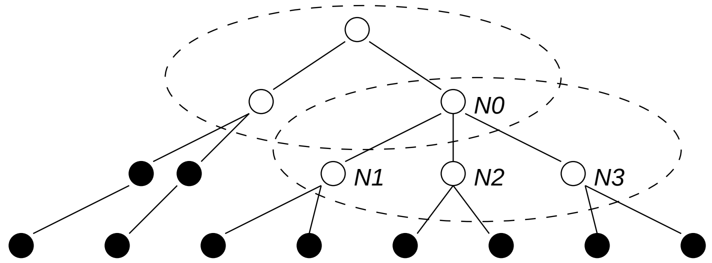

Figure 7: Determining environment for node *N1*.

In fact, it is obvious that if the unique complete x-tree exists, then it is reached by a constant number of one-step expansions regardless of the leaf expansion order. Thus, the actual leaf selection policy is indeed fully insignificant: the expansion is *order-independent*.

In most macro systems known to us the macro expansion result does depend on the expansion order. Usually, the expansion is defined to take place in a strict depth-first left-to-right (or top-down) order. In particular, this order affects macro variables (in contrast to macro parameters), whose currently effective values are determined by their most recent definitions encountered by the expander. Moreover, in many systems it is possible to redefine macros, and the visibility of a macro definition is similar to the visibility of a (global) macro variable definition. Some disparities between the left-to-right bias and machine-level code generation were hinted at the end of Sec. 2.1.2. More generally, the bias may complicate the maintenance of the code: a change in a minor macro can propagate forward through the whole source program and affect any later macro call. (Note that in a "purely functional" general-purpose macro system without macro variables and macro redefinitions the bias would be insignificant.)

Due to its order-independence, our macro system has no left-to-right bias. Instead, we (have to) rely on the symmetric inherited context, i.e. the expansion-time environment, which can be examined with domain-specific attributes. The advantage of the order-independence is the increased maintainability: a change in a macro definition may propagate only downwards—not left, right, or upwards—to the descendants of each call of the particular macro in an expansion tree; and even in the downward case, the modules constituted by the lower-level macros are guarded by the module interface. Deep macro hierarchy is in principle enabled by the modularity (see Sec. 2.1.3) and in practice encouraged by the order-independence: the former guarantees global consistency and the latter facilitates extreme local optimization. Specifically, because of the downwards-only propagation of modifications, it is reasonable to make each coding decision on the lowest possible level of the macro hierarchy.

3.3 Layered expansion

Much of the value of the order-independence is in the fact that there is always the option to fix the expansion order in any way that for some reason seems desirable. Accordingly, it may sometimes be reasonable to divide the macro hierarchy into explicit layers. This means that the expander processes the macro calls on a layer-by-layer basis, i.e. in a breadth-first fashion within a single layer until all the leaves of the expansion tree are calls of macros marked to belong to the next-lower layer. Such a strategy has the following rationale: before processing the next layer, the expander may step aside for a moment and some "external" layer-specific global optimization procedure may temporarily take over. What the optimizations are like is of no concern to the expander, which in due time recovers the control and starts afresh. (The expander is not astonished by the news that the external optimizer has removed from the tree all other nodes than the original root and the fresh set of leaves. By modularity, the code remains consistent even after such a drastic action, as long as the optimizer respects the environment of the root.)

In the case of machine-level code generation, likely examples of external optimizations are *code compaction* [11, 16] as a postprocessing phase and, say, global enhancement of the register allocation as an intermediate optimization phase. Indeed, as depicted even in Fig. 8, we suggest that the inherently local macro expansion and more conventional global optimization procedures can be profitably combined.

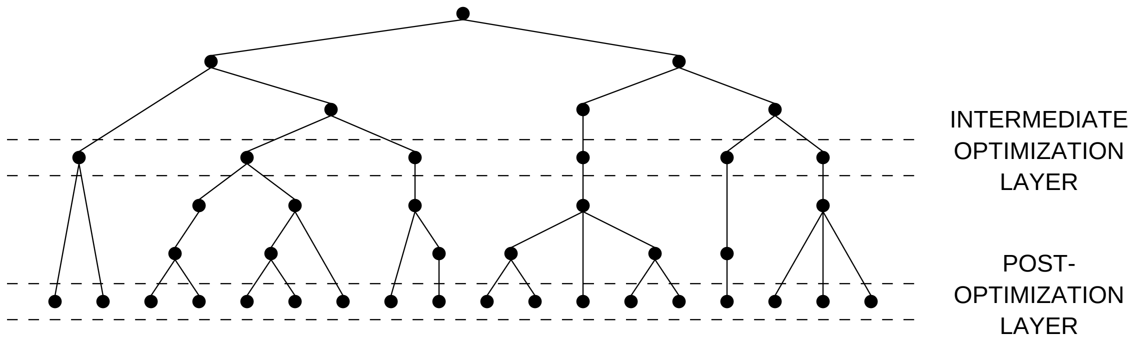

Figure 8: Expansion with two conversion layers.

In Sec. 1, we remarked that the domain-specific actions of Knuth's TEX are all conversion-integrated, taking place only after the whole macro expansion. By 'conversion' we mean non-transparent language translations based on no explicit rules, as opposed to macro expansion. High-level programming language compilation is a typical example of conversion. (Actually, the TEX conversion is found to be a hybrid between compilation and interpretation; in the case of pure compilation, domain-specific attributes—such as the height of the text accumulated on the current page—could not be available for the user.) In principle, conversion-integrated domain-specific actions have a great potential, since their implementation method is left unspecified. As the above layer-specific global optimizations can be classified as conversions, we may say that the proposed macro expansion scheme also provides a reasonable hook for conversion-integrated domain-specific actions even in the middle of expansion.

4 Conclusions

The implemented ReFlEx 1.0 is a demonstration prototype of a special-purpose macro expander whose domain is machine-level code generation. It has the following distinctive features:

- Order-independence.
- Crucial importance of domain-specific attributes.
- Explicit role of expansion-integrated domain-specific actions.

Ordinary macro systems do not have order-independence. Many special-purpose macro systems, such as TEX or traditional macro assemblers, provide a few domain-specific attributes; an early but elaborate example of macro-based—but non-hierarchical—code generation with optimization-oriented domain-specific attributes is given in [3] (as opposed to ours, these attributes can be implemented through a hidden general-purpose macro expander in the manner sketched in Sec. 2.1.2). However, in these special-purpose macro systems all domain-specific actions are conversion-integrated.

Why could the proposed macro expansion scheme complement the more conventional machine-level code generation methods such as mentioned in Sec. 2.1.1? First, these pattern matching methods effectively process program descriptions consisting of character strings, whereas we explicitly try to become aware of the dynamic utilization of the processor's physical resources. Second, our macro definitions are modular and our expansion is order-independent, whereas the patterns of a pattern matcher are not independent of each other: if some patterns and their associated costs are about to be changed, the prediction of all the consequences in advance is difficult.

The principles of the proposed domain-specific macro expansion scheme could perhaps be applied to tasks in which the solution consists of a sequence of tightly coupled subtasks to which a set of limited resources—such as CPU registers—must be allocated. In such cases, it is important not only what is done but also how it is done. In particular, domain-specific macro expansion enables user-controlled hierarchical resource allocation, which may be worth a try if the set of resources is so heterogeneous that global optimization methods do not work well—as it is the case with the register set of many embedded special-purpose processor architectures. The fact that TEX macro expansion proper does not incorporate more domain-specificity is not surprising as the main elements in the rather homogeneous resource set are horizontal and vertical space. Finally, to give you are an even better intuitive idea of the task type we mean, we more or less spontaneously mention two rough everyday examples: to travel from city A to city B, and to prepare a dinner.

References

[1] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 4, pp. 491–516. October 1989.

[2] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sundarsanam, S. Tjiang, and A. Wang. Challenges in code generation for embedded processors. In [29], pp. 48–64.

[3] B.W. Arden, B.A. Galler, and R.M. Graham. The MAD definition facility. *Communications of the ACM*, vol. 12, no. 8, pp. 432–439. August 1969.

[4] AT&T Microelectronics. DSP1610 digital signal processor information manual (MN90-020DMOS). Allentown (Pennsylvania, USA), December 1992.

[5] P.J. Brown. *Macro Processors and Techniques for Portable Software.* Wiley, London (UK), 1974.

[6] M. Campbell-Kelly. *An Introduction to Macros.* Macdonald, London (UK), 1973.

[7] A.J. Cole. *Macro Processors* (second edition). Cambridge University Press, Cambridge (UK), 1981.

[8] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, vol. 29, no. 6 (Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation), pp. 242–256. June 1994.

[9] H. Emmelmann, F.-W. Schöer, and R. Landwehr. BEG—a generator for efficient back ends. *ACM SIGPLAN Notices*, vol. 24, no. 7 (Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation), pp. 227–237. July 1989.

[10] D.J. Farber. A survey of the systematic use of macros in systems building. *ACM SIGPLAN Notices*, vol. 6, no. 9 (Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation), pp. 29–36. October 1971.

[11] J.A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490. July 1981.

[12] J.A. Fisher and B.R. Rau. Instruction-level parallel processing. *Science*, vol. 253, no. 5025, pp. 1233–1241. September 1991.

[13] C.W. Fraser, D.R. Hanson, and T.A. Proebstring. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, pp. 213–226. September 1992.

[14] J.G. Ganssle. *The Art of Programming Embedded Systems.* Academic Press, San Diego (California, USA), 1992.

[15] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge (UK), 1986.

[16] S.M. Kafka. An assembly source level global compacter for digital signal processors. *Proceedings of 1990 International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 1061–1064. IEEE, 1990.

[17] W. Kent. Assembler-language macroprogramming: a tutorial oriented toward the IBM 360. *ACM Computing Surveys*, vol. 1, no. 4, pp. 183–196. December 1969.

[18] D.E. Knuth. *The TEXbook.* Addison-Wesley, Reading (Massachusetts, USA), 1984.

[19] L. Lamport. *LATEX: A Document Preparation System* (second edition). Addison-Wesley, Reading (Massachusetts, USA), 1994.

[20] J. R. Larus. Assemblers, linkers, and the SPIM simulator. In D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Appendix A. Morgan Kaufmann, San Mateo (California, USA), 1994.

[21] E. Lassila.[1] A macro expansion approach to embedded processor code generation. In presentation material of Workshop on Interaction between Compilers and Computer Architectures. San Jose (California, USA), February 1996.

[22] E. Lassila.[1] ReFlEx—an experimental tool for special-purpose processor code generation. Report B15, Digital Systems Laboratory, Helsinki University of Technology. Espoo (Finland), March 1996.

[23] E. Lassila.[1] A macro expansion approach to embedded processor code generation. *Proceedings of EUROMICRO 96 Conference*, pp. 136–142. IEEE Computer Society Press, Los Alamitos (California, USA), 1996.

[24] E. A. Lee. Programmable DSP architectures. *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19 (Part I), and vol. 6, no. 1, pp. 4–14 (Part II). October 1988 and January 1989.

[25] S. Y.-H. Liao. Code generation and optimization for embedded digital signal processors. Ph. D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Cambridge (Massachusetts, USA), 1996.

[26] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded DSP microprocessors. *Proceedings of 32nd Design Automation Conference*, pp. 599–604. ACM, 1995.

[27] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pp. 235–253. May 1996.

[28] V. K. Madisetti. *Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*. Butterwoth-Heinemann, Newton (Massachusetts, USA), 1995.

[29] P. Marwedel and G. Goossens (eds.). *Code Generation for Embedded Processors*. Kluwer, Boston (Massachusetts, USA), 1995.

[30] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, vol. 3, no. 4, pp. 214–220. April 1960.

[31] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP design tool requirements for embedded systems: a telecommunications industrial perspective. *Journal of VLSI Signal Processing*, vol. 9, no. 1, pp. 23–47. January 1995.

[32] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York (New York, USA), 1987.

[33] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, Chichester (UK), 1992.

[34] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, vol. 30, no. 6 (Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation), pp. 1–12. June 1995.

[35] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, vol. 14, no. 7, pp. 41–47. July 1981.

[1] See `ftp://saturn.hut.fi/pub/reflex/README` for an electronic copy.