Synthesizing Safety Controllers for Interactive Web Services

Claus Brabrand
brabrand@daimi.au.dk

December 21, 1998

Abstract

We show how to use high-level synchronization constraints, written in a version of monadic second-order logic on strings, to synthesize safety controllers for interactive web services. On top of this, we introduce a syntactic level macro language that permits the language to be transparently extended with new constructs. These new constructs will appear as if they were an inherent part of the original language. Together, the macros and constraints, provide a means for extending the original language with sophisticated high-level concurrency abstractions.

Contents

Introduction

The thesis consists of four chapters, the contents of which will be briefly outlined below.

Chapter 1

The first Chapter, *"The Constraint Language"*, introduces the idea of restricting the execution of an interactive web service according to a set of global safety requirements specified in monadic second-order logic on strings. This is done by extending the source code of the service program with labeled points. The execution of the service will give rise to a sequence of labels, corresponding to the labels past during execution. It is this sequence that is restricted by the safety requirements.

We shall also introduce a notion of triggers, that will take us beyond the boundaries of regularity otherwise imposed by the logic. Technically, the safety requirements are translated into counter automata, which are essentially DFA's augmented with a notion of integer counters paralleling the logic level trigger concept. These counter automata will be the core of a controller process associated with the web service, that is given the power to delay (even indefinitely) the passing of labels that when passed will cause the safety requirements to be violated.

Prohibiting two consecutive A labels, not having a B between them, yields mutual exclusion between the labels A and B. More sophisticated abstractions can and will be introduced.

Even though highly sophisticated concurrency mechanisms can be implemented, they are hard to capture as abstractions. This calls for some form of abstraction mechanism, which is the topic of the next Chapter.

Chapter 2

The second Chapter, *"The Macro Language"*, will introduce a macro language, that allows for the definition and preservation of parameterized abstractions. The Chapter will commence with a brief macro survey, investigating a few of the most popular macro languages. Hereafter, attention is turned towards our macro language.

Our macro language operates on a syntactic level as opposed to the lexical level of operation of most other macro languages. The macro language thus operates on parse trees, rather than uninterpreted sequences of characters (that is, raw text). All macros are assigned a syntactic category, a non-terminal type, upon definition, to which the body of the macro and all invocations must comply. Similarly, all formal macro parameters are typed with a non-terminal, to which all actual macro parameters are expected to comply. This provides an intuitive way of adding together parse trees according to the productions in the grammar.

Along with the definition of a macro, one can specify the exact syntax invocations are enforced to have. This enables a macro to be tailored to have the right look-and-feel, conveying the meaning and nature of the abstraction.

Since we operate on parse trees, our macros will know enough of the structure of the language to automatically support alpha conversion of identifiers. This will avoid the unintentional identifier clashes that are common to most other macro languages.

All this will provide a means for transparently extending the language with new constructs that will appear as if they were part of the original language.

Chapter 3

The third Chapter, *"Synthesis"*, will show how the two preceding independent languages can be combined so as to provide high-level concurrency abstractions as if they were an inherent part of the original language. It will also be shown how the two languages can provide abstractions of varying nature, ranging from basic primitives to entire new concepts. Also, the Chapter will illustrate how the macro language can gradually "evolve" the original language. The entire Chapter can be perceived as a conclusion on what can be achieved when combining our two languages.

Chapter 4

The final Chapter, *"The Runtime System"*, is an article "A Runtime System for Interactive Web Services", that will explain the runtime model we have used in <**bigwig**>.

Implementation Status

All the work presented in this thesis, has been incorporated into the <**bigwig**> language (see http://www.brics.dk/bigwig/") which is an intellectual descendant of the **MAWL** language. <**Bigwig**>is a domain specific language with a **C**-like syntax for generating interactive web services from high-level specifications.

At the time of writing, everything presented in the thesis has been implemented, excepting the four features; timeout, dead, event, and split.

Thesis Homepage

We have provided a small homepage for the thesis, which is available at the URL: "http://www.daimi.au.dk/~brabrand/thesis/". It contains references to the macro libraries mentioned and the examples from the thesis, that can be seen pretty printed in HTML (with and without macro expansion) and tried out.

Chapter 1

The Constraint Language

1.1 Introduction

When programming in a concurrent environment, programmers often need to make sure that certain properties hold—or, dually, that certain properties are never violated. In this chapter we shall present a general solution to this problem.

Problems of this kind are often neglected, in fact sometimes even ignored. This is at least true for cgi-based Internet services that quite often place their fate in the hands of probabilistics rather than sound programming.

We shall emphasize that our aim is not to perform a thorough investigation of concurrent programming, but to present a framework in which many aspects of concurrent programming can be uniformly handled.

Our solution is to permit various requirements to be stated along with the "usual" service code. Given this as input, the compiler will in turn produce a service program that is guaranteed to obey the specified requirements. Technically, the compiler will produce a centralized component, called a *controller* that will be running along with the associated sessions. For a diagram of the overall compilation process see Figure 1.1.

The controller process will control the service, prohibiting it from behaving in an illegal manner (with respect to the requirements). Each time a session desires to pass certain points deemed (by the service programmer) to be critical or otherwise of interest to the controller, the session will ask the controller for permission to continue.

The controller will thus act as a big brother to the sessions by stalling them, granting permission to continue only when (or if) the entire service has reached a state in which this is safe. At this point the session will resume execution from whence it paused.

One could of course argue that the introduction of such a centralized component may cause bottlenecks. However, we claim that two factors render this danger minimal—namely that of fast computers and slow networks. In general,

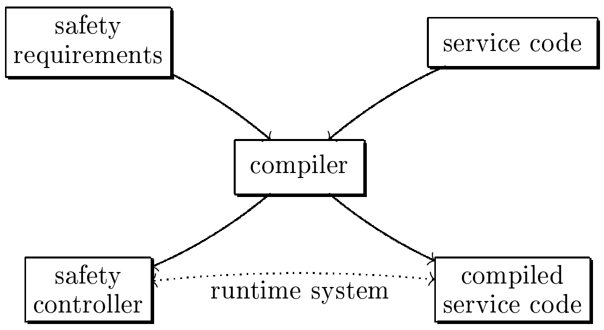


Figure 1.1: The compilation process.

the speed of a modern network as opposed to that of a contemporary web-server easily permits overhead associated with the processing of cgi-requests. Even to-day, the most accessed web servers have glowing networks while the cpu spends most of its time "sleeping". In other words, the network is by far the worst bottleneck.

First (in Section 1.2), we shall say nothing of how these requirements are specified, but only look at the way the session interacts with the controller, whilst ignoring the details of exactly how the controller works and how it is produced. In the remaining parts of this chapter, we shall turn our attention towards the specification of such requirements.

1.2 Session-Controller Interaction

Essential to the entire system is the interaction between the runtime safety controller and the session code through the runtime system (as depicted in Figure 1.1). The statement **wait** provides this interaction. The syntax of this construct can be seen in Figure 1.2.

In the following, we shall look at the semantics of the **wait** statement and the surface of the controller.

We shall use the term *control logic* for the inside of the controller (please note that the word is not inherently biased towards the use of logic). We shall perceive this control logic as a black box that somehow does all the magic. For an overview of the components of the controller see Figure 1.3.

The **wait** construct essentially gives the programmer a way of introducing labeled points in the service code, that may be of interest to the controller. The controller in turn has the power to restrict the execution of the service by stalling the sessions at these labeled points in such a way that certain stated global safety requirements are satisfied.

stm_list	::=	*stm**
stm	::=	...
	|	**wait** *id* ;
	|	**wait** { *waitbranch_list* }
waitbranch_list	::=	*waitbranch*$^+$
waitbranch	::=	**case** *id* : *stm_list* **break** ;
	|	**case** ! *id* : *stm_list* **break** ;
	|	**dead** : *stm_list* **break** ;
	|	**timeout** *exp* : *stm_list* **break** ;

Figure 1.2: The syntax of wait-statements.

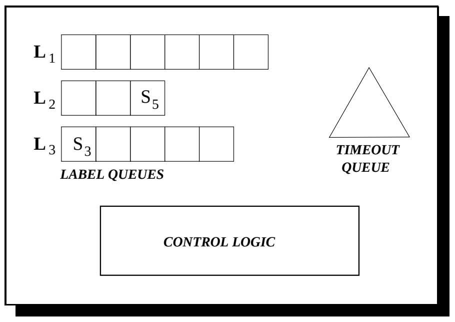


Figure 1.3: The components of the controller.

1.2.1 The Short Wait

The syntax of the short wait statement is: "**wait** L;". Whenever a session reaches such a statement, the following happens. The session will contact the controller through the runtime system, thus requesting permission to pass the label (in this case "L") and suspend (implemented as "unix-sleep") execution. The controller responds by placing the requesting session in a queue corresponding to the label it requested permission to pass (that is, a queue "L"). The controller is thus equipped with one queue for each label appearing in the service program. The controller in Figure 1.3 has three queues, namely L_1, L_2, and L_3 with two sessions S_3 and S_5 waiting on the labels L_3 and L_2, respectively. The timeout queue will be explained later.

When (or if) the controller reaches a state in which the session can continue without violating the requirements, it will inform the stalled session that it is safe to continue after which the session will do just that.

In order for this to be fair, the controller uses queues plus a token ring (see Figure 1.4) in order to determine which session is the next one allowed to pass a given label. The requests are of course placed in queues, so that a session issuing a wait on a given label always is guaranteed to be allowed to continue before any other sessions doing the same thing, only later.

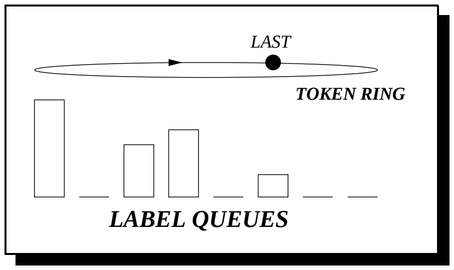


Figure 1.4: The Token Ring.

We shall say that a queue L is *enabled* if and only if a label L can be passed without the requirements being violated.

The token ring is there to ensure that no non-empty enabled queues are inspected more often than others. To this end, the controller will perpetually circle all the queues looking for ones that are non-empty and enabled. Whenever such a queue L is found, the affected session is awaken and given the message that it has been granted permission to pass the label L. We shall say that the controller has *taken* the label L.

The token ring will remember which queue was last taken, so the controller can start its circling of the queues with the queue immediately following the one last serviced. When the controller has done one full circle of the queues without taking any labels, it will go to sleep until new requests arrive (continued circling would not amount to anything).

Example—A Critical Region

A critical region can easily be protected by sticking two wait statements around it as in example 1.5. This of course assumes that the control logic somehow guarantees mutual exclusion between the two labels (enterR and exitR).

```
session S() {
    ...
    wait enterR;
    /* critical region */
    wait exitR;
    ...
}
```

Figure 1.5: A critical region.

1.2.2 Generalizing Wait

The short **wait** will suffice for many cases. Sometimes, however, it would be nice with something a bit more expressive. Imagine a scenario containing two resources that are of equal importance (in the sense that none is preferred over the other) and require exclusive access. This could of course be achieved by a few global variables and critical regions. However, if we were to permit a session to branch on the state of the controller, by letting it wait on a number of labels, we could implement the imagined scenario quite elegantly (see Figure 1.6).

Exactly where execution resumes now depends on the controller that responds by telling the session which label was passed. The execution thus resumes at the appropriate point. This syntax was chosen because it is reminiscent of a **switch** in the language **C**.

Having added the possibility of specifying several labels in a **wait** statement, the simple strategy of placing the sessions in queues no longer suffices—we need something more. We must be able to atomically remove a session from all of its queues, when the controller *takes* one of its labels allowing it to continue. In order to accommodate this need, we have linked all the entries derived from the same **wait** statement into something we shall call a *wait chain*. This way the controller can follow the wait chain and remove all entries associated with a given session. Such a link can be seen in Figure 1.7 where the session S_3 apparently is waiting for one of the labels L_1 and L_3 to be taken (the timeout element in the wait chain will be explained below).

As one can observe in Figure 1.8, the short **wait** is merely a special case of the more general **wait**. Still, it is nice to have the short **wait** as a shorthand.

```
session  S() {
    ...
    wait {
       case enterR1:
          /* critical region1 */
          wait exitR1;
          break;
       case enterR2:
          /* critical region2 */
          wait exitR2;
          break;
    }
    ...
}
```

Figure 1.6: Two critical resources.

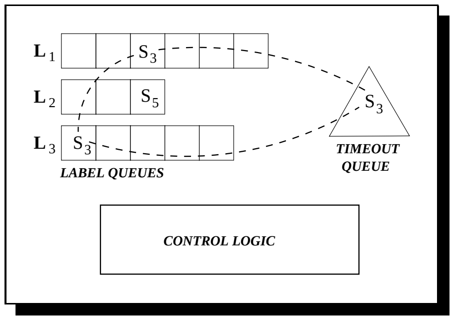


Figure 1.7: The controller (with a wait chain).

```
                   wait {
                      case L:
wait L;     ≡            /* empty */;
                         break;
                   }
```

Figure 1.8: Short wait in terms of the general wait.

1.2.3 Timeout

As an additional feature, we have included a timeout branch in the **wait** statement. This gives the service programmer the possibility of timing out during a **wait** which would otherwise not be possible. The expression associated with the branch is required to be an integer that will specify a time bound in seconds. The semantics of a **wait** statement with a timeout branch is the following. If the session cannot be granted permission to pass any of the labels in the wait statement within the given time bound, the session will be informed by the controller that a timeout occurred. At this point, the session will continue by executing the statement associated with the timeout branch.

In Figure 1.9, the session will wait for permission to pass the label L in which case the statement S will be executed. If permission is not granted within τ seconds, the controller will remove the wait-chain, awake the waiting session and inform it that a timeout has occured. The session will thus cease waiting and execute the corresponding timeout statement S'.

```
wait {
    case L: S;
       break;
    timeout τ: S';
       break;
}
```

Figure 1.9: Timeout example.

Seconds were chosen as the time unit here because it did not seem sensible to tighten the precision. This is because of the delay associated with the shipping of the respons over the network. The time bound should not be relied upon for high-precision tasks, as it might be the case that the controller is busy circling the queues at the precise moment when the time has gone. Rather, one should think of it as if the controller had the right to timeout a session at any point in time after τ seconds after having received and processed the wait request. And indeed it will, only perhaps not precisely τ seconds later.

A timeout branch will also give rise to an entry in the wait chain—the entry is placed in a special timeout queue (see Figure 1.7). The timeout queue has a tree-like form from the fact that it is implemented using a tree-based priority queue.

It will not make sense to specify more than one timeout branch because the one with the lowest time bound will deterministically always win. Thus we shall enforce that there be only one such.

1.2.4 Dead

A wait statement is said to be *dead*, when none of the involved labels can ever be taken by the controller. The idea with the dead feature is to allow a waiting service to act accordingly when or if the controller discovers that it is in fact dead. Unfortunately, as shown in Section 1.9 this is in general undecidable.

A dead branch in a wait statement also containing a timeout branch will have no effect whatsoever because the timeout would always eventually occur. To this end, we will not permit the two branches to appear in the same wait statement. In fact, to enhance readability we require that any dead or timeout branch be specified as the last branch in a wait statement.

1.3 SyCoLogic

In this section we shall turn our attention towards the safety requirements and look inside the black box—the control logic.

We shall place three demands on the safety requirements. They should be *robust*, *succint*, and specified in an *intuitive* notation. In other words, they should be easy to formulate as well as modify. Therefore, logic seems as the natural basis for the specification language. First of all, logic is in its nature much closer to the language in which informal requirements are written. Furthermore, it is very easy to add, remove and change parts of a logic-based specification as opposed to specifications in terms of automata—the latter being highly sensitive to even minor changes. Almost the same arguments applies to regular expressions. Furthermore, logic is in general much more succint than automata or regular expressions.

Because of this, we have based the specifications on second-order monadic logic for strings (a.k.a. **M2L-Str**). This choice is further motivated by the existence of a very well functioning tool called **MONA** (see the project homepage: http://www.brics.dk/mona/) for compiling **M2L-Str** formulas into minimal deterministic finite automata.

M2L-Str is very expressive and succint logic. However, there exists formulas, ϕ, for which the corresponding MDFA (minimal-state deterministic finite automaton), A_ϕ, is of non-elementary size. That is, if ϕ is of size n, then the corresponding automaton A_ϕ can be of size $2^{\cdot^{\cdot^{2}}}\}n$.

This is of course, seen from a complexity point of view, very bad news. But seen from the point of view of language design, though, it is useful to have such a succinct logic as the basis of ones specification language. A succinct logic will simply imply shorter specifications that are easier to write.

Given such a safety requirement, ϕ, and a sequence of labels $\omega \in \Sigma^*$, the formula ϕ will either evaluate to **true** or **false** when interpreted over ω. The former case is denoted by $\omega \models \phi$, the latter by $\omega \not\models \phi$. In this way, the formula ϕ induces a language, namely the set of sequences of labels for which the formula is true, that is,

$$L(\phi) = \{\omega \in \Sigma^* \mid \omega \models \phi\}$$

Using the **MONA** tool, such a formula ϕ is now turned into *the* unique corresponding MDFA A_ϕ.

We shall characterize an automaton A by a five tuple $(Q, \Sigma, \hat{q}, \rightarrow, F)$, where Q is a finite set of states, Σ is a finite set of labels, $\hat{q} \in Q$ is the initial state, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation (or transition function from $Q \times \Sigma$ to Q, if A is deterministic), and $F \subseteq Q$ is the set of acceptance states (a.k.a. final states).

By this convention, the automaton above A_ϕ can be specified as (Q_ϕ, Σ_ϕ, $\hat{q}_\phi$, $\rightarrow_\phi$, F_ϕ).

We define the relation $\rightsquigarrow \subseteq Q \times \Sigma^* \times Q$ (abbreviated as $q \overset{\omega}{\rightsquigarrow} q'$ for $(q, \omega, q') \in \rightsquigarrow$) inductively in terms of $\rightarrow$ by:

$$\begin{aligned} & q \overset{\epsilon}{\rightsquigarrow} q \\ & q \overset{\omega}{\rightsquigarrow} q' \ \textbf{ iff } \ \exists q'' \in Q, \sigma \in \Sigma : q \xrightarrow{\sigma} q'' \wedge q'' \overset{\omega'}{\rightsquigarrow} q' \wedge \omega = \sigma\omega' \end{aligned}$$

The language associated with the automaton can now be defined by:

$$L(A_\phi) = \{\omega \in \Sigma^* | \exists q_\phi \in F_\phi : \hat{q}_\phi \overset{\omega}{\rightsquigarrow}_\phi q_\phi\}$$

MONA guarantees that $L(\phi) = L(A_\phi)$. It is this automaton that will be used as a controller to restrict the execution of the service. It has the property that a sequence of labels $\omega \in \Sigma^*$, is accepted by A_ϕ ($\omega \in L(A_\phi)$), if and only if it satisfies the safety requirement ϕ (that is, $\omega \models \phi$).

1.3.1 Restricting Execution

With the introduction of labeled points in the service code, each service run will give rise to a (finite or infinite) sequence of labels. Namely, the sequence of labels the service passed during execution. It is exactly this sequence that will be restricted through our use of safety requirements.

A service S can be perceived as an **infinite** state automaton $A_S = (Q_S$, Σ_S, $\hat{q}_S$, $\rightarrow_S$, $F_S = Q_S)$. The transition relation $\rightarrow_S$ corresponds to passing labels during execution. Note that this automaton is neither guaranteed to be finite nor deterministic.

Assuming that $\Sigma_S = \Sigma_\phi$ (in the following referred to as Σ), we can define the product automaton $A_S \times A_\phi$ as ($Q_S \times Q_\phi$, Σ, $(\hat{q}_s, \hat{q}_\phi)$, $\rightarrow_{S\times\phi}$, $F_S \times F_\phi$). The transition relation is defined compositionally as:

$$(q_s, q_\phi) \xrightarrow{\sigma} (q'_s, q'_\phi) \Leftrightarrow q_s \xrightarrow{\sigma}_S q'_s \wedge q_\phi \xrightarrow{\sigma}_\phi q'_\phi$$

From this we obtain:

$$\forall \omega \in \Sigma^* : \omega \in L(A_S \times A_\phi) \Leftrightarrow \omega \in L(A_S) \wedge \omega \in L(A_\phi)$$

In other words $L(A_S \times A_\phi) = L(A_S) \cap L(A_\phi)$. Again, this product automaton might not be finite (if A_S is not). If ω is accepted by the product automaton, then ω specifies an execution of the service S ($\omega \in L(A_S)$) **and** furthermore an execution that does not violate the safety requirement ϕ ($\omega \models \phi$). Thus, the product automaton has exactly the behavior we want our system to have.

Of course, we simply cannot run a service and then when (or rather if) it finishes, decide whether it was a *valid* execution or not—depending on whether the automaton ended up in an accept or reject state. Also, we want the requirements to always be satisfied. Therefore, we want to run the finite automaton A_ϕ induced from the requirements along side the service, while taking transitions corresponding to passing labels. Since we do not want to violate the safety requirements at any point, we require that we never reach a non-acceptance state in the automaton A_ϕ.

Now, all we need to do is make sure that:

- the service asks the control automaton for permission each time it desires to pass a label; and

- the control automaton only grants permission to pass labels, whose corresponding transition does not take us to a reject state.

In this way, the requirement, ϕ cannot be violated at any point in time.

1.3.2 Prefix Closing ϕ

We prefix close all formulas, because it does not make sense to restrict a service by a formula that is not prefix closed. Hence, the formula $\exists t : L(t)$, has no effect whatsoever, since it will be prefix closed and thus valid on all strings.

The prefixing of formulas is easily done on automata level. A prefix closed minimalized automaton has at most one reject state. Taking advantage of this fact, we only need to define the transitions that take us to acceptance states and in this way assume that all the rest will take us to the reject state. Behold the automaton $M_{A,B} = (\{q, q', q''\}, \{A, B\}, q, \{(q, A, q'), (q, B, q), (q', A, q''), (q', B, q), (q'', A, q''), (q'', B, q'')\}, \{q, q'\})$ depicted in Figure 1.10.

Using an implicit reject state it can be shortened to the automaton $(\{q, q'\}, \{A, B\}, q, \{(q, A, q'), (q, B, q), (q', B, q)\}, \{q, q'\})$ illustrated in Figure 1.11. Since all the states shown are acceptance states, we shall omit the double circles.

Henceforth, we shall depict automata with implicit reject state.

1.3.3 SyCoLogic Syntax

Here is the syntax for the SyCoLogic ($\underline{\text{Sy}}$nthesis of $\underline{\text{Co}}$ntroller) language.

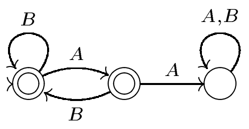


Figure 1.10: The MDFA $M_{A,B}$ fully represented

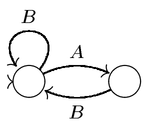


Figure 1.11: The MDFA $M_{A,B}$ with implicit reject-state

constraint	::=	**constraint** {*constraintbody_list* }
constraintbody_list	::=	$constraintbody^+$
constraintbody	::=	*label*
	$\mid$	*trigger*
	$\mid$	*formula* ;
label	::=	**label** *id_list* ;
	$\mid$	**label priority** *id_list* ;
trigger	::=	**trigger** *id* **when** *trigexp* == *trigexp* ;
trigexp	::=	*trigexp* + *trigexp*
	$\mid$	*trigexp* − *trigexp*
	$\mid$	+ *trigexp*
	$\mid$	− *trigexp*
	$\mid$	*intconst*
	$\mid$	# *id*
formula	::=	**true**
	$\mid$	**false**
	$\mid$	*id* ∘ *id*
	$\mid$	*id* (*id*)
	$\mid$	(*formula*)
	$\mid$	! *formula*
	$\mid$	*formula* • *formula*
	$\mid$	**all** *id* : *formula*
	$\mid$	**is** *id* : *formula*
	$\mid$	**restrict** *formula* **by** *id*

where $\circ \in \{==, !=, <, >, <=, >=\}$ and $\bullet \in \{\&\&, ||, =>, <=>\}$

As can be seen, constraints are comprised of labels, formulas, and triggers. The first is used to declare the labels to be used in the previously explained **wait** statements. The formulas are basically standard monadic second-order logic but with a single nonstandard construct, **restrict-by**, that will be explained below. The triggers present a way of taking us beyond the boundaries of regularity otherwise imposed by the logic of the formulas, as will be explained later.

It is possible to specify several formulas in one constraint, however it has the exact same effect as if they were specified individually. That is, the requirements "**constraint** { ϕ; ϕ'; }" and "**constraint** { ϕ } **constraint** { ϕ'; }" mean the exact same thing, and will both ultimately be taken as: "**prefix**(ϕ) $\wedge$ **prefix**(ϕ')".

Actually, we only use monadic first-order logic simply because we have not found any examples that needs the second-order sets. However, there is absolutely no reason why we cannot incorporate second-order constructs into SyCoLogic.

There are a couple of reasons why labels are explicitly declared and not just inferred. Due to the existence of the priority concept, we need to specify which labels have priority (and which not). Also, we have trigger-labels that definitely need declaration. Finally, the rest of the <**bigwig**>language is a **C**-like declarative language, so it thus respects the overall look-and-feel.

One may be surprised to find that we have not included predicates. The reason being, as previously hinted, that we have a full scale macro language built on top of the syntax that will serve these needs.

When several formulas are specified, they must of course all be satisfied. Thus,

Example—a formula restricting a label

Consider the following formula, specifying that there must be a B between any two A's—yielding mutually exclusive access between A and B (see Figure 1.12).

$$\forall t, t'' : t < t'' \wedge A(t) \wedge A(t'') \Rightarrow (\exists t' : t < t' < t'' \wedge B(t'))$$

In SyCoLogic, the formula will be specified as in Figure 1.12.

```
all t0:
    all t2:
        t0<t2 && A(t0) && A(t2) =>
            (is t1:
                t0<t1<t2 && B(t1))
```

Figure 1.12: A formula—mutex(A,B)

1.4 SyCoLogic Semantics

All constructs have straightforward mathematical semantics, save one, **restrict-by**. The semantics is defined relative to an environment $E : Id \hookrightarrow \mathbb{N}$ (or $Id \rightarrow \mathbb{N}_\perp$), that holds the values of all free time-variables (a.k.a. position-variables) that will specify positions in the sequence of labels (or string) $\omega \in \Sigma^*$. The string ω will also be perceived as a function $\omega : \mathbb{N} \hookrightarrow \Sigma$ mapping each position to the label at that position in the string.

Also, we shall define the relation $\models \subseteq ((Id \hookrightarrow \mathbb{N}) \times (\mathbb{N} \hookrightarrow \Sigma) \times formula)$. $(E, \omega, \phi) \in \models$ will be abbreviated as $E \models_\omega [\![\phi]\!]$. Also, we shall write $\omega \models \phi$ for $\perp_E \models_\omega [\![\phi]\!]$, where $\perp_E$ is the empty environment.

Before we proceed with the definition of $\models$, we need to present a notational convention. If $f : D \rightarrow E$ is a function, and d_0 and e_0 members of D and E, respectively, then we define $f[d_0 \mapsto e_0]$ to be:

$$f[d_0 \mapsto e_0](d) := \begin{cases} e_0 & \text{, if } d = d_0 \\ f(d) & \text{, otherwise} \end{cases}$$

We are now ready to specify the relation $\models$. It will be defined by structural induction in terms of the syntax of ϕ.

$$\begin{array}{lll}
E \models_\omega [\![\mathbf{true}]\!] & \textbf{iff} & true \\
E \models_\omega [\![\mathbf{false}]\!] & \textbf{iff} & false \\
E \models_\omega [\![t == t']\!] & \textbf{iff} & E(t) = E(t') \\
E \models_\omega [\![t\ != t']\!] & \textbf{iff} & E(t) \neq E(t') \\
E \models_\omega [\![t >= t']\!] & \textbf{iff} & E(t) \geq E(t') \\
E \models_\omega [\![t <= t']\!] & \textbf{iff} & E(t) \leq E(t') \\
E \models_\omega [\![t > t']\!] & \textbf{iff} & E(t) > E(t') \\
E \models_\omega [\![t < t']\!] & \textbf{iff} & E(t) < E(t') \\
E \models_\omega [\![L(t)]\!] & \textbf{iff} & \omega[E(t)] = L \\
E \models_\omega [\![(\phi)]\!] & \textbf{iff} & E \models_\omega [\![\phi]\!] \\
E \models_\omega [\![\ !\phi]\!] & \textbf{iff} & \neg E \models_\omega [\![\phi]\!] \\
E \models_\omega [\![\phi\ \&\&\ \phi']\!] & \textbf{iff} & E \models_\omega [\![\phi]\!] \wedge E \models_\omega [\![\phi']\!] \\
E \models_\omega [\![\phi\ ||\ \phi']\!] & \textbf{iff} & E \models_\omega [\![\phi]\!] \vee E \models_\omega [\![\phi']\!] \\
E \models_\omega [\![\phi => \phi']\!] & \textbf{iff} & E \models_\omega [\![\phi]\!] \Rightarrow E \models_\omega [\![\phi']\!] \\
E \models_\omega [\![\phi <=> \phi']\!] & \textbf{iff} & E \models_\omega [\![\phi]\!] \Leftrightarrow E \models_\omega [\![\phi']\!] \\
E \models_\omega [\![\mathbf{all}\ t : \phi]\!] & \textbf{iff} & \forall n \in \mathbb{N} : n < |\omega| \Rightarrow E[t \mapsto n] \models_\omega [\![\phi]\!] \\
E \models_\omega [\![\mathbf{is}\ t : \phi]\!] & \textbf{iff} & \exists n \in \mathbb{N} : n < |\omega| \wedge E[t \mapsto n] \models_\omega [\![\phi]\!]
\end{array}$$

1.4.1 Restrict-by

In this section, we shall take $SyCoLogic^+$ to mean the language specified by the syntactic category $formula$ above and $SyCoLogic$ to be the same but without the $\mathbf{restrict-by}$ construct.

The formula "**restrict** ϕ **by** t" means that the formula ϕ is restricted in such a way that time-variables introduced through quantifiers are allowed only to specify positions in the string before the time (or position) t.

Instead of directly specifying the semantics of the **restrict − by** construct, we shall exhibit a translation function that can be applied to a formula in order to desugar this construct. We define $[\![\cdot]\!] : \mathit{SyCoLogic}^{+} \rightarrow \mathit{SyCoLogic}$ as follows:

$$
\begin{array}{lcl}
[\![\textbf{true}]\!] & = & \textbf{true} \\
[\![\textbf{false}]\!] & = & \textbf{false} \\
[\![I \circ I']\!] & = & I \circ I' \\
[\![I(I')]\!] & = & I(I') \\
[\![(\ \phi\ )]\!] & = & (\ [\![\phi]\!]\ ) \\
[\![!\ \phi]\!] & = & !\ [\![\phi]\!] \\
[\![\phi \bullet \phi']\!] & = & [\![\phi]\!] \bullet [\![\phi']\!] \\
[\![\textbf{all}\ t:\ \phi]\!] & = & \textbf{all}\ t : [\![\phi]\!] \\
[\![\textbf{is}\ t:\ \phi]\!] & = & \textbf{is}\ t : [\![\phi]\!] \\
[\![\textbf{restrict true by}\ \tau]\!] & = & \textbf{true} \\
[\![\textbf{restrict false by}\ \tau]\!] & = & \textbf{false} \\
[\![\textbf{restrict}\ I \circ I'\ \textbf{by}\ \tau]\!] & = & I \circ I' \\
[\![\textbf{restrict}\ I(I')\ \textbf{by}\ \tau]\!] & = & I(I') \\
[\![\textbf{restrict}\ !\ \phi\ \textbf{by}\ \tau]\!] & = & !\ [\![\textbf{restrict}\ \phi\ \textbf{by}\ \tau]\!] \\
[\![\textbf{restrict}\ (\ \phi\ )\ \textbf{by}\ \tau]\!] & = & (\ [\![\textbf{restrict}\ \phi\ \textbf{by}\ \tau]\!]\ ) \\
[\![\textbf{restrict}\ \phi \bullet \phi'\ \textbf{by}\ \tau]\!] & = & [\![\textbf{restrict}\ \phi\ \textbf{by}\ \tau]\!] \bullet [\![\textbf{restrict}\ \phi'\ \textbf{by}\ \tau]\!] \\
[\![\textbf{restrict all}\ t:\ \phi\ \textbf{by}\ \tau]\!] & = & \textbf{all}\ t:\ t < \tau\ \ {=}{>}\ \ (\ [\![\textbf{restrict}\ \phi\ \textbf{by}\ \tau]\!]\ ) \\
[\![\textbf{restrict is}\ t:\ \phi\ \textbf{by}\ \tau]\!] & = & \textbf{is}\ t:\ t < \tau\ \&\&\ (\ [\![\textbf{restrict}\ \phi\ \textbf{by}\ \tau]\!]\ ) \\
[\![\textbf{restrict restrict}\ \phi'\ \textbf{by}\ \tau'\ \textbf{by}\ \tau]\!] & = & [\![\textbf{restrict}\ [\![\textbf{restrict}\ \phi'\ \textbf{by}\ \tau']\!]\ \textbf{by}\ \tau]\!]
\end{array}
$$

The reason why we have included this construct in the language will be clarified below.

1.4.2 Forbid/Allow

Often, a service programmer needs to prohibit a label, L, whenever a formula, ϕ, is true on the sequence of labels, ω seen thus far (that is, $\omega \models \phi$). To this end, we want to provide an intuitive syntax like:

$$\textbf{forbid}\ \mathsf{L}\ \textbf{when}\ \phi$$

Exactly this can easily be achieved through the **restrict-by** construct. We shall define the **forbid-when** construct in terms of another useful construct, **allow-when** (with the obvious semantics), that is in turn defined in terms of the **restrict-by** construct.

$$
\begin{array}{lcl}
\textbf{allow}\ \mathsf{L}\ \textbf{when}\ \phi & := & \textbf{all}\ \mathsf{now} : \mathsf{L}(\mathsf{now}) {=}{>}\ \textbf{restrict}\ \phi\ \textbf{by}\ \mathsf{now} \\
\textbf{forbid}\ \mathsf{L}\ \textbf{when}\ \phi & := & \textbf{allow}\ \mathsf{L}\ \textbf{when}\ (!\phi)
\end{array}
$$

Notice that all formulas obtained from using **allow** and **forbid** at the outermost level are automatically prefix closed.

The reason why we have not included these in the language, is because they can be defined in terms of the rest of the language using our generalized macro concept (see Section 3.2.1).

Of course, we could also have chosen these two over **restrict-by**. This choice is partly motivated by the fact that **restrict-by** has an obvious self-contained semantics and partly by our overall design approach. By this we mean, that we only include the basic building blocks (or primitives) in our core language and then build everything else up in terms of these.

Using, the **forbid-when** construction, one can easily define *mutex* (see Figure 1.13) in a very intuitive manner:

```
forbid A when
    is t: A(t) &&
        (all tt: t<tt => !B(tt));
```

Figure 1.13: mutex(A,B) in terms of **forbid-when**.

As can be seen below, the definition here corresponds exactly to the one previously given (see Figure 1.12).

$$\begin{array}{ll}
 & \textbf{forbid}\ A\ \textbf{when}\ \exists t : A(t) \wedge (\forall t' : t < t' => \neg B(t')) \\
\equiv & \forall t'' : A(t'') \Rightarrow \neg[\textbf{restrict}\ (\exists t : A(t) \wedge (\forall t' : t < t' \Rightarrow \neg B(t')))\ \textbf{by}\ t''] \\
\equiv & \forall t'' : A(t'') \Rightarrow \neg[\exists t : t < t'' \wedge A(t) \wedge (\forall t' : t < t'' \Rightarrow (t < t' \Rightarrow \neg B(t')))] \\
\equiv & \forall t'' : A(t'') \Rightarrow [\forall t : \neg(t < t'' \wedge A(t)) \vee (\exists t' : t < t'' \wedge (t < t' \wedge B(t')))] \\
\equiv & \forall t'' : A(t'') \Rightarrow [\forall t : (t < t'' \wedge A(t)) \Rightarrow (\exists t' : t < t'' \wedge (t < t' \wedge B(t')))] \\
\equiv & \forall t, t'' : t < t'' \wedge A(t) \wedge A(t'') \Rightarrow (\exists t' : t < t' < t'' \wedge B(t')) \\
\equiv & mutex(A, B)
\end{array}$$

1.5 Beyond Regularity—Triggers

Let us for a moment consider the reader/writer-problem (bounded by the following constraints):

- At any given time there must be at most one thread writing.
- While there are threads reading there must not be any threads writing.
- While there are threads writing there must not be any threads reading.
- Writers have priority over readers (that is, if a request for permission to write is given, no new readers are allowed).

For reasons of simplicity, we shall ignore the last point in the following. The full-scale problem will be treated in Section 3.2.4.

Unfortunately, this cannot be expressed in **M2L-Str** since we would have to somehow remember the **unbounded** number of readers at any given moment. The positions at which there are no readers in progress are exactly those where the number of enterR and exitR labels occurring before that position are the same. This is non-regular in the same way as the language $\{a^n b^n \mid n \geq 0\}$ and therefore cannot be expressed in **M2L-Str**. We can of course constrain the problem to any fixed maximum number of readers (N) corresponding to $\{a^n b^n \mid N \geq n \geq 0\}$ which is indeed a regular language (it is finite) and hence expressible in **M2L-Str**.

In order to overcome this hurdle, we have invented a notion of **triggers**, that will take us beyond regularity.

1.5.1 Triggers

Triggers are constructs that need to be explicitly declared. The following declaration...

$$\textbf{trigger}\ \ \mathsf{noR}\ \ \textbf{when}\ \ \#\,\mathsf{enterR}\ \ ==\ \ \#\,\mathsf{exitR}\ \ ;$$

...will give us a label (in this case noR) that will "fire" (that is, the edge labeled noR will be taken in the resulting automaton) exactly once each time the equation becomes true. In the above case, noR will fire each time the number of enterR and exitR labels goes from being unequal to equal.

Each trigger T will give rise to a counter γ_T in the control logic. This counter, will increase and decrease with the firing of labels in such a manner that it is zero precisely when the expression specified in the trigger is satisfied. When this happens, the counter label will itself fire.

We shall restrict the triggers in such a way that each label appears at most once in each trigger, otherwise a trigger could pass the zero value without firing. Counters can thus be represented as a four tuple, the first constituent is the name of the trigger label, the second and third constituents are the sets of labels that when firing will cause the value of the counter to increase respectively decrease by one. The last constituent is an integer, namely the initial value of the counter.

$$(\gamma, \gamma_{inc}, \gamma_{dec}, \gamma_{init}) \in \Gamma \times \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathbb{Z} \text{ where } \gamma_{inc} \cap \gamma_{dec} = \emptyset.$$

We shall say that a label is in *positive* or *negative* position in the trigger (**trigger** T **when** $te == te'$) when it is in the set $pos(te) \cup neg(te')$ or $neg(te) \cup pos(te')$, respectively. The functions will be defined below. The set of labels in positive position are those that increase the trigger. Similarly, the negative position labels are those that decrease the counter.

The functions $pos, neg : trigexp \to \mathcal{P}(\Sigma)$ and $init : trigexp \to \mathbb{Z}$ used are defined in the following:

$$\begin{array}{ll}
pos(te+te') = pos(te) \cup pos(te') & neg(te+te') = neg(te) \cup neg(te') \\
pos(te-te') = pos(te) \cup neg(te') & neg(te-te') = neg(te) \cup pos(te') \\
pos(+te) = pos(te) & neg(+te) = neg(te) \\
pos(-te) = neg(te) & neg(-te) = pos(te) \\
pos(\#L) = \{L\} & neg(\#L) = \emptyset \\
pos(n) = \emptyset & neg(n) = \emptyset
\end{array}$$

$$\begin{array}{l}
init(te+te) = init(te) + init(te') \\
init(te-te) = init(te) - init(te') \\
init(+te) = init(te') \\
init(-te) = -init(te') \\
init(\#L) = 0 \\
init(n) = n
\end{array}$$

The four tuple we shall use given a trigger T (**trigger** T **when** $te == te'$) is:

$$\gamma_T = (\mathsf{T}, pos(te) \cup neg(te'), neg(te) \cup pos(te'), init(te) - init(te'))$$

Since $pos(te) = neg(-te)$, we get that:

$$\gamma_T = (\mathsf{T}, pos(te-te'), neg(te-te'), init(te-te'))$$

This counter has the property that the trigger is zero precisely when is trigger-expression is satisfied (proof omitted).

The trigger in the above example can thus be specified by the four-tuple: $\gamma_{\mathsf{noR}} = (\mathsf{noR}, \{\mathsf{enterR}\}, \{\mathsf{exitR}\}, 0)$.

We shall use a relation $\vdash \subseteq trigexp \times (\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathbb{Z})$ in order to determine whether we have a valid trigger or not. We shall write $(te, (\gamma_{inc}, \gamma_{dec}, \gamma_{init})) \in \vdash$ as $\vdash te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init})$.

A trigger

trigger T **when** $te == te'$

is said to be *valid* if and only if

$$\vdash te - te' : \gamma$$

The relation $\vdash$ is defined by structural induction on the syntax of *trigexp*.

$$\frac{\vdash te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init}) \qquad \vdash te : (\gamma'_{inc}, \gamma'_{dec}, \gamma'_{init})}{\vdash te + te' : (\gamma_{inc} \,\dot\cup\, \gamma'_{inc}, \gamma_{dec} \,\dot\cup\, \gamma'_{dec}, \gamma_{init} + \gamma'_{init})} \;, \textbf{if} \begin{array}{c} \gamma_{inc} \cap \gamma'_{inc} = \emptyset \\ \textbf{and} \\ \gamma_{dec} \cap \gamma'_{dec} = \emptyset \end{array}$$

$$\frac{\vdash te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init}) \qquad \vdash te : (\gamma'_{inc}, \gamma'_{dec}, \gamma'_{init})}{\vdash te - te' : (\gamma_{inc} \dot{\cup} \gamma'_{dec}, \gamma_{dec} \dot{\cup} \gamma'_{inc}, \gamma_{init} - \gamma'_{init})} \text{, } \mathbf{if} \begin{array}{c} \gamma_{inc} \cap \gamma'_{dec} = \emptyset \\ \mathbf{and} \\ \gamma_{dec} \cap \gamma'_{inc} = \emptyset \end{array}$$

$$\frac{\vdash te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init})}{\vdash +te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init})}$$

$$\frac{\vdash te : (\gamma_{inc}, \gamma_{dec}, \gamma_{init})}{\vdash -te : (\gamma_{dec}, \gamma_{inc}, -\gamma_{init})}$$

$$\frac{}{\vdash \#L : (\{L\}, \emptyset, 0)}$$

$$\frac{}{\vdash n : (\emptyset, \emptyset, n)}$$

The above can easily be applied to check whether a trigger is valid or not. Actually, we have that (proof omitted):

$$\vdash te - te' : (\gamma_{inc}, \gamma_{dec}, \gamma_{init}) \Leftrightarrow \begin{array}{l} \gamma_{inc} = pos(te - te') \wedge \\ \gamma_{dec} = neg(te - te') \wedge \\ \gamma_{init} = init(te - te') \end{array}$$

A Trigger Example—Reader/Writer

The reader/writer mentioned above can now be easily solved by using a trigger noR to determine when there are readers in progress.

SyCoAnalysis

As can be seen in Figure 1.14, the triggers can be included in formulas in order to restrict labels. Of course, this also gives the programmer the possibility of restricting the triggers themselves. It is not clear what effect a trigger T, restricted by the formula $\forall t : \neg T(t)$, should have when it fires. We list three obvious possibilities here:

- T will simply not “fire”.
- T fires, causing the automaton to enter the reject state.
- Runtime error!

```
constraint {
    label enterR, exitR, enterW, exitW;

    mutex(enterW, exitW);
    trigger noR when #enterR == #exitR;
    allow enterW when (all t: !enterR(t)) ||
       (is t: noR(t) && (all tt: t<tt => !enterR(tt)));
    forbid enterR when (is t: enterW(t)) &&
       (all tt: t<tt => !exitW(tt));
}

session reader() {
    wait enterR;
    /* reading ... */
    wait exitR;
}

session writer() {
    wait enterW;
    /* writing ... */
    wait exitW;
}
```

Figure 1.14: The Reader/Writer Problem (without priorities).

Choosing the first would render the entire trigger concept highly unreliable, as they would simply not fire when they in fact were supposed to, causing them to behave in a rather unexpected manner. After the automaton has entered the reject state as in the second case, no label is ever enabled—driving the system to a grinding halt. For these reasons, we have chosen the third.

Of course, this choice is made in the absence of an analysis guaranteeing that triggers are never restricted. Whether a trigger is disabled or not can easily be seen by inspecting the automaton. It simply amounts to determining whether there are transitions labeled with a trigger label going to the reject state. If this is the case, the compiler should issue an error.

Triggers are in nature *uncontrollable* (as in [3])—meaning that they should occur without the service programmer being able to influence it.

1.5.2 Formalizing Counter Automata

A *counter automaton* is a six-tuple $(Q, q_0, \Sigma, F, C, \rightarrow)$, where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- Σ is a finite set of labels (or input symbols);
- $F \subseteq Q$ is the set of acceptance states (or final states);
- C is a finite set of counters: $\Gamma \times \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathbb{Z}$, where Γ is the set of all counter names; *and*
- $\rightarrow \subseteq (Q \times (\Gamma \rightarrow \mathbb{Z}) \times \mathcal{P}(\Gamma) \times (\Sigma \cup \Gamma) \times Q \times (\Gamma \rightarrow \mathbb{Z}) \times \mathcal{P}(\Gamma))$ is a transition relation.

For ordinary finite-state automata, the concept of state is entirely modeled by the elements of Q. However, we need to also incorporate the values of all counters into the notion of state. This could be described as an element in $\Gamma \rightarrow \mathbb{Z}$, that is, a function that for each counter gives the value. Finally, we also need to know something about which of the counters that are to fire actually have, which calls for a third element in our notion of state.

The initial state is $(q_0, S_0, \emptyset) \in (Q \times (\Gamma \rightarrow \mathbb{Z}), \mathcal{P}(\Gamma))$, where:

$$\forall(\gamma, \gamma_{inc}, \gamma_{dec}, \gamma_{init}) \in C: \ S_0(\gamma) = \gamma_{init}$$

No counter may have a label both in its increase and decrease set, that is:

$$\forall(\gamma, \gamma_{inc}, \gamma_{dec}, \gamma_{init}) \in C: \ \gamma_{inc} \cap \gamma_{dec} = \emptyset$$

As usual, we shall abbreviate the transition relation $(q, S, Z, \sigma, q', S', Z')$ as $(q, S, Z) \xrightarrow{\sigma} (q', S', Z')$. Also, we shall constrain the relation so that:

$$(q, S, Z) \xrightarrow{\sigma} (q', S', Z') \text{ , } \textbf{where } \sigma \in \Sigma \textbf{ implies}$$

$$\begin{aligned} & Z = \emptyset \textbf{ and} \\ & \forall(\gamma, \gamma_{inc}, \gamma_{dec}, \gamma_{init}) \in C : S'(\gamma) = \begin{cases} S(\gamma) + 1 & \text{, } \textbf{if } \sigma \in \gamma_{inc} \\ S(\gamma) - 1 & \text{, } \textbf{if } \sigma \in \gamma_{dec} \\ S(\gamma) & \text{, } \textbf{otherwise} \end{cases} \textbf{ and} \\ & Z' = \{\gamma \mid (\gamma, \gamma_{inc}, \gamma_{dec}, \gamma_{init}) \in C : S'(\gamma) = 0 \wedge S(\gamma) \neq S'(\gamma)\} \end{aligned}$$

The requirement that Z be empty will ensure that no label transitions are *taken* when there are counters that are to fire. The next requirement will make sure that the appropriate counters are updated according to the definition of the counter. The last requirement will make sure that Z' will contain the counters that become zero.

$$(q, S, Z) \xrightarrow{\gamma} (q', S', Z') \text{ , } \textbf{where } \gamma \in \Gamma \textbf{ implies } Z' = Z \setminus \{\gamma\} \textbf{ and } S = S'$$

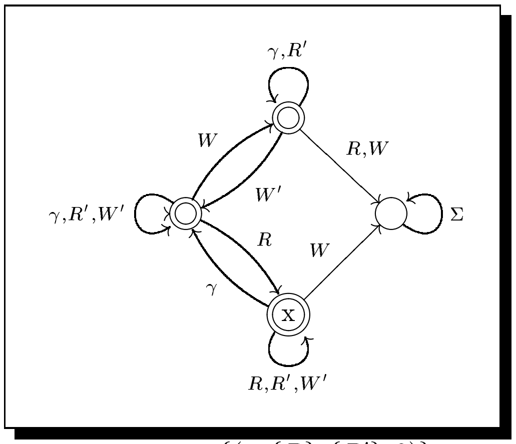
This requirement is to make sure that all counters that are to fire actually will, one by one. Notice, that the sequence in which the counters fire is unspecified, as in our system.

That no counters are restricted can also be formulated as a static requirement on the transition relation:

$$\begin{aligned} & \forall q \in F,\ S \in (\Gamma \rightarrow \mathbb{Z}),\ Z \in \mathcal{P}(\Gamma) :\ \forall \gamma \in Z : \\ & \qquad \exists q' \in F,\ S' \in (\Gamma \rightarrow \mathbb{Z}),\ Z \in \mathcal{P}(\Gamma) :\ (q, S, Z) \xrightarrow{\gamma} (q', S', Z') \end{aligned}$$

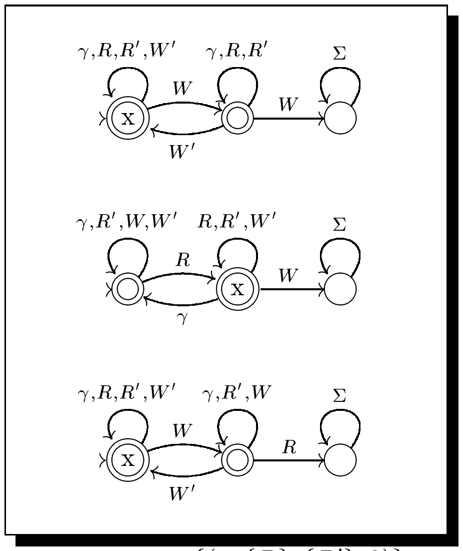
1.5.3 A Counter Example

As an example of a counter automaton, we have taken the R/W example (without writer priority) from Figure 1.14. The resulting control logic, that comprises an automaton and a set of counters, can be seen in the Figures 1.15 and 1.16. Note that we have in both figures shortened the label names from noR, enterR, exitR, enterW, and exitW, to γ, R, R' ,W, and W', respectively. Figure 1.15 shows the "big" automaton obtained as the conjunction of all (three) (individually prefixed) formulas. In order to avoid the state explosion problem, we actually use FDFAs (factorized deterministic finite-state automata) or rather factorized counter automata as our control logic (as proposed in [4]). Each formula will give rise to a factor in the FDFA. Figure 1.16 shows the three counter automata factors, which are equivalent to the product automaton. The advantage of using FDFAs is remarkable. For instance, n independent mutex constraints would yield a product automaton with $2^n + 1$ states, whereas the factorized ones make do with $3n$ states. Both of the control logics display a scenario where there are two readers in progress, as can be seen from the value of the counter and the current states of the automata marked by "x".



counters : $\{(\gamma, \{R\}, \{R'\}, 0)\}$
value$(\gamma) = \boxed{2}$

Figure 1.15: The product counter automaton from R/W



counters : $\{(\gamma, \{R\}, \{R'\}, 0)\}$
value$(\gamma) = \boxed{2}$

Figure 1.16: The factorized counter automata from R/W

1.5.4 Ensure—Another way beyond regularity

We have also considered another concept taking us beyond regularity which we have dubbed anonymous counters (or counter constraints). An anonymous counter must be declared so:

$$\textbf{ensure}\ \textit{trigexp} \circ \textit{trigexp}\ ;$$

Unlike the triggers above, the anonymous counters are not integrated into the formulas. Instead they silently ensure that the associated expression is never violated.

However, the expressibility associated with these counters is strictly contained in that of the triggers.

Given such an anonymous counter, (**ensure** $te \circ te'$), one can construct a corresponding trigger, (**trigger** T **when** $te == te'$) and some formula that will simulate the counter precisely (that is, have the exact same behavior). This will be briefly sketched in the following.

The idea is (when $\circ$ is '>=') to forbid those labels that may want to decrease the corresponding counter whenever the counter has fired signaling that it has reached zero and none of the labels that increment has fired (causing the counter to have a positive value) since. If the counter is initially zero, one also has to forbid the decreasing labels when there has never been any that causes it to increase.

See Figure 1.17 for an example of anonymous counter simulation.

```
ensure #A - #B >= #C - #D ;

≡

trigger  T  when  #A - #B == #C - #D ;

forbid B when (all t: !(A(t) || D(t))) ||
    (is t: T(t) && (all t': t<t' => !(A(t) || D(t))));

forbid C when (all t: !(A(t) || D(t))) ||
    (is t: T(t) && (all t': t<t' => !(A(t) || D(t))));
```

Figure 1.17: An anonymous counter simulated by a trigger and two formulas.

The rest of the cases can be handled in terms of ">=" (see Figure 1.18)—provided we ignore those **ensure** that have initially violated expressions.

Because the original triggers were integrated in the formulas unlike the anonymous counters, there are certain requirements that are easily expressible with the triggers and impossible with **ensure**. An example of this could be if one wanted to forbid a label after the second time a trigger fired.

```
ensure te == te';   :=   ensure te >= te'; ensure te' >= te;

ensure te != te';   :=   ensure te >= te'+1; ensure te' >= te + 1;

ensure te <= te';   :=   ensure te' >= te;

ensure te > te';    :=   ensure te >= te' + 1;

ensure te < te';    :=   ensure te' >= te + 1;
```

Figure 1.18: Ensure te $\circ$ te', for $\circ \in \{==, != , <=, >, <\}$

Because of this, we have chosen the **trigger** concept over **ensure**.

1.6 Negated Labels

Experiments have shown that it is very helpful to be able to react appropriately when issuing a wait on a label that is not enabled. An example of this could be if the service programmer wanted to use a resource (that required exclusive access) only if it happened to be available. Meaning that he does not want to wait for it to become available if it was not readily so.

One approach would be (as shown in Figure 1.19) to introduce a **default**-branch in the *waitbranch* with the obvious semantics.

```
forbid L when φ;

wait {
    case L:
        S
    default:
        S'
}
```

Figure 1.19: Negated Labels - "default"

Of course, this can be achieved through explicit programming. It amounts to, for each restriction of the given label, to explicitly add complementary constraints in such a way that this new label would be restricted when and only when the original label was not (see Figure 1.20).

```
forbid L when $\phi$;
allow $L_{neg}$ when $\phi$;

wait {
    case L:
        S
    case $L_{neg}$:
        S'
}
```

Figure 1.20: Negated Labels - up to the programmer

As can be seen, this is a rather tedious approach especially if there are many such restricting formulas. Of course, all this could also be handled by the compiler.

Yet another approach is to introduce the notion of a *negated label*, written “!L”. The idea is that for all labels, L, there is a negated version, “!L”, that is enabled precisely when L is not. When such a label is “taken” it has no effect on the automaton (that is, the automaton will not change state).

The appealing factors concerning this solution compared with the first is that it can be implemented in the controller preserving the scheduling model and at almost no cost whatsoever. The problem with the first is that it is not clear when the default branch should be scheduled. If scheduled by adding a “default” queue to the token ring, we would have to check all other branches in the wait every time the token reaches this queue in order to make sure that none of them were enabled. Also, a **wait** statement, waiting for two labels to become disabled cannot be readily expressed with the default concept. To this end, we have included this negated label mechanism.

```
forbid L when $\phi$;

wait {
    case L:
        S
    case !L:
        S'
}
```

Figure 1.21: Negated Labels - “!L”

1.7 Priority Labels

As a feature we have introduced one level of *priority* on labels. This will in the controller give rise to another level of queues with their own private token ring (as illustrated in Figure 1.22). The controller will every time the automaton changes state (that is, when a label is taken) do one full circle of the priority queues in order to guarantee that any such are taken before it resumes circling the original token ring.

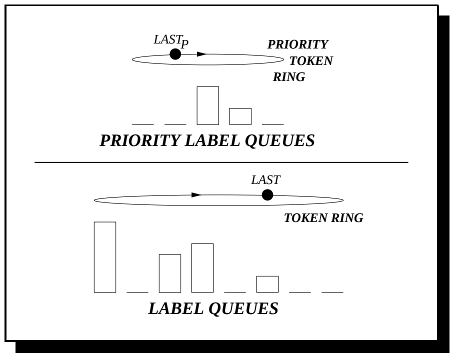


Figure 1.22: Two token rings to implement priority.

We emphasize that the use of such labels is highly dangerous since it may cause starvation of the labels that do not have priority.

1.8 Events

In this section we shall briefly consider the possibility of dealing with external events in our system. These are basically labels but with the exception that they occur in an uncontrollable fashion, much like the trigger labels, but based on external factors, such as when the harddisk or the memory fills up, or when the service coredumps, etc. The idea is that a service then could have special sessions (event handlers) waiting on these labels, ready to react appropriately (see Figure 1.23). However, this feature has not been implemented, nor fully explored.

1.9 Dynamic Deadlock Detection (D^3)

It would be useful to the service programmers if the system could somehow inform a waiting session when it is blocked by a dead **wait** statement, thus permitting the session to react appropriately.

Unfortunately this is not possible to compute in full generality. This can be proved by a reduction argument showing that if this was somehow possible, we would be able to solve the halting problem.

```
session S() {
    wait HardDiskFull;
    /* remove something... */
}
```

Figure 1.23: HardDiskFull event handling

1.9.1 The Halting Reduction

Below we have sketched a simple language with three syntactic categories; arithmetic expressions, boolean expressions, and commands:

$$\begin{array}{lcl} a & ::= & n \mid x \mid a_1 + a_2 \mid -a_1 \\ b & ::= & \neg b_1 \mid b_1 \wedge b_2 \mid a_1 \geq 0 \\ c & ::= & x := a \mid \textbf{if } b \textbf{ then } c \mid c_1 \ ; \ c_2 \mid \textbf{while } b \textbf{ do } c \end{array}$$

The language is clearly Turing equivalent, rendering the halting problem undecidable (with respect to this language). The goal is now to show that if we could decide precisely when a label is dead, then we could also solve the undecidable halting problem. Logically, this corresponds to the following deduction rule:

$$\frac{\text{Halting } \textit{undecidable} \text{ , } \textit{solve } D^3 \Rightarrow \textit{solve} \text{ Halting}}{D^3 \ \textit{undecidable}}$$

Formally, for every program H (belonging to the command category) in the above language, we construct an automaton such that a certain label, say π, is *reachable* (or *live*) if and only if the program H terminates:

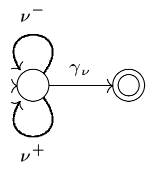
$$H\downarrow \iff \pi \textit{ live in :} \qquad \begin{tikzcd} {} \arrow[r] & {[\![H]\!]} \arrow[r, squiggly] & \bigcirc \arrow[r, "\pi"] & \circledcirc \end{tikzcd}$$

Here, $[\![\cdot]\!]$ is the translation function that given a program produces a corresponding automaton (in which π is assumed not to occur), and the state immediately following $[\![H]\!]$ with a squiggled arrow signifies the acceptance state in the automaton obtained from H. The term acceptance state is perhaps a bit misleading, as it is only used to specify how automata are composed from their immediate constituents, using structural induction. The acceptance state in the automaton $[\![H]\!]$ is the non-acceptance state left to the acceptance state in the diagram above. The squiggled arrow is thus used as a notation for combining automata. Incidently, the problem above equivalent to that of determining whether the language associated with the above automaton recognizes anything.

A key idea in the translation $[\![\cdot]\!]$ from programs to automata is that variables and program state are modeled by the counter concept in our automata. Each variable will give rise to a counter holding the value of the variable.

Each variable x in the program will give rise to a counter $(\gamma_x, \{x^+\}, \{x^-\}, 0)$, which we shall refer to by the same name (x). For each such counter, we have labels to increase (x^+) and decrease (x^-) it along with the counter label itself (γ_x) used when the counter fires (that is, when the counter becomes zero).

Furthermore, to simulate evaluation of arithmetic and boolean expressions, each point (in AST sense) will also give rise to a counter. We shall implicitly label all points in the program. A program $a_1 + a_2$ will be labeled as, $({a_1}^{\ell_1} + {a_2}^{\ell_2})^{\ell}$. Each such label, ℓ, will give rise to a counter $(\gamma_\nu, \{\nu^+\}, \{\nu^-\}, 0)$. The simulation of the evaluation of the arithmetics expression will assume that its constituents have computed ${a_1}^{\ell_1}$ and ${a_2}^{\ell_2}$, the values of which reside in the counters $(\gamma_{\nu_1}, \{\nu_1^+\}, \{\nu_1^-\}, 0)$ and $(\gamma_{\nu_2}, \{\nu_2^+\}, \{\nu_2^-\}, 0)$. Since we have **while** loops and our automaton requires a finite number of counters, we need a mechanism to "reset" these counters (induced by the points in the program) in order to be able to reuse them. We propose the following:



reset ν

The only way through the automaton is when ν becomes zero so that we take the γ_ν transition. In this way we can stick this automaton in whenever we need to reset a counter.

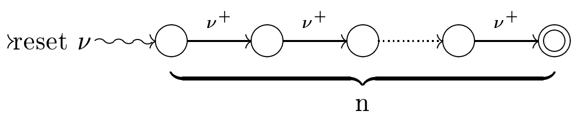
We now proceed by structural induction, depicting the translation for each of the three syntactic categories separately.

Arithmetic Expressions

In translating arithmetic expressions to automata, we maintain the following invariant:

There is at least one way to the acceptance state, and for all such ways, ν will obtain the value of the arithmetic expression. Furthermore, the state is preserved.

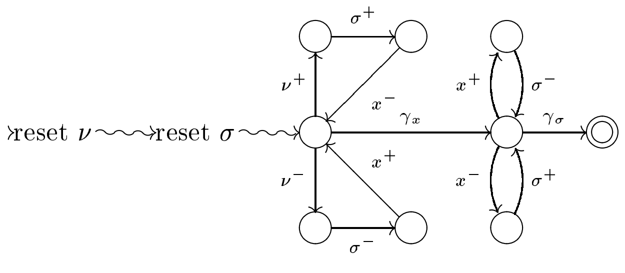
The first translation rule (for constant n) is simple:



$[\![n]\!]$

We just increase/decrease the counter ν n times depending on the sign of n.

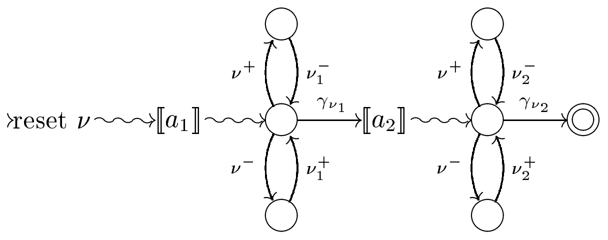
The next rule for variables x is more demanding:



$[\![x]\!]$

All this is needed to ensure that the value in the counter x is unaffected by us inspecting its value. A counter, σ, (associated with this automaton only) remembers the value of x so that we are able to reconstruct the value before finishing. The only way along the long arrow is γ_x, in which case the original value of the counter x would have been reconstructed in the counters ν and σ. The only way to the acceptance state is when the counter σ fires, in which case the value of the counter x would have been reconstructed. Basically, we just copy the value of the counter x to ν, using a "temporary" counter σ.

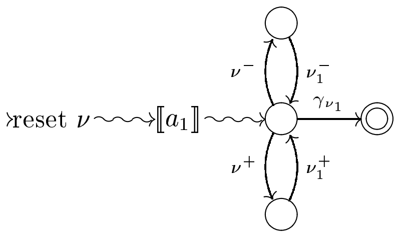
The third rule is addition $(a_1 + a_2)$. Since this is one of the inductive cases, we assume that $[\![a_1]\!]$ and $[\![a_2]\!]$ give us the automata corresponding to a_1 and a_2 with results in ν_1 and ν_2, respectively. We now need to combine these two automata in a sensible way, reflecting the addition:



$[\![a_1 + a_2]\!]$

As always, we begin by resetting ν after which the automaton corresponding to a_1 is "run" yielding a value ν_1. This value is transfered to ν. Finally, the second automaton is run and the result is transfered to ν. When we reach the accept state, ν will clearly hold the sum of ν_1 and ν_2 as desired.

The final arithmetic case, $-a$, is straightforward and presents no new ideas:



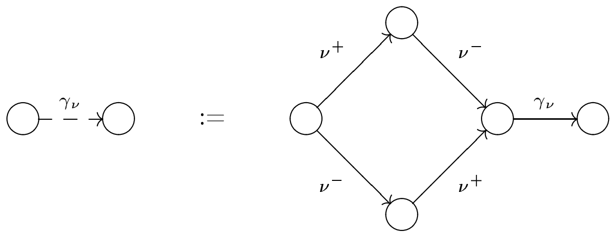
$[\![-a_1]\!]$

Boolean Expressions

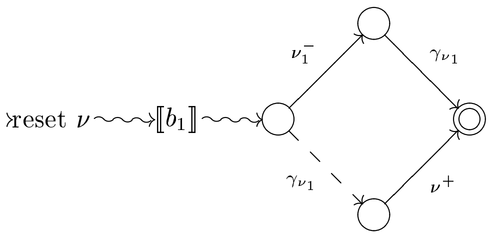
As with the arithmetic expressions we have an invariant to maintain, this one being:

There is at least one way to the acceptance state, and for all such ways, ν will obtain the value zero if the boolean expression evaluates to false, and one if the boolean expression evaluates to true. Furthermore, the state is preserved.

We shall introduce a notational convention, that will shorten some of the automata:



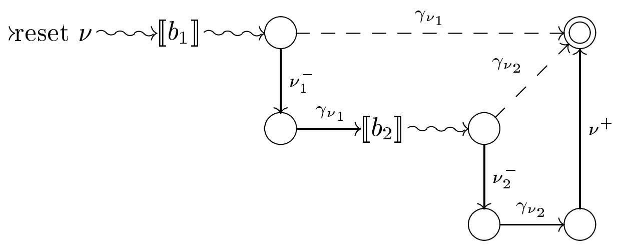
The rule for negation ($\neg b_1$) will give rise to the following automaton during the translation process:



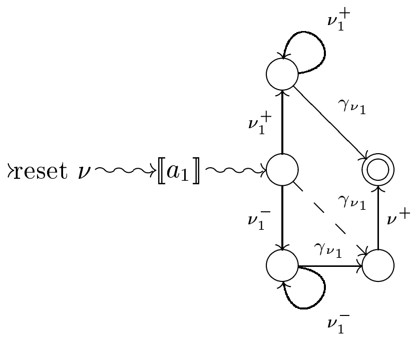
$[\![\neg b_1]\!]$

Initially ν is reset and the automaton corresponding to b_1 is run, leaving the result in ν_1. The invariant now guarantees that ν_1 is either zero or one. If it is zero, we need to produce a one in ν since we are negating the boolean expression, and vice versa. If the value of the counter ν_1 was zero, the label γ_{ν_1} will be taken, as the counter ν_1 will fire after which we increase the value of the counter ν, producing a one. This corresponds to the lower route in the diagram above. If on the other hand, it was one, we can go to the acceptance state taking the upper route (that will decrease it and let it fire).

The final two boolean rules, $b_1 \wedge b_2$ and $a_1 \geq 0$, are presented below:



$[\![b_1 \wedge b_2]\!]$



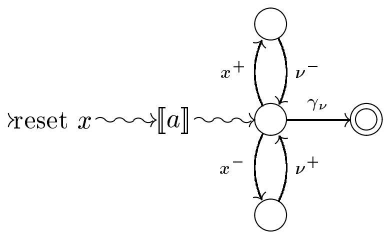
$[\![a_1 \geq 0]\!]$

Commands

The invariant for commands reads:

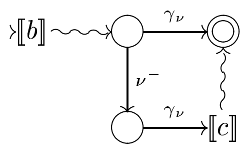
> There is a way to the acceptance state if and only if the command (program) halts. Furthermore, the state of the automaton corresponds to the state of the program.

Regarding assignment ($x := a$), we reset x, run the automaton induced by a and transfer the result to the counter x, as done in the arithmetic case x:



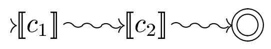
$[\![x := a]\!]$

The translations for the three remaining commands are portrayed below:

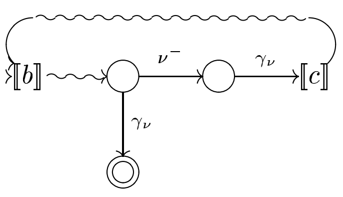


$[\![$**if** b **then** $c]\!]$

Initially, we run the automaton resulting from the translation of the boolean expression b. If this expression evaluates to **false**, the value of the counter ν_1 will be zero and the counter will fire, causing us to take the transition labeled γ_{ν_1}, ignoring the automaton $[\![c]\!]$. If on the other hand, the expression was **true** (the value of the counter was one), we can take the downward route. This will decrease the value of the counter by one, after which it will fire and takes us to the automaton $[\![c]\!]$, that is "executed". After execution, we end up in the acceptance state displayed in the diagram.



$[\![c_1 \ ; \ c_2]\!]$



$[\![$**while** b **do** $c]\!]$

Conclusion

Resulting from the invariant for commands we have that for any program H (belonging to the syntactic category c):

$$H\downarrow \iff \pi\ live\ in: \qquad \begin{tikzcd} \rightarrowtail [\![H]\!] \arrow[r, squiggly] & \bigcirc \arrow[r, "\pi"] & \circledcirc \end{tikzcd}$$

Therefore, if we could determine precisely when a given label is dead, we could solve the halting problem. This proves that the dynamic deadlock detection is undecidable.

1.9.2 An Approximate Solution

As shown above, dynamic deadlock detection is undecidable. This does however **not** mean that we cannot determine liveness at all. It simply states that in general the problem is undecidable. Fortunately there are many cases, even interesting ones, for which we actually **can** determine liveness in the above sense.

If we did not have any counters, the deadlock detection would be trivial. One could simply for each state color the graph induced by the automaton and take the union of all labels occurring on the edges. By taking the complement, one obtains the deadsets (that is, the set of labels that can never occur). This calculation would be precise because if a label is unreachable from the current state (that is, all instances of this label lead to the reject state), then the label is dead.

The problem with the counters is that a counter may itself be dead (that is, it can never reach zero—so that γ_ν never occurs). This might of course render a part of the automaton inaccessible which will kill off the labels that only occurred there. Again, these newly dead labels might kill off new counters and so forth...

If we were able to precisely say when a counter is dead, we could compute liveness precisely using the above scheme, disregarding all edges concerning the counter. In other words this is the heart of our griefs—and incidently the problem that is undecidable.

However, we can exploit the fact that if a counter at a given point is positive and all the negative contribution labels associated with it are all dead, then clearly the counter is dead. Similarly with the signs reversed.

We introduce the function $D : Q \to \mathcal{P}(\Sigma)$, that for each state, q, gives the set of labels that are dead at the state q.

$$(\gamma > 0 \land neg(\gamma) \subseteq D(q)) \lor (\gamma < 0 \land pos(\gamma) \subseteq D(q)) \Rightarrow \gamma\ dead\ at\ q.$$

Here, $neg(\gamma)$ and $pos(\gamma)$ give the sets of labels that decrease and increase the value of the counter γ. Notice that the implication is uni-directed, a biimplication would mean that we were able to solve D^3 precisely. It is from this fact the word approximation stems.

Pre- vs. Re- Calculation

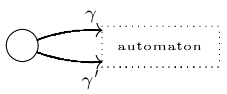
We are now presented with a choice. Either we could try to pre-calculate all possibilities and simply look them up when they occur, or we could re-calculate the dead sets every time a counter dies (having initially calculated the scenario where all counters are live).

Pre-Calculation

Unfortunately, it is not generally the case that:

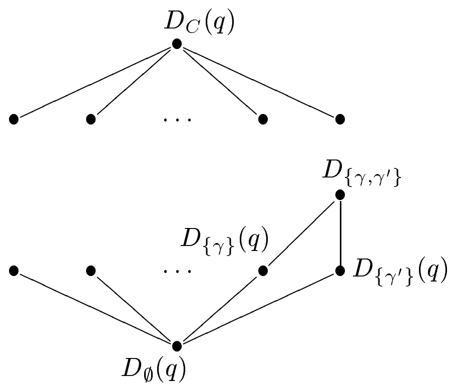
$$D_{S_1}(q) \cup D_{S_2}(q) = D_{S_1 \cup S_2}(q)$$

where $D_S : Q \to \mathcal{P}(\Sigma \cup C)$ is the D function above generalized to give for each state q, the set of labels and counter labels that are dead as a direct consequence of the set S of counters being dead. Therefore, we have to take all possibilities of dead counters into account. That is, we cannot infer what is dead as a result of two counters from what is dead as a result of the two, individually. The following example will illustrate this:



As can be seen in the example, the entire dot-framed automaton is live if one (or none) of the two counters γ and γ' are dead, but not if both are.

This means that we have an exponential dependency on the number of counters, in that we have to do $O(2^{|C|})$ individual pre-calculations, if C is the set of all counters. In fact, this can be perceived as a complete lattice $\langle \{D_S(q) \mid S \subseteq C\}, \subseteq \rangle$:



Re-Calculation

The exponential factor, and the fact that during any run we will only take one path from the base to the top (of the figure above) are strong arguments against the pre-calculation scheme. Fortunately, it is possible to (re-)calculate the deadsets in time $\Theta(V+E)$. The algorithm below will do just that. This is fast enough for supporting an incremental solution, where we have a process running, for instance in parallel, recalculating the dead sets every once in a while.

The Algorithm

Clearly, all nodes belonging to the same Strongly-Connected-Component (abbreviated SCC from now on) will have the same live and dead sets. For this reason we can save time, by first calculating these components using the conventional SCC-algorithm found in [5]. However, the algorithm has been strongly adapted to our scenario. When we are done, we can calculate the local live set of any component as the labels occurring on an edge going from a node in the component. Now the live sets are readily computable as the local live set of the component itself plus the local live sets of all the components reachable from it.

Because $\left(G^T\right)^T = G$ and $SCC(G) = SCC(G^T)$, we can reverse the order of G and G^T in comparison to the traditional SCC-algorithm. That way we can completely ignore the labels when calculating the transposed graph and when performing the DFS. Also, the edges now have the right orientation so that the local live sets can be easily collected during the coloring phase (step six below).

Let S be the set of dead counters at the current state, q. The algorithm proceeds as follows:

① $G := G \setminus \gamma$ -edges, $\forall \gamma \in S$.

② Color G starting at q.

③ G:=colored(G)—everything uncolored is clearly dead.

④ Compute G^T, ignoring labels (that is, edge-annotations).

⑤ Perform a depth first search $DFS(G^T)$ in order to calculate the *finishing times*—which are the numbers obtained from a post-order depth-first traversal of the graph.

⑥ Color(G) in descending finishing times order. Every time a new node (the one with the highest finishing time) is chosen, we add an entry in the SCC-table. During the coloring from this node, all the labels encountered are accumulated in a set (*localliveset*, that will later constitute the live set for this SCC). As a color, we install a "backpointer" (the number of the current SCC) and every time a node is colored, we add this node to a set (*Reachlist*) holding all the member-nodes of the current SCC. Every time an already colored node (with a color different from the current) is encountered, we have reached an edge between this SCC and another. The identity (number) of the target component is added to the *Connected − to* set in the SCC-table. The three sets; *Connected − to*, the *Reachlist* and the live set mentioned above are recorded in the SCC-table for the current component. When we get stuck (we cannot color anymore), we chose a new node as explained above and repeat the process. This is repeated until everything is colored and we thereby have found all the SCC's.

⑦ Calculate the live sets from the local ones. This is done by running through them in the order in which they appear in the SCC-table and taking the union with all local live sets from all components in the *Connected − to* set.

⑧ Live sets are reversed to dead sets (by taking complement with respect to L, the set of all labels).

The first three points will ensure that trivially unreachable parts of the graph are disregarded. The steps four, five and six will find the strongly connected components. Finally, steps six, seven and eight will extract the dead sets from the information given by the SCC-analysis.

The algorithm is linear in the size of the graph $\Theta(V+E)$, which is roughly $O(|L|\,|Q|)$, the number of labels times the number of states in the automaton.

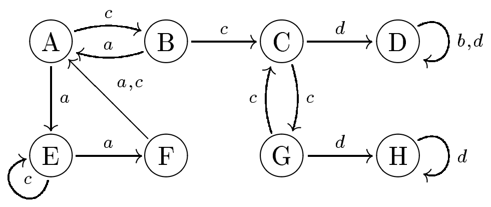
This algorithm can if desired run incrementally as a parallel process in the sense that it looks for dead counters and when one or more are found, it applies the above algorithm to re-calculate the dead sets.

Example

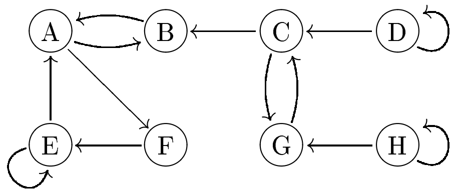
Consider the following automaton, where the current state is A and $\gamma \in S$:

$$\begin{tikzcd}
X \arrow[r, "b"] \arrow[d, bend left, "d"] & A \arrow[r, bend left, "c"] \arrow[d, "a"] & B \arrow[l, bend left, "a"] \arrow[r, "c"] & C \arrow[r, "d"] \arrow[d, bend left, "c"] & D \arrow[loop right, "b"] \\
Y \arrow[u, bend left, "b"] & E \arrow[r, "a"] \arrow[loop below, "c"] & F \arrow[ul, "c"] & G \arrow[l, "\gamma"] \arrow[u, bend left, "c"] \arrow[r, "d"] & H \arrow[loop right, "d"]
\end{tikzcd}$$

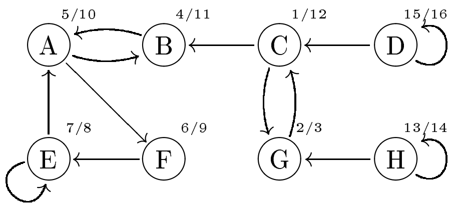
Because $\gamma \in S$ (that is, γ is dead), the γ-edge is removed according to step 1. In step 3, the states X and Y get eliminated because the current state is A and there are no paths from A to X or Y. In other words, the automaton will look like this immediately after step 3:



In step 4, we calculate the transposed automaton without annotations on the edges, after which the automaton looks like this:



In step five, we do a depth first search resulting in:

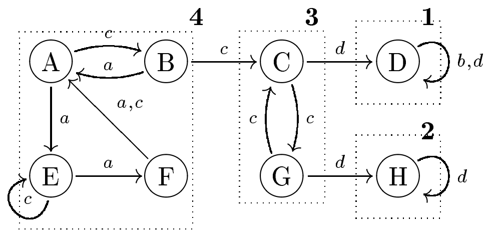


The first number (before the slash) signifies the time we reach the node and the second number is the finishing time for the node. As can be seen, we have (arbitrarily) chosen C as the first node and when we could no longer color, we (arbitrarily) chose H, and finally D.

After we have completed step six, we will have a SCC-Table:

No.	Reachlist	local live set	Connected-to
1	{D}	{b,d}	∅
2	{H}	{d}	∅
3	{C,G}	{c,d}	{1,2}
4	{B,A,E,F}	{a,c}	{3}

Corresponding to the following SCC-division of the automaton:



Finally, after the steps seven and eight, we will have the desired live and dead sets:

No.	live set	dead set
1	{b,d}	{a,c}
2	{d}	{a,b,c}
3	{b,c,d}	{a}
4	{a,b,c,d}	$\emptyset$

We have thus presented an incremental linear-time (in the size of the automaton) algorithm for finding the sets of dead labels at each point in the automaton. However, the algorithm only works on the product automata and does not generalize to the factorized automata we use (as explained in Section 1.5.3). Because of this, we do not use it.

1.10 Conclusion

We have seen how the introduction of labels in the service source code along with safety requirements could produce along with the service, a controller that ensured the requirements were never violated. Also, we introduced the notion of triggers that allowed us to transgress the borders of regularity, in a way that was highly compatible with the logic. All this allowed us to specify intuitively various concurrency mechanisms, such as mutual exclusion and the more sophisticated reader/writer problem. In Chapter 3, we shall see more examples of high level concurrency mechanisms, implemented this way.

Unfortunately, our solution lacks a means for defining and preserving the essence of such mechanisms, making them available as general concepts. As we shall see in the Chapters 2 and 3, this problem has a very general solution.

Chapter 2

The Macro Language

2.1 Introduction

It is very often the case that a fragment of a program captures an abstraction that has a self-contained meaning independent of the program as a whole. Such a fragment will thus have a tendency of occurring in numerous programs and often even several times in the same program, yet with slight variations that depend on the immediate context.

It would be useful, if the programmer could somehow define such abstractions in a manner that is flexible enough to permit certain parts of the abstraction to be parameterized and then, for each occurrence in the program tailor this unspecified part according to its context.

Of course, a language could readily contain a vast number of primitives (with intuitive syntax). However, such a language, static in nature, could never even hope to capture all desirable abstractions. A language designed with this viewpoint would grow indefinitely constantly requiring revisions (and thus compiler updates). An inflexible and costly affair.

However, another solution exists. We shall allow the programmer to dynamically compose and preserve such abstractions, making them available to other programs and programmers. This is reminiscent of a function library but with extreme syntactic flexibility.

We shall distinguish between *function style* and *primitive style* syntax. We will not give a concise definition here, but rather illustrate the differences through an informal example. Consider the language **C** augmented with a **repeat-until** construct (semantics, unspecified). A function style usage could look something along the lines of: **repeat_until**$(x = x * y; y = y - 1; , y == 0)$; whereas a primitive style usage might read: **repeat** $\{x = x * y;\ y = y - 1;\ \}$ **until** $(y == 0)$;. The second really has the look and feel of the **C** language, with enforced braces, parentheses and semi-colon as statement terminator. Enforced in the sense that it would be treated as an error if one were to omit them.

Thus, the crucial difference here is one of syntax. Superficial some may argue, but it does have the non-negligible advantage of making programmer defined constructs appear as if they were part of the original language itself. In other words, the difference is *transparency*—a recurring concept in computer science. Additionally, the syntax may help convey the meaning of and reflect the nature of the abstraction.

Of course, we do not get all the benefits of language primitives. Some such may, for instance, require non-trivial demands on usage, to be determined through analysis, that in turn would enable efficient implementation that would not otherwise be possible.

We would like to build a framework that allows for such fragments to be specified and used. To this end, we have dusted off an old, well-known and widely used concept, namely macros. We present a macro language in which the above is possible.

The macro language presented is currently hardwired into <**bigwig**>, but the ideas of the macro language are completely general (that is, they are in no respect specific to <**bigwig**>) and can be transferred to any modern programming language. In fact, we seriously consider creating a tool **MetaFront** for fast generation of compiler front-ends inherently supporting such macros.

2.1.1 Conception

The concept of macros is by no means a new invention. Webster defines the adjective *macro* as "intended for use with relatively large quantities" and the noun as "a single computer instruction that stands for a sequence of operations" (originating in 1959). This of course also applies to parameterized cases. Macros quickly caught on and became an extremely widespread structuring and information-hiding mechanism for low-level machine code languages. However, with the improvement in compiler technology in the early 70s, and hence the success and availability of numerous high-level languages, macros were confined to a less prominent existence. Since then the notion of macros has been used in many other contexts that really have nothing to do with text expansion, for instance, keyboard macros.

Although a powerful syntactic abstraction mechanism, macros even today remain but a second-rated feature of most modern high-level programming languages. This is mainly due to the many inherent inconveniences directly resulting, we claim, from their often lexical nature. We will attempt to lift the notion of macros to operate on syntax rather than on sequences of characters or lexical tokens.

Conceptually, our macros should be thought of as operators on parse trees as illustrated in Figure 2.1. The figure nicely conveys the intuition behind our macros. The white parts are written by the service programmer and the grey part is written by the macro programmer.

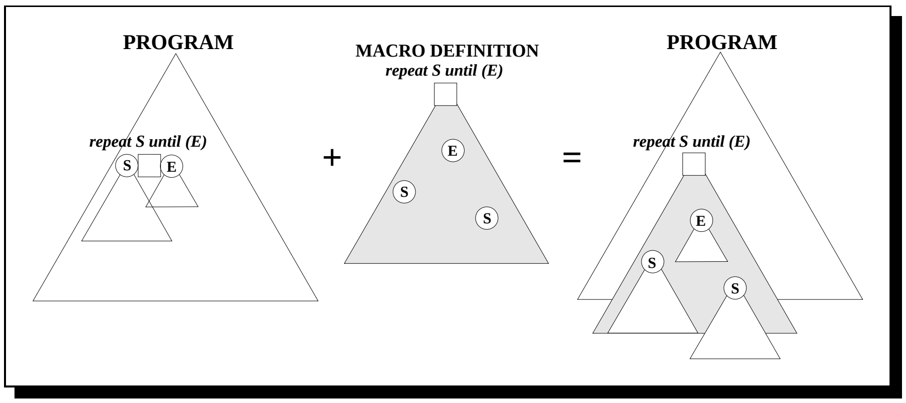


Figure 2.1: Macros—Operators on parse trees

2.1.2 Demands

We shall place several demands on our macro language and on the implementation.

As mentioned above, we want our macros to work on parse trees instead of raw (`ascii`) texts. That is, we want them to operate on a syntactic level rather than a lexical one, permitting only parse-consistent operations. Also, we want them to be completely general in the sense that they should work on all the non-terminals of the grammar. Additionally, they should constitute a closed scope, alpha converting identifiers to ensure this.

We require that the macros be *transparent* so that they will not bother the compiler writer. Finally, one must be able to invoke the macros through a *syntax* flexible enough to permit the language to be transparently extended.

Regarding the implementation, we should be able to *pretty print* the code both with and without macro expansion. Our second implementation demand is that, the compiler should issue *sensible error messages.* By this we require that any errors should be reported exactly where they occur—even if in the body of a macro. Also, for all non-parse errors, one should get the complete trail of macro invocations and arguments corresponding to the particular erroneous program point so as to aid debugging.

2.1.3 Customization

Since the macros can embody an abstraction with a meaning in itself that can be communicated through invocation, it could very well be the case that the macro programmer is not the same person as the service programmer.

Also, one could imagine a library of domain specific macros that were tailored for certain types of services. For instance, a macro library that was customized for the development of web shopping sites, with shopping baskets, customers, shelves, products and the like appearing as if they were part of the original language.

2.2 A Brief Macro Survey

Most programming languages come with macros in some form or another. Macros can be divided into two categories (lexical and syntactical) depending on the level on with they operate.

2.2.1 Lexical Level Macros

The vast majority of macro languages operate purely on a lexical level. By this we mean that they only support substitution of tokens with *arbitrary* sequences of characters. The tokens may be parameterized, in which case the substitution sequence may contain place-holders (or gaps) for the parameters, that themselves are but arbitrary sequences of characters. The crucial point here is that these macros have no knowledge of the syntax of the language whatsoever. Conceptually they constitute a preprocessor, that is, a separate first step in compilation, although rarely implemented as such for reasons of efficiency. In the following section we will look at a few such macro languages. The **C** Macro preprocessor is very representative here, in the sense that it has most of the inconveniences inherent to this category of macros.

After this, we will briefly consider two more rather different macro languages belonging to this category, namely the universal unix standard macro preprocessor called **m4** and the **TEX** macros.

The C Macro Preprocessor—Cpp

In (ansi) **C**'s macro preprocessor, **Cpp**, macros are declared as follows:

#define *name replacement–text*
#define *name* (*arguments*) *replacement–text*

Subsequent occurrences of the identifier specified as the name will be substituted with the replacement-text. If the definition contained arguments (a comma separated list of identifiers), the corresponding identifiers in the replacement-text would be replaced with the arguments supplied upon invocation.

The macros are completely independent of the rest of the language which has some counterintuitive effects (see Figure 2.2).

This program has three peculiarities, one for each macro. Since the substitution is purely lexical, macro expansions may have surprising effects such as the one in the figure (taken from [1]) where $\boxed{\texttt{square(y+1)}}$ becomes $\boxed{\texttt{y+(1*y)+1}}$ and not the expected $\boxed{\texttt{(y+1)*(y+1)}}$. Similarly, the macro invocation of *pos* will "steal" the following **else** (**C** resolves the ambiguous **else** by attaching it to the closest previous **if**) causing, most likely, an unintended behavior. Additionally,

```
#define square(x) x*x

#define pos(x) if (x<0) x=-x

#define swap(a,b) { int x; x=a; a=b; b=x; }

if (square(y+1)<x) pos(x);
else swap(x,y);
```

Figure 2.2: Peculiar **C** example

because the macro language is independent of scope, the substitution may cause inadvertent identifier clashes as is the case with the invocation of the last of the macros, swap. After execution of swap(x,y), x and y will not have been swapped, as the programmer probably aimed for.

Notice finally that "swap(x,y);" has two statements—one compound statement corresponding to the body of the macro swap and one empty statement ";". This does not pose any problems here, but imagine placing this somewhere that expected only one statement, for instance in an **if-else** statement. This can be "hacked", by placing the compound statement in a **do-while** statement with a constant **false** condition ensuring one and only one execution of the body. Here, a final semi-colon would be allowed, terminating the **do-while** statement. Unfortunately, the **Cpp** macro programmer needs to be aware of such pitfalls.

If the body of the macro contains syntax-errors, they will only be discovered when the macro is invoked. Furthermore, such macro errors will be reported as if they occurred at the calling point. Because of this, **Cpp** macros are quite often very hard to debug. Of course, this is implementation specific.

Also, **Cpp** macros have restricted syntax as they can only be specified in the usual function-like manner.

Cpp has a few control language constructs available at preprocessing-time for doing conditional inclusion; #**if**, #**ifdef** and **defined**. Also, there is a construct #**undef** that, not surprisingly, undefines a macro and can be used to simulate local macro definitions.

The intricate details of how, when and in what sequence **Cpp** expands macro calls are rather complex.

In contrast to **C**'s otherwise static scope rules, the scope rules for the macros are dynamic. Consider Figure 2.3 with two macro definitions one of which is defined in terms of the other that is subsequently redefined. A macro is available from the point of definition and onward, lexically speaking. However, since **Cpp** does not expand the body of a macro until the time of invocation, the call to B will cause a new macro call (to A). The identity of A is determined in the *current* environment and not the one at the point where B was defined. Hence

B expands into 42 and not 87. The arrows in the figure each represent one level of expansion.

```
#define A 87
#define B A
#undef A
#define A 42

B → A → 42
```

Figure 2.3: **Cpp** has dynamic macro scope

Cpp has a controversial approach to recursive macros. As explained above, the body of a macro is only treated at invocation time and the recursive case is no different. No errors or warnings are signaled, instead, **Cpp** will keep track of which macros are in the process of being expanded and ignore calls to such macros. To this end, recursive and mutually recursive macros will get expanded one level. For instance, x in Figure 2.4 would at runtime evaluate to 10.

```
int x = 7;
#define x (3+x)

x → (3+x) ≡ 10
```

Figure 2.4: Recursive macros in **Cpp**

Regarding the order of expansion for nested macro calls, we distinguish between *applicative order of reduction* as opposed to *normal order of reduction* (henceforth abbreviated *AOR* and *NOR*, respectively). The two terms are taken from the **λ-calculus**, and are two ways of determining the sequence of reductions in the calculus. Macro expansion is in many ways similar to beta-reductions in the **λ-calculus**. *NOR* will expand the outermost macro first, yielding a *call-by-value* semantics, while *AOR* commences with the innermost resulting in *call-by-name.*

However, **Cpp** uses none of the two schemes. When **Cpp** encounters a macro call, it scans the arguments, while expanding any macro calls into the body of the outermost call. Finally, the whole body of the outermost macro is scanned (implying a rescan of any of the arguments) and any macro invocations are expanded.

This is illustrated by the example in Figure 2.5, that when run produces the error "`macro 'id' used with too many (2) args`", corresponding to the third expansion sequence in the figure.

#**define** id(x) x
#**define** one(x) id(x)
#**define** two(x) a,b

one(two)
$\rightarrow_{AOR}$ one(a,b)
$\rightarrow_{AOR}$ *arity-error 'one'!*

one(two)
$\rightarrow_{NOR}$ id(two)
$\rightarrow_{NOR}$ two
$\rightarrow_{NOR}$ a,b

one(two)
$\rightarrow_{C}^{2}$ id(a,b)
$\rightarrow_{C}$ *arity-error 'id'!*

Figure 2.5: The Order of Expansion in **Cpp**

The above can be exploited to piece together a macro call partly coming from the body of a macro and partly from the actual arguments as in Figure 2.6.

#**define** succ(x) ((x)+1)
#**define** call7(x) x(7)

call7(succ) $\rightarrow$ succ(7) $\rightarrow$ ((7)+1)

Figure 2.6: Piecing together a call

When expanding the macro call7, its argument, the sequence of characters "succ", is scanned, but since "succ" has no arguments (no parentheses) contrary to the definition of the macro succ, **Cpp** does not treat is as a macro invocation. Hereafter **Cpp** scans the text produced and will at this point discover a macro call to succ and expand it.

M4—Unix Macro Preprocessor

The entire basis for **m4** is highly different. **M4** it is not tailored for any specific language, rather it acts as a universal preprocessor, independent of the target language. Originally though, **m4** was the rational **Fortran** preprocessor called "ratfor". It bares quite a few similarities to the **Cpp** macros. **M4** has some thirty-odd built-in macros, some of which have side-effects, such as **define** used for defining new macros (see Figure 2.7).

```
define(`square',`eval($1 * $1)'

square(3)  →  eval(3 * 3) →  9
```

Figure 2.7: A macro in **m4**: **square**

The macro **define** takes two arguments in parentheses, the first is the name, while the second is the body. The arguments in the body are ciphers preceeded by a dollar-sign and thus the number of arguments to a macro is implicitly specified as the maximal such number mentioned in the body. Whenever a macro is invoked with too many or too few arguments, they are assumed to be NULL or ignored, respectively. The preprocessor distinguishes between plain text and quoted text. Quoted text can be arbitrarily nested and will always evaluate to the text within, stripping away the outermost quotes, without expanding any macros. This supplies a simple way of delaying the expansion-time for macros. Note for instance that the quotes around the invocation of the built-in macro **eval** are essential and serve to delay the invocation until the arguments are supplied. The result of a macro expansion is as in **Cpp** reread to expand any new macro calls. This is exactly when the **eval** macro will be invoked. It will evaluate its argument using 32-bit signed integer arithmetic, using **C**-like arithmetic syntactic conventions.

The remaining built-in macros can be placed in one of the following categories:

- evaluation of simple arithmetic expressions;
- simple string operations;
- file inclusion;
- conditional branching;
- system calls;
- management of (multiple) output files;
- explicit stacking of macro definitions; and
- dumping various macro information.

Apart from the syntax of macro declarations, the overall result is more or less the same as in **Cpp**, with the same shortcomings such as no alpha conversion and restrictive invocation syntax. In the following we will mention the main differences.

As opposed to **Cpp**, the macro scope in **m4** is static and all macros, even the ones in the body of others get expanded immediately. Hence a call to a macro defined as B in Figure 2.3 in **m4**, would yield 87, as the body of B would have been expanded into 87, literally.

Recursion is handled in a much more intuitive manner—**m4** will simply reject any such attempts, by issuing an error and abort compilation.

For further details on **m4** we refer to the table in Section 2.2.3.

TEX Macros

TEX provides a somewhat different macro language. Here, one can define along with the macro, the syntax of the call. It is then enforced that all invocations comply with this particular syntax which is also used for matching arguments. Macros are completely integrated with a full-scale interpreted compile-time language that also guides processing. **TEX** does not receive any input and is fully interpreted on compile-time, deterministically producing its **dvi** output for which there is no concept of runtime.

TEX macros are defined by the **def** construct, or similar variants, followed by the name of the macro preceeded by a backslash. Hereafter follows a sequence of tokens and arguments. The arguments are identified by ciphers preceeded by the token "#", bounding the number of arguments by nine. Finally comes the body of the macro, which is taken to be whatever is written between two balancing brackets. Of course, the body can also contain corresponding argument usages, the treating of which is deferred to invocation-time, as with the **Cpp** macros, yielding dynamic macro scope.

```
\def \vector #1[#2..#3]{
   $({#1}_{#2},\ldots,{#1}_{#3})$
}

\vector x'[0..n-1]  →  $({x'}_{0},\ldots,{x'}_{n-1})$
```

Figure 2.8: A **TEX**macro example: \vector

Figure 2.8 shows a definition of a macro called vector plus an invocation of it. The macro is defined to expect its arguments properly delimited by square brackets and two dots. The invocation will of course ultimately generate $\boxed{(x'_0, \ldots, x'_{n-1})}$.

TEX's level of operation is lexical in a slightly different way than that of **Cpp** and **m4**, as the body of a macro gets tokenized at definition time, that is, it is not completely raw text. Except from a few very special cases, this gives the same behaviour. If, for instance, the invocation syntax is redefined after a macro containing invocations of other macros is declared, any such calls resulting from the invocation of the macro itself (thus with an old invocation syntax) will still be treated as such.

Invocations are ambiguous in the sense that there are several possibilities of binding actuals to formals. The argument of a delimited argument is defined to be the shortest (possibly empty) sequence of tokens with properly nested {...} groups that is followed by the particular list of separator-tokens.

As a consequence, it is not possible to directly nest macros such as the one above, as the ']' in the innermost call would terminate the outermost call. Of course, this can be amended by explicitly wrapping brackets around the second call.

TEX has a different approach to the order of expansion, namely *NOR*. Outer macro calls are handled before inner ones. This *call-by-name* semantics is due to the fact that the compile-time control language is completely integrated with the macros and is to react on macros calls that might change the environment for things within the call. However, expansion of a macro can be delayed, so as to treat the inner macro calls first, through the **expandafter** construct, yielding *AOR*.

Recursive macros are handled, or rather not handled, so as to produce eternal expansion. Of course, the idea here is that the control language can halt this process when, for instance, some compile-time variable reaches a certain value.

The intricate details for determining exactly how actuals are bound to formals, on the other hand are relatively complicated with excepting rules for special tokens, such as whitespaces, curly brackets, plus constructs **par** and **long** for instructing the parser when to abort treating a macro invocation.

TEX also provides a way to aid the "programmer" in debugging his macros. **TEX** will print the actual-formal bindings of all macro invocations, for which the compile-time variable **tracingmacros** is positive. However, this information gets logged along with a lot of other information.

The **TEX** macro language has been successfully used to extend **TEX** to, for instance, **LATEX** and **BibTex**.

2.2.2 Syntax Level Macros

Contrasting lexical level macros, we have the other category, the *syntax level macros*. As hinted by the name, these macros are closely coupled with the syntax of the language. The picture here, however, is quite different and there exist very few such examples. We shall briefly consider one, namely **Scheme**'s Hygienic macros.

Scheme Hygienic Macros

In **Scheme** it is possible to specify macros as syntactic transformers. In order to facilitate this, **Scheme** provides a pattern matching language for specifying such macro transformers, as can be seen in Figure 2.9 where we have presented an example of a **Scheme** macro and. Before any evaluation takes place, **Scheme** will transform all macros, using pattern matching to choose which syntactic transformations to apply using NOR beta-reductions. Hence, there is no concept of macros on evaluation-time.

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and b) b)
    ((and b ...)
      (if b (and ...) #f))))
```

Figure 2.9: A **Scheme** macro example: **and**

A **Scheme** macro will preserve the lexical scope of the program and alpha convert all identifiers to avoid conflicts with other identifiers.

Scheme's macros are syntactic, because they enforce the syntactic structure of **Scheme** and know enough about it to alpha convert identifiers. However, there is only one syntactic category, namely expressions, the syntax of which is explicit on runtime. All **Scheme**'s macros must comply with the prefix syntactic style of **Scheme**.

2.2.3 Comparison

In this section we will juxtapose the four macro languages we looked at in the previous along with our macro language <**bigwig**> that is the topic of the rest of this chapter. The table below exhibits characterizing properties for each of the five macro languages.

First, the level of operation that reveals the nature of the macro concept, then the existence and nature of any control language interpreted at compile-time along with the macros. After this comes a lot of cosmetic aspects of the macros. We take *transparency* to mean whether the macro user is or needs to be aware of them. However, we do acknowledge that this is perhaps a rather subjective category. Hereafter, we present semantic aspects dealing with ambiguities, scope, and overall behaviour. Finally, we exhibit implementation specific properties such as the support for pretty printing with and without the macros expanded and the trailing of errors to aid debugging.

Property\Language	**Cpp**	**m4**	**TEX**	**Scheme**	**<bigwig>**
Level of operation	lexical	lexical	lexical	(syntatic)	syntactic
Macro computation	no	limited	full-scale	patterns	no
Invocation syntax	*id*(...)	*id*(...)	arbitrary	(*id* ...)	arbitrary
Argument syntax	*id*	$*[0-9]*	#*[1-9]*	patterns	*id*
Typed arguments	no	no	no	no	yes
Transparency	no	no	yes	yes	yes
Macro ambiguities	none	none	shortest	none	greedy
Macro backtracking	no	no	no	patterns	no
Order of expansion	mixed	inner*	outer*	outer	inner
Macro scope	dynamic	static	dynamic	dynamic	static
Local macro scope	no	yes	yes	yes	yes
Multiple definitions	no	no	no	patterns	split
Alpha conversion	no	no	no	yes	yes
Recursive definitions	1 level	rejected	loop	pat./loop	rejected
Pretty printing	no	no	no	no	yes
Error trailing	no	N/A	(no)	no	yes

Table: Comparing five macro languages.

At first glance, macro languages may appear relatively indistinguishable in behaviour, but as the table above shows, they are in fact highly different and in lots of respects. No two of the above macro languages are quite the same.

One important issue that discerns our macro language from the rest is the fact that it has been *designed* and that as an individual entity, contrary to the others, that sort of evolved from other applications. Although our macro language has been designed as a separate entity, it has been developed along side the rest of the <**bigwig**> language, for which it has always played a conscious role. For these reasons, the <**bigwig**> macro language has been guided by an overall design strategy, as an intuitive and comprehensive extension to a **C**-like language.

2.3 Syntax

In this section we will present the syntax for defining macros along with a lot of examples, gradually increasing in complexity. After this, we shall look at some naming conventions and conclude with comparing macros and functions.

macro_list	:	*macro**
macro	:	**macro** < *nonterm* > *id macroparam*
macroparam	:	< *nonterm id* > *macroparam*
	|	*id macroparam*
	|	*token macroparam*
	|	::= { *macrobody* }

The first production for the non-terminal *macroparam* is for declaration of macro arguments. The next two are for adding separators either in the form of an identifier or a token. The last one is to terminate the macro header and thus requires the *macrobody* the type of which, of course, corresponds to the non-terminal type of the macro, written after the keyword **macro**. All macros are in <**bigwig**> declared *before* the actual service, so that one first extends the language by declaring macros (typically by including some libraries) and then specifies the actual service for the "fixed" extended language. This could easily be modified so as to allow macros to be declared within the service code.

2.3.1 Examples

Let us look at some examples. To underline the generality of our macros, we will exhibit examples from several different syntactic categories.

One of the simplest macros one could write, would be a macro that does not take any arguments as is the case for the macro **pi** in Figure 2.10.

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

Figure 2.10: A very simple macro: **pi**

When declared it will appear to the programmer as if the *floatconst* syntactic category had been extended with a production **pi**. This is different from a lexical macro in that the macro invocation of **pi** is only allowed in places where a *floatconst* would be.

Figure 2.11 shows a macro defining a new construct **maybe**, that takes one argument, namely a statement and executes it with 50% probability.

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

Figure 2.11: A macro taking an argument: **maybe**

Consider the *regexp* category in the <**bigwig**> grammar (which is available at http://www.brics.dk/bigwig/langspec/grammar.html). As one can see there is something called **star**, for Kleene's star on regular languages, signaling zero-or-more. However, there is nothing called **plus** for one-or-more. Such a construct could easily be defined by a macro. This will be a nice example (see Figure 2.12) of a macro that uses token separators to enforce a particular syntax. The macro definition contains two *tokens* (corresponding to the third production of *macroparam*), namely the two parentheses. The compiler will thus automatically enforce that invocations of the macro **plus** contain the two parentheses in the sense that it would be a syntactic error to omit them. In this way the macro author can tailor his macros to have the desired look-and-feel.

```
macro <regexp> plus ( <regexp R> ) ::= {
    concat(R,star(R))
}
```

Figure 2.12: A macro definition with tokens separators: **plus**

Of course, this could be abused to write macros that expected horrific syntax with, for instance, unbalanced parentheses of varying types. But this comes with extreme flexibility. So the macro programmer should take some care when designing the macro's syntax.

A similar macro, but from a completely different syntactic category, is the macro **never** in Figure 2.13.

```
macro <formula> never ( <id L> ) ::= {
    all t: !L(t)
}
```

Figure 2.13: Another macro with token separators: **never**

Let us now define the macro implicitly referred to in Figure 2.1; **repeat-until**. This can not surprisingly be done in terms of **while**, see Figure 2.14. This could of course have been done easier in terms of **do-while**, but doing it in terms of **while** will illustrate a point. This macro will take two arguments, S and E.

As can be seen in Figure 2.14 this really has the look-and-feel of **C**. The **repeat-until** construct is *transparent* in the sense that it appears to the programmer as if it really was in the language.

The macro uses its statement argument S twice. Since we do not use DAGs (reasons explained later) it will be present twice in sub parse trees resulting from invocations of this macro. For this reason, it would probably be a better idea to write it a little differently as in Figure 2.15. Another reason exists, further motivating the choice for the second version. This explanation, however, is deferred to the section on pretty printing.

There is no reason why the body of a macro-definition cannot contain another macro invocation, as is the case with our next example. Here, we have a new macro **forever** defined in terms of **repeat-until**. Actually, such a macro will

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
    {
        S
        while (!E) S
    }
}

session S() {
    ...
    x = 1;
    repeat {
        x = x*y;
        y = y-1;
    } until (y==0);
    ...
}
```

Figure 2.14: A macro taking two arguments: **repeat-until** and an invocation

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
    {
        bool first = true;

        while (first || !E) {
            S
            first = false;
        }
    }
}
```

Figure 2.15: Another version of **repeat-until**

only be expanded once, namely when the other macro is parsed, yielding static macro scope, but all this will be explained later.

```
macro <stm> forever <stm S> ::= {
   repeat S until (false);
}
```

Figure 2.16: A macro defined in terms of another

Even an entire service could be written as a macro (see Figure 2.17).

```
macro <service> my <stringconst C>
                  hello-world service! ::= {
   service {
      session Hello() {
         html HelloWorld = <html>
            <h1><font color=[color]>Hello World</font></h1>
         </html>;
         exit plug HelloWorld[color = C];
      }
   }
}

my "blue" hello-world service!
```

Figure 2.17: An entire service as a macro

Naming Conventions

Before we proceed, we will establish some naming conventions. In order to do so, let us look at Figure 2.14 again. The macro itself will be referred to by its first identifier (**repeat** in the Figure). We will refer to $\boxed{\texttt{S}}$ and $\boxed{\texttt{E}}$ in the macro definition header as the *formal arguments* of the macro. Correspondingly, $\boxed{\texttt{\{ x = x*y; y = y-1; \}}}$ and $\boxed{\texttt{y == 0}}$ will be called the *actual arguments* of the macro. As for $\boxed{\texttt{S}}$, $\boxed{\texttt{E}}$, and $\boxed{\texttt{S}}$ in the body of the macro definition, we will use the term *macro argument usage.* Finally, we shall call **until** a *macro separator* and refer to the tokens (the two parentheses and the semi-colon) by the term *macro separator tokens.*

2.3.2 Macros vs. Functions

Macros may at a first glance appear similar to functions, but there are significant differences. A macro takes *syntax* as arguments on *compile-time*, whereas a function takes *values* as arguments on *runtime*.

Macros are in their very nature non-recursive, so we definitely still need functions. Additionally, functions have several other advantages over macros, like different parameter mechanisms, space economy, and type information.

One should bear in mind that the macros are in no way designed to replace functions. We want both abstraction mechanisms.

In the following we focus on the benefits of macros over functions as they are in **C** (and many other similar languages). To this end, we have sketched a list of benefits from the macros that does not come with such functions.

1. *Genericity.* They work on all non-terminals. So far we have seen examples of macros defined for *floatconst*, *statement*, *regexp*, *formula*, and the *service* category.

2. *Uniformity.* Abstractions are defined in the same manner for all syntactic categories. This is especially useful in <**bigwig**>, since it basically is an ensemble of domain specific languages with constraints, regular formats, html, and so on.

3. *Transparency* (internally). The compiler has no need for binding calls to definitions.

4. *Overloading.* They are independent of the type system and hence provide a means for defining type overloaded constructions.

5. *Efficiency.* No function calls are required.

6. *Call-by-name.* They provide this (alternate) parameter mechanism and code that is repeated over an over, yet with slight variations, can be written once and reused indefinitely through a parameterized macro.

7. *Syntax.* The macros can transparently extend the original language (see for instance the invocation of the **repeat-until** construction in Figure 2.14).

Of course, the functions can mimic some of these characteristics depending on the nature of the function abstractions.

If the original language was designed according to *Tennent's Principle of Abstraction...*

> "Any semantically meaningful syntactic class can
> in principle be used as the body of an abstraction."

...the "function" concept would be able to dual the first point, and to some degree the second. The third, however, cannot be mimicked by the functions, as a function is itself a citizen of the language. If the original language is polymorphic or dynamically typed, the fourth point is non-applicable. Function in-lining can make the fifth point apply to non-recursive functions. Finally, the sixth and seventh points may also be inherent to the original language.

For functional languages with only one syntactic category and several parameter mechanisms the macros may, appear rather unappealing. However, for languages like **C**, it provides a comprehensive and intuitive extension with the above benefits and without having to redesign the entire language.

2.4 Semantics

In this section we will look at the semantic properties of our macros. We will look at how ambiguities are resolved and what to do about recursion. Also, the scope aspects will be covered extensively, this includes alpha conversion and the possibilities of introducing local scopes. Finally we shall look at the order in which nested invocations take place, and its implications.

2.4.1 Resolving Ambiguities

When using the macros, some cases of ambiguity arise. For instance, it is not clear how invocations of a macro defined to take, for instance, two consecutive statement-lists should be parsed. As with the dangling **else** in **C**, we have disambiguated by convention. We have chosen the macros to be greedy, again because this seemed the most intuitive thing to do. Thus the macro mentioned would parse as much as it could as the first argument, while leaving the second empty.

As mentioned earlier one can write a lot of nonsense with the macros. In fact, one can write macros for which there are no possible invocations. This is, for instance, the case with the macro in Figure 2.18. Because of the production $exp \rightarrow exp + exp$, the parser would never stop anywhere with a plus as the next token while parsing the first expression argument. Of course, the compiler could warn the programmer of such macros.

```
macro <exp> add <exp E1> + <exp E2> ::= {
   E1 + E2
}
```

Figure 2.18: A macro that has no possible invocations

2.4.2 Recursion

We have chosen to reject attempts to define macros in terms of themselves, for the simple reason that their expansion will loop forever. Contrary to **TEX**, we do not have a compile-time language interpreted during parsing that could halt this expansion at some point. Also, we did not find **Cpp**'s solution with one level expansion very intuitive.

2.4.3 Scope

We would like to ensure that the macros preserve lexical scope, so that no local identifiers introduced by a macro inadvertedly clash with or shadow identifiers from outside scopes. Furthermore, we would like to make sure that the body of each macro constitutes a closed scope. That is, a scope that is inaccessible in both directions. In this way the expansion of a macro becomes safe in the sense that it will not cause any symbol clashes. In terms of Figure 2.1, no identifiers from the white and gray areas clash with each other. The reason for this is that macros in this way are independent of their immediate context unless through their arguments. This will enable only macros with self-contained meanings to be written.

Static Scope

Consider a new macro defined in terms of an old one. We want the invocation of the old macro to take place *once*, namely when the new macro is defined, and not each time the new macro is invoked. This will give us static macro scope. We have chosen static scope, because it is by far the most intuitive and widely used. Also, the scope rules of <**bigwig**>, for which we have implemented this macro language, are all static.

Concatenating Identifiers

We provide a mechanism for transgressing these scope boundaries—the motivation will be evident later. Because of this, we have introduced an operator for concatenating identifiers: $\boxed{\sim}$. Our identifier concatenation operator thus resembles the ##-operator in **C**, except that ours is explicit in the grammar as a production on identifiers:

$$\begin{array}{lll} id & ::= & tIDENTIFIER \\ & | & id \sim id \end{array}$$

Two identifiers, `x` and `y`, will when concatenated be represented internally as the string "`x__y`" (separated by two underscore characters). In order to assure that such identifiers do not clash with normal ones, we have prohibited identifiers beginning with, ending with, and having two consecutive underscores.

Without having introduced our alpha conversion rules, this operator may seem worthless. The idea with this construction is that it provides a mechanism for bypassing the scope boundaries. Another equally important motivation for this construct is that it permits the generation of a lot of new identifiers parameterized by a single. This will be extensively exploited in examples to come.

Alpha Conversion

Without alpha conversion, the expansion of macros becomes dangerous (as with x in the body of the macro swap from Figure 2.2). Of course, the programmer could just rename x to xyz87 or something even more arbitrary. However, this is not always a safe thing to do either, even if all introduced identifiers all have different arbitrary names. Nesting such macros or using them twice could cause errors—of course, this depends on the nature and scope rules of the "target" language.

In any case, we do not want this to be left to the programmer. Instead, we would like the compiler to alpha convert all such identifiers in a sensible way. But exactly how to do this is not at all evident.

Determining which identifiers are locally introduced in the body of a macro implies full-scale symbol checking. In order to avoid this expansion-time symbol checking, we instead alpha convert *all* identifiers. The scope of a macro will become closed. In fact, the scope of a macro body (the grey area in Figure 2.1) will be packed into the immediately surrounding scope frame, yet it will not clash with any of its identifiers. Thus, while expanding macros we have no reason to be aware of any symbol or scope information and yet the macros will behave exactly as if we were. Furthermore, the alpha conversion is done before the symbol checking so that this phase need not be aware of the macros. Thus, the two phases become independent.

We will suffix (using the identifier concatenation operator "$\sim$") all identifiers in the grey area (in the sense of Figure 2.1) of the macro with a number. This can never clash with normal identifiers because they can never be constructed—an identifier cannot begin with a cipher. This number will enumerate the macro invocation taking place so that the number will be "fresh" for each macro invocation. In this way, all declarations and corresponding usages in the same macro body will refer to the same identifier as before alpha conversion.

Consider Figure 2.19. The dummy code contains three macro invocations, two of the macro **repeat** as defined in Figure 2.15 and one to a macro **swap**, defined in the figure. Both of the macros introduce local variables that will automatically be alpha converted, which is evident from the macro expanded output in Figure 2.20.

As previously mentioned, the identifier concatenation is for bypassing the alpha converter. We shall thus use the following function $\alpha : Id \rightarrow \{\mathbf{true}, \mathbf{false}\}$ to determine whether an identifier should be alpha converted or not.

$$\begin{aligned} \alpha(I) &= \begin{cases} \mathbf{false}, & \text{if } I \in \mathit{amaenv}, \\ \mathbf{true}, & \text{otherwise.} \end{cases} \\ \alpha(I \sim I') &= \alpha(I) \wedge \alpha(I') \end{aligned}$$

As one can see, identifiers being part of arguments through the identifier concatenation, are never converted, and thus provide access to the macro. We shall see why this is so interesting later.

```
macro <stm> swap <type T> <id X> <id Y> ::= {
    {
       T t;
       t = X;
       X = Y;
       Y = t;
    }
}

service {
    session S() {
       int first;
       int t;

       repeat {
          first++;
          repeat t++; until (t>10);
          swap int t first;
       } until (t>42);
       t=t*2;
    }
}
```

Figure 2.19: A dummy service: alpha

```
service {
  session S() {
    int first;
    int t;

    {
      bool first~1 = true;

      while (first~1 && !t>42) {
        {
          first++;
          {
            bool first~2 = true;

            while (first~2 && !t>10) {
              t++;
              first~2 = false;
            }
          }
          {
            int t~3;

            t~3 = t;
            t = first;
            first = t~3;
          }
        }
        first~1 = false;
      }
    }
    t = t * 2;
  }
}
```

Figure 2.20: The alpha converted expanded code for alpha

Suppressing Alpha Conversion

We have also considered making it possible for the programmer to suppress the alpha conversion. This could for instance be done by prefixing an identifier with a $\boxed{\text{'}}$ (which in **Scheme** and many other languages has the semantics that what is written subsequently is to be taken literally—which is basically what would be done). The quote could then be applied to the name of the macro in the definition header with the result that no identifiers in the macro would be alpha converted. Or, the quote could be applied to individual identifiers in the body of the macro, suppressing alpha conversion.

All this provides a means for a macro to access its surrounding scope. Thus, such macros would be context dependent and situation specific, jeopardizing the self-contained meaning property. For this reason we have not included the alpha-suppression operator in our macro language.

Nesting Macro Definitions

Sometimes it would be nice to introduce a local macro, the usages of which are restricted to a certain area (or scope) in the program. This will also help reduce name-space pollution. Since we have static macro scope, we only need to provide and maintain a stacking mechanism for the definitions in order to obtain a scoped solution.

To this end, we have introduced two macro stacking meta directives #**require** and #**end**. The directive #**require** acts as an inclusion directive taking a file as a constant argument (much like that of #**include**), but only macros are allowed within the file. All #**require** directives will give rise to a new macro stack frame in which all macro definitions from the specified file will be mentioned (with a pointer to the actual definition, kept in a global macro set). Whenever requiring a file that has already been required, #**require** will not re-parse the file, only add a dummy frame to the macro stack, stating that the appropriate file has been re-required. The corresponding popping mechanism is supplied by #**end**. The two directives are entirely independent, in that we do not require them to balance, nor do we require an equal number of them. When we meet a #**end** directive, we pop the top frame from the macro stack. If this frame is a dummy re-requisition element, we throw it away. If, on the other, hand it is a "real" macro stack frame, with macro definition pointers, we follow all the pointers and mark the appropriate macros as dead. Macro definitions in the global macro set are completely ignored by the macro parser.

This simple mechanism will avoid multiple inclusions and break circular inclusion, as any re-inclusions will not be re-parsed.

Encapsulation

At some point, we considered adding the concept of *encapsulation*, called **package**, comprising a set of macros and a sequence of local *toplevel* syntax. The idea was to alpha convert the local syntax inside the package in order for it to constitute a closed scope, only available to the macros within the package that in turn were visible from the outside. As with the macros, the packages should be specifiable in flexible syntax that also allows for parameterization. This would permit the programmer to present to the user, for instance a statistics package like the one in Figure 2.21.

```
package normal ( <exp M> , <exp S> ) {
   float my = M;
   float sigma = sqrt(S);

   macro <id> mean ::= {
      my
   }

   macro <id> variance ::= {
      sigma
   }

   macro <exp> observe ::= {
      Φ^{-1}((random()-my)/sigma)
   }
}
```

Figure 2.21: A macro package: **normal**

In order to be able to use multiple distinct instances of this package, clearly the programmer would have to give each such instance a name. For each subsequent invocation, the programmer would have to explicitly state which instance he is referring to. Since identifiers are our atomic entity for providing links in the source program, the naming and instance referencing should be specified by means of identifiers.

Instead of supplying this additional package concept, we shall present a way of achieving this, but using the basic macro concept as it is. The **normal** package and its three macros can instead be written as four separate macros linked together through concatenated identifiers as in Figure 2.22. One of these macros will serve as a declaration macro, simulating the package instantiation.

As one can see, a **normal** can be "declared" as an identifier, seemingly much like that of declaring for instance an integer, invoking the first macro. The identifier will be used to construct the "local" data needed (my and sigma in the example above). Henceforth the other macros are available *through* this identifier. That is, the "local" data on which to operate is again constructed from the supplied identifier. This way one can have multiple **normal**'s working on distinct "local" data. We repeat the phrase previously stated: "identifiers are our atomic entity for providing links in the source program". Indeed, this is true

```
macro <toplevel_list> normal ~ N
        ( <exp M> , <exp S> ) <id X> ; ::= {
   float X~my = M;
   float X~sigma = sqrt(S);
}

macro <id> E [ <id X> ] ::= {
   X~my
}

macro <id> Var ( <id X> ) ::= {
   X~sigma
}

macro <exp> observe ( <id X> ) ::= {
   Φ^-1((random()-X~my)/X~sigma)
}
```

Figure 2.22: The **normal** package in terms of regular macros

for the above.

This technique shall be employed several times in the next Chapter.

2.4.4 Order of Expansion

In the presence of nested macro calls, we expand the innermost first, yielding AOR or *call-by-value* because we thought this was the most intuitive. Since we do not have any compile-time language interpreted during parsing, there are only two issues for which this choice matters. Namely, efficiency and alpha conversion.

Clearly, if the outermost macro threw its arguments away it would be more efficient to expand this one first, ignoring the second completely. On the other hand, if it replicated its argument several times, it would be better to expand the innermost call once and not once for every newly generated invocation. Thus, efficiency does not dictate a clear choice.

Since alpha conversion may side-effect identifiers in a macro invocation, it is not independent of the order of expansion, which is illustrated by the example in Figure 2.23.

As can be seen in the figure, the order of expansion does matter with respect to alpha conversion. The program resulting from using AOR will fail symbol checking as an identifier clash will be discovered. The NOR version, however, has two distinct identifiers in it and is indeed valid. Nonetheless, we shall still

macro <*decl*> declare ::= {
    int x;
}

macro <*decl_list*> double (<*decl* D>) ::= {
    D D
}

double (**declare**)
    $\rightarrow_{AOR}$ **double** (int x~87;)
    $\rightarrow_{AOR}$ int x~87; int x~87;

double (**declare**)
    $\rightarrow_{NOR}$ **declare declare**
    $\rightarrow_{NOR}$ int x~87; **declare**
    $\rightarrow_{NOR}$ int x~87; int x~88;

Figure 2.23: Alpha Conversion and the Order of Expansion

go by the *call-by-value* strategy, because we claim it is the most intuitive. One would expect any legal invocation of the macro **double** to give an error. The alpha conversion should be a property of the **declare** macro alone and have nothing to do with the macro **double**.

2.5 Implementation

In the following we will look at the implementation of the macros (as they are in <**bigwig**>).

2.5.1 Parsing

Often a compiler front-end is written with the parser generators Yacc/Bison and Lex/Flex. However, since we need to direct parsing according to the macro definitions and the tools above provide no direct way of doing so, we have written our own parser for the <**bigwig**> language.

Fortunately, the <**bigwig**> grammar on which our macros are based is $LL(1)$. This is mainly due to the fact that it resembles **C**, which is basically $LL(1)$. However, a few syntax revisions have taken place in order for it to comply with the $LL(1)$ demands. In this way we could straightforwardly construct a predictive top-down recursive descent parser on which to base the parsing of macros. For this reason, we have not considered the possibility of parsing bottom-up.

However, instead of making one big static $first$ table, we have made a $first$ function ($first : TOKEN \rightarrow \{\mathbf{true}, \mathbf{false}\}$) for each non-terminal that will given a token tell us whether the token can be reached from the non-terminal. This choice will be elaborated and motivated later.

We see no reason, other than efficiency, why the macros could not support trial-and-error parsing. However, for this precise reason, we have no intention of doing backtracking.

2.5.2 Parsing Macro Definitions

The parsing of macro definitions can conceptually be separated into two tasks, namely the parsing of the macro header and the body. We shall look at the two tasks individually in the following. Of course, the two tasks are integrated so as to support macro calls in macro definitions.

Parsing the Macro Header

The parsing of the macro header is straightforward. It amounts to parsing according to the macro grammar presented in Section 2.3, while building an appropriate structure. The structure built from the **repeat-until** macro definition from Figure 2.14, is illustrated in Figure 2.24. As one can see, all macro definitions are placed in one global set (implemented as a hash-table) each mapping to a structure. Supporting local macro definitions amounts to maintaining a stack of such definitions rather than a global table. The structure begins with a *macro definition* node stating the name and resulting type of the macro at hand. This is followed by a sequence of *macro parameter* nodes of various kinds corresponding to the four *macroparam* productions in the macro grammar. The last node in this structure is always guaranteed to be of kind *body* and will appropriately contain a reference to the parse tree body of the macro. The type of this parse tree will of course be the one specified in the beginning of the macro (*statement* in the case of **repeat-until**). The reason why the line number and filename information is placed here and not, as one would expect, in the initial macro definition node, is due to the *split* feature that will be explained later. While parsing the macro header, the macro parser will collect all formal macro arguments in an environment to be used for recognizing the macro argument usages in the body of the macro. This formal macro argument environment, is a partial function from identifiers to non-terminal types: $fmaenv : Id \hookrightarrow NT_{type}$. However, it is implemented as a list of pairs: $fmaenv : [(Id, NT_{type})]$. Since, we treat one macro definition at a time, we only ever need one $fmaenv$. Naturally, the compiler checks that no formal argument is defined twice or has the same name as the macro itself. After having constructed the corresponding $fmaenv$, which becomes [(S,<stm>),(E,<exp>)] for the **repeat-until** example), we are ready to parse the body of the macro.

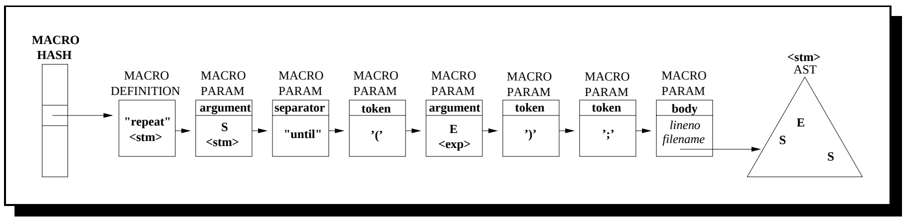


Figure 2.24: A Macro Header: (**repeat-until**)

Parsing the Macro Body

In order to recognize the macro argument usages in the body of the macro, we need to parse relatively to the constructed $fmaenv$. Each time we get the *next* token from the lexer, we check to see if it is an identifier and whenever so, we look it up in the $fmaenv$. If it is not there, it is considered a normal identifier, but if it is, the following happens.

If we are at the right spot—that is, the non-terminal we are about to parse is the same as the type of the formal macro argument (registered in the $fmaenv$)—we create a space in the parse tree for the future actual argument, after which we continue usual parsing. If, however, we are not *yet* at the right spot—that is, there exists a sequence of productions in the grammar that will take us there—the *first* functions, that embody the structure of the grammar, will take us there.

First Functions vs. A First Table

The *first* functions are constructed directly from the grammar in an obvious manner. Each non-terminal A will give rise to a first function depending on its productions. This function will take the *next token* and answer **true** or **false** depending on whether there exists a derivation from the non-terminal A having the *next token* as its first terminal.

There are two reasons why we have not optimized this using a statically computed *first* table (given by a fixed-point computation), instead of *first* functions. First of all, it is very easy to modify the functions when modifying the grammar, since only the affected productions need modification. This is untrue for the tables that would need to be totally recomputed since they may be completely altered from even minor modification. Secondly, whether a given terminal can be derived as the first terminal from a non-terminal, depends not only on static information, but also on dynamic information, namely the defined macros. Of course, this could be split into a static part and some dynamic data structure, but we have not looked any further at this.

The result of parsing the macro header and body should be a parse tree with a number of holes in it for each formal macro argument. Now, the macro is ready to be invoked.

2.5.3 Parsing Macro Invocations

As with macro argument usages above, whenever the lexer returns an identifier, we look it up in the global macro set to see if it is the name of a macro. If this is the case, the *first* functions will direct parsing to the appropriate non-terminal (the resulting type of the macro). Once this is done, the parsing of the macro invocation begins. This amounts to running through the macro parameter chain doing one of two things depending on the kind of the macroparam node.

If the macroparam node specifies either a separator or a token, corresponding to the second and third cases in the syntax of macros definitions, the compiler checks to see if the lexer returns such an identifier or token. If not, an error is issued. If however, it is a macroparam node specifying an argument (the first macroparam production), the compiler will parse as much as it can of the syntactic category specified in this node (recall that the macro parser was greedy by definition). The resulting parse tree will be stored in something we have called an actual macro argument environment ($amaenv : Id \hookrightarrow AST$), which is basically a partial function from identifiers to parse trees, holding the actual macro arguments for each formal macro argument of the macro.

Once the chain has been successfully traversed, the compiler has filled up the *amaenv* and the construction of the resulting parse tree can begin. The compiler retrieves the parse tree corresponding to the body of the macro at the end of the macroparam chain. The job is now, given a macro body and an *amaenv*, to produce a resulting parse-tree (as depicted in Figure 2.1). The compiler now begins copying this parse tree since we do not want any DAGs. We will not allow DAGs because of simplicity and the fact that the analysis phases want to attach information specific to the individual points in the program such as type, symbol and error information. Each time a macro argument usage is encountered, the compiler initiates copying of the argument's corresponding parse tree, which it finds in the *amaenv*, onto the macro argument usage (recall that these had space left in them for this purpose). Also, the fact that the argument has been expanded is added to the macro argument usage. During all this copying, some identifiers will get alpha-substituted, but the details of this is deferred to the section on scope.

The parsing of macro invocations, of course, occurs simultaneously with the parsing of macro definitions as we want to be able to construct new macros in terms of old ones. Recall, from the discussion of static scope, that the call to the old macro in the body of the new one would be generated *once*, namely during parsing of the body of the new macro and not each time the new macro is invoked. The expansion of the invocation of the old macro call would thus be stored inside the body of the new one at the end of its *macroparam* chain, copied along whenever the new macro is invoked.

2.5.4 Representation

The representation of macros can be put in two categories: implicit and explicit. In the following, we shall look at both. Consider the macro and invocation in Figure 2.25. The invocation of the macro IDxy with the actual argument B,C will, along with the A and D before and after, construct the identifier-list A,X,B,C,Y,D. We shall now utilize this example to see how the invocation is represented internally in the parse tree employing each of the two different schemes.

All the macro representations we have considered, however, have one inherent flaw. Once you have constructed the parse tree, it contains all macro information, hence you cannot rearrange it and expect to be able to pretty print it correctly. To this end, if some rearranging is to be done (for instance, desugaring or parse tree optimization), one has to go about it differently—we present a few possibilities.

One could add information to the existing parse tree that can be useful to subsequent phases, add the information externally in the form of separate parse trees, or simply do the optimizations once it has been established that there are no more errors (and the pretty printer has done its job).

```
macro <id_list> IDxy ( <id_list I> ) ::= {
    X,I,Y
}

... A,IDxy(B,C),D ...
```

Figure 2.25: A macro and an invocation

Implicit Representation

The first representation we shall consider is one in which the macros are almost completely transparent. All nodes are augmented with macro specific information that states whether the given node contains a call or an argument and if so, what the definition of the macro is and where it is defined. Also, all nodes contain information in the form of a pointer, a *prev-pointer*, which is a pointer to the nearest macro call or argument. This way, when an error is discovered one can follow the prev-pointers and write out the trail of macro calls and arguments along with the error. The prev-pointers are depicted as dotted arrows in Figure 2.26 that shows the example from Figure 2.25 represented using this implicit representation scheme. The normal arrows symbolize next-pointers (a sequence of any category has been represented as a list, linked together through the use of next-pointers).

Since there are no explicit macro nodes, all information regarding macros has to be packed into parse tree nodes. That is, the macro information has to be written in the parse tree node following the point where the macro call or argument reside. As one can see, the macros impose no new nodes on the

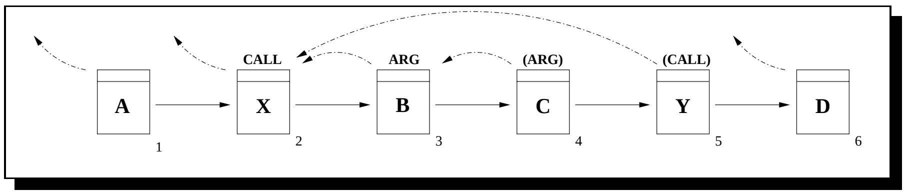


Figure 2.26: Implicit Macro Representation

parse tree. It has exactly the same nodes as if one had written the identifier list A,X,B,C,Y,D directly.

However, having only one piece of macro information in each node is not enough. If two macro calls are made without any parse tree nodes in between (as is for instance with **forever** in Figure 2.16 where the calls to **forever** and **repeat-until** occur in the same place, so to speak), a single macro information entity is not enough. The two calls thus have to be packed into the same parse tree node which requires us to generalize the scheme so that we instead speak of sequences of macro information. The same problem will also arise when having two arguments or even a call and an argument that are not separated by something that will give rise to a parse tree node.

In order for the pretty printer to be able to print the source code without macro expansions, it has to be able to recognize which nodes are part of a macro call or argument—that is, where the macro call or argument ends. Thus, parse tree nodes that are not a call or an argument themselves, but are part of the body of a macro or part of an actual argument, have to be tagged with appropriate information. This is what is meant by CALL and ARG in parentheses in Figure 2.26.

As one can imagine, the construction of parse trees respecting this scheme is somewhat tedious and error prone—but it gets worse. There are scenarios we would like to be able to handle, that cannot be represented employing this representation. This is the case if we supplied no identifiers to the argument of the IDxy macro. Nothing is a valid identifier-list and the construction A,IDxy(),D would yield a valid identifier-list, namely A,X,Y,D. The problem we are faced with here is that, unlike the scenario above, we have no obvious place to put the macro invocation and (empty) actual macro argument information. An even worse example is the identity macro on for instance identifier-lists that takes the empty identifier-list.

At the time of writing, identifier-lists cannot be empty. However, we plan to make this possible and then extend the weeder to check that such lists are not empty whenever they should not be. Only because a macro returns an empty lists does not mean that the overall result is empty, as in the example above. The above argument applies equally to, for instance, *sesarg_list*s that can indeed be empty.

Because of this, we looked for alternative representation schemes.

Explicit Representation

Instead of trying to make the macros transparent by packing all related information into usual parse tree nodes, we considered representing the macros explicitly. For each macro call we insert a special parse tree node that tells us if we are faced with a macro call, how the macro is defined, and where (see Figure 2.27). This special node has two pointers, a standard next-pointer to designate what follows the macro call, and a special macro-pointer (depicted by dashed arrows) that points to the body of the macro. This way, the parse tree has much more structure. Pretty printing without macro expansions is easy, the pretty printer simply does not follow the special macro-pointers and prints out appropriate information supplied by the special macro call nodes.

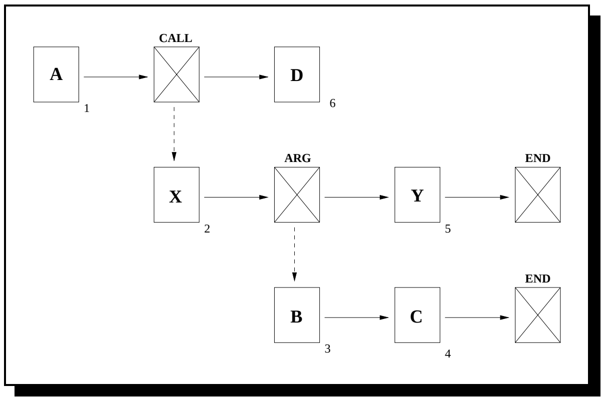


Figure 2.27: Explicit Macro Representation

Since the parse tree for each syntactic list category now is a tree, we can no longer do a fast iterative traversal (that is, without recursion). However, if we stack a pointer to a special node explicitly whenever we meet one, we can continue iteration through the body and retrieve where we are to resume on the top of the stack when we encounter a special end node inserted for this purpose. The reason for the special end-node, as opposed to a null-pointer termination, is that a body of a macro could contain lots of other lists that had nothing to do with macros and were terminated by a null-pointer. In this way, we do not have to be aware of where we are in the parse tree, we just pop from the stack whenever we meet an end node.

All that has been said here for macro calls is of course equally valid for macro arguments.

Of course, this representation requires all subsequent phases to be aware of macros in order for them to traverse them using the explicit stacking as described above. However, there is a way around this drawback.

2.5.5 Weaving

In order for the macros to appear transparent to subsequent phases, we have inserted a separate phase, namely a weaving phase. However, conceptually it should be thought of as a part of the parsing phase.

We have extended all the parse tree nodes so that they all have two pointers to other parse tree nodes. This will give us two distinct ways of traversing the parse tree. One is the one constructed during parsing. The other is constructed by the weaver. The weaver will traverse the parse tree, using an explicit stack for efficiency, and chain all non-macro nodes together as depicted in Figure 2.28. This way, the phases can chose whether they want to see the macro explicitly or not at all. For instance, the pretty printer can take can take the route where the macros are explicit so that it can pretty print them easily. While an analysis phase can take the weaved route ignoring the macros completely.

The weaver will thus give us transparency, of course, for the price of extra memory consumption.

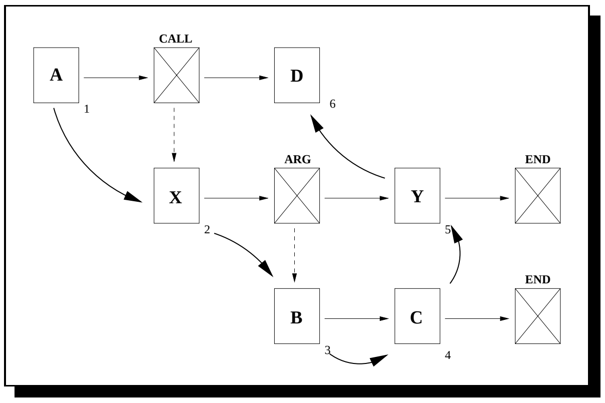


Figure 2.28: Weaving the parse tree

2.5.6 File Inclusion over the Internet

Like most languages we have made a way to lexically include files. This is done through the #**include** directive that will take a string constant designating a file with the effect that the lexer will open and read it. As a special feature, we have extended this so that the programmer may specify an URL (beginning with “http”) in which case the lexer will automatically fetch the file over the Internet at the designated location.

In **C** the preprocessor taking care of lexical inclusion, distinguishes between strings specified in quotes and angled braces. The former will cause the lexer to find the file in the current working directory, while the latter will correspond to some predefined location. Similarly, we have included the two variants of lexical inclusion. However, we have chosen the latter to be individually definable, defaulting to "<**bigwig**> macro central" (http://www.brics.dk/bigwig/macro/). We use the humorous extension "bigmac" for <**bigwig**> macro (library) files.

2.6 Communicating Macro Information

In this section we will look at how the compiler handles and communicates information on macros to the programmer. This will happen in two cases. First of all when the compiler reports errors. Secondly, when the compiler pretty prints the code. Recall that these were our initial demands on the implementation.

If we did not have to communicate macros to the programmer, we could simplify the entire implementation a lot. We would not need any internal representation of macros in the sense that once expanded as plain sub-parse-trees they could be completely ignored. After the parsing, we would thus have a completely normal parse-tree. But since we do, we shall look at how this can be extensively simplified, so that the fewest possible phases have to worry about them.

2.6.1 Error Reporting

Usually, error reporting is done "on the spot" in a compiler, meaning that the compiler signals any such precisely when they are determined.

Recall that we wanted the compiler to report along with the error, the complete trail of macro invocations and arguments corresponding to the erroneous point in order to aid debugging. Since a lot of phases will potentially be reporting errors, using the above scheme, would imply that a lot of phases needed to be aware of the macros.

One idea, however, entails macro transparency for these phases. We shall attach an error field to all the nodes in the parse tree. Whenever the compiler discovers an error, it will report nothing, only assign this field an appropriate error message. We can then see the reporting of errors as a special case of pretty printing, where we silence the pretty printer except for when it reaches non-empty error-fields.

This way, the analysis phases (that may want to report errors) can take the weaved route around the parse tree, ignoring all macros.

Thus, the pretty printer is the only phase that needs to be aware of the macros in order to print out the macro trail of calls and arguments. The pretty printing of errors will be elaborated below.

Recall the dummy service "alpha" from before. If we change the "`t++;`" to "`x++;`", we have an undeclared identifier within two macro invocations. The error printer, will report this appropriately:

```
*** alpha.wig:28:
    Identifier ‘x’ not declared
      in macroargument ‘S’
    in macrocall 'repeat' (alpha.wig:28) defined in [alpha.wig:2]
    in macroargument 'S'
    in macrocall 'repeat' (alpha.wig:26) defined in [alpha.wig:2]
```

2.6.2 Pretty Printing

Pretty printing the source code with all macros expanded is straightforward. It basically amount to a recursive traversal of the parse tree according to the non-terminals of the grammar while printing out appropriately the terminals.

However, as initially mentioned, we want to be able to pretty print the source code with as well as without macro expansions. Without macros would, of course, correspond to what the programmer wrote. To this end, the pretty printer phase, unlike all the other phases, needs to be aware of how the macros are represented internally in order to print them out correctly.

Fortunately, the internal representation of macros is explicit so that we can take the route in the parse tree that allows us to see and thus stack all the macrocalls and arguments. The explicit stacking of macro calls and arguments is done completely as in the weaver in order to let us continue from where we descended into a body or an actual argument in order to print it.

The reason why we use an explicit stack, and not just the (implicit) call stack using recursion, is that it will be available at any point during pretty printing. This has the nice property that whenever we encounter an error, we can run through this stack and see the precise history of macro calls and arguments. This is required in order to print out sensible error messages and it is precisely because of this we can perceive the error reporting as an instance of pretty printing. This way, the programmer and especially the macro programmer can easily trail the error which will facilitate debugging.

Using this scheme we have made a very general pretty printer. It has two parts *pretty* and *print* corresponding to non-terminals and terminals. The pretty part will recursively descend the parse tree according to the grammar. Whenever it needs to print out a terminal or to change the current indentation, it will instruct the "printer" to do so.

The "printer" is a *domain specific terminal printer* meaning that it is capable of printing out information for various domains. Currently, the **<bigwig>** terminal printer supports four domains: `ascii-text`, LaTeX, <HTML> and the special *error* mode.

Actually, the "printer" is implemented by means of function pointers pointing to domain specific functions that will print out information tailored for the particular domain. In order to pretty print for a given domain, all that needs to be done is to set the function pointers to the routines for printing that domain and call the pretty printer with the parse tree.

The terminals have been categorized, so that the printer can print the categories out differently, for instance, emphasizing keywords. The categories are:

- keyword;
- identifier;

- constant entity (bool, int, float, and string);
- macro entity (call, formal, argument use, separator, non-terminal);
- indention (nextline, indent, unindent); and
- whatever (the rest).

Since the pretty printing is done from the parse tree, the output thus does not necessarily include of all the delimiting characters the programmer wrote. For instance, the macro in Figure 2.25 invoked with the empty identifier-list would give us A,X,Y,D and not A,X,,Y,D when printed with all macros expanded.

Pretty Printing HTML

Our most sophisticated output is clearly that of the HTML domain. Here, keywords are printed out in bold face and with a hyperlink to the appropriate documentation page for the particular keyword.

Macro identifiers are by default printed out in blue and with a link to a page containing the definition of the macro. Similarly, all non-terminals in the macro definition header contain a hyperlink to the particular non-terminal in the grammar.

In order to fully utilize the possibilities of HTML, we have augmented the set of function pointers with two pairs, one for identifiers and one for expressions. One will be called before, the other after having printing out the corresponding syntactic category.

Thus, we can make an entire identifier a hyperlink which is exactly what has been done. After symbol-checking we have made sure that all associated identifiers contain unique numbers with the (arbitrary) convention that declarations are positive and usages are the same, only negative. This has been exploited to visualize use/def-chains with hyperlinks from identifier usages to definitions.

Similarly, the type of an expression can be obtained by clicking on it, in which case it will be highlighted (in green) and its type will appear in the browser's status bar. Incidently, this will also explicitly reveal precedences of operators, as one can easily see the extent of an expression.

Finally, we have done the exact same thing with errors so that they will be highlighted in an appropriate red color and with the convention that when clicked on, a small alert-window will pop up stating the error along with of course the trail of macro calls and arguments, if any.

In order for the pretty printer to visualize symbol, type, and error information in these ways, it will necessarily have to print information that has been symbol-type- and otherwise checked. To this end, when printing without expansions, we do not let the pretty printer print information from the *amaenv*. Rather, we make sure, during parsing, to keep a pointer from the *amaenv* to the copy of the actual macro argument, if there is exactly one such, so that we can follow this one when printing macro actual arguments. Unlike the ones in the *amaenv*, these are part of the parse tree and have been through the various checking phases and thus contain the relevant information. Clearly, we can only do this if the macro argument has exactly one macro usage. If there are no usages, the actual will never get analyzed, and if there are more than one, the different usages may contain different type, symbol, and error information.

However, maintaining such a pointer to the actual argument by assigning it when the macro call is constructed is not enough. It does not work with nested macro calls. A correctly handled macro call might later get copied because of an outer macro call, messing up the last usage pointer. To this end, we—much like stop-and-copy garbage collection—install a *forward* pointer in the argument back to the copied *amaenv* that points to the argument. Whenever we subsequently copy the argument and establish its location, we can go back and update the call's *amaenv* so that it will point to the copied argument.

For this reason, there is a small difference between the two **repeat** macros presented in Figures 2.14 and 2.15. Since the first one has two usages of its statement argument S, we cannot visualize it, whereas for the second we are able to visualize the actual with type, symbol, and error information. Because of this (and the fact that there is no blow-up in the size of the parse-tree), we prefer the second.

To aid service development, we have made a useful feature in the compiler. If so instructed, it will upon compilation cause the browser to autoload the pretty printed HTML source code. In this way, the programmer can easily find the errors—especially if there are macros involved as the body of a such is no further than a click away.

Also, when a service is installed, the compiler can be made to pop up a (web interfaced) service management page in the programmers browser.

2.7 The Split Feature

We have generalized the definition of macros, so as to allow multiple macros bearing the same name, but under certain conditions. Clearly, we cannot cope with two macros that are identical except for the body. They need to agree on the result type and on all arguments, both the name and type, up until the point where they eventually differ, and this must be on either a separator or a token—that is, one of them has a separator or a token the other does not have. This is for instance the case with the two macros **si/si-sinon** (French for **if/if-else**) in Figure 2.29, that *split* on the identifier **sinon**. Incidently, it is in this way possible to "redefine" the entire syntax so as to permit the programmer to program in, for instance, French (see http://www.brics.dk/bigwig/macro/le.bigmac for an example of this).

We cannot have two macros that are identical up until a point where they differ on the types of an argument as is the case with the two first macros in Figure 2.30. The parser would not in general be able to *guess* which one to parse without back-tracking. The exact same thing goes for two macros, one of which was terminated while the other had a formal, as with the first and third macros in Figure 2.30.

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

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

Figure 2.29: Macro Split Example: **si/si-sinon**

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

macro <stm> si ( <exp E> ) <exp E> ::= {
    ...
}

macro <stm> si ( <exp E> ) ::= {
    ...
}
```

Figure 2.30: Illegal Splitting

While parsing a macro header, the parser will now see if another macro by that name already existed. If so, it will move along the macroparam chain and continually verify that the new macro complies with the arguments of the one already defined. When they eventually split (on a separator or a token), it will begin construction of a new macroparam chain corresponding to the differing part of the new macro. This chain will be added to the separator/token macroparam node on which they differed in the form of a *split-pointer* to the new macroparam chain.

Whenever the parser reaches a macro invocation, it will continue as usual through the macroparam chain while collecting the actual macro arguments. Once it reaches a macroparam node where it has two possibilities (this can only be a separator or a token), it will check the next token to see if it is the specified separator or token. If this is the case, it will continue along the macroparam chain, otherwise it will follow the split-pointer and thus proceed along the alternate macroparam chain.

This split feature is exactly the reason why the line number and filename information is placed in the last node in the macroparam chain and not in the macro definition node (see Figure 2.24). The parser does not know which macro it is parsing until it has actually reached the end node.

The split feature can thus be exploited to make a set of macros with different arities as in Figure 2.31. However, this can only make macros with bounded arities.

```
macro <exp> and () ::= {
    true
}

macro <exp> and ( <exp E1> ) ::= {
    E1
}

macro <exp> and ( <exp E1> , <exp E2> ) ::= {
    (E1 && E2)
}

macro <exp> and ( <exp E1> , <exp E2> , <exp E3> ) ::= {
    (E1 && E2 && E3)
}
```

Figure 2.31: A Group of Macros with varying arities

2.8 Future Work

HTML macros

There are many ideas for future work. First of all, it would be useful if the macros were extended to cope with **html**-macros. Such macros should probably be invoked in a begin/end tag-style as would thus allow the programmer to define his own **html**-tags. For instance, the programmer could define a macro **mystyle** that would cause anything written between <**mystyle**> and </**mystyle**> to appear in bold, italic, and blue.

Flexible Functions Syntax

Functions could be given the same flexible syntax, where all arguments are assumed to be expressions and instead hold type information about the argument (as in <**int**>). Similarly, the result type would be specified where the macro holds its result non-terminal type. The body of such a function would of course still be a statement. The next step in the generalization of functions would perhaps be to support different parameter mechanisms like <*int>, <"int">, and <'int'> for *call-by-reference*, *call-by-name*, and *call-by-need*, respectively. Being a completely orthogonal augmentation, this really has nothing to do with macros.

Type Annotations

Another idea has to do with type annotations to the macro definition header. One could allow the programmer to place type restrictions on expression arguments that in turn would be verified by the type checker. In this way, the programmer would get even more sensible errors, when attempting to use a macro in an unfortunate way.

Macros with Requirements on Usage

This could be generalized even further. One could extend our work, allowing constraints of perhaps varying nature to be defined along with the macros. The compiler would, in turn, check (or rather analyze) the utilizations during compilation accepting only usages that comply with the specified demands.

A Front-end Generator—*MetaFront*

By far the most prominent idea is the construction of a meta parser that would take the specification of a grammar as input and produce an entire compiler front end. We have seriously considered constructing such a tool, **MetaFront**, that should inherently support macros on all specified non-terminals, pretty printing for various domains, and all the other ideas presented in this chapter. We would also like for the tool to support the ideas presented in YakYak [2]—that is, the possibility of adding logical side-constraints to all productions in the grammar that, in turn, would be enforced by the parser. Clearly some work also needs to be done with regards to classifying exactly which grammars we are able to embed with macros.

2.9 Conclusion

We have seen how a general macro language could be designed and implemented. Also, we have seen how the macros provide a means for extending the language and how the syntactic flexibilities allowed for such extensions to be transparent.

Additionally, we have seen how various design choices, have impacted on the overall behavior of our macro language. For instance, how the immediate treatment of macro calls in macro definitions yielded static macro scope rules.

Furthermore, we have seen how to error printing can be expressed as a special case of pretty printing, enabling us to ignore the macros in all phases but this one. And how to aid macro debugging by providing along with the complete trail of macro calls and arguments.

Once again, we state that the ideas from the macro language presented are in no respect specific to <**bigwig**> and can be incorporated into the vast majority of other programming languages.

We are now ready to see how the two independent language introduced can be combined to produce highly sophisticated concurrency control abstractions.

Chapter 3

Synthesis

3.1 Introduction

Both the constraint language and the macro language are interesting in their own right, but when put together they form something perhaps even more so. This will allow for complex concurrency control mechanisms to be defined and used as if they were already in the language itself. Thus, programmers that are unexperienced with concurrent aspects, will be able to handle them through the use of various macro libraries. The synthesis of such mechanisms will be the topic of investigation in this chapter.

Instead of presenting one big example, we shall exhibit a couple of small but illustrative ones. It should be clear that all of these easily scale to large scenarios.

3.2 A Chain of Development

In this section we will develop several well-known concurrency abstractions. They will constructed so as to gradually increase in sophistication and all of them will be constructed in terms of the ones previously introduced. This will thus also constitute an intriguing example of how a language can be "evolved" through the use of macros.

All examples have different flavors. Some will appear as they were primitives, others as predicates, and others still will seem as if they were entire concepts.

3.2.1 Allow/Forbid-when

As previously promised, we were going to introduce the **allow-when** and **forbid-when** constructs through the use of macros. As in Section 1.4.2, the first one could be defined in terms of the **restrict-by** construct and the latter in terms of the first (see Figure 3.1).

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

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

Figure 3.1: Adding two primitives: **allow-when** and **forbid-when**

3.2.2 Mutex

Now the time has come to add another interesting macro, namely **mutex** as considered in Figure 1.13. As shown in this figure, it can easily be constructed using **forbid-when**. See Figure 3.2 for the definition of the macro.

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

Figure 3.2: **mutex** in terms of **forbid-when**

An Example Service

With this new construct, we are now ready to synthesize our first interactive web service from the macros and constraints. We have placed all the macros introduced so far in the library: "http://www.brics.dk/bigwig/macro/thesis.bigmac", which will be included by the service. The service (in Figure 3.3) uses two macros directly, the invocations of which we have underlined. Thes service consists of four constituents at top level: a constraint part, a document declaration, and two sessions. The constraint part declares two labels A and B which are constrained by the macro **mutex**. The next part is a global variable declaration of a constant html variable called Doc. The document has two holes lab and no that can be plugged with various information. This is followed by two sessions sesA and sesB, that will be repeating a simple task until the client checks a quit-now radio button. The session sesA will repeatedly ask the controller for permission to pass label A. When permission to continue is granted, it will increase its local counter variable i to count the number of iterations and show the document, with "A" and the value of i plugged into the appropriate holes. The other session, sesB, will do the same thing, only for B.

Due to the **mutex**(A,B) safety requirement, this service, will never be able to pass two A labels without having passed a B in between. The second request to pass the label A will be delayed until this is a legal thing to do—that is, until someone has passed a B.

3.2.3 Region

The idea of separating two mutex-related **wait** statements as above, hence rendering them independent, although doable, it is perhaps not the most representative usage. For real services, such two **wait** statements are more likely to occur around critical regions, implementing exclusive access. Clearly, the programmer does not want to explicitly add the **mutex** requirement and the related **wait** statements each time he require exclusive access. Instead, it would be nice if we were able to lift this so as to provide a higher level of abstraction for the programmer. A level of abstraction that would hide away all the details of labels and constraints. Once again we put the macros to work.

We shall employ the package simulation ideas described in Section 2.4.3, defining a declaration macro and a usage macro (see Figure 3.4).

Clearly, this is useful to programmers that do not want to deal with concurrency aspects themselves. The programmer only needs to "declare" a region, invoking the macro **region**. Subsequent **exclusive** statements with the same "declared" identifier will operate on the same region, granting exclusive access to their statements within, so to speak.

The flavor of these two macros is perhaps somewhat different from what we have seen thus far. This is really as if we have extended the <**bigwig**> language with a whole new *concept* that was not there before—the concept of regions.

Assume, for instance, we were to protect a global variable, say x, using the region concept. The *identifier* supplied to the region, declaring it, could very well be x itself. Since the macros operate on concatenations of x and A or B, there is no clash with the declaration of x, regardless of how it is defined.

3.2.4 Resource

In this section, we shall look at an even more sophisticated example. Recall the Reader/Writer problem from Section 1.5 and the solution to the simplified case without priorities in Figure 3.5. Using both the ideas and the macros presented above, we shall define the concept of *resources*. There are two ways of accessing a resource, namely that of *reading* and *writing*—corresponding to the reader-writer problem. Giving the writers priority is handled by introducing yet another **trigger**, this time counting the number of sessions that would like to write but not yet have. The readers are thus only allowed when the label R∼P has never been *taken* or when this new trigger has *fired* at some point after which no one has transgressed R∼P. This will give the writers permission

```
#include <thesis.bigmac>

service {
   constraint {
      label A,B;
      mutex(A,B);
   }

   const html Doc = <html>
      <h1><[lab]><[no]></h1><hr>
      Quit Now? (
      Yes <input type=radio name=quit value=true> /
      No <input type=radio name=quit value=false checked>
      )
   </html>;

   session sesA() {
      int i;
      bool quit;

      repeat {
         wait A;
         i++;
         show plug Doc[lab = "A", no = i] receive [quit = quit];
      } until (quit);
   }

   session sesB() {
      int i;
      bool quit;

      repeat {
         wait B;
         i++;
         show plug Doc[lab = "B", no = i] receive [quit = quit];
      } until (quit);
   }
}
```

Figure 3.3: A Service Example: mutex

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

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

Figure 3.4: **region** in terms of **mutex**

over the readers, as no new readers are allowed when someone wants to write. Readers are of course still blocked while someone is in the process of writing.

3.2.5 Protected

Now we have reached the final example in this gradual language evolution chain. We shall now make a minor extension to the **resource** macro above, providing us with a declaration modifier. As can be seen in Figure 3.6, we have now made it possible to transparently get the reader/writer functionality as if it was a direct property of the variable declared, simply by sticking the "keyword" **protected** in front. This is completely as, for instance, one does with the **const** modifier when declaring constant variables. All this **protected** macro does is, upon invocation to declare the associated variable using the specified name and type and invoke the **resource** macro.

One might want to consider using the split feature here, adding a similar yet slightly longer macro to support initialization expressions also.

3.3 An Example Service: RW

The example service presented in Figure 3.7 uses the protected macro and will when run nicely illustrate the reader-writer problem. As one can see x is (transparently) declared as a protected integer, meaning that we can use the **reader** and **writer** macros on it, protecting it accordingly.

When the reader session R is run, it will flash a message saying that it is waiting onto its associated html reply file (see Section 4 for further details). This way, if it is blocked by writers, the client will see this appropriate message

```
macro <toplevel_list> resource <id R> ; ::= {
   region R;

   constraint {
      label R∼enterR, R∼exitR;
      label R∼P;

      trigger R∼noR when #R∼enterR == #R∼exitR;
      trigger R∼noW when #R∼P == #R∼B;

      allow R∼enterR when never (R∼P) ||
         (is t: R∼noW(t) && (all tt: t<tt => !R∼P(tt)));
      allow R∼A when never (R∼enterR) ||
         (is t: R∼noR(t) && (all tt: t<tt => !R∼enterR(tt)));
   }
}

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

macro <stm> writer (<id R>) <stm S> ::= {
   {
      wait R∼P;
      exclusive (R) S
   }
}
```

Figure 3.5: **resource** in terms of **region**

```
macro <toplevel> protected <type T> <id V> ; ::= {
   T V;

   resource V;
}
```

Figure 3.6: **protected** in terms of **resource**

```
#include <thesis.bigmac>

service {
   protected int x;
   int y;

   session R() {
      flash <html>Waiting to Read...</html>;
      reader (x) {
         y = x;
         show <html>Reading...</html>;
      }
      exit <html>Done Reading!</html>;
   }

   session W() {
      flash <html>Waiting to Write...</html>;
      writer (x) {
         x = y + 1;
         show <html>Writing...</html>;
      }
      exit <html>Done Writing!</html>;
   }
}
```

Figure 3.7: A Service Example: RW

(in the browser). If, however, it is not blocked, the session will enter the reader part, thus blocking writers, read x and show a message stating that it is busy reading. We intend this message to simulate the actual reading and thus give the client time to experiment, starting new sessions before letting the session terminate "reading". Similarly for the writing session W.

Please note, that this is a highly dangerous example! The reason being, that the client might decide to "leave" the session when it is occupying a resource, blocking the entire service. An even worse thing to do, would be to **exit** or **break** computation while blocking such a resouce. In general one should never place **show**s or **exit**s within the reader- or writer-statement. The example above is intended only for illustrating the reader-writer problem only. The idea of placing **show**s within reader-writer statements should not be copied, unless fully intended.

3.4 Other Examples

3.4.1 Alternation

Before proceeding, we shall introduce a useful macro (see Figure 3.8), **more-recently-than**, taking two identifiers (labels) and evaluating to *true* if and only if the last of the two on the string seen so far is the first one. That is, if the first is more recently seen than the second.

```
macro <formula> more - recently <id A> than <id B> ::= {
    is t: A(t) && (all tt: t<tt => !B(tt))
}
```

Figure 3.8: more-recently

On session level this is not very interesing. On a global service level, however, this has some concurrent aspects. To solve this, we shall introduce two labels, A and B. Through the use of constraints we make sure that initially only A is enabled and that passing an A, will enable B and disable A, and vice versa. Clearly, at any point in time exactly one of them is enabled, the other not. We alternate between the two statements by branching on the state of the controller, using the generalized **wait**. The result can be seen in Figure 3.9.

3.4.2 Synchronization

The next example we shall look at is the task of synchronizing two sessions. This can be done by letting both sessions pass two labels in sequence, a requst-for-synchronization label and an acknowledge-synchronization label, whilst restricting them appropriately.

```
macro <toplevel> alternation <id A> ; ::= {
   constraint {
      label A~first, A~second;
      forbid A~first when
         more-recently A~first than A~second;
      forbid A~second when
         more-recently A~second than A~first ||
            never(A~first);
   }

macro <stm> alternate ( <id A> ) <stm S1> and <stm S2> ::= {
   wait {
      case A~first:
         S1
         break;
      case A~second:
         S2
         break;
   }
}
```

Figure 3.9: Alternation

Before specifying the actual macros, we shall as with the alternation example, introduce a macro **second-latest** (see Figure 3.10), that will help us. This time, however, the macro introduced is somewhat specialized, in the sense that it is probably not much useful, other than for building the synchronization macro.

```
macro <formula> second - latest of ( <id A> , <id B> )
                is <id C> ::= {
    is t1: is t2: t1<t2 &&
       C(t1) && (A(t2) || B(t2)) &&
          (all t3: t1<t3 && t3 != t2 =>
             !(A(t3) || B(t3));
}
```

Figure 3.10: A rather specialized macro: **second-latest**

We are now ready to define the synchronization macros, however, we shall refrain from giving a detailed explaination of the constraints involved.

```
macro <toplevel> synchronization <id S> ; ::= {
    constraint {
       label S~Req, S~Ack;
       forbid S~Req when
          second-latest of (S~Req, S~Ack) is S~Req;
       forbid S~Ack when
          second-latest of (S~Req, S~Ack) is S~Ack;
    }

macro <stm> synchronize ( <id S> ) ; ::= {
    {
       flash <html>Synchronizing, please wait...</html>;
       wait S~Req;
       wait S~Ack;
    }
}
```

Figure 3.11: Synchronization

The above really only applies to cases where there are only two sessions, but this is probably true for a wide varity of services. One could well imagine a whole library dedicated to two-party communication, with lots of special constructions and primitives that are relevant in such scenarios.

3.5 Conclusion

We have seen how the language could be gradually extended with new concepts. In the end, the language evolution chain of macros developed, counted six levels:

$$\textbf{restrict} \Rightarrow \textbf{allow} \to \textbf{forbid} \to \textbf{mutex} \to \textbf{region} \to \textbf{resource} \to \textbf{protected}$$

This can also be seen by clicking on these macros in the HTML pretty printed source code for the example service above, RW.

Also, the macros introduced illustrate different "flavors". The first macros, **allow-when** and **forbid-when** are reminiscent of real language *primitives*, whereas **mutex** seems more like a *predicate*. The **region** and **resource** macros are more of the order of actual *concepts*, while the **protected** macro is used as a declaration *modifier*.

More importantly, we have seen how various concurrency control mechanisms could easily be synthesized. Similarly, lots of other mechanisms can be constructed. We have from the macros and constraint language obtained highly sophisticated mechanisms that can be used as if they were originally part of the <**bigwig**> language.

Chapter 4

The Runtime System

Bibliography

[1] Brian W. Kernighan, D. M. R. *The (ansi) C Programming Language.* Prentice Hall, Inc., 1978.

[2] Niels Damgaard, M. I. S., and Klarlund, N. YakYak: Parsing with Logical Side Constraints, 1998.

[3] Sandholm, A. Decentralized Control of Discrete-Event Systems using Monadic Second-Order Logic. ., August 1998.

[4] Sandholm, A., and Schwartzbach, M. I. Distributed Safety Controllers for Web Services. In *FASE 98* (1998), pp. 270–284.

[5] Thomas H. Cormen, C. E. L., and Rivest, R. L. *Introduction to Algorithms.* McGraw-Hill, Inc., 1989.