Partial Evaluation of an Object-Oriented Imperative Language

Morten Marquard *Bjarne Steensgaard*

Department of Computer Science - University of Copenhagen
Universitetsparken 1, DK–2100 Copenhagen Ø, Denmark
Electronic mail: `marquard@diku.dk` or `rusa@diku.dk`

April 30, 1992
(printed September 25, 1992)

Abstract

The development of a fully automatic online partial evaluator for a simple object-oriented imperative language is described. The source language is a language invented for the project. This is to our knowledge the first description of a partial evaluator for an object-oriented imperative language.

Historically online specializers have had problems having good termination properties while generating sufficient specialization. A technique is described, which to some extent solves these problems. An almost identical technique has been developed simultaneously and independently at Stanford University.

Online imperative partial evaluators generate *explicators* at program points where the program state is generalized during the symbolic computations. Program states are generalized at e.g. the entrance of loops with what is known as "static variables under dynamic control". Explicators are assignment statements, that perform the assignments, which were at first thought reducible, but have to be performed anyway. A new technique for generating *explicators* is described. The technique solves the problems connected with variables being enclosed in objects and gives the residual program a more natural structure than one you get using the previously described techniques for generating explicators.

A novel technique for partially unrolling loops during specialization is also described.

Contents

List of Figures

List of Tables

List of Examples

Chapter 1

Introduction

This report describes our work constructing an online partial evaluator for an object-oriented imperative language.

Writing a partial evaluator for an imperative language is not an easy task. In object-oriented languages the state is enclosed in objects. One of the reasons for undertaking the project was the hope that the enclosing of state in objects would make partial evaluation easier than it is, using an ordinary imperative language as the source language. This report illustrates that this is only partly the case, as new problems appeared when performing partial evaluation of object-oriented languages. These problems include: (1) forced use of method names from the source program in a number of special cases, (2) generation of explicators is complicated by objects enclosing their state, (3) objects are first class higher order values, and (4) the mere size of an object-oriented language.

The developed partial evaluator uses online techniques rather than offline techniques. All the other partial evaluators developed at DIKU use offline techniques. The choice of online techniques was almost forced by the fact that the language has *both* assignments and higher order values (objects). With higher order values, it may not be possible at the time of partial evaluation to determine exactly what code is to be executed when a method is invoked (a function or procedure call). Since the executed code may have side effects not only locally but also globally due to invocation of other methods, everything could very easily degenerate to be annotated as *unknown* or *dynamic*. Using an abstract interpretation that only throws away a minimum of information can help tame this behavior. This does however imply using *online* techniques.

Online partial evaluators have until recently had problems being aggressive enough to perform sufficient specializations and at the same time lenient enough to terminate for a sufficiently large class of source programs. We have developed a technique to overcome this problem. The group at Stanford University working with the online partial evaluator Fuse has independently developed almost the same technique as ours but for another type of programming language. We think our techniques are superior to theirs with a slight margin.

Object-oriented languages are typically rather complex compared with e.g. applicative languages and so is the resulting partial evaluator. The development of the partial evaluator is described by a step by step development, which hopefully will make it easier to

understand the final algorithm.

The first step is a formal description of the source language. The second step is an algorithm for abstract interpretation of programs, where we do not throw any information away. This algorithm will fail to terminate for a very large class of programs. The third step is the development of an improved abstract interpretation algorithm, which will terminate for a sufficiently large class of programs. The class of programs, that the improved algorithm may fail to terminate on, is the class of programs, which will fail to terminate for any instantiation of the unknown (dynamic) input and the programs that contain a region with an infinite loop regardless of such instantiations. The region containing the infinite loop does not necessarily have to be executed. The last step is to add generation of program fragments to the algorithm. These program fragments can then be combined by a postphase, which prints the residual program. Further more, the developed partial evaluator is improved in two more steps to improve the quality of the residual programs.

During the work developing the partial evaluator we have developed a novel technique for generating *explicators*, which are needed in online partial evaluators of imperative languages [Meyer 91]. The fact, that all variables are enclosed in objects, makes the generation of explicators more difficult than in ordinary imperative languages. The developed technique overcomes these difficulties. The developed technique gives the residual program a more natural structure than it would have had, using the techniques described in e.g. [Meyer 91].

The work combines knowledge from two areas of computer science, which are rarely combined. Most scientists working with program transformations such as partial evaluation will not know much about object-oriented programming, and most scientists working with object-oriented programming will not know much about program transformation. The report consequently contains short introductory descriptions of the two areas.

We would like to thank our supervisor Professor Neil D. Jones for his supervision of this project. Despite the long tradition at DIKUof constructing offline partial evaluators, he accepted that we wanted to use online techniques. He supplied us with valuable references to relevant literature in the most critical phases of this project. He is also one of the founders of the Topps-group at DIKU, which we are proud to be members of. The research environment in this group is simply incredible. Finally we would like to thank Associate Professor Eric Jul for introducing to us the language Emerald, which has served as the basis for the language used in this project. He also kindly supplied us with an office of our own in the final period of working with this project. This has helped us tremendously. Grethe Hejlesen has read a couple of chapters of this report and helped correct the prose. We thank her for her valuable time.

Chapter 2

What is partial evaluation?

Partial evaluation is also called *program specialization*. Given a (*source*) program and part of the input to the program, partial evaluation is the process of constructing a new (*residual*) program that behaves as the original program given the same input and the static input used to perform the partial evaluation. That it is possible to construct such a program is given by Kleene's s-m-n theorem [Kleene 52]. Normally the expectations to the residual program is that it will execute at least as fast as the source program. For many programs, the result of partial evaluation is a residual program that executes an order of magnitude faster than the source program. The justification for partial evaluation is just these speedups.

Partial evaluation can be used to make compilers from interpreters. The compilers may not be industrial strength compilers, but in evolving environments as the object-oriented community where there is currently being developed a lot of compilers for experimental languages, partial evaluation may be a valuable tool.

In this chapter we will give a more thorough introduction to what partial evaluation is, what the theory people say you can do with partial evaluation, and what partial evaluation actually is used for.

2.1 Specializing Programs with Partially Static Input

As mentioned, partial evaluation is the technique used to generate residual programs (also called *specialized programs*) given a source program and part of the input to this program. To illustrate what this mean, we can consider the functional program listed below (written in ML [Milner 90]) that implements the power function, computing the value of the first argument lifted to the power of the second argument. The example can be dated back to [Ershov 77].

```
fun power(x,n) =
   if n = 0 then
      1
   else if even(n) then
      sqr(power(x, n div 2))
   else
      x * power(x, n-1)
```

Partial evaluation of this program with the information that the second argument is 5 generates a residual program that implements the "power of 5" function. An example of such a residual program is:

```
fun power5(x) =
   x * sqr(sqr(x))
```

This program could in fact have been generated by a partial evaluator, if such a partial evaluator exists (to our knowledge there are no reported partial evaluators for ML). The techniques used to achieve this result would be *symbolic computation* to eliminate the tests in the residual program, and *unfolding* to eliminate the recursive calls. These are two of the three major techniques for partial evaluation:

1. Symbolic computation
2. Unfolding
3. Program point specialization

Program point specialization is the generation of specialized program points. In functional programming languages, program points are often function definitions. In imperative languages, program points are basic blocks in simple languages, and procedures, functions, and statements in structured languages. If the power function as defined above was only a fragment of a program, where the power function at one point was called with 5 as the second argument, and at another point called with 7 as the second argument, then specialization would lead to generation of two specialized functions implementing respectively the "power of 5" and the "power of 7" functions. This is program point specialization. If there can be more than one specialized function in the residual program for each function in the source program, the specialization is said to be *polyvariant*. If there can only be one residual function for each function in the source program, the specialization is said to *monovariant*. Monovariant specialization is not very useful.

It is rather obvious that the shown specialized version of the power function will be faster than the general function. If the first argument had been the known argument rather than the second argument, then the specialized function would not be much faster than the specialized version (if faster at all). This because no computations could be performed at the time of specialization. The relative speed of the specialized programs compared with the source program thus depend heavily on the problem and on what arguments are known at specialization time. For the kind of programs that does a lot of checking or dispatching on part of the input (e.g. interpreters, general pattern matching routines, ray tracers, etc.) a considerable speedup (runtime of residual program compared with runtime

of source program) can be observed. A speedup between 5 and 50 is normal. Recently research has commenced on techniques to predict the speedup before partial evaluation is performed [Andersen 92].

There is a question to consider when talking about partial evaluation. What does it mean that two programs behave the same way? The interesting issue in this context is: when does the specialized program behave the same way as the source program (of course with the same static input as the specialized program is generated with respect to.) The answer to this question influences the design of a partial evaluator. If both programs terminate, then the question is rather easy to answer. If the source program terminates then the specialized program should also terminate. But what if the source program does not terminate for some input, where the specialized program does? Is this acceptable behavior? If the answer is no, then the partial evaluation is called *strict*. If the answer is yes, then the partial evaluation is called *nonstrict* [Bondorf 90b].

A nonstrict partial evaluator may be more aggressive in its attempt to perform computations at the time of specialization than a strict partial evaluator. Programs specialized with nonstrict partial evaluators may show better speedups than programs specialized with strict partial evaluators. Nonstrict partial evaluation may change the run-time complexity of the residual program to be better than the source program's.

In the Department of Computer Science at the University of Copenhagen (DIKU), much work has been put into constructing *self-applicable* and *automatic* partial evaluators, also called *autoprojectors*. Self-applicable partial evaluators can as the name suggests be applied to themselves. This implies that the partial evaluator is implemented in the source language of itself (this will be explained more elaborately in the next section). Automatic partial evaluators are automatic in the sense that they do not need programmer annotations of programs to be specialized.

2.2 The Futamura Projections

The theoreticians of Computer Science have pointed out a number of nice things that are possible with self-applicable partial evaluators. It is possible to generate compilers from interpreters and even to generate compiler generators. The rules stating how to do this is called the *Futamura Projections* after the Japanese Computer Scientist Yoshihiko Futamura that first formulated them [Futamura 71].

The basic idea is that an interpreter is just a program that takes two arguments; the first argument being the program to interpret, and the second argument being the input (arguments) to this program. Specializing the interpreter with respect to a program yields a program that behaves just as the program to be interpreted. The clue being that the computations in the interpreter that does syntax checking and dispatching on language constructions can be performed at the time of specialization. The residual program should thus ideally only perform the real computations performed by the program to be interpreted. In other words, the partial evaluation process has performed a compilation of the program from the language that the interpreter implements to the language generated by the specializer.

This is the first Futamura projection. If the source language of the partial evaluator,

MIX, (the name has historical reasons) is $\mathcal{S}$, the residual programs are programs of the target language $\mathcal{T}$, the partial evaluator is implemented in the language $\mathcal{M}$, and the interpreter, int, (that must be written in $\mathcal{S}$) implements the language $\mathcal{L}$, then the first Futamura projection can be written as:

$$\mathrm{p}_{\mathcal{T}} = [\![MIX]\!]_{\mathcal{M}}\ \mathrm{int}_{\mathcal{S}}\ \mathrm{p}_{\mathcal{L}}$$

where we have used subscripting to indicate the language. The semantic brackets $[\![\ ]\!]$ are here used to denote the meaning (or execution) of a program.

The second and third Futamura projections are based on the important observation that an implementation of a partial evaluator is a program that takes two arguments; the first argument being the program to be specialized, and the second argument being the static input parameters to the program. If the partial evaluator is written in its own source language ($\mathcal{M} = \mathcal{S}$) then the first argument to the partial evaluator can be the partial evaluator itself. This is called *self-application.*

If the first argument to MIX is MIX itself and the second argument is an $\mathcal{L}$ interpreter, then we observe that the "inner" MIX only needs a program $\mathrm{p}_{\mathcal{L}}$ to generate a specialized program $\mathrm{p}_{\mathcal{T}}$. From this we can see that the result of applying MIX to MIX and an interpreter is a *compiler* that takes any program written in $\mathcal{L}$ and generates the same program written in $\mathcal{T}$. This is the second Futamura projection and can be written as:

$$\mathrm{L2Tcomp}_{\mathcal{T}} = [\![MIX]\!]_{\mathcal{S}}\ \mathrm{MIX}_{\mathcal{S}}\ \mathrm{int}_{\mathcal{S}}$$

where "L2Tcomp" denotes a compiler from $\mathcal{L}$ to $\mathcal{T}$.

The third Futamura projection illustrates what happens if MIX is given itself as both first and second argument. We observe that the "second" MIX (first argument to "first" MIX) only needs an interpreter as second argument to generate a compiler. The result of applying MIX to MIX and MIX is thus a *compiler generator* that given an interpreter written in $\mathcal{S}$ implementing the language $\mathcal{L}$ generates a compiler from $\mathcal{L}$ to $\mathcal{T}$. This is the third Futamura projection:

$$\mathrm{cogen}_{\mathcal{T}} = [\![MIX]\!]_{\mathcal{S}}\ \mathrm{MIX}_{\mathcal{S}}\ \mathrm{MIX}_{\mathcal{S}}$$

An interesting feature of cogen is that the result of applying cogen to MIX is cogen itself! This may seem to be an unusable feature, but it may actually be usable if you change to a new version of MIX.

The three Futamura projections are just special cases of Turchin's metasystem transitions[1]. A recent paper describing the use of metasystem transitions is [Gluck 91].

2.3 Offline vs. Online Partial Evaluation

A partial evaluator can be constructed after two basically different strategies. Partial evaluators constructed after these strategies are called either *online* or *offline* partial evaluators.

[1] We have not been able to find a reference to the correct paper.

An offline partial evaluator is a multi-stage program. There is a prephase, a specialization phase, and usually also a post-phase. The algorithms are offline because the specialization algorithm must be preceded by an analysis yielding information that guides the specialization.

In the prephase a *binding time analysis* is performed. This analysis divides language constructions into two categories stating that they will either be *static* (known) at partial evaluation time and thus can be reduced by the partial evaluator, or they will be *dynamic*, which means that they may not be reduced at the time of partial evaluation. We will not describe the actual workings of a binding time analysis. The interested reader is referred to [Henglein 91], [Bondorf 90b] [Bondorf 90a], [Consel 90], [Bondorf 88], [Nielson 88]. Offline partial evaluators for higher order languages does usually also have a *closure analysis* in the prephase. The result of a closure analysis is some information about what closures may be applied at the application points in the program. A closure analysis is a prerequisite of a binding time analysis for higher order languages if the binding time analysis is performed by abstract interpretation.

The specializing phase of offline partial evaluators uses the binding time information to aid the specialization. Language constructions that are marked as static do either not appear at all in the residual program or appear in a reduced form. Language constructions that are marked as dynamic are copied into the residual program (also called being *left residual*). The job of the specializer is thus not to decide when to reduce an expression and when to leave it residual since that is determined by the binding time analysis. The specializer performs the symbolic computation, the unfolding and the program point specialization guided by annotations of the program to be specialized. That the specialization is guided by the binding time annotations of the source program means that a good separation of binding times is necessary to obtain good results. Often programs to be specialized have to be written and rewritten with the binding time annotation in mind.

The postphase of offline partial evaluators usually performs last-minute optimizations, reductions, and unfolding. Some partial evaluators also perform arity raising in the postphase [Romanenko 90].

Online partial evaluators have no prephase with a binding time analysis. They are one phase specializers[2].

Online partial evaluators have a single phase where the specialization is done and usually also a postphase that prints the generated program and performs some last-minute optimizations. That the specializer is not guided by a binding time separation of the program but has to decide whether to reduce or residualize on basis of actual data may result in better specialization. Also, programs to be specialized do not have to be written with specialization and binding time separation in mind. Online specializers are typically slower than offline partial evaluators because they have to maintain more information on what has already been done to ensure termination.

[2]There has recently been a discussion concerning the definition of online algorithms in the internet newsgroup `comp.theory`. There is plenty of different definitions of what makes an algorithm online. Some of these exclude the existence of general online partial evaluators. There is general agreement that online algorithms may not use global information as e.g. yielded by a prephase analysis. In the partial evaluation community it is generally accepted that a one-phase specializer is called an online specializer.

Until recently, online partial evaluators also had difficulties with at the same time being aggressive enough to generate sufficiently specialized programs and lenient enough to terminate for a sufficiently large class of source programs. These difficulties have more or less been solved by the group of people working with the Fuse online partial evaluator [Weise 91b] (and independently also by the authors of this thesis).

The reason for online partial evaluators being able to generate better specialized programs for source programs that is not written with binding time separation in mind, is that the online partial evaluators do not have any problems with variables that can have a known value at one point of the computations and an unknown value at another point. The offline partial evaluators usually have problems with this due to the granularity of the information yielded by the binding time analysis. The problems can be somewhat solved by more advanced binding time annotations, but the binding time annotations are (safe) approximations that will never be better than the information obtained by performing symbolic computations with the actually occurring values (as done in online partial evaluators).

It has been suggested that offline partial evaluators should be used for imperative languages where variables may change from being bound to a unknown value to being bound to a known value (or the opposite way around) in a single basic block or structured statement [Meyer 91].

It is widely believed that it is not possible to make online partial evaluators self-applicable. This is partially disproved by [Gluck 91]. The paper describes an online partial evaluator that is self-applicable. The partial evaluator has the usual (for online partial evaluators) problem with termination and requires user annotations of programs to control unfolding. No fully automatic self-applicable partial evaluators using online techniques have yet been developed.

2.4 Existing Partial Evaluators

There are a lot of existing partial evaluators. Here we will only mention a few of them. Those we mention have either been fore-runners in the field of partial evaluation, are well known in the partial evaluation community, or have direct relevance to our work.

There also exist partial evaluators for logic programming languages [Bondorf 90c, Sahlin 91], but since they are irrelevant to us, we have omitted description of such partial evaluators here.

Schism

A partial evaluator for a higher order subset of Lisp. Originally required user annotations of the program. Is now fully automatic. Is still a subject of ongoing research. An overview is given in [Consel 88]. It does not perform a separate closure analysis before the binding time analysis, but instead it integrates the two in a polyvariant higher-order binding time analysis [Consel 90]. It handles partially static structures as mentioned in [Mogensen 88].

LambdaMix

A partial evaluator for the untyped lambda calculus. Uses type inference for binding time analysis of variables and occurrence analysis of functions instead of a binding time analysis for variables by abstract interpretation and a closure analysis for functions. An interesting point about this partial evaluator is that the specialization is proven to be correct whenever the binding time annotations of the source program are correct. Described in [Gomard 91b]. The partial evaluator is strict. The specialization does fail to terminate for some programs.

Similix

Probably the most widespread partial evaluator at the moment. The source language is a subset of Scheme. Developed here at DIKUby Anders Bondorf and Olivier Danvy. Uses abstract interpretation in the binding time analysis and closure analysis. Cannot handle partially static structures[3]. A fully automatic nontrivial partial evaluator [Bondorf 90a].

Fuse

An online partial evaluator for a subset of Scheme. Most termination difficulties are overcome while generating sufficient specializations of functions. Fuse has been developed with the programmers in mind, and has successfully been used for a number of applications in the area of scientific programming. Fuse has backends that can generate both Scheme and C code. Fuse can handle partially static datastructures and uses graphs as an intermediate representation of programs. This makes some extra postphase optimizations possible. Relevant references are: [Weise 90], [Ruf 91] and [Weise 91b].

ImperativeMIX

A self-applicable partial evaluator for a simple imperative language with basic blocks (and without procedures). The partial evaluator is described in [Gomard 91a]. An important lesson from this partial evaluator is that a lifetime analysis for variables may be necessary to prevent blowup of the residual program.

2.5 Application Areas for Partial Evaluation

In addition to the generation of compilers and compiler generators, partial evaluation can be used for almost all programs that takes more than one input argument, and where part of the input may be known before the rest.

In the area of computer graphics, partial evaluation has successfully been used for raytracing [Mogensen 86], sorting of polygons [Goad 82] , and bitmap manipulating operations [OKeefe 91]. For the raytracer and the polygon sorting program, partial evaluation where the scene is constant (but not the point of view) yielded speedups between 10 and

[3]Anders Bondorf, one of the authors of Similix is currently working to remedy this.

500 times. For these examples the typical pattern was quite clear that the time for partial evaluation plus the time for compilation and execution of the residual program was a lot less than the execution time for the source program. The used partial evaluators were offline partial evaluators.

In the area of neural networks, partial evaluation has been used to optimize a neuron interpreter to speed up the program. Partial evaluation where the constant parameters was just the topology of the network and the learning rate and momentum resulted in residual programs that were 25–50% faster than the source program. The partial evaluator was an offline partial evaluator for a very small subset of C [Jacobsen 90].

Partial evaluation has also with success been applied to scientific computing. Here the online partial evaluator Fuse has been used. The reported problems handled successfully include the N-body problem (speedup around a factor of 40) and numerical integration using Runge-Kutta integration (speedup around a factor of 7) [Berlin 90]. An interesting observation was that the residual code was more suited for parallelization than the source program code.

Yet another application area of partial evaluation is spreadsheets. Spreadsheet programs contain formula interpreters, and partial evaluation can successfully be used to generate spreadsheets specialized with respect to a specific set of formulas. This is really just a special case of generating a compiled program from an interpreter and the program input (the formulas).

An interesting observation is that many famous algorithms have been rediscovered by partial evaluation of very simple algorithms performing the same task. If one expect such behavior, then partial evaluation can be used to generate the often complicated algorithms from simpler ones. This feature has been exploited in the company ICAD to generate fast specialized bitmap manipulation algorithms from a very general algorithm [OKeefe 91].

Chapter 3

Object-Oriented Programming Languages

This purpose of this chapter is to give a short introduction to the object-oriented programming paradigm. This introduction contains a brief explanation of several buzz-words used in the object-oriented community.

The object-oriented paradigm has emerged as a promising paradigm for programming in the large, i.e. analysis, design, implementation and maintenance of large scale systems. The reason for this success is probably that the only entities handled in the systems are objects. Objects have some properties and can interact with other objects. Each object is an autonomous entity with private data and a collection of methods. The object's internal state can only be modified and inspected by invoking methods in the object. The invocation of a method in an object is often called *sending a message* in the object-oriented community. The message supposedly sent contains information on which method to invoke and the arguments of the invocation. We will use the term *invoke* throughout the report. The uniform way of looking at programming makes it easy to write programs. Some people claim that when using the same paradigm and concepts in the whole life-cycle of a system there will not be the traditional big steps from analysis to design to implementation etc.

The object-oriented philosophy originate from the language Simula [Nygaard 70], designed in the late 60's. However, the object-oriented paradigm did not become widespread until the late 70's when Smalltalk [Xerox 81] was developed. In Smalltalk, everything from integers to language constructions are objects (also e.g. an if-construction).

Object-oriented languages are in widespread use today. Many of the languages used are hybrid languages, i.e. existing conventional languages where object-oriented features has been added later, e.g. C++ [Stroustrup 86,Stroustrup 89], Flavors [Moon 86] and CLOS [DeMichiel 87,Bobrow 86]. However, some pure object-oriented languages have been designed from scratch, e.g. Trellis/Owl [Schaffert 86], Eiffel [Meyer 90], Emerald [Raj 91] and Self [Ungar 87].

There is not general agreement on a definition of what requirements a language should fulfill to be regarded as object-oriented. There are several definitions, of which [Wegner 86], [Meyer 88], [Booch 91], and [Blair 89] are the most used. The oldest and

most widely known definition is [Wegner 86]. The one closest to our perception of object-orientedness is [Blair 89]. The definitions in [Meyer 88] and [Booch 91] are taught to many newcomers to the area of object-oriented programming. Not all languages that are object-oriented according to one definition will be object-oriented according to another definition.

In Section 3.1 we will take a closer look at why object-oriented programming languages have become so widespread and successful. Section 3.2 explains some of the buzz-words from the object-oriented community. In Section 3.3 we will discuss some problems that occur when we look at object-oriented programming languages. Section 3.4 describe some similarities between objects and closures. In Section 3.5 we describe a concept we call *transposition*; the difference between datastructures of traditional languages and object-oriented programming languages.

3.1 Why use the Object-Oriented Paradigm

There are several reasons for the success of the object-oriented paradigm.

One major advantage is the uniform concepts of objects whose interface is described by an abstract data type. The clear interface to the object, the encapsulation and the hiding of the internal state of the object, makes them easy to handle and moves the effort of programming from "how to do it" to "what to do". An example of an object that supplies the programmer with a very useful level of abstraction could be an object describing time as illustrated in Example 3.1.

Example 3.1 Describing time by an object
Consider representing the notion of time as objects with the following interface:

type *TimeType*
function +[**Time**] → [**Time**]
function −[**Time**] → [**Time**]
function ∗[**Integer**] → [**Time**]
function /[**Integer**] → [**Time**]
function >[**Time**] → [**Boolean**]
function >=[**Time**] → [**Boolean**]
function <[**Time**] → [**Boolean**]
function <=[**Time**] → [**Boolean**]
function =[**Time**] → [**Boolean**]
function ! =[**Time**] → [**Boolean**]
function *getSeconds* → [**Integer**]
function *getMicroSeconds* → [**Integer**]
function *asString* → [**String**]
function *asDate* → [**String**]
end *TimeType*

The programmer can use such time objects without worrying about representation, how to compare times, etc. This is all taken care off by the object itself. The programmer only has to worry about what to do with the time values since the other implementation details have been taken care of.

There is nothing spectacular in having an abstract datatype representing time. You can probably even find such an abstract datatype in some non object-oriented languages.

The real message is that in object-oriented languages, the creation of new abstract datatypes is very easy and is encouraged.

Inheritance is also a very important reason for the success. Inheritance provides a fast way to reuse of code. An implementation of an object can be reused when making specialized versions of objects. Inheritance can be used both to make specializations of specific abstractions (e.g. extending a *stack* object with a method *height*) and when creating new abstractions (e.g. creating a *stack* abstraction from a *deque* abstraction).

When invoking a method at a given point in the program, the implementation of the target object does not have to be well known. Ideally, one object can be used wherever another object can be used that has a matching interface. Some languages do however impose some limitations on this. If the method invoked at a certain program point is *plus*, then it does (in principle) not matter whether the target object is an integer object, a floating point object, a matrix object, or even a time object. Another example is the objects representing expressions in a programming language like Scheme. Conditional expressions can be represented by one kind of objects, function application expressions by another kind of objects, etc. If each expression object has a method *eval*, then you can evaluate an arbitrary expression just by invoking the expression object's *eval* method. Adding a new kind of expression will not influence evaluation of expressions, if the expression objects all implement the method *eval*.

The abstract data type part of objects is also very important, because when the interface to all objects is decided on, then the implementation of the different objects can be supplied by different programmers.

The implementation of an object can be changed independently of the rest of a program as long as the interface is kept unchanged. When writing a program one can thus focus more on the way components of the program interact with each other than on the internal workings of the different components. One can choose a simple, slow, but fast-to-write implementation in the beginning and then replace the first implementation with another implementation later.

Finally the size of the object-oriented community and the great number of people that think highly of object-oriented programming may be seen as a sign of quality. This can be seen in computer magazines where nearly every product is promoted as being "object-oriented".

3.2 Features of Object-Oriented Languages

In this section we will explain some of the features and buzz-words from the object-oriented community.

Objects

The main entity of object-oriented programming languages is of course objects. Basically, an object is the encapsulation of some data and some operations.

The encapsulation of data is supposed to be strict. This means that there should be no possible way to inspect the interior state of objects except by invoking methods in the

object. The representation of an object's state is thus completely invisible to the user of an object.

Objects communicate by invoking methods in each other. A method is local to an object, so there may exist several methods with the same name (in different objects) that has different arity and different functionality. When the object-oriented paradigm was just emerging, it was found convenient to explain method invocations by sending messages. New programmers where then not limited by their intuition that methods with the same name should have the same arity, etc.

Some people like to view objects as providers of services. When an object sends a message (invokes a method in another object), then the receiver of the message provides a service, i.e. the object is a server for the object invoking the method.

Information hiding and data abstraction

Information hiding, or protection, is an important issue of objects. Only the methods of an object can inspect and change the internal state of the object. The behavior of an object is defined only by its response to invocations. That means that the user of an object has no way of knowing how the object is actually implemented. The object is a kind of *black box* with buttons that can be pushed and lamps that can be turned on and off. The interface or external view of an object can be described by an abstract data type.

Classes and Metaclasses

In most object-oriented languages objects are described by classes. A class describe the abstract data type of the objects made from the class (the interface), and the class provide the implementation of the object's methods. Classes are used when creating object.

A class can for example be the creator of a stack object. The abstract data type of a stack can be described by the following four lines where we have used a notation borrowing from ML:

$$
\begin{array}{ll}
\mathit{push}: & \mathit{'a} \rightarrow \mathit{unit} \\
\mathit{pop}: & \mathit{unit} \rightarrow \mathit{'a} \\
\mathit{isEmpty}: & \mathit{unit} \rightarrow \mathit{bool} \\
\mathit{top}: & \mathit{unit} \rightarrow \mathit{'a}
\end{array}
$$

A stack can however be implemented in different ways, i.e. by using a list, an array, or another datastructure.

In some languages, (e.g. Smalltalk) classes are themselves objects with a single method usually called *New*. Classes are then said to be instances of *metaclasses*. Metaclasses are again objects that are instances of another metaclass. This relation does however tend to give problems because you have many kinds of entities in the system, as shown in Figure 3.1

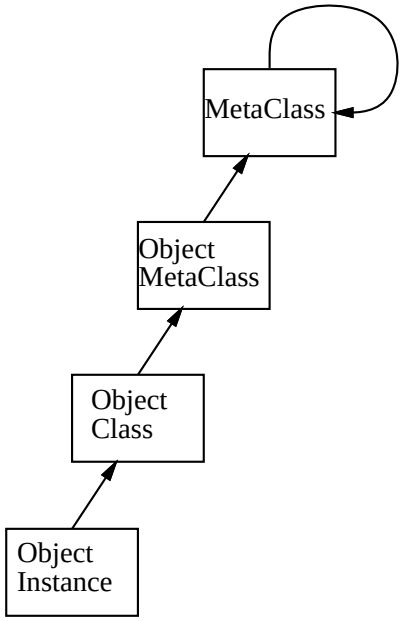

Figure 3.1: Instance – class – metaclass relation

Object Constructors

In languages like Emerald, objects are created by evaluating object constructor expressions. There are no classes, metaclasses etc. Object constructors can be used to create objects that behave like class objects as shown in example 3.2.

Example 3.2 Using object constructors to simulate classes
Consider the following program fragment:

```
const aClass == object aClass
   function New[] → [result]
      result ←
         object anObject
            ⋮
         end anObject
   end New
end aClass
```

An object constructor expression has the structure "**object** *Name* ... **end** *Name*". When the expression is evaluated, an object as described by the body of the object constructor expression, is created. In the example this is used to create an object each time the method *New* is invoked. The expression is evaluated in the assignment statement assigning a value to variable *result*. At the end of the method body, the value assigned to this variable is returned as the function value.

Using object constructors to simulate classes remove the superfluous levels shown in Figure 3.1. Using object constructors rather than classes to create objects furthermore has the advantage that you can make one-of-a-kind objects.

Subtypes

It is very common to define a type system for objects that is based on the objects' interface. The actual implementation of an object with a given interface is not a part of the type.

In object-oriented languages, you are allowed to replace one object with another object if the interfaces of the two objects match. Insisting on an exact match is not desirable (and nobody does that). The "other" object should at least be allowed to have more methods than the one it is replacing. The rules describing what objects may replace others impose an ordering on the types of objects.

The set of objects that can be multiplied with an integer (has a *times* method that requires an integer argument) is larger than the set of objects that can both be multiplied and divided by an integer. The type of the first set of objects is thus more general than the type of the second set of objects. The second type is a *subtype* of the first type.

The subtype relationship can be extended to not only relate two types where one type describe an extension of the other. Different languages use different extensions of the subtype relationship. Many modern object-oriented languages use the *conformance* relation. The conformance relation is the largest relation on types describing interfaces to objects that ensure that you will never introduce the error of invoking a nonexisting method if you replace an object of one type with an object of a subtype. A more formal definition of the conformance relation can be found in [Blair 89].

Inheritance

Inheritance is a mechanism originating from Simula, that makes it possible to create new classes by extending or specializing existing classes. The new class is a *subclass* of the original one; the old a *superclass* of the new. The subclass inherits both the variables and the methods from its superclass, and the subclass can then extend or overwrite methods inherited from its superclass.

Inheritance can be used twofold. Either if you want to extend the definition of an abstract data type, or if you want to use the implementation of a class to define a new class. For example, if the class *Deque* describes objects with the following interface:

```
type Deque
   procedure addtop 'a
   procedure removetop
   procedure addbottom 'a
   procedure removebottom
   function gettop → 'a
   function getbottom → 'a
end Deque
```

then, a class *Stack* can be implemented by inheriting implementation from a class *Deque* in the following way:

```
class Stack
    inherits Deque
    export push, pop, isEmpty, top
    procedure push = addtop
    procedure pop = removetop
    function top = gettop
end stack
```

One distinguishes between single and multiple inheritance. With single inheritance, a class can only inherit implementation from one superclass, while multiple inheritance makes it possible to inherit implementation from several classes.

Inheritance imposes an ordering on the set of classes. With single inheritance, the ordering can be illustrated by a forest (of trees). With multiple inheritance, the ordering can be illustrated by a directed acyclic graph (DAG). The structure on classes is often used when writing browsers aiding the programmer trying to find an existing class for use in new programs.

Several languages treat subclasses as subtypes. This often makes it possible to write programs that are type correct but still fail because nonexisting methods are invoked. People working theoretically with object-oriented programming do generally agree that the hierarchy on types and the hierarchy on classes should be distinct hierarchies.

Dynamic binding

In an object-oriented programming language, a special kind of polymorphism (often called ad-hoc polymorphism) can be used to perform similar actions on completely different objects.

Suppose we have an array of different objects, we can print out representations of all these objects using a simple loop:

```
for i := 1 to array.length do
    (array.getElement[i]).print
```

All the objects in the array must of course have a *print* method.

Even though the objects may be of different kinds, we can print out all of them. The *print* method is polymorphic in the sense, that when the method *print* is invoked, a representation of the target object is printed.

It is often said that the binding of the name *print* to some code is *dynamic*.

3.3 Problematic features

Pure versus hybrid languages

Object-oriented languages can be separated into two categories: the pure and the hybrid languages. In pure languages, all data is represented by objects. That means that integers, floating point numbers, characters and filesystems all are represented as objects. Some object-oriented languages like Smalltalk [Goldberg 89] and Ellie [Andersen 91a] also

represent language constructions like if- and loop-constructions as objects. In the hybrid languages, there is a mixture of objects and other data entities as we know them from other programming languages like Pascal, C, Lisp, ML, etc. Examples of hybrid object-oriented languages are C++ [Stroustrup 86], CLOS [Bobrow 86], and Eiffel [Meyer 90].

There may be a lot of reasons for using hybrid languages. One reason that is valid for languages like e.g. C++ and ObjectPascal is that many programmers are not used to the object-oriented paradigm and should be allowed to make the transition to object-oriented programming in their own pace. C++ and ObjectPascal are languages with add-on object-oriented features. Another reason might be efficiency. Pure object-oriented languages are typically not very efficient. The latest compiler for the language Self does to some extend show that this need not be the case [Chambers 91].

What is inheritance?

Inheritance is by many people regarded as a very important element of object-orientedness. It is however not quite clear what inheritance is or rather how it should be implemented and what semantics it has. We will try to clarify the problems.

Inheritance is a tool for reusing the implementation of one kind of objects when writing the implementation of another kind of objects. In practice there are two ways of implementing inheritance. The first way is the simplest and consists simply of copying definitions of reused methods and variables while resolving possible name clashes. This can be done by a simple early phase of the compiler. The second way is to reuse the already generated object code. This requires some smart handling of object references and representation.

There is only an observational difference between the two approaches if the notion of subtyping is tied to inheritance. If the ordering of types is unrelated to inheritance or if the language does not have a strong type system, then there is no observational difference. If the first approach can be regarded as a valid approach to inheritance, then inheritance is nothing but an advanced form of syntactic sugar.

There is general agreement that subtyping and subclassing is not the same thing [Black 91], [Fischback 91], [Hutchinson 91], [Palsberg 91b]. Many people in the object-oriented community does however not like to think of inheritance as syntactic sugar. From our discussions with people disliking the thought we have learned that their dislike originate from use of languages where subtyping was the same thing as subclassing. With the risk of sounding arrogant, we believe that people will grow to understand that inheritance *can* be regarded as syntactic sugar and that the other implementation is used for space/time efficiency.

3.4 Objects can be regarded as closures

Objects in an imperative language can be implemented using closures from higher-order languages and assignments to variables. This has been known for a long time. We have been told that some of the early papers on Scheme by Abelson and Sussman show this. A recent paper describing how to implement objects using closures is [Reddy 88]. Here

we will just show by an example how this can be done. We will not use this in the rest of the report.

Suppose we have an object created by evaluation of the following object constructor:

```
object ObjectName
   var a
   function not[b] → [c]
      var d
      d ← b.not[]
      c ← d
   end not
end ObjectName
```

An entity behaving in exactly the same way is an ML closure created by evaluation of the following expression:

```
let val a = ref []
    fn method arglist =
    in
       case method of
            "not" =>
                  (case arglist of
                        b ::[] =>
                           let val d = ref [] in
                               (d := b "not" []; !d)
                           end
                      | _ => raise Match)
          | _ => raise Match
    end
```

3.5 Transposition

When programming in non object-oriented languages you often focus on functions and procedures operating on data. In object-oriented languages you focus on objects and add features to the objects as needed. The difference can be seen by considering a program that does a lot of dispatching on the type of an argument. This is often the case when implementing e.g. interpreters.

When programming an interpreter in a functional language, you first parse the string and build an abstract syntax of the program, and then you traverse the syntax tree while dispatching on the type of the elements in the syntax tree.

Writing an interpreter in an object-oriented language can be done in a similar way. It is however also possible to use the benefits of the object-oriented paradigm. First you parse the program and build an abstract syntax of the program, where all nodes in the abstract syntax tree are objects with the method *execute*. Then you invoke the method *execute* in the root object of the tree, resulting in an interpretation of the program. We call this way of interpreting a program for "interpretation using an *active syntax*", because the program's syntax tree is able to interpret itself!

The difference in view of computations is very similar to viewing a matrix either row-wise or column-wise. Using the object-oriented way, it is very easy to add a new language construction to an interpreter because you only have to perform changes in one spot. Using normal dispatching in the interpreter, you should make changes in all places where dispatching is performed. Adding an operation that should work on all kinds of language constructions is on the other hand made more complicated using the object-oriented technique. It requires changes in a lot of places, where the changes in the dispatching interpreter might be performed in one spot.

Chapter 4

Our Source/Target Language

This chapter will describe the object oriented imperative language we have designed for use in this project. We call the language Simili. To make things easy, it was required that the language be small and simple, but still realistic for practical use.

Our language is a "baby" Emerald [Raj 91], i.e. Emerald without a strong type system and without the possibility of distributed and parallel programs. We have chosen Emerald as basis for our language because it is the only object oriented language we know of, that does not have classes and inheritance. Our language is thus untyped, based on objects, and still realistic for practical use. Inheritance can be mimicked in our language by adding some syntactic sugar.

We will avoid the notion of classes and class variables as found in some object oriented languages. We will however insist on strict data encapsulation.

Structured statements like while loops will be replaced by conditional jumps. Composite expressions will not be allowed. This in order to keep things simple.

4.1 Design Decisions

The Goal of the language

To simplify partial evaluation, the language must be kept small. The language should support objects, i.e. the encapsulation of data and methods. The methods should be programmed in an imperative style.

Inheritance is syntactic sugar

Inheritance is a way of reusing code. Methods can be added or modified in subclasses inheriting methods and variables from a superclass. In many typed object oriented languages a subclass is at the same time a *subtype* of the class (type) it inherits from. The idea of subtyping is that wherever an instance of a type can be used so can instances of its subtypes.

Researchers working with type systems in object-oriented languages now generally agree that the class hierarchy should be separated from the type hierarchy [Black 91],

[Fischback 91], [Hutchinson 91], [Palsberg 91b].

In untyped languages and in languages where types are unrelated to inheritance, inheritance is only a textual sort-hand used to reuse code and save the programmer some work [Palsberg 91b]. In the programming language Emerald, inheritance is actually handled in the desugaring phase of the compiler. The core language of Emerald has no notion of inheritance.

We will not make inheritance an integrated part of the language. It is possible to add some syntactic sugar to our language to mimics inheritance. We will however not do this.

Object Constructors rather than Classes

In many object oriented programming languages all objects are instances of a class. To have a single object one has to define a class for the object and then create the instance. This turns out to be very inconvenient when performing partial evaluation. What we really need is a more clean and general language construct to create objects.

The programming language Emerald uses *object constructors*. An object constructor is an expression that when evaluated returns a new object. Classes as found in other languages can be simulated by an ordinary object. Such an object should have a method "New" that returns a new object. This is illustrated in Example 4.1.

Example 4.1 Simulating classes with object constructors
The object constructor is an expression of the following form: "**object** *Name* ... **end** *Name*". A pair-"class" can then be defined as follows:

```
const Pair ==
   object PairClass
      function New[] → [result]
         result ←
            object aPair
               ⋮
            end aPair
      end New
   end PairClass
```

When the method "New" is invoked by an expression like "Pair.New[]" an object created by evaluating the object constructor "aPair" is returned.

Some object oriented languages do not enforce strict data encapsulation. In these languages it is possible to inspect and modify an object's data without using the object's methods. We suspect that this will complicate partial evaluation. We insist on strict data encapsulation.

Procedural and functional methods

We have chosen to distinguish between methods that may have side effects and methods that may not have side effects. This should make partial evaluation easier. When the partial evaluator handles functions, i.e. methods without side effects, it can assume no side effects will happen.

Basic blocks versus structured statements

It is well known that all language constructs influencing flow of control through a program can be translated into conditional goto-statements. Additionally, some flow-graphs are not easily mapped into the usual language constructs in languages like Pascal, etc.

In order to keep things simple we will not have any explicit loop constructs in the language. Early experiments have also shown that these are undesirable for partial evaluation. We will settle for labels and conditional goto statements in our core language.

Composite expressions

A composite expression is an expression that is made up by other expressions. An example is $(2 + 3) + 4$. Early experiments have shown that composite expressions may complicate specialization of programs. To avoid these complications we will insist that all intermediate results are assigned to variables. Composite expressions will not be allowed.

Built-in objects versus basevalues and primitives

We would like all data to be objects. There will be no exceptions like integers, strings, characters, lists, or anything. They must all be objects too. Formally, the only necessary builtin object is the object called **nil**. It is an object without any methods. Integers, Characters, Strings, and other objects can then be simulated in a manner similar to what is seen in the lambda calculus. It is however practical to have some other builtin objects. These can be added without major changes to the language if they are objects.

4.2 What the language has

In this section we will informally describe our language. We will give small examples of how to use the different language constructs. Figure 4.2 shows the concrete syntax of our language.

Object Constructors

Objects are created by "**object** ..." expressions. The objects can have an arbitrary number of methods and instance variables. Each object constructor expression has an initialization part that is executed each time the expression is evaluated.

The instance variables are in the scope of all methods in the object. Each method can have a number of local variables that are private to the method. Local variables can *shadow* instance variables. Recursive methods have independent sets of local variables for each step of the recursion.

Procedural and Functional Methods

The methods are split into two categories: *procedures* and *functions*. A procedure method can have side effects on instance variables and does not return a value. It can also invoke

procedure methods in other objects. A function method is not allowed to have any side effects on instance variables and is not allowed to invoke procedure methods because these can have side effects. Function methods always return a single value.

Function methods have a special return variable that must be assigned the value to be returned before the execution of the method body is finished. This is illustrated by example 4.2.

Example 4.2 An object with a double-function
Suppose we would like an object with a function that given an integer, x, computes $2 \times x$. We can code the function in the following way:

```
const doubleobject == object aDoubleObject
   function twice[x] → [result]
      label: result ← x.times[2]
             return
   end twice
end aDoubleObject
```

Initialization of Objects

The initialization part of an object constructor expression is executed each time the expression is evaluated. It is thus possible to initialize the instance variables of the object. The part of the environment mapping method variables to values is in the scope of all expressions evaluated inside the initialization part. It is thus possible to let the initial values of an objects instance variables depend on the environment it was created in. In example 4.3 a program fragment is given illustrating initialization using formal parameters.

Example 4.3 An object with a function returning a cons-cell (object)
Suppose we would like to simulate a class, where we initialize the objects when created. This could be done in the following way:

```
const consClass == object aConsClass
   function New[hd, tl] → [result]
      label: result ← object consObject
                var head   var tail
                function head[] → [result]
                   lbl: result ← head return
                end head
                function tail[] → [result]
                   lbl: result ← tail return
                end tail
                initially
                   lbl: head ← hd
                        tail ← tl
                        return
                end initially
             end consObject
             return
   end New
end aConsClass
```

When we create objects of the consClass, we call the function *New* with two parameters. These parameters are used to initialize the created object's variables in the initialization part of the object constructor.

The jump instructions

There are both conditional and unconditional jump instructions. The meaning of these should be obvious. Within any body (*IBody*, *PBody*, and *FBody*) it is only allowed to jump to labels within the same body.

The **return** keyword signals that execution of the current method should end. There has to be at least one occurrence of the keyword **return** in each method. The return value of a function method is the value assigned to the *ResultName* variable. This variable has to be assigned a value during execution of the function body. There is no argument to the return instruction as there is in C.

Builtin Objects

The most important object is the "nil" object. Formally, we do not need other builtin objects in the language. Everything else can be encoded as in the pure lambda calculus. An example of this can be found in [Palsberg 91a].

To make the language realistic and useful for programming we decided to make it possible to manipulate integers, strings, chars, and booleans as objects. In Appendix A we have listed the interface to the builtin objects. The "names" used to denote the builtin objects are also given in this appendix. In the syntax specification we have denoted all these names by the nonterminal *BaseValueName*.

Instance Variables and Local Variables

An object may have instance variables. These variables are local to an object and are shared among all methods of the object. In addition to instance variables we have method variables. We have three kinds of method variables, *formal parameters*, *local variables*, and *result names*. Formal parameters contain the values of the arguments passed to the method. Local variables are declared explicitly. They are not initialized when the method is called. Result names are used to return values from functions; they do not exists in procedures.

Assignment to formal parameters is not allowed.

Weak Type System

The language is weakly typed. This means that type errors cannot be detected at compile time. We assume that all programs are without errors.

4.3 Syntax of Simili

The abstract syntax of Simili is illustrated in Figure 4.1. We have used a slightly modified version and of the Extended Backus-Naur Form (EBNF). A "*" after a nonterminal means zero, one, or more occurrences. A "+" means one, or more occurrences. Square brackets and parentheses are all part of the syntax. A "?" after a nonterminal means zero or one

occurrence. The keywords of the language are in **boldface**. The elements in a list may be separated by commas to improve readability. Terminals and nonterminals can be grouped together with braces, e.g. $\{\ a\ b\ \}^+$ means one or more occurrence of a followed by b, while $a\ b^+$ means one occurrence of a succeeded by one or more occurrences of b.

The list of *IVarName*'s in object constructor expressions is the list of declared instance variables. The list of *LVarName*'s in method definitions is the list of declared local variables.

Program	::=	*ConstDecl*.*
ConstDecl	::=	**const** *ConstName == Exp.*
BasicExp	::=	*ConstName* | **self** | *IVarName* | *LVarName*
	|	*FormalPar* | *ResultName* | *BaseValueName*
Exp	::=	*BasicExp*
	|	*BasicExp == BasicExp*
	|	*BasicExp.FnctName*[*BasicExp**]
	|	**object** *ObjectName* (*IVarName**)(*MethodDef**) *IBody.*
MethodDef	::=	*ProcName*[*FormalPar**](*LVarName**) *PBody*
	|	*FnctName*[*FormalPar**]→[*ResultName*](*LVarName**) *FBody.*
IBody	::=	$\{$ *label InitiallyStatement Jump* $\}^+$.
PBody	::=	$\{$ *label Statement Jump* $\}^+$.
FBody	::=	$\{$ *label FnctStatement Jump* $\}^+$.
InitiallyStatement	::=	*IVarName* ← *Exp*
	|	**skip**
	|	*InitiallyStatement1 InitiallyStatement2.*
Statement	::=	*IVarName* ← *Exp*
	|	*LVarName* ← *Exp*
	|	*BasicExp.ProcName*[*BasicExp**]
	|	**skip**
	|	*Statement1 Statement2.*
FnctStatement	::=	*LVarName* ← *Exp*
	|	*ResultName* ← *Exp*
	|	**skip**
	|	*FnctStatement1 FnctStatement2.*
Jump	::=	**goto** label
	|	**if** Exp **then goto** label **else goto** label
	|	**return**.

Figure 4.1: The (abstract) syntax of the language Simili

The abstract syntax does not have a deterministic parser. We have consequently defined a parsable syntax for the language. This syntax is illustrated in Figure 4.2.

Program	::=	*ConstDecl** **;**.
ConstDecl	::=	**const** *ConstName* == *Exp*.
BasicExp	::=	*ConstName* | **self** | *IVarName* | *LVarName* | *FormalPar* | *ResultName* | *BaseValueName*
Exp	::=	*BasicExp* | *BasicExp* == *BasicExp* | *BasicExp*.*FnctName*[*BasicExp**] | **object** *ObjectName IVarDecl* MethodDef* Initially* **end** *ObjectName*.
MethodDef	::=	**procedure** *ProcName* [*FormalPar**] *LVarDecl PBody* **end** *ProcName* | **function** *FnctName* [*FormalPar**] → [*ResultName*] *LVarDecl* FBody* **end** *FnctName*.
IVarDecl	::=	**var** *IVarName*$^+$.
LVarDecl	::=	**var** *LVarName*.
Initially	::=	**initially** *IBody* **end initially**
IBody	::=	{ *label* **:** *InitiallyStatement*$^+$ *Jump* }$^+$.
PBody	::=	{ *label* **:** *ProcStatement*$^+$ *Jump* }$^+$.
FBody	::=	{ *label* **:** *FnctStatement*$^+$ *Jump* }$^+$.
InitiallyStatement	::=	*IVarName* ← *Exp* | **skip**.
ProcStatement	::=	*IVarName* ← *Exp* | *LVarName* ← *Exp* | *BasicExp*.*ProcName*[*BasicExp*] | **skip**.
FnctStatement	::=	*LVarName* ← *Exp* | *ResultName* ← *Exp* | **skip**.
Jump	::=	**goto** *label* | **if** *Exp* **then goto** *label* **else goto** *label* | **return**.

Figure 4.2: The concrete syntax of the language Simili

4.4 Syntactic sugared Simili

In practice we do not want to bother with labels and jumps when writing Simili programs. Consequently we have defined a sugared version of the language. In the sugared version, variables are declared by use of a keyword, procedure and function methods are indicated by a keyword, object constructors and methods are explicitly terminated, and language constructs other than **goto** has been introduced to control the flow of control in the program. The sugared version of the syntax is illustrated in Figure 4.3. The mapping of syntax from the sugared version of the syntax to both the parsable and the abstract syntax should be obvious.

There are two possible formats for comments in the programs. Everything from a percent sign (%) to the end of the line is considered a comment. Everything surrounded by "(*" and "*)" is also considered a comment.

```
Program          ::= ConstDecl* ;.

ConstDecl        ::= const ConstName == Exp.

BasicExp         ::= ConstName | self | IVarName | LVarName
                   | FormalPar | ResultName | BaseValueName

Exp              ::= BasicExp
                   | BasicExp == BasicExp
                   | BasicExp.FnctName[BasicExp*].
                   | object ObjectName IVarDecl* MethodDef* Initially end ObjectName.

MethodDef        ::= procedure ProcName [FormalPar*]
                       LVarDecl* ProcStatement* end ProcName
                   | function FnctName [FormalPar*] → [ResultName]
                       LVarDecl* FnctStatement* end FnctName.

IVarDecl         ::= var IVarName+.

LVarDecl         ::= var LVarName.

ProcStatement    ::= IVarName ← Exp
                   | BasicExp.ProcName[BasicExp*].
                   | LVarName ← Exp
                   | skip
                   | if Exp then ProcStatement* { elseif Exp then ProcStatement* }*
                       { else ProcStatement* }? end if
                   | repeat ProcStatement* until Exp
                   | loop { ProcStatement | ExitStatement }* end loop
                   | while Exp do ProcStatement* end while.

ExitStatement    ::= exit when Exp.

Initially        ::= initially IBody end initially

InitiallyStatement ::= IVarName ← Exp
                   | skip
                   | if Exp then InitiallyStatement*
                       { elseif Exp then InitiallyStatement* }*
                       { else InitiallyStatement* }? end if
                   | repeat InitiallyStatement* until Exp
                   | loop { InitiallyStatement | ExitStatement }* end loop
                   | while Exp do InitiallyStatement* end while.

FnctStatement    ::= LVarName ← Exp
                   | skip
                   | if Exp then FnctStatement* { elseif Exp then FnctStatement* }*
                       { else FnctStatement* }? end if
                   | repeat FnctStatement* until Exp
                   | loop { FnctStatement | ExitStatement }* end loop
                   | while Exp do FnctStatement* end while
                   | ResultName ← Exp.
```

Figure 4.3: The Concrete Syntax of Sugared Simili

4.5 Operational Semantics

In this section we will describe the semantic of Simili. The semantics will be described using deduction rules as described by e.g. [Kahn 87].

We represent an object reference by an OID (Object ID). The OIDs can e.g. be the set of integers. We distinguish between builtin objects and objects made by evaluating object constructors in the program. None of the builtin objects have an internal state.

The state of all the existing program objects is represented by a mapping from OIDs to code and environments. Environments are mappings from names into OIDs.

We will represent constants as instance variables in a special object. This is done to reduce the amount of mappings in the description of the semantics. We use the "nil" object for this purpose.

We assume that the builtin objects has also been given OIDs. This is reasonable since the set of builtin objects is countable. The mapping of textual representation of builtin objects to OIDs is given by the mapping γ. The OID of the truth object is given by the constant $\mathsf{true} = \gamma(\textit{"true"})$. The OID of the falsity object is given by the constant false.

We have divided the definitions of the used semantic objects into three categories: basic semantic objects, compound semantic objects, and semantic functions. The definition of all the basic semantic objects used to define the semantics of Simili is listed in Figure 4.4. We have written the definition of the semantic objects so the nil object have a special status being neither a program object or a base value (builtin object).

FormalPar	$\in$	FORMALPAR
LVarName	$\in$	LVARNAME
ResultName	$\in$	RESULTNAME
LocalName	$\in$	LOCALNAME = FORMALPAR $\cup$ LVARNAME $\cup$ RESULTNAME
IVarName	$\in$	IVARNAME
FnctName	$\in$	FNCTNAME
ProcName	$\in$	PROCNAME
MethodName	$\in$	PROCNAME + FNCTNAME
Ω	$\in$	Abstract_Object = Program_Object + nil
nil	$\in$	Program_Object
bv	$\in$	Builtin_Object
β	$\in$	OID = Builtin_Object + Abstract_Object
true	$\in$	OID
false	$\in$	OID
$\widehat{\beta}$	$\in$	OID list
$\mathcal{B}$	$\in$	Body = BBlist

Figure 4.4: Basic semantic objects used to define the semantics of Simili.

In the derivation rules we will use many semantic objects. The most important of these are the compound semantic objects denoted by the symbols Ω (Current Object), τ (environment mapping method variables to OIDs), and ρ (the state of all existing objects, the global store). The signatures of all the used compound semantic objects are shown in Figure 4.5.

$$
\begin{array}{rcl}
\text{Code} & \in & \text{CODE} = \\
 & & \quad \text{PROCNAME} \overset{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{LVARNAME list} \times \text{Body} \ + \\
 & & \quad \text{FNCTNAME} \overset{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{RESULTNAME} \times \text{LVARNAME list} \times \text{Body} \\
\tau & \in & \text{LocalEnv} = \text{LOCALNAME} \overset{\text{fin}}{\mapsto} \text{OID} \\
\imath & \in & \text{InstanceEnv} = \text{INSTANCENAME} \overset{\text{fin}}{\mapsto} \text{OID} \\
\rho & \in & \text{Object\_Store} = \text{program\_object} \overset{\text{fin}}{\mapsto} (\text{CODE} \times \text{InstanceEnv})
\end{array}
$$

Figure 4.5: Compound semantic objects used to define the semantics of Simili.

In Figure 4.6 we have listed the used semantic functions. The mapping ϵ maps a list of syntactic method definitions into the chosen semantic representation for the methods of an object. The mapping **DoBaseFnctCall** implements the semantics of all builtin objects.

$$
\begin{array}{rcl}
\gamma & \in & \text{BaseValueName} \mapsto \text{Builtin\_Object} \\
\epsilon & \in & \text{MethodDef*} \mapsto \text{CODE} \\
\textbf{DoBaseFnctCall} & \in & \text{OID} \times \text{FNCTNAME} \times \text{OID list} \mapsto \text{OID}
\end{array}
$$

Figure 4.6: Semantic functions used to define the semantics of Simili.

We use the symbol ρ_0 for the global store, where there only exists a mapping for nil. The code-part of this is empty. The symbol τ_0 is used to denote the empty local environment. The symbol $\imath_0$ is used to denote the empty instance environment. Further we use the notation $\rho_{\text{env}}(\Omega)$ for the second component of $\rho(\Omega)$, and ρ_{code} for the first component.

We use some syntactic sugar for updating ρ:

$$\rho[\Omega,\ \mathit{Name} \mapsto \beta]$$

is syntactic sugar for

$$\rho[\Omega \mapsto (\rho_{code}(\Omega) \times \rho_{env}(\Omega)[\mathit{Name} \mapsto \beta])]$$

and

$$\rho[\Omega \overset{\text{env}}{\mapsto} \imath]$$

is syntactic sugar for

$$\rho[\Omega \mapsto <\rho_{code}(\Omega),\ \imath>]$$

We use the notation $\hat{\text{a}}$ for a list of "a". We use the notation $\widehat{\text{a}_n}$ for the list $[\text{a}_1, \ldots, \text{a}_n]$.

Judgment forms

We use some judgment forms that are different from what can be seen elsewhere, e.g. the formal definition of ML [Milner 90,Milner 91] or the different semantics described in [Kahn 87]. This will be done use the following rules as an example:

$$\vdash \mathbf{FnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_i}) \Rightarrow \beta : \rho \rightarrow \rho' \tag{4.1}$$

$$\frac{\begin{array}{c}\Omega,\ \tau \vdash [\![\mathit{Exp}]\!] \Rightarrow \beta : \rho \rightarrow \rho' \\ \tau' = \tau[\mathit{LVarName} \mapsto \beta]\end{array}}{\Omega \vdash [\![\mathit{LVarName} \leftarrow \mathit{Exp}]\!] : \rho,\ \tau \rightarrow \rho',\ \tau'} \tag{4.2}$$

All names to the left of the turnstile symbol ($\vdash$) are semantic objects that are used but not changed in the rule. They can be thought of as prerequisites. Everything after the colon (:) describes modifications of semantic objects. The new values can be found after the arrow symbol ($\rightarrow$). The part of the judgment form immediately after the turnstile symbol is a pattern used to select which rule is to be used. The pattern can be syntactic as in rule 4.2 and is then enclosed in semantic brackets ($[\![\ ]\!]$). It can also use a qualifying symbol as in rule 4.1 (**FnctCall**). Finally, a rule can return new semantic objects. These are listed after the double arrow symbol ($\Rightarrow$).

The first rule (4.1) should be read in the following way. Given a pattern of the form **FnctCall**(...), and a global store, ρ, it evaluates to an object, β, while the global store is modified from ρ to ρ'. The rule could be summarized to: when the sentence is evaluated to β, the store is modified to become ρ'.

The second rule (4.2) should be read in the following way. Given the current object, Ω, a syntactic object matching the pattern $[\![\mathit{LVarName} \leftarrow \mathit{Exp}]\!]$, a global store, ρ, and a local environment, τ, the effect is a modification of both the global store and the local environment. The modification of the global store is equal to the modification to the global store caused by evaluation of the expression, *Exp*. The modification of the local environment is specified by the second line of the deduction rule.

In some of the formulas we use the notation

% Error - ...,

which means that an error situation has occurred. If this happen, then we do not consider the given text a program at all. We have included these rules to specify the error situations explicitly.

One type of error situation we have chosen not to catch or even specify is the use of undeclared variables. We assume that this is caught while parsing the program.

The formulas

Figure 4.7 defines the meaning of running a program. Running a program you take a program, a ConstName from the program, a ProcName and FnctName from the object the ConstName correspond to, and finally some arguments, Args, that you want to execute the program with. First the program is loaded, by evaluating the constant declarations. Then, the ConstName and arguments, Args, is evaluated, and then the procedure with the name ProcName is executed in the store found from loading the program. Executing the procedure can cause changes to the store. Finally, the function with the name FnctName is called, returning a value, that is the value returned by the program. It is important to notice, that "ConstName" must be a constant name within the program. ProcName and FnctName must be the names of a procedure and a function within the object ConstName correspond to. The arguments to the procedure, Args, must be a list of either ConstNames from the program, a builtin object or the nil object. The object returned from the program can be any object in the new object-store, ρ'.

$$\frac{\begin{array}{c}\vdash [\![\mathit{Program}]\!] \Rightarrow \rho \\ \rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \Omega \\ \rho \vdash [\![\mathit{Args}]\!] \Rightarrow \widehat{\beta} \\ \vdash \mathbf{ProcCall}(\Omega,\ \mathit{ProcName},\ \widehat{\beta}) : \rho \rightarrowtail \rho' \\ \rho' \vdash \mathbf{FnctCall}(\Omega,\ \mathit{FnctName},\ [\,]) \Rightarrow \beta\end{array}}{\vdash [\![\mathit{Program}\ \mathit{ConstName}\ \mathit{ProcName}\ \mathit{Args}\ \mathit{FnctName}]\!] \Rightarrow \beta}$$

Figure 4.7: Semantic rules for the input–output function of a program.

In Figure 4.8 we show the semantics for constant declarations. The first rule show us the meaning of a program, i.e. a global store. The second rule says, that evaluating a sequence of constant declarations, is the same as evaluation the last constant declaration and then evaluating the rest. The third rule explain the meaning of a single constant declaration. First we evaluate the expression in the given environment modifying the global store. Then we update the new global store, so that the nil object's instance environment contains a mapping from the *ConstName* to the result of evaluating the expression. It is important to observe that when finding the meaning of a program, it is

$$\frac{\text{Program} = \text{ConstDecl*} \qquad \vdash [\![ConstDecl*]\!] : \rho_0 \rightarrow \rho}{\vdash [\![Program]\!] \Rightarrow \rho}$$

$$\frac{\vdash [\![ConstDecl*]\!] : \rho \rightarrow \rho' \qquad \vdash [\![ConstDecl]\!] : \rho' \rightarrow \rho''}{\vdash [\![ConstDecl*\ ConstDecl]\!] : \rho \rightarrow \rho''}$$

$$\frac{\Omega = \mathsf{nil} \qquad \tau_0 \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \qquad \rho'' = \rho'[\mathsf{nil}, ConstName \mapsto \beta]}{\vdash [\![\textbf{const}\ ConstName == Exp]\!] : \rho \rightarrow \rho''}$$

Figure 4.8: Semantic rules for constant declaration part

done in an environment, where the current object always is nil. The constant declaration

const *myself* == **self**

will thus lead to the binding of the constant "myself" to nil. Further, all expressions on top level are evaluated with the empty local environment, τ_0.

It is important to notice, that during load of a program, all expression been evaluated must only contain references to constant names, ConstNames, that has been loaded before the actual load. This also means, that during evaluation of an expression, the current ConstName is undefined, but can, in the case of an object expression, be referred to by **self**.

Figure 4.9 shows the meaning of basic expressions. The meaning of a "ConstName" can be found in instance environment of the nil object. The value of an instance variable can be found in the instance environment of the current object in the global store, i.e. $\rho_{\text{env}}(\Omega)$. Local variables can be found in the local environment, τ. The value of a "BaseValueName" is found through the mapping γ shown in Figure 4.6. We have not listed a special rule for the nil object even though this formally is required. In this context we assume that nil is a member of the Builtin_Object domain.

$$\frac{}{\rho \vdash [\![ConstName]\!] \Rightarrow \rho_{\text{env}}(\mathsf{nil})(ConstName)}$$

$$\frac{}{\Omega \vdash [\![\mathbf{self}]\!] \Rightarrow \Omega}$$

$$\frac{IVarName \in \text{Dom}(\rho_{\text{env}}(\Omega))}{\Omega, \rho \vdash [\![IVarName]\!] \Rightarrow \rho_{\text{env}}(\Omega)(IVarName)}$$

$$\frac{IVarName \notin \text{Dom}(\rho_{\text{env}}(\Omega)) \quad \text{\% Error - this should not happend !!!}}{\Omega, \rho \vdash [\![IVarName]\!] \Rightarrow}$$

$$\frac{LVarName \in \text{Dom}(\tau)}{\tau \vdash [\![LocalName]\!] \Rightarrow \tau(LocalName)}$$

$$\frac{LVarName \notin \text{Dom}(\tau) \quad \text{\% Error - this should not happend !!!}}{\tau \vdash [\![LocalName]\!] \Rightarrow}$$

$$\frac{\beta = \gamma(BaseValueName)}{\vdash [\![BaseValueName]\!] \Rightarrow \beta}$$

Figure 4.9: Semantic rules for basic expressions

When fetching a value from an environment, we first check that the name is within the domain of the environment, i.e. we check to see if the variable has been given a value. If not, we have an error situation, which we assume cannot happen during computation, i.e. it is not a program, if such a situation can occur.

Figure 4.10 describe the meaning of expressions. The meaning of the equality expression is simple. Find the meaning of both basicexpressions. If they are the same, return true else return false. The meaning of a function call is also simple. When the meaning of the receiving object and the arguments are found, find the meaning of the **FnctCall** and return this result. The meaning of the object constructor expression is first to find a new *OID* that is not already in the global store. Update the global store, so it also contain the new object, with the code part bound to the methods defined, and let the instance environment be empty, $\imath_0$. Then find the meaning of the initially body where the current object is the newly created, Ω', and return the OID of the new object with the modified global store.

$$\frac{BExp = Exp \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta}{\Omega,\ \tau,\ \rho \vdash [\![Exp]\!] \Rightarrow \beta}$$

$$\frac{\Omega,\ \tau,\ \rho \vdash [\![BExp_1]\!] \Rightarrow \beta \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp_2]\!] \Rightarrow \beta}{\Omega,\ \tau,\ \rho \vdash [\![BExp_1 == BExp_2]\!] \Rightarrow \mathsf{true}}$$

$$\frac{\Omega,\ \tau,\ \rho \vdash [\![BExp_1]\!] \Rightarrow \beta_1 \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp_2]\!] \Rightarrow \beta_2 \qquad \beta_1 \neq \beta_2}{\Omega,\ \tau,\ \rho \vdash [\![BExp_1 == BExp_2]\!] \Rightarrow \mathsf{false}}$$

$$\frac{\begin{array}{c}\Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta \\ \Omega,\ \tau,\ \rho \vdash [\![BExp_j]\!] \Rightarrow \beta_j, \quad \text{for } j \in \{1, \ldots, n\} \\ \vdash \mathbf{FnctCall}(\beta,\ FnctName,\ \widehat{\beta_n}) \Rightarrow \beta' : \rho \rightarrow \rho'\end{array}}{\Omega,\ \tau \vdash [\![BExp.FnctName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \beta' : \rho \rightarrow \rho'}$$

$$\frac{\begin{array}{c}\Omega' \notin \mathrm{Dom}(\rho) \\ \rho' = \rho[\Omega' \mapsto <\epsilon(MethodDef*),\ \imath_0>] \\ \Omega' \vdash [\![IBody]\!] : \rho',\ \tau \rightarrow \rho'',\ \tau'\end{array}}{\Omega,\ \tau \vdash [\![\mathbf{object}\ ObjectName(IVarName*)(MethodDef*)\ IBody]\!] \Rightarrow \Omega' : \rho \rightarrow \rho''}$$

Figure 4.10: Semantic rules for expressions

Figure 4.11 describe the meaning of statements. The first three rules gives the semantics of assignment. First find the meaning of the expression, and then assign the found value to the variable in the proper environment or store. Observe that you cannot assign values to formal parameters. The meaning of the **skip** statement is very simple. The meaning of a procedure call is the meaning of **ProcCall** with the meaning of the receiving object and, the procedure name and the arguments. The modified store is returned. Finally the meaning of composite statements is, first execute the first then the second.

$$\frac{\Omega,\ \tau \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \qquad \rho'' = \rho'[\Omega,\ IVarName \mapsto \beta]}{\Omega,\ \tau \vdash [\![IVarName \leftarrow Exp]\!] : \rho \rightarrow \rho''}$$

$$\frac{\Omega,\ \tau \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \qquad \tau' = \tau[LVarName \mapsto \beta]}{\Omega \vdash [\![LVarName \leftarrow Exp]\!] : \rho,\ \tau \rightarrow \rho',\ \tau'}$$

$$\frac{\Omega,\ \tau \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \qquad \tau' = \tau[ResultName \mapsto \beta]}{\Omega \vdash [\![ResultName \leftarrow Exp]\!] : \rho,\ \tau \rightarrow \rho',\ \tau'}$$

$$\frac{}{\vdash [\![\mathbf{skip}]\!] :}$$

$$\frac{\begin{array}{c}\Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta \\ \Omega,\ \tau,\ \rho \vdash [\![BExp_j]\!] \Rightarrow \beta_j, \quad \text{for } j \in \{1, \ldots, n\} \\ \vdash \mathbf{ProcCall}(\beta,\ ProcName,\ \widehat{\beta_n}) : \rho \rightarrow \rho'\end{array}}{\Omega,\ \tau \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] : \rho \rightarrow \rho'}$$

$$\frac{\Omega \vdash [\![Statement1]\!] : \rho,\ \tau \rightarrow \rho',\ \tau' \qquad \Omega \vdash [\![Statement2]\!] : \rho',\ \tau' \rightarrow \rho'',\ \tau''}{\Omega \vdash [\![Statement1\ Statement2]\!] : \rho,\ \tau \rightarrow \rho'',\ \tau''}$$

Figure 4.11: Semantics of statements

Figure 4.12 shown the meaning of a body (Body) and a single basic block. The meaning of a body is simply the meaning of the first basic block in the body. The meaning of a basic block is the meaning of first executing the statement and then the Jump part.

$$\frac{\mathcal{B} = [\![Body]\!] \quad \mathcal{B} = [\![BB \ \ldots]\!] \quad \mathcal{B}, \Omega \vdash [\![BB]\!] : \rho, \tau \rightarrow \rho', \tau'}{\Omega \vdash [\![Body]\!] : \rho, \tau \rightarrow \rho', \tau'}$$

$$\frac{\Omega \vdash [\![Stmnt]\!] : \rho, \tau \rightarrow \rho', \tau' \quad \mathcal{B}, \Omega \vdash [\![Jump]\!] : \rho', \tau' \rightarrow \rho'', \tau''}{\mathcal{B}, \Omega \vdash [\![label\ Stmnt\ Jump]\!] : \rho, \tau \rightarrow \rho'', \tau''}$$

Figure 4.12: Semantic rules for basic blocks

Figure 4.13 describe the meaning of a Jump. The meaning of a **goto** is simply the meaning of the basic block to go to. The meaning of **return** is nothing. The meaning of an if-statement is: first find the meaning of the expression. If it is true, then the meaning is the meaning of the first basic block, if false it is the meaning of the other. If the meaning of the expression is neither true nor false, then the program has no meaning (a type error?).

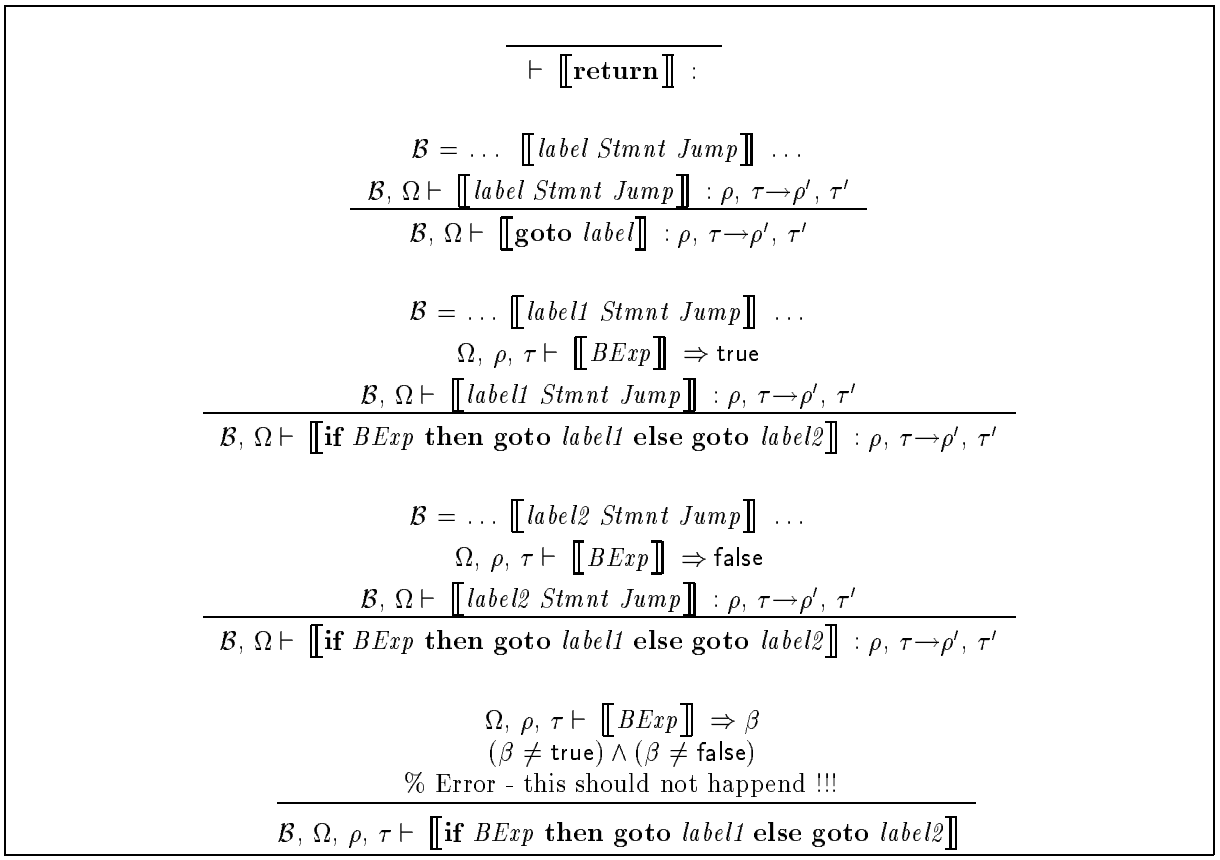

Figure 4.13: Semantics for Jump

Figure 4.14 describes the meaning of a procedural method invocation. The first rule describe the meaning of a procedure call on a builtin object. It a simply the meaning of **DoBaseProcCall** that has no side effect. The second rule descibes the meaning of invoking the nil object, which is obviously an error. The third and fourth rules describes the meaning of a procedure call on an object defined in the program. First find the actually method in the code-part of the global store. This is done a strange way. Lookup in the global store and fetch the code-part of the result. If not found wor, else the formal parameters, local variables and procedure body is returned. Then initialize the formal parameters to the arguments given. Finally execute the body of the procedure.

$$\frac{\%\ \textit{should not happen - error !!!}}{\vdash \mathbf{ProcCall}(\mathsf{bv},\ \mathit{ProcName},\ \widehat{\beta_i})}$$

$$\frac{\%\ \textit{Nil invoked - error !!!}}{\vdash \mathbf{ProcCall}(\mathsf{nil},\ \mathit{ProcName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName}) \\ \%\ \textit{Method Not Understood - error !!!}\end{array}}{\rho \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_i},\ \widehat{\mathit{LVarName}_k},\ \mathit{PBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName}) \\ i \neq j \\ \%\ \textit{Message Not Understood - error !!!}\end{array}}{\rho \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_j},\ \widehat{\mathit{LVarName}_k},\ \mathit{PBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName}) \\ \tau = [\mathit{FormalPar}_i \mapsto \beta_i],\quad \text{for } i \in \{1, \ldots, j\} \\ \Omega' \vdash [\![\mathit{PBody}]\!] : \rho,\ \tau \rightarrow \rho',\ \tau'\end{array}}{\vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta_j}) : \rho \rightarrow \rho'}$$

Figure 4.14: Semantics of procedural method invocation

Figure 4.15 described the meaning of a function-method invocation. It is very simular to a procedure-method invocation, the only difference is that it is possible to invoke funcion-methods in a builtin object. This is handled by the **DoBaseFnctCall** stuff, defined in Figure 4.6. Further the result of invoking a function-method is given in the local environment, τ, by the ResultName. If the ResultName is not in the domain of the local environment after executing the body of the method an error occur, else the result is simply returned. The third rule describe the meaning of a function call on program

$$\frac{\vdash \mathbf{DoBaseFnctCall}(\mathsf{bv}, \mathit{FnctName}, \widehat{\beta_j}) \Rightarrow \beta :}{\vdash \mathbf{FnctCall}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta_j}) \Rightarrow \beta}$$

$$\frac{\%\ \mathit{Nil\ invoked\ -\ error\ !!!}}{\vdash \mathbf{FnctCall}(\mathsf{nil},\ \mathit{FnctName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ \%\ \mathit{Method\ Not\ Understood\ -\ error\ !!!}\end{array}}{\rho \vdash \mathbf{FnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_i},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ i \neq j \\ \%\ \mathit{Message\ Not\ Understood\ -\ error\ !!!}\end{array}}{\rho \vdash \mathbf{FnctCall}(\Omega',,\ \mathit{FnctName},\ \widehat{\beta_j})}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_j},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ \tau = [\mathit{FormalPar}_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \Omega' \vdash [\![\mathit{FBody}]\!] : \rho,\ \tau \rightarrow \rho',\ \tau' \\ \mathit{ResultName} \in \mathrm{Dom}(\tau') \\ \beta = \tau'(\mathit{ResultName})\end{array}}{\vdash \mathbf{FnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_j}) \Rightarrow \beta : \rho \rightarrow \rho'}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_j},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ \tau = [\mathit{FormalPar}_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \Omega' \vdash [\![\mathit{FBody}]\!] : \rho,\ \tau \rightarrow \rho',\ \tau' \\ \mathit{ResultName} \notin \mathrm{Dom}(\tau') \\ \%\ \text{Error - ResultName not assigned to any value during evaluation}\end{array}}{\rho \vdash \mathbf{FnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_j})}$$

Figure 4.15: Semantics of functional method invocation

objects, while the fourth rule describe it on builtin objects. The rules are similar to those of procedure call.

This page intentionally left blank.

Chapter 5

Ideas, Problems and Strategies

5.1 Background

Partial evaluation has since the mid 80's been an area of major interest in the Topps group at the Department of Computer Science at the University of Copenhagen. At first the work was concentrated on developing partial evaluators for first order applicative languages. Having gained a better understanding of the technology, partial evaluators were constructed for higher order functional languages. At approximately the same time, a partial evaluator for a 0'th order (no functions or procedures) imperative language was developed, and also a partial evaluator for a declarative language (a subset of Prolog).

Recently, work has commenced on constructing a partial evaluator for a first order imperative language with pointers as data values. The presence of pointers, functions and procedures makes that a very difficult task.

Hoping that things would be easier if data and functions/procedures operating on these data were encapsulated in objects, we started to work on constructing a partial evaluator for an object-oriented imperative language. An object-oriented imperative language can be viewed as a higher order imperative language using the higher order values (the objects) in a stringent way.

The historical development of partial evaluation at the Department of Computer Science at the University of Copenhagen is illustrated by the graph shown in Figure 5.1.

5.2 Theses

The main objective of this project is to prove the thesis that it is possible to construct a partial evaluator for an object-oriented imperative language. This has not been done before and will therefore move the frontier of research in the area of semantics-based program transformation.

Another thesis that we would like to prove or disprove is that objects in the source language of a partial evaluator is an advantage. This of course only if the source language is imperative.

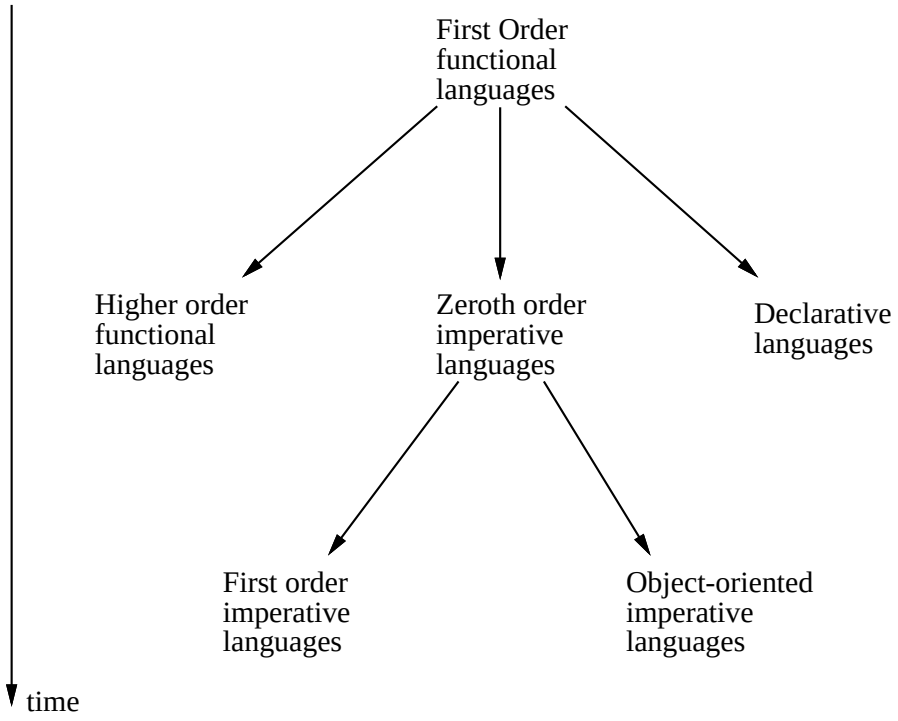

Figure 5.1: The history of partial evaluation at DIKU

```
const compute == object something
   function power[x, n] → [result]
      var temp
      if n.eq[0] then
         result ← 1
      elseif n.even[] then
         temp ← n.div[2]
         temp ← self.power[x, temp]
         result ← temp.times[temp]
      else
         temp ← n.minus[1]
         temp ← self.power[x, temp]
         result ← x.times[temp]
      end if
   end power
end something
```

Figure 5.2: A program implementing the power function

5.3 Problem Statement

It is essential to construct a partial evaluator for an object-oriented imperative language. A trivial specializer which only replaces the formal parameters with the known (static) input values does not satisfy our goal.

The partial evaluator should perform symbolic computations with the known values. Conditional branches and loops which are entirely controlled by the known input values should be removed from the program. Assignments to variables should be removed whenever possible. The specializer should also perform a polyvariant specialization of object constructors and methods. Objects created by the same object constructor in the source program may therefore be created by different object constructors in the residual program as we expect them to be.

We shall give examples of the programs we want to specialize by our partial evaluator. Further we shall sketch the structure of the specialized programs.

Using a more or less functional subset of the source language, we can write a program implementing the power function. The power function is the classical example used in almost all descriptions of partial evaluators for functional languages. The power function takes two arguments and compute the value of the first argument lifted to the power of the second argument. In our language, the power function has to be a method in an object. It can be implemented as illustrated in Figure 5.2.

Specializing the power function with the second argument being (say) 5, we expect the partial evaluator to generate a program with a structure similar to that of the program illustrated in Figure 5.3.

An implementation of Ackermann's function is given in Figure 5.4. The function is recursive and thus contains a loop. In the loop the first argument is *a static variable*

```
const compute == object something
   function power5[x] → [result]
      var temp
      temp ← x.times[1]
      temp ← temp.times[temp]
      temp ← temp.times[temp]
      result ← x.times[temp]
   end power5
end something
```

Figure 5.3: The specialized power function

```
const compute == object something
   function Ackermann[m, n] → [result]
      var newm
      var newn
      if m.eq[0] then
         result ← n.plus[1]
      elseif n.eq[0] then
         newm ← m.minus[1]
         result ← self.Ackermann[newn, 1]
      else
         newn ← n.minus[1]
         newm ← m.minus[1]
         newn ← self.Ackermann[m, newn]
         result ← self.Ackermann[newm, newn]
      end if
   end Ackermann
end something
```

Figure 5.4: An implementation of Ackermann's function

under dynamic control [Bondorf 90b]. That a loop is under dynamic control means that it is the value of a dynamic variable that controls when the loop should be exited. The partial evaluator should be able to handle this classical problem correctly.

Partially unfolding some of the recursion in the implementation of Ackermann's function may yield faster residual programs. The first argument to the function may also be removed in the function of the residual program. If the specialization is performed so the first argument to Ackermann's function is (say) 2, the residual program should have a structure similar to that of the program listed in Figure 5.5. If the partial evaluator does not perform the unfolding, the residual program should have the same structure as the source program. We do not require the partial evaluator to generate a residual program where the function is partially unfolded, but will rather say that it is an optional extra.

```
const compute == object something
    function Ackermann2[n] → [result]
        var newn
        if n.eq[0] then
            result ← 3
        else
            newn ← n.minus[1]
            newn ← self.Ackermann2[newn]
            result ← self.Ackermann1[newn]
        end if
    end Ackermann2
    function Ackermann1[n] → [result]
        var newn
        if n.eq[0] then
            result ← 2
        else
            newn ← n.minus[1]
            newn ← self.Ackermann1[newn]
            result ← self.Ackermann0[newn]
        end if
    end Ackermann1
    function Ackermann0[n] → [result]
        result ← n.plus[1]
    end Ackermann0
end something
```

Figure 5.5: The structure of an unfolded version of Ackermann's function

One of the most common uses of partial evaluation is the application to interpreters. The partial evaluator should be powerful enough to perform translation in the usual way. In Figure 5.6 we have listed an interpreter for a simple imperative language in which programs can only have two variables x and y[1]. The variable x is initially set to the input value. The value of the variable y after the computations are finished is the return value. The vocabulary of the language is given in Figure 5.7. If the integers' size is unbound, the language is *Turing complete*. The interpreter is written in the usual style using dispatching on the language constructions. We have assumed the program to be an object that has a method *getline* that returns string objects with the statements.

Consider specialization of the interpreter function with respect to a representation of the following program that multiplies the argument with two:

ifX0 5, addy, addy, subx, goto 0, halt

We expect the residual program to have the same structure as the program listed in Figure 5.8.

In object-oriented programming languages, it is possible to create objects at run-time. An example of a program that creates objects could be an interpreter for the above language that operated with one object acting as the store of the two variables, and one object for each statement in the program to be interpreted. The interpreter should create all the statement objects guided by a string representing the program. The statement objects should have methods performing the actions dictated by the semantics of the statement and passing on control to the next statement object to be executed. The exact details of such an interpreter and a listing of an interpreter program operating after this principle is not in the scope of this section. Specializing such an interpreter with respect to the same program as before should also yield a residual program with the same structure as the program listed in Figure 5.8.

This example illustrates two important points. If the program to be interpreted is given as the known input to the partial evaluator, then the partial evaluator should show its ability to handle construction of objects under static control. If the program to be interpreted is the unknown input, then the partial evaluator will have to show that it is able to handle a construction of objects in dynamic loops.

5.4 Choice of Strategies

This section will describe some solution strategies and our choices of which to use. The properties concern the overall mechanism of the partial evaluator.

5.4.1 Strictness

The partial evaluator may be strict or nonstrict as defined in Chapter 2. We will not require the partial evaluator be strict since this complicates the algorithm and requires extra work arguing that the partial evaluator actually is strict.

[1]The example is taken from [Jones 92]

```
const imperative == object language
    function interpret[program, startx] → [result]
        var instruction
        var x   var y
        var code   var temp
        x ← startx
        y ← 0
        instruction ← 0
        loop
            code ← program.getline[instruction]
            instruction ← instruction.plus[1]
            exit when code.equal["halt"]
            if code.equal["addx"] then
                x ← x.plus[1]
            elseif code.equal["addy"] then
                y ← y.plus[1]
            elseif code.equal["subx"] then
                x ← x.minus[1]
            elseif code.equal["suby"] then
                y ← y.minus[1]
            else
                temp ← code.getslice[5, -1]
                temp ← temp.asInteger[]
                code ← code.getslice[1, 4]
                if code.equal["ifX0"] then
                    if x.eq[0] then
                        instruction ← temp
                    end if
                elseif code.equal["ifY0"]
                    if y.eq[0] then
                        instruction ← temp
                    end if
                else % code.equal["goto"]
                    instruction ← temp
                end if
            end if
        end loop
        result ← y
    end interpret
end language
```

Figure 5.6: An interpreter for a minimal language

`halt` `addx` `addy` `subx` `suby` `ifX0` *linenumber* `ifY0` *linenumber* `goto` *linenumber*

Figure 5.7: The vocabulary of the simple imperative language

```
const imperative == object language
   function interpret[startx] → [result]
      var x
      var y
      x ← startx
      y ← 0
      loop
         exit when x.eq[0]
         y ← y.plus[1]
         y ← y.plus[1]
         x ← x.minus[1]
      end loop
      result ← y
   end interpret
end language
```

Figure 5.8: The structure of the specialized interpreter

5.4.2 Automatic Partial Evaluation

If the partial evaluation process does not require any user annotation of the program to be specialized, the process is said to be automatic. In the past it has been quite common to start by requiring user-annotations when trying to use partial evaluation on a new type of source language. When the technique is well understood, the nonautomatic partial evaluators are replaced by an automatic ditto.

Even though we explore a new type of language to apply partial evaluation to, we require that the partial evaluator is automatic. The technology of partial evaluation has by now reached a state where it should be possible to construct an automatic partial evaluator even for a new type of language.

5.4.3 Termination Properties

Besides the termination properties of the generated residual programs, it makes sense to talk about the termination properties of the partial evaluator itself. The partial evaluation process may fail to terminate for some source programs. To our knowledge there does not exist any nontrivial automatic partial evaluator that is guaranteed to terminate for all syntactic correct source programs. Nonautomatic partial evaluators (those requiring hand-annotations of programs to be specialized) cannot be said to have any termination properties.

A general requirement, which we also insist upon, is that the partial evaluator terminates even if the source program contains loops with static variables under dynamic control. This was also part of the problem statement in the previous section. In our language there are two kinds of loops we must be aware of: recursive and iterative.

Objects can also be created in loops under dynamic control. We require that the partial

```
const a == object devious
   function New[dyn] → [result]
      result ← object Dummy
         var temp
         function recur[] → [result]
            var aVar   var newDyn
            newDyn ← temp.minus[1]
            if temp.zero[] then
               aVar ← a.New[newDyn]
               result ← aVar.recur[]
            else
               result ← nil
            end if
         end recur
         initially
            temp ← dyn
         end initially
      end Dummy
   end New
end devious
   ⋮
b ← a.New[SomeValue]
c ← b.recur[]
```

Figure 5.9: A "loop" of objects creating objects creating objects ...

evaluation process terminates even if objects are created in loops under dynamic control. The creation of objects in loops has an extra twist to it, since a newly created object can have a method that creates a new object of the same kind and subsequently invokes this same method in the new object. This is illustrated by the program in Figure 5.9. We require that the partial evaluation process terminates even if the source program contains such "loops" under dynamic control.

Some partial evaluators have so strong termination properties, that even if the source program never terminates because of an infinite loop, then the symbolic computations during partial evaluation will still terminate, and the partial evaluation process terminates. We do not require the partial evaluator to terminate if the source program contains nonterminating loops under static control.

5.4.4 Self Application

The second and third Futamura projections require the partial evaluator to be self applicable. This again requires the partial evaluator to be written in its own source language.

We find that the final partial evaluator will be very complicated and large because of the many different language constructs in our language. Because of the sheer size of

the program, we would like to implement it in a language with strong type checking and other facilities to help the programmer and not in our own language which is very difficult to write large-scale programs in. This implies that the partial evaluator will not be self applicable. The partial evaluator may later be implemented in its own source language and may even be self applicable, but we will not attempt this as part of this project.

However, a good way to investigate the possibility of self application is to test whether partial evaluation of an interpreter with respect to an original program yield a specialized program, more or less equal to the original program. This can be stated in the equation,

$$p' = [\![Mix]\!] \text{ sint } p$$

where p' is the residual program, Mix is the partial evaluator, sint is a self-interpreter, and p is the original program. If p' is close to p, then it should not be too difficult to achieve self applicability of a specializer.

5.4.5 Representation of Unknown Values

Consider an undecidable `if`-statement where a variable is assigned the value 5 in one branch and the value 4 in the other branch. What is the value of the variable after the execution of the `if`-statement? Since we cannot decide at the time of partial evaluation what branch will be taken, the value must be regarded as an unknown value. Such unknown values can be represented in several ways.

Suppose we represent unknown values by a special abstract value (let us call it *dynamic*) and use a closure analysis (object constructor analysis) to tell us what object constructors a certain unknown value may be created by. An invocation of a procedural method in an unknown object is then problematic. All variables, that are assigned values in the procedures possibly invoked, must then be annotated as being dynamic. The value of any expression involving instance variables is also unknown, so many tests will be undecidable and many targets of invocations will be unknown. If there is more than one object constructor with a procedural method with matching name and number of arguments, then all invocations of methods from inside the method body will have to be treated as "dynamic" invocations way even if the target of the "inside" invocation is known.

An alternative is to represent the value of a variable with an unknown value by the set of values that the variable might be assigned to. The domain of builtin values is infinite, so to ensure finiteness in the cases where an unknown value may contain an infinite number of values, we have to have special symbolic values representing all the integers and all the strings. The number of objects possibly constructed during program execution are also unbounded, so we must also have symbolic values representing an unbounded number of objects constructed in a loop. Finally, we must have a symbolic value representing the truly unknown values. These are the unknown input values of the program. With this representation of unknown values, the effect of invoking a method in an unknown object is under much stricter control.

We think that if we represent the unknown values with a simple symbolic value (dynamic) during abstract interpretation, then we shall loose too much information for any

real life programs. This is closely related to our language having both closures and side effects and cannot immediately be related to partial evaluation of languages not combining these two features.

It may be possible to rewrite a program to prevent a great loss of information, but our attitude to partial evaluation is that it should be applicable to real programs without too many modifications (if any). We will therefore represent the value of a variable, with an unknown value, by the set of values that the variable may contain. We do not know of any other partial evaluators representing the unknown values in this way, i.e. by a set of values.

5.4.6 Online vs. Offline Techniques

A fundamental choice to be made is whether the partial evaluator should use online or offline techniques. Each has its strong and weak sides. Using online techniques usually make good termination properties harder to achieve than if using offline techniques. More information must be accumulated and maintained in order to ensure termination, making online techniques slower than offline techniques. The generated residual programs are often larger using online techniques than using offline techniques. On the other hand, using online techniques makes it possible to retain more information than using offline techniques, because it is making the way for better specialization in a number of cases.

Since we want to represent the value of a variable with an unknown value by the set of values that the variable might be assigned to, the natural choice is to use online techniques. The same symbolic computations would have to be performed independent of whether online or offline techniques are used due to our chosen representation of unknown values. However, online techniques introduces some overhead due to the bookkeeping necessary to identify loops. It is our guess that the overhead using online techniques is less than or equal to the overhead incurred by the closure analysis (object constructor analysis) and the binding time analysis needed by the offline methods. Using online techniques will lead to a slower partial evaluator than using offline techniques. Again, this is only because of our choice of representation of unknown values.

We think that it is possible to achieve the desired termination properties of the partial evaluator even if we use online techniques. We have several ideas as how to achieve this. In Section 5.6 we will elaborate on these ideas.

5.5 Problem Analysis

5.5.1 Explicators

When using online techniques the decision of whether to specialize or residualize an assignment statement is to be taken when the statement is encountered. This, however, cannot be determined when the statement is encountered! Even if it seems that an assignment statement can be specialized, it may later turn out that the assignment has to be performed in the residual program anyway. Consider the following program fragment:

```
a ← 4
if dynamic.test[] then
    a ← 5
else
    b ← b.plus[1]
end if
```

When first looking at the assignment of the integer value 4 to the variable a it looks as if the assignment can be performed solely at specialization time. In the first branch of the undecidable if-statement the variable is however assigned a new value while it is not assigned a new value in the other branch. The value of the variable a is unknown after the if-statement and the two assignments to the variable should be left in the residual program.

One way to ensure that the two assignments are inserted in the residual program is to compare the two environments (mapping names to symbolic values) resulting from symbolic execution of the two branches of the if-statement. If a variable has different values in the two environments and one of the values are global, then assignment statements are inserted to ensure that the variable indeed will have these values in the residual program. The generated assignment statements are called *explicators*. The technique is described in further detail in e.g. [Meyer 91][2].

The technique can only be used if all variables that we need to generate explicators for are in the current scope. In our language this is not always the case since the state of other objects can be modified by method invocations in the branches of if-statements. Another technique has to be invented for our use. The technique we have invented will be sketched in the next section.

5.5.2 Global vs. Known Values

Not all assignments of known values to variables can be specialized so they do not appear in the residual program. This is because some values may be known without being globally accessible.

Our language supports initialization of objects. If the initialization of an object requires references to e.g. some of the unknown input objects, then the object cannot be constructed at load-time by a constant declaration in the residual program. Such objects will have to be created at run-time and will not be accessible from all parts of the residual program. These objects may be known to exist when flow of control reaches a given point in the source program, but since the reference to the object is not globally accessible we cannot specialize assignments of such known (but not global) values.

By trying to make as many objects as possible bound to globally known constants we make more specialization possible. The necessary criteria for moving the creation of an object from run-time to load-time is that only globally accessible values are used during initialization of the object.

[2]The term *explicators* were first coined by A.P. Ershov. We have however been unable to find the reference to the relevant paper.

5.5.3 Methods in Unknown Input Objects

The unknown input values may be objects with both functional and procedural methods and may have a state that can change. If we assume no knowledge whatsoever about the unknown input values, then we do not even know if they have references to the globally accessible values of the source program. A procedural method in an unknown input object may thus theoretically invoke all methods in the global values, and also in objects passed on to them as arguments. If we do not put any restrictions on the allowed behavior of the procedural methods in the unknown input objects, then we cannot assume any knowledge of the parts of the program state that it is possible to modify. Objects created by evaluation of methods in the unknown input values are themselves considered unknown input values. This is clearly unsatisfactory and makes partial evaluation virtually impossible. The reason that this has not been a problem with previous partial evaluators is that they do not allow higher order values as input values to the program to be specialized.

We could impose the restriction on the unknown input values that they are not allowed to have procedural methods. Another possible restriction is that the methods of unknown input objects may not modify the state of any object created by the object constructors of the source program (program objects). Another restriction could be that the unknown input values may not invoke methods in the program objects. We should add and enforce a third and stronger condition. Unknown input objects are not allowed to invoke *any* methods in program objects.

We will impose the last of these restrictions on the allowed behavior of the unknown input values. This will allow the partial evaluator to remove and/or rename methods at will as long as the functionality of the program is the same. The restriction also implies that we cannot specialize a program that pass on a container object as an argument to any unknown input object and expects that unknown input object to modify the contents of the container object. Probably the semantics of the generated residual program will be undefined (the program contains errors) or be different than the semantics of the source program.

5.5.4 Forced Use of Method Names

Consider an invocation where the target of the invocation can be both objects created by an object constructor of the source program and basevalues and/or unknown input values. We cannot modify the method names for the builtin values or the unknown input values so we are forced to use the same method name in the residual program as in the source program. This can potentially give us problems because we want the partial evaluator to be polyvariant with respect to specialization of methods. If there is only one invocation of a method with a forced method name, then there is no problem in generating a specialized method. If there are several invocations, then we cannot generate several specialized methods in the same object, since the method names are forced upon us.

One way to tackle this problem is to copy the method definitions from the source program into the residual program and make sure that there are explicators for all used variables in these methods. An alternative strategy is to generate a specialized method usually the first time such a method is needed. Subsequently the partial evaluator are

to generate a new more general method, with the same name, to replace the existing method in the same object constructor. The environment used to generate the new (more) general method is found by generalizing the previously used environment and the current environment.

We want the partial evaluator to use this last strategy. To our knowledge neither of the two strategies are described in the literature and the latter seems the most interesting to investigate.

5.5.5 Postphase Optimizations

Unfolding Methods

For partial evaluators for applicative languages, unfolding functions is a very important part of the process. Unfolding a function application saves the time required to pack and unpack parameters and the result, and to perform the function lookup. Unfolding method invocations can be divided in two important cases: methods in the current object, and methods in other objects.

Methods in the current object can be unfolded the same way as in applicative languages. Direct recursive methods cannot be unfolded. The recursion may be (partially) unrolled but this is another matter. Because the actual parameters are basic expressions there is no need for occurrence counting [Bondorf 90b] to determine if unfolding may lead to duplicating computations (it never does) so unfolding is somewhat simpler. The only extra problem is if an instance variable is used both as actual parameter to and modified in a procedural method. In that case an extra local variable is needed to preserve the call-by-value semantics of the invocation.

Methods in another object can only be unfolded if the accessed variables all are in the current objects scope. This will mostly be the case if the methods do not refer to any instance variables.

We want to unfold the invoked methods in the current object unless they are directly recursive. We will not unfold invoked methods in other objects. We fear that this will make the residual programs harder to read and understand and that the analysis to determine if this nonlocal unfolding is possible will increase specialization time without improving program speed considerably. Further, invocation of methods with empty bodies, i.e. only skip statements should be removed from the program.

Merging Objects

In many actual programs some objects will be entirely local to another object in the sense that methods in the "local" objects are only invoked from the other object. From our experience with the small programs we have constructed and specialized by hand, this is often the case in specialized programs.

We would like to merge such objects into one object, so only one object constructor has to be evaluated instead of two. Depending on the compiler for the language, merging objects will probably make the residual program faster. The merging of objects can also make extra unfolding of methods possible.

A necessary condition of two objects to be merged is that one of the objects is accessible in at least all the places where the other object is accessed. Also the two objects may never be compared in a test using the equality (==) operator.

Two objects to be merged should have disjoint sets of methods (methodnames). Merging may be possible even if the sets of methodnames are not disjoint. This would however (if possible) require some clever renaming.

We want to merge objects. We do not want to make advanced analyses to determine the accessibility of parent objects, so we will only merge objects if both are globally accessible. The merging should be performed after the specialization process is finished.

Garbage Collection

The source program might contain constant declarations which are never used. There may also be methods, instance variables and local variables which are never used. Such unused parts may be propagated into the residual program, where even more variables may have become superfluous because the computations they were required for has been performed symbolically. Such *garbage* may be removed by a post phase of the partial evaluator. We will include this kind of garbage collection in the partial evaluator.

The program may also create objects that are never used in any way. This can only be detected during abstract interpretation. An object can be regarded as unused if it never has a method invoked and never is used in a undecidable comparison using the equality (==) operator. We do not want to collect this kind of garbage objects as we do not think the extra computation cost is worthwhile.

If two local variables are *alive* [Aho 86] in nonoverlapping parts of a method body, then the two variables may be replaced by one. Having the partial evaluator perform this transformation may make the residual program harder to read, and a good optimizing compiler would perform the optimization anyway. We will not require the partial evaluator to perform this optimization.

Transforming Recursion to Iteration

Interpreters written in the *active syntax* style illustrated in Appendix D will when specialized yield programs where recursion is used to model iteration in the program the interpreter is specialized with respect to. This does not matter much if the compiler for the target language is written so tail recursion is optimized into direct jumps. The transformation of direct tail recursive methods into loops could however easily be performed by a post phase to the partial evaluator. We will not perform this kind of transformation.

5.6 Ideas for Construction of the Partial Evaluator

In the previous sections we have stated what we want the partial evaluator to do in terms of our expectations to specialization of some example programs and a list of qualities we want it to have when tackling certain problems. In this section we will briefly describe some of our ideas about *how* we will construct a partial evaluator that will achieve these

goals. We will describe how we are going to let the partial evaluator generate fragments of residual code during abstract interpretation of the source program. How we are going to develop the partial evaluator. How to ensure the termination properties we want. Finally how the partial evaluator should generate explicators for variables in other object's scope. The technical details will be explained exhaustively in later chapters of the report; this is only to serve as a brief introduction.

5.6.1 Code Generation During Abstract Interpretation

A general strategy we will use is the generation of fragments of residual code during abstract interpretation of the program. This is closely related to our choice of online techniques.

We will perform an abstract interpretation of the program using symbolic values. For all symbolic computations we decide either to specialize the computation or to generate a fragment of the residual program performing the same computation, but with actual values. Assignment of values we know will be globally accessible in the residual program is only performed symbolically. So is invocations of functional methods if the value returned is globally accessible. Example 5.1 illustrates the technique. As phrased in [Ruf 91] we perform a symbolic execution of the program while generating a *trace* of the computations that could not completely be performed at the time of specialization. These computations are those, which have to be performed by the residual program when the unknown input values become available. It is not phrased in the same way but it is nevertheless the same technique as described in [Gluck 91].

Example 5.1 Generation of code during abstract interpretation
Consider the following object constructor expression:

```
object craze
    var xxx    var yyy
    function nonsense[dyn] → [result]
        a0: result ← 4
            if dyn.test[] then goto a1 else goto a2
        a1: result ← 5
            return
        a2: skip
            return
    end nonsense
    initially
        b0: xxx ← self.nonsense[dynamic]
            yyy ← xxx.plus[1]
            return
    end initially
end craze
```

Let us assume that we during the abstract execution of a program are about to execute the above object constructor expression in an environment where the name *dynamic* maps to an unknown value. As part of the evaluation of the object constructor expression we symbolically execute the initialization part.

The assignment of the value 4 to the variable *result* is performed completely symbolically. The two branches of the undecidable branch instruction are then executed independently and a residual test and branch instruction is generated. In the first branch the assignment is also performed symbolically.

Depending on which branch is taken, the value returned is either 4 or 5. Since the two assignments were only performed symbolically explicators must be generated to make sure the assignments (that was first thought to be specializable) are generated anyway. The function returns an unknown value so code is generated both for the method and for the invocation. The value assigned to the variable *yyy* is unknown so this assignment is also added to the trace of computations to be performed by the residual program.

5.6.2 Stepwise Development of a Partial Evaluator

We shall develop the partial evaluator using a stepwise strategy starting from the formal semantics for the source language. Briefly, the steps in the development process are:

1. Give the semantics of the source language
2. Define the domain of abstract values to be used and give the semantics for a naïve abstract interpretation
3. Give the abstract interpretation the required termination properties
4. Add code-generation to the abstract interpretation semantics
5. In languages with methods, functions and/or procedures, add facilities for reuse of these
6. Develop the post phase of the partial evaluator

The starting point is a formal description of the semantics of the source language. Preferably the semantics should be an operational semantics. Having defined the domain of abstract values to be used in the partial evaluator, the standard semantics can be modified to give a *nonstandard semantics*. The nonstandard semantics describes how to pursue all possible execution paths while performing the symbolic computations. Termination properties are irrelevant at this stage.

The first stages are rather trivial. The third stage does however require some ingenuity. Some kind of mechanism must be invented ensuring that the required termination requirements are fulfilled. The abstract interpretation of the program should terminate while still investigating all possible program paths and using environments that abstracts all possible environments. Ensuring proper termination properties has historically been one of the big problems with online partial evaluation.

The next stage is adding code generation to the semantics developed so far. For languages with assignment this involves generating explicators or something equivalent to this.

A code is generated for all execution paths that are investigated. The previous phases did not do anything to reuse the generated code or already investigated execution paths. This might be worth doing for some languages (if not all). This can be done using different degrees of ingenuity. An example of a smart technique, which optimizes the reuse of functions using inferred type information is described in [Ruf 91].

The last phase of the construction of the partial evaluator is the development of the postphase performing last minute optimizations and combining all generated program fragments to the final residual program.

We believe that the strategy described here can be used to develop an online partial evaluator for any desired language. This is notable since there is not much material on the process of developing a partial evaluator.

5.6.3 How to Ensure the Termination Properties We Want

The source of the problem with termination is loops. Both loops that never will terminate and loops that we cannot determine when will terminate. We are only interested in the last type of loops. The cure is well known: generalization of program state. If the program state at all points where a loop is closed is exactly as general as or less general than the program state at the entrance of the loop, then the abstract interpretation can be said to have investigated all possible execution paths in the loop.

Since we want the specializer to be polyvariant in the generation of methods we cannot permit generalization over all invocations of a given method. We have to find a technique to identify the loops where there is a genuine chance of a recursive loop under dynamic control in the program. We would also like a technique for identifying genuine iterative loops under dynamic control to avoid mindless generalization of program states whenever two execution paths within the same method can lead to execution of the same basic block.

We want to identify recursive loops by maintaining information, which is an abstraction of a call stack. We are only interested in detecting the loops, where we cannot determine if they will terminate or not. Only information about methods containing undecidable branches will thus be kept in our abstraction of the stack call. If we during our symbolic computations are about to invoke a method that we have stored information about in our call stack abstraction, then we have found a loop we must take precautions against. A similar technique can be used for detecting iterative loops. The call stack abstraction can also be used to identify the object creation loops illustrated by the program in Figure 5.9. This is out of the scope of this section and will be explained in a later chapter.

Our technique only identifies loops under dynamic control. Loops under static control are not detected. This is exactly the properties we want.

5.6.4 How to Generate Explicators

Generating explicators in the current object is no big problem. In this case the conventional techniques can be used. When explicators are needed in other objects the conventional techniques cannot be used. As stated in the previous section we have to invent a new technique for this situation. Example 5.2 illustrate the situation.

Example 5.2 Explicators may be needed in other objects
Consider the following program fragment to be symbolically executed in an environment where the variable x reference a container object with a single cell that can be modified by the method *set*:

```
x.set[4]
if dynamic.test[] then
    x ← x.set[5]
end if
```

The first modification of the cell in the object referenced by x is first performed completely symbolically. The second modification is performed conditionally so both modifications should be performed in the residual program.

In this simple example the problem can be solved by inserting an invocation of a method performing the necessary assignment. It is however no problem to construct a larger example, where the object, which the explicator should be inserted in, is not accessible from the point in the program, where we identify the need for the explicator.

Instead we use another technique we have invented for the purpose. We add some information to the symbolic values showing where the value is assigned. This is only relevant for symbolic values representing known, globally accessible values. When the need for generation of explicators arises, the symbolic value has the information necessary for the generation of the assignment and inserting this at the right place in the residual program.

The explicator assignments are inserted in the residual program in the position where the original assignment would have been, if it could not have been performed completely symbolically (which it really could not). Using this technique therefore makes the residual program resemble the source program more than if the traditional techniques are used.

This page intentionally left blank.

Chapter 6

Abstract Interpretation with Symbolic Values

The first step in our development of a partial evaluator is the development of an abstract interpretation algorithm to perform the essential symbolic computations. In this chapter we describe such an algorithm. The termination properties of the algorithm are very poor. This will not concern us in this chapter.

The purpose of the abstract interpretation is to explore all possible execution paths while maintaining safe approximations to the actual program states that can occur during actual program execution with the unknown input parameters replaced by actual objects.

6.1 Maintaining a Maximum of Information

The program state is in our language a global environment and a set of closures each with an environment. All environments maps names to *object references*. The nearest equivalent of this in ordinary imperative languages is *pointers*.

During abstract interpretation the references are replaced by abstract values. At a given point on a possible execution path, an abstract value should represent all the possible actual values at the same point on the execution path during actual execution of the program. Because of the possibility of conditional jumps where it is undecidable which branch to take, this may be more than one object. The symbolic values thus have to be sets of *object values*. There is one object value denoting all the unknown input values. All other object values denote individual objects. In this chapter we do not have object values representing an unbounded number of either builtin objects or program object.

In other words, we get information on exactly what objects a variable can denote during actual execution of the program. If a variable at a specific point on a specific execution path can be bound to either a stack object or one of two different queue objects depending on the unknown input values then the symbolic values will be able to accommodate this information. There is however no way to reflect invariants of the kind "`x` is never bound to the same value as `y`" in the program to be specialized.

6.2 The Semantics of the Abstract Interpretation

The semantics of the abstract interpretation is in many way similar to the semantics of the standard interpretation. The major differences are the handling of conditional jumps and method invocations.

If it cannot be determined which of the branches is to be taken in a conditional jump, then both branches are taken and the results generalized. The result of taking a branch in a conditional jump is the program state at the end of the method or initialization part. The generalized result must of course be a safe approximation to the result of taking either branch. Program states are generalized by using the least upper bound of the two program states. This ensures that whatever branch is taken, the resulting store is still a safe description of the possible states.

The partial order on program states will be defined later. Informally, it is based on the partial order on symbolic values used for each object reference in the program state. The used partial order on symbolic values is equal to subset inclusion.

Because the symbolic values are sets of object values, the target of an invocation may not always be well known. If it is not well known what the target of an invocation is, then we call the invocation a *nonmanifest invocation*. We will also call abstract values denoting more than one object a *nonmanifest value*.

In a nonmanifest method invocation all the possible "branches" are taken. The branches in this case are the methods in the different objects that may possibly be invoked. The program state after the invocation is again computed by finding the least upper bound of all the branches' program states after the invocation.

This strategy of taking more than one branch is safe in the sense that all possible execution paths are pursued. It is however unsafe in the sense that it may lead to test expressions in conditional jump instructions evaluating to neither of the boolean values. Nonexisting methods may also be invoked in objects or a wrong number of parameters may be used in invocations. This does not necessarily reflect faulty programs as we might pursue branches that will never be taken. The abstract interpreter must in a reasonable way handle all error situations that might happen due to taking "impossible" execution paths.

We assume that there is *minimal interaction* between the unknown input values and the rest of the program (as stated in Chapter 5). In order to (re)define what this mean, we define the set of *program objects* to be all the objects created by the constant declarations of the program and all objects created by object constructors inside program objects. There is minimal interaction between the unknown input values and the rest of the program *if and only if* methods in program objects are invoked only by statements and expressions in program objects *and* the only functional methods returning program objects are methods in program objects.

The minimal interaction property implies that a function in an unknown input object will never return a program object. Objects returned by functional methods in unknown input objects can themselves safely be considered being unknown input objects. If the unknown input objects are later instantiated with real objects, the value returned may either be one of the given objects, a builtin object, or an object created at runtime (but

not a program object). The builtin objects fulfill the minimal interaction property and the other objects are required to do so.

The abstract interpretation is what is sometimes called an interpretation with non-standard semantics. The semantics of the interpretation is formally defined in Figure 6.4 to Figure 6.8 using the same notation as in Chapter 4. The semantic domains used are different. They are shown in Figure 6.1 and 6.2. The semantic rules for abstract interpretation of constant declarations, basic expressions, and sequences of basic blocks are not given in this chapter. They are textually equal to the semantic rules given in Chapter 4 (Figures 4.8, 4.9, and 4.12).

The used partial order on symbolic values is simply equal to subset inclusion. The used partial order on environments is the natural extension to mappings. The part of the program state we are interested in comparing is a local variable environment and a finite mapping from object identifications to instance variable environments. The used partial order is thus the extension of the partial order on environments. The partial orders used can be defined as follows:

$$\begin{array}{ll}
\text{(abstract values)} & x \sqsubseteq y \Leftrightarrow x \subseteq y \quad \text{(subset inclusion)} \\
\text{(abstract environments)} & x \sqsubseteq y \Leftrightarrow \forall\, n \in \mathrm{Dom}(x) : x(n) \sqsubseteq y(n) \\
\text{(program states)} & (x_1, x_2) \sqsubseteq (y_1, y_2) \Leftrightarrow \forall\, n_1 \in \mathrm{Dom}(x_1) : x_1(n_1) \sqsubseteq y_1(n_1) \wedge \\
 & \qquad\qquad \forall\, n_2 \in \mathrm{Dom}(x_2) : x_2(n_2) \sqsubseteq y_2(n_2)
\end{array}$$

The definitions of the used basic semantic objects are listed in Figure 6.1. The "Arg" semantic object is new. It describes the possible arguments to the abstract interpreter.

FormalPar	∈	FORMALPAR
LVarName	∈	LVARNAME
ResultName	∈	RESULTNAME
LocalName	∈	LOCALNAME = FORMALPAR ∪ LVARNAME ∪ RESULTNAME
IVarName	∈	IVARNAME
ConstName	∈	CONSTNAME
FnctName	∈	FNCTNAME
ProcName	∈	PROCNAME
MethodName	∈	METHODNAME = PROCNAME + FNCTNAME
Ω	∈	Abstract_Object = Program_Object + nil
bv	∈	Builtin_Object
Arg	∈	ARG = CONSTNAME + BaseValueName + `nil` + `Dynamic`
InputDynamic	∈	INPUTDYNAMIC = {InputDynamic}
x		OID = Builtin_Object + Abstract_Object + INPUTDYNAMIC
true	∈	OID
false	∈	OID
$\mathcal{B}$	∈	Body = BBlist

Figure 6.1: Basic Semantic Objects for Nonstandard interpretation

The definitions of the used compound semantic objects are listed in Figure 6.2. What was formerly object references (OIDs) are now AbsVals that are sets of OIDs.

If x is a set, then $\mathcal{P}(x)$ is the corresponding *powerset*. That is, if s is a set of integers, then $\mathcal{P}(x)$ is the set of sets of integers.

The signatures of the semantic functions used in the semantic rules are given in Fig-

$$
\begin{array}{rcl}
\beta & \in & \text{AbsVal} = \mathcal{P}(\text{OID}) \\
\widehat{\beta} & \in & \text{Arglist} = \text{AbsVal list} \\
\text{Code} & \in & \text{CODE} = \\
 & & \quad \text{PROCNAME} \xrightarrow{\text{fin}} \text{FORMALPAR list} \times \text{LVARNAME list} \times \text{Body} \; + \\
 & & \quad \text{FNCTNAME} \xrightarrow{\text{fin}} \text{FORMALPAR list} \times \text{RESULTNAME} \times \text{LVARNAME list} \times \text{Body} \\
\tau & \in & \text{LocalEnv} = \text{LOCALNAME} \xrightarrow{\text{fin}} \text{AbsVal} \\
\imath & \in & \text{InstanceEnv} = \text{IVARNAME} \xrightarrow{\text{fin}} \text{AbsVal} \\
\rho & \in & \text{Object\_Store} = \text{Abstract\_Object} \xrightarrow{\text{fin}} (\text{CODE} \times \text{InstanceEnv})
\end{array}
$$

Figure 6.2: Compound Semantic Objects for Nonstandard Interpretation

ure 6.3. The γ function maps the name of the builtin objects into the object values denoting these objects. The ϵ is used to transform a list of method definitions into a function that maps methodnames into method definitions. The **DoBaseFnctCall** function implements semantics of all the builtin objects. The function should be robust in the sense that it should be able to handle invocations of nonexisting methods and invocations with a wrong number of arguments. The "Manifest" function is a predicate that determines if a symbolic value is manifest or not. A symbolic value is manifest if the set only contains one element, and that element is not InputDynamic. The predicate "NonManifest" is the logical complement to "Manifest".

$$
\begin{array}{rcl}
\gamma & \in & \text{BaseValueName} \mapsto \text{Builtin\_Object} \\
\epsilon & \in & \text{MethodDef*} \mapsto \text{CODE} \\
\textbf{DoBaseFnctCall} & \in & \text{Builtin\_Object} \times \text{FNCTNAME} \times \text{Arglist} \mapsto \text{AbsVal} \\
\text{Manifest} & \in & \text{AbsVal} \mapsto \text{Boolean} \\
\text{NonManifest} & \in & \text{AbsVal} \mapsto \text{Boolean}
\end{array}
$$

Figure 6.3: Semantic functions for the Nonstandard Interpretation

The semantic rule in Figure 6.4 is a slight modification of the rule in Figure 4.7. The difference being that the list of arguments now may include the name `Dynamic`. We have not given any rule for interpretation of this name, as this is the only place where the name may occur, and that it obviously represents the symbolic value: {InputDynamic}.

$$
\frac{\begin{array}{c}
\vdash [\![\mathit{Program}]\!] \Rightarrow \rho \\
\rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \Omega \\
\rho \vdash [\![\mathit{Arg*}]\!] \Rightarrow \widehat{\beta} \\
\vdash \textbf{ProcCall}(\Omega,\ \mathit{ProcName},\ \widehat{\beta}) : \rho \rightarrow \rho' \\
\vdash \textbf{FnctCall}(\Omega,\ \mathit{FnctName},\ [\,]) \Rightarrow \beta : \rho' \rightarrow \rho''
\end{array}}{\vdash [\![\mathit{Program\ ConstName\ ProcName\ Arg*\ FnctName}]\!] \Rightarrow \beta}
$$

Figure 6.4: Semantic rules for the input–output function of a program.

The semantic rules in Figure 6.5 describes the semantics of nonstandard interpretation of expressions. The first rule describes the semantics of an expression that is only a basic

expression and is textually equal to the similar rule in Chapter 4. The semantics of the equality operator is described by the next three rules. The interesting point is that if either of the two basic expressions evaluate to a nonmanifest value and there is a chance that the two actual values might be equal, then the expressions evaluates to a symbolic value describing *both* the truth and the falsity value.

The fifth rule describes the interpretation of a function invocation expression. Since the target of the invocation may be more than one object, the program state and the result value is the least upper bound of the program states and result values stemming from interpretation of each possible branch. Note that **FnctCall** has another functionality than in Chapter 4. The semantic description of **FnctCall** is given in Figure 6.8.

Because the abstract interpreter takes all branches in undecidable branch situations, we can no longer suffice with requiring that the semantic object Ω' gets a value that is not in the domain of the ρ semantic object. Instead we require that the value is in the "Abstract_Object" semantic domain and require that a new value is used each time. In other words, we assume that the set inclusion test is *generative*. We use this assumption silently in the rest of the Thesis.

The semantic rules in Figure 6.6 describe the semantics of nonstandard interpretation of statements. Most rules are textually identical to the similar rules in Chapter 4. Assignment to local variables are now described by one semantic rule instead of two. The rule describing invocation of procedural methods is changed because the target may be more than one value. The change is similar to the change in the rule for functional method invocation. Also **ProcCall** has another functionality than in Chapter 4. The semantic rules describing **ProcCall** are to be found in Figure 6.8.

The semantic rules in Figure 6.7 describe the semantics of nonstandard interpretation of the syntactic category Jump. The two first rules are textually identical to the similar rules in Chapter 4. The last four rules describe interpretation of conditional jumps between basic blocks. The interesting thing to note is that both branches are interpreted if the test may evaluate to both boolean values or to some unknown input value. If the test evaluates to something that cannot be a truthvalue, then none of the branches are investigated as a runtime error will occur if this execution path is ever investigated.

The semantic rules for abstract interpretation of method invocations are given in Figure 6.8. The first seven rules describe the handling of different error situations: invoking a procedural method in a builtin object, invoking methods in the **nil**, invoking nonexisting methods, and using a wrong number of arguments. Rule number eight and nine describes the semantics of invoking methods in unknown input objects. The program state is not changed (the minimal interaction criteria), and the result of invoking a functional method can safely be approximated by an unknown value as explained earlier.

The third from the last rule describes the semantics of invoking a functional method in a builtin object. The rule uses the semantic function **DoBaseFnctCall**. This semantic function is different from the one used in Chapter 4 because it has to be fault tolerant. It should return the minimal symbolic value (the empty set) if a nonexisting method is invoked or a wrong number of arguments are used.

The two last rules describe ordinary method invocations in program objects. The rules are textually equal to the similar rules in Chapter 4.

$$\frac{Exp = BExp \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta}{\Omega,\ \tau,\ \rho \vdash [\![Exp]\!] \Rightarrow \beta}$$

$$\frac{\Omega,\ \tau,\ \rho \vdash [\![BExp_1]\!] \Rightarrow \beta \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp_2]\!] \Rightarrow \beta \qquad \mathrm{Manifest}(\beta)}{\Omega,\ \tau,\ \rho \vdash [\![BExp_1 == BExp_2]\!] \Rightarrow \{\mathsf{true}\}}$$

$$\frac{\Omega,\ \tau,\ \rho \vdash [\![BExp_1]\!] \Rightarrow \beta_1 \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp_2]\!] \Rightarrow \beta_2 \qquad \beta_1 \cap \beta_2 = \emptyset}{\Omega,\ \tau,\ \rho \vdash [\![BExp_1 == BExp_2]\!] \Rightarrow \{\mathsf{false}\}}$$

$$\frac{\Omega,\ \tau,\ \rho \vdash [\![BExp_1]\!] \Rightarrow \beta_1 \qquad \Omega,\ \tau,\ \rho \vdash [\![BExp_2]\!] \Rightarrow \beta_2 \qquad (\beta_1 \cap \beta_2 \neq \emptyset) \vee \mathrm{NonManifest}(\beta_1) \vee \mathrm{NonManifest}(\beta_2)}{\Omega,\ \tau,\ \rho \vdash [\![BExp_1 == BExp_2]\!] \Rightarrow \{\mathsf{true},\ \mathsf{false}\}}$$

$$\frac{\begin{array}{c}\Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta \\ \Omega,\ \tau,\ \rho \vdash [\![BExp_j]\!] \Rightarrow \beta_j, \quad \text{for } j \in \{1, \ldots, n\} \\ \vdash \mathbf{FnctCall}(x,\ FnctName,\ \widehat{\beta_n}) \Rightarrow \beta_x : \rho \rightarrow \rho_x, \quad \text{for } x \in \beta \\ \rho' = \sqcup \{\rho_x\}\ , \quad \text{for } x \in \beta \\ \beta' = \sqcup \{\beta_x\}\ , \quad \text{for } x \in \beta\end{array}}{\Omega,\ \tau \vdash [\![BExp.FnctName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \beta' : \rho \rightarrow \rho'}$$

$$\frac{\begin{array}{c}\Omega' \in Abstract\_Object \\ \rho' = \rho[\Omega' \mapsto <\epsilon(MethodDef\,*),\ \imath_0>] \\ \Omega' \vdash [\![IBody]\!] : \rho',\ \tau \rightarrow \rho'',\ \tau'\end{array}}{\Omega,\ \tau \vdash [\![\mathbf{object}\ ObjectName(IVarName\,*)(MethodDef\,*)IBody]\!] \Rightarrow \Omega' : \rho \rightarrow \rho''}$$

Figure 6.5: Semantic rules for nonstandard interpretation of expressions

$$\frac{\Omega, \tau \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \quad \rho'' = \rho'[\Omega, IVarName \mapsto \beta]}{\Omega, \tau \vdash [\![IVarName \leftarrow Exp]\!] : \rho \rightarrow \rho''}$$

$$\frac{\Omega, \tau \vdash [\![Exp]\!] \Rightarrow \beta : \rho \rightarrow \rho' \quad \tau' = \tau[LocalName \mapsto \beta]}{\Omega \vdash [\![LocalName \leftarrow Exp]\!] : \rho, \tau \rightarrow \rho', \tau'}$$

$$\frac{}{\vdash [\![skip]\!] :}$$

$$\frac{\begin{array}{c}\Omega, \tau, \rho \vdash [\![BExp]\!] \Rightarrow \beta \\ \Omega, \tau, \rho \vdash [\![BExp_j]\!] \Rightarrow \beta_j, \quad \text{for } j \in \{1, \ldots, n\} \\ \vdash \mathbf{ProcCall}(x, FnctName, \widehat{\beta_n}) : \rho \rightarrow \rho_x, \quad \text{for x} \in \beta \\ \rho' = \sqcup_{\text{if}} \{\rho_x\}, \quad \text{for x} \in \beta\end{array}}{\Omega, \tau \vdash [\![BExp.ProcName[BExp_1 \ \ldots \ BExp_n]]\!] : \rho \rightarrow \rho'}$$

$$\frac{\Omega \vdash [\![Statement1]\!] : \rho, \tau \rightarrow \rho', \tau' \quad \Omega \vdash [\![Statement2]\!] : \rho', \tau' \rightarrow \rho'', \tau''}{\Omega \vdash [\![Statement1 \ Statement2]\!] : \rho, \tau \rightarrow \rho'', \tau''}$$

Figure 6.6: Semantics for nonstandard interpretation of statements

$$\frac{\mathcal{B}= \ldots \ [\![label\ Stmnt\ Jump]\!] \ \ldots \qquad \mathcal{B}, \Omega \vdash [\![label\ Stmnt\ Jump]\!] : \rho, \tau \rightarrow \rho', \tau'}{\mathcal{B}, \Omega \vdash [\![\textbf{goto}\ label]\!] : \rho, \tau \rightarrow \rho', \tau'}$$

$$\frac{}{\vdash [\![\textbf{return}]\!] :}$$

$$\frac{\begin{array}{c}\Omega, \rho, \tau \vdash [\![BExp]\!] \Rightarrow \beta \\ (\mathsf{true} \in \beta) \wedge (\mathsf{false} \notin \beta) \wedge (\mathsf{InputDynamic} \notin \beta) \\ \mathcal{B}= \ldots \ [\![label1\ Stmnt\ Jump]\!] \ \ldots \\ \mathcal{B}, \Omega \vdash [\![label1\ Stmnt\ Jump]\!] : \rho, \tau \rightarrow \rho', \tau'\end{array}}{\mathcal{B}, \Omega \vdash [\![\textbf{if}\ BExp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] : \rho, \tau \rightarrow \rho', \tau'}$$

$$\frac{\begin{array}{c}\Omega, \rho, \tau \vdash [\![BExp]\!] \Rightarrow \beta \\ (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \in \beta) \wedge (\mathsf{InputDynamic} \notin \beta) \\ \mathcal{B}= \ldots \ [\![label2\ Stmnt\ Jump]\!] \ \ldots \\ \mathcal{B}, \Omega \vdash [\![label2\ Stmnt\ Jump]\!] : \rho, \tau \rightarrow \rho', \tau'\end{array}}{\mathcal{B}, \Omega \vdash [\![\textbf{if}\ BExp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] : \rho, \tau \rightarrow \rho', \tau'}$$

$$\frac{\begin{array}{c}\Omega, \rho, \tau \vdash [\![BExp]\!] \Rightarrow \beta \\ ((\mathsf{true} \in \beta) \wedge (\mathsf{false} \in \beta)) \vee (\mathsf{InputDynamic} \in \beta) \\ \mathcal{B}= \ldots \ [\![label1\ Stmnt1\ Jump1]\!] \ \ldots \\ \mathcal{B}= \ldots \ [\![label2\ Stmnt2\ Jump2]\!] \ \ldots \\ \mathcal{B}, \Omega \vdash [\![label1\ Stmnt1\ Jump1]\!] : \rho, \tau \rightarrow \rho_1, \tau_1 \\ \mathcal{B}, \Omega \vdash [\![label2\ Stmnt2\ Jump2]\!] : \rho, \tau \rightarrow \rho_2, \tau_2 \\ \rho'' = \rho_1 \sqcup \rho_2 \\ \tau'' = \tau_1 \sqcup \tau_2\end{array}}{\mathcal{B}, \Omega \vdash [\![\textbf{if}\ BExp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] : \rho, \tau \rightarrow \rho'', \tau''}$$

$$\frac{\begin{array}{c}\Omega, \rho, \tau \vdash [\![BExp]\!] \Rightarrow \beta \\ (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \notin \beta) \wedge (\mathsf{InputDynamic} \notin \beta)\end{array}}{\Omega \vdash [\![\textbf{if}\ BExp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] : \rho, \tau \rightarrow \rho_0, \tau_0}$$

Figure 6.7: Semantics for nonstandard interpretation of Jump

$$\frac{}{\vdash \mathbf{ProcCall}(\mathsf{bv},\ ProcName,\ \widehat{\beta_i}) : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \mathbf{ProcCall}(\mathsf{nil},\ ProcName,\ \widehat{\beta_i}) : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \mathbf{FnctCall}(\mathsf{nil},\ FnctName,\ \widehat{\beta_i}) \Rightarrow \{\ \} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(ProcName)\end{array}}{\vdash \mathbf{ProcCall}(\Omega',\ ProcName,\ \widehat{\beta_i}) : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(FnctName)\end{array}}{\vdash \mathbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta_i}) \Rightarrow \{\ \} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_i},\ LVar\widehat{Name_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(ProcName) \\ i \neq j\end{array}}{\vdash \mathbf{ProcCall}(\Omega',\ ProcName,\ \widehat{\beta_j}) : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_i},\ ResultName,\ LVar\widehat{Name_k},\ FBody> = \rho_{\mathrm{code}}(\Omega')(FnctName) \\ i \neq j\end{array}}{\vdash \mathbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta_j}) : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \mathbf{ProcCall}(\mathsf{InputDynamic},\ ProcName,\ \widehat{\beta_i}) :}$$

$$\frac{}{\vdash \mathbf{FnctCall}(\mathsf{InputDynamic},\ FnctName,\ \widehat{\beta_i}) \Rightarrow \{\mathsf{InputDynamic}\} :}$$

$$\frac{\vdash \mathbf{DoBaseFnctCall}(\mathsf{bv},\ FnctName,\ \widehat{\beta_i}) \Rightarrow \beta :}{\vdash \mathbf{FnctCall}(\mathsf{bv},\ FnctName,\ \widehat{\beta_i}) \Rightarrow \beta :}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ LVar\widehat{Name_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(ProcName) \\ \tau' = [FormalPar_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \Omega' \vdash [\![PBody]\!] : \rho,\ \tau' \rightarrow \rho',\ \tau''\end{array}}{\vdash \mathbf{ProcCall}(\Omega',\ ProcName,\ \widehat{\beta_j}) : \rho \rightarrow \rho'}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ ResultName,\ LVar\widehat{Name_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(FnctName) \\ \tau' = [n_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \Omega' \vdash [\![FBody]\!] : \rho,\ \tau' \rightarrow \rho',\ \tau''\end{array}}{\vdash \mathbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta_j}) \Rightarrow \tau''(ResultName) : \rho \rightarrow \rho'}$$

Figure 6.8: Semantics for nonstandard interpretation of method invocation

The abstract interpretation will fail to terminate on many programs. Especially all programs containing iterative or recursive loops that are controlled by an expression whose value depends on the unknown input will fail to terminate. This is what can be expected since the algorithm has not been designed to achieve termination.

6.3 How to Prove Correctness

First of all we should define what a safe approximation to a program state means. This is typically done by defining an *abstraction* function $\mathcal{A}_{\mathrm{v}}$ that maps concrete values to symbolic values, and defining a *concretization* function $\mathcal{C}_{\mathrm{v}}$ mapping symbolic values to sets of concrete values. Since we can create objects at runtime this can be a hard thing to do and we have not attempted to do so.

Using the definition of the concretization function $\mathcal{C}_{\mathrm{v}}$ on symbolic values we can define a concretization function $\mathcal{C}_{\mathrm{s}}$ on abstract program states. A concrete program state $\mathcal{PS}_{\mathrm{con}}$ is approximated by an abstract program state $\mathcal{PS}_{\mathrm{abs}}$ if $\mathcal{PS}_{\mathrm{con}} \in \mathcal{C}_{\mathrm{s}}(\mathcal{PS}_{\mathrm{abs}})$.

The proof of correctness of approximating the program state during any possible execution path should then be an inductive proof over the syntactic constructs of the language; e.g. for each statement $\mathcal{S}$ we would have to prove:

$$\forall\, \mathcal{PS}_{\mathrm{con}} \in \mathcal{C}_{\mathrm{s}}(\mathcal{PS}_{\mathrm{abs}}) :$$
$$((\mathcal{PS}'_{\mathrm{con}} = [\![\mathcal{S}]\!]_{\mathrm{con}}\mathcal{PS}_{\mathrm{con}}) \wedge (\mathcal{PS}'_{\mathrm{abs}} = [\![\mathcal{S}]\!]_{\mathrm{abs}}\mathcal{PS}_{\mathrm{abs}})) \Rightarrow (\mathcal{PS}'_{\mathrm{con}} \in \mathcal{C}_{s}(\mathcal{PS}_{\mathrm{abs}}))$$

Here we have used the $[\![\ ]\!]$ brackets with a subscript to denote that they are not used to set of syntactic constructs but as application of the semantic function transforming program states. Using these brackets in this way follows the usage in e.g. [Jones 92].

In addition to the proofs of correctness of program state we have to prove that the abstract interpretation does not break on legal programs. This can be proven by proving that the interpretation does not break on any syntactic correct program.

We have not attempted any of these proofs but we do not believe it will be hard to do them.

Chapter 7

Ensuring Termination

The simple abstract interpretation described in the previous chapter does not terminate very often. In this chapter we describe how to improve the termination properties of the abstract interpretation by generalizing symbolic values. We generalize in three different looping situations. The resulting abstract interpreter is very safe. It has termination properties similar to those of the partial evaluators Fuse and Similix.

The termination properties can be summarized as: the abstract interpretation fails to terminate when the program being interpreted itself fails to terminate for any input *or* when the program contains a nonterminating region regardless of input.

7.1 Generalizing in speculative loops leads to termination

Using the terminology of [Weise 91b] we define a *speculative loop* to be a loop where the test of whether or not to exit the loop depends on the unknown input to the program. Speculative loops are the most common reasons for nontermination of the abstract interpretation described in the previous chapter. When we identify a speculative loop we create a generalized abstract program state. The generalized program state should be at least as general as the abstract program state when we entered the loop and as the abstract program state when we identified the loop (when the loop is closed). Abstract interpretation of the body of the loop is then repeated using the generalized abstract program state.

For iterative loops the program state is given by the object store, ρ, and the local environment, τ. For recursive loops the program state is given by the object store, ρ, and the vector of arguments, $\widehat{\beta}$. The generalization of the abstract program states and the abstract interpretation of the loop is repeated until the abstract program state is unchanged between subsequent iterations of the loop. When this happens, a fixpoint is reached and a safe approximation to the effect of executing the loop any number of times has been computed.

It is possible to get even better termination properties if one considers as dangerous all loops encountered while trying out one of several possible branches. Using this technique

it is possible to give the partial evaluator so good termination properties that the specialization process only will fail to terminate if the source program will fail to terminate for any instantiations of the unknown input values. Using this strategy will however make the final partial evaluator less aggressive. For this reason we have chosen only to regard speculative loops as being dangerous.

Using any mechanism for generalizing in loops requires a new domain of symbolic values. We also need a technique for detecting speculative loops.

7.1.1 Extending the domain of symbolic values

There exists an infinite number of builtin objects. To prevent infinite loops due to use of different builtin objects in each iteration of a loop, we introduce object values representing all integers, all strings, or all characters. These object values are to be used when generalizing program states in loops.

If we want to generalize program states in loops using the least upper bound operator, then we have to define a new partial order on symbolic values. The partial order must of course be extended to environments and program states. The new partial order should be defined so the least upper bound of two symbolic values is the union of the two sets of object values with the exception that if the result contains two or more object values representing builtin objects of the same kind (integers, characters, or strings) then those object values are replaced by the object value representing all builtin objects of this kind.

Using this new partial order when generalizing program states in speculative loops prevents object values that denote builtin objects from accumulating in such loops. Under the assumption that no objects are created in loops, the number of program state generalizations, which can be performed in a speculative loop without reaching a fixpoint, is now finite.

Objects can however be created in loops and thus render this argument useless. The number of objects possibly created in a speculative loop is unbounded. All objects created by the same object constructor in a speculative loop must therefore be represented by a finite number of object values. We will use a single object value. Given this, only a finite number of generalizations can be made in a speculative loop before reaching a fixpoint.

In order to ensure termination of the abstract interpretation we must thus extend the previously used domain of symbolic values. The symbolic values are still sets of object values, but the domain of object values is to be extended. We must add values denoting all integers, all characters, all strings, to the previously used domain of object values. We must also add one object value for each object constructor in the source program. These last object values denote all objects created by the object constructors in speculative loops.

7.1.2 Detecting Iterative Loops

Our source language has basic blocks and jump instructions. This means that we can have iterative loops in the body of a method. The loops are entirely local to single methods.

In the partial evaluator Fuse, speculative loops are identified by maintaining a stack

```
x ← 2
while x.neq[1] do
    x ← x.minus[1]
    if dynamic.test[] then
        y ← 4
    end if
end while
```

(a)

```
label0: x ← 2
        goto label1
label1: skip
        if x.neq[1] then label2 else label5
label2: x ← x.minus[1]
        if dynamic.test[] then label3 else label4
label3: y ← 4
        goto label4
label4: skip
        goto label1
label5:
```

(b)

Figure 7.1: A nonspeculative loop

with call information for all functions that the thread of control is passing through on the current execution path [Weise 91b]. Whenever a conditional branch expression is encountered where it is not possible at specialization time to determine what branch is taken, a *conditional marker* is pushed onto this stack. Before each function call, Fuse inspects the stack to see if information on the function to be called is on the stack below the highest conditional marker (stack growing upwards!). If it is, a possible speculative loop has been identified. This scheme can of course be adapted to basic blocks instead of functions.

Another scheme is to maintain a set of information on the encountered basic blocks that had an undecidable conditional jump. Before jumping to a basic block, the set must be consulted. If the basic block to jump to is described in the set, a speculative loop is encountered.

The last scheme has the advantage of faster lookups than the stack scheme. Both schemes are conservative and will sometimes claim to have found speculative loops where there actually are no speculative loops. The stack scheme is a little more conservative than the set scheme. The stack scheme will claim the program fragment in Figure 7.1 to contain a speculative loop whereas the set scheme will not. If the variable x was assigned the value 4 instead of the value 2 in the start of the program fragment, then both schemes would claim the fragment contained a speculative loop.

In a recursive loop the same basic blocks with undecidable jump instructions can be executed several times without being in a iterative loop. The stack or the set used to identify speculative iterative loops must consequently be local to each symbolic execution of a method body.

7.1.3 Detecting Recursive Loops

Using a closure representation of objects as described in Section 3.4, we would have a recursive loop if a method in an object invoked another method in the same object. We do not consider this as recursion in our language. Recursion occurs when execution of a method m in an object x leads to invocation of the same method m in the same object

x.[1]

We can identify speculative recursive loops by the same methods as for iterative loops. That is, either use a stack of information on method invocations and conditional markers, or use a set of information on method invocations where the method body contains an undecidable branch instruction.

Undecidable branch instructions in this context are either undecidable conditional jumps or nonmanifest invocations. A method invocation where the target of the invocation is a symbolic value containing a single object value denoting an unbounded number of run-time objects is regarded as an undecidable branch. We will adapt the meaning of the term *manifest symbolic values* to denote symbolic values containing only a single object value, which denotes anything but the InputDynamic symbolic value. The meaning of the term *manifest invocation* is modified in the same way.

Since recursive loops can be partitioned into recursion of functional methods and recursion of procedural methods, the fastest solution is to use two sets or stacks — one for procedural methods and one for functional methods. We have decided to use the scheme using sets to detect loops as described above rather than using stacks. We have decided to use only a single set for method invocations rather than one for functional and one for procedural methods. Using more than one set will just clutter up the final formulas.

7.1.4 Detecting Object Creation Loops

A third kind of dangerous loops is a loop where the undecidable jump instruction is in a method belonging to a new object each time. The objects are created in the loop. This is a loop construction that is often ignored in abstract interpreters [Jones 91].

A fragment of a program containing such a loop is sketched in Figure 7.2. To avoid nontermination when performing abstract interpretation of programs of this kind, all the objects created in the loop should be represented by a single object value. In the example this is all the objects created by the *Dummy* object constructor.

We can detect this kind of loop using the techniques for detecting recursive loops if we take special precautions when creating new objects during abstract interpretation. Suppose we maintain a set of information about method invocations as described above. When we are about to evaluate an object constructor expression, then we should examine the set. If it contains information about invocations of methods in objects created by the same object constructor as the one we are to evaluate, then invocations of the same methods in the object we are about to create are dangerous. Invocations of these methods will be dangerous until the invocations we found information on in the set has terminated.

For the example program in Figure 7.2, imagine that the set of information is empty when we encounter the following method invocation:

[1] Using the closure encoding of objects we see that the branch taken on the static dispatching on the method name has to be the same for two *closure*-recursive calls to be considered recursive by us. This can of course be generalized (independent of language) to count all branches of static tests. This might be an area of future research.

```
const a == object devious
   function New[dyn] → [result]
      result ← object Dummy
         var temp
         function recur[] → [result]
            var aVar   var newDyn
            newDyn ← temp.minus[1]
            if temp.zero[] then
               aVar ← a.New[newDyn]
               result ← aVar.recur[]
            else
               result ← nil
            end if
         end recur
         initially
            temp ← dyn
         end initially
      end Dummy
   end New
end devious
   ⋮
b ← a.New[SomeValue]
c ← b.recur[]
```

Figure 7.2: Program fragments illustrating an object creation loop

b.recur[]

During the abstract interpretation of the body of *recur* we encounter the undecidable conditional jump and add to the set information about the method being invoked. When we are to interpret the object constructor in the body of the *New* method in the object *a*, we inspect our set of information and find information on the invocation of *recur*. We must now record the information that an invocation of the method *recur* in the object we are about to create is dangerous. This is valid until the current invocation of recur is finished. When we after the creation of the object encounter the method invocation

aVar.recur[]

we identify the dangerous loop.

Fuse as described in [Ruf 91,Weise 91b] does not handle this kind of loops even though the authors claim that Fuse has the same termination properties as we claim our algorithm has. Fuse does actually handle this kind of loop [Weise 91a]. The strategy used in Fuse is basicly that for each closure the call stack information at the time of creation of the closure is preserved. This call stack information is then used to detect speculative loops when the closure is applied. The algorithm used in Fuse will detect all the loops that our algorithm will detect. Their algorithm is however overly conservative in the sense that it often detects dangerous loops where there actually are none and our algorithm would

not lead to detection of a dangerous loop. Their algorithm will not identify more actual dangerous loops than our algorithm will (except by mistake).

7.2 Designing the new Algorithm

Generalizing in speculative loops is the fundamental technique used to ensure termination, but there are still a number of problems we have to address. These include where to restart symbolic execution after program state generalization, handling nested speculative loops, and how to treat objects created in speculative loops. These matters will be discussed in this section.

7.2.1 Iteration of symbolic execution of loops

When we identify a speculative loop, we create a generalized program state more general than both the present program state and the program state when last entering the loop. The abstract interpretation with the generalized program state can continue from two different points on the execution path: the present point, or the point where the speculative loop was first entered. The result of the abstract interpretation is the same regardless of what strategy is chosen, but the flow of control in the abstract interpreter is influenced and in the end also the structure of the generated residual programs.

To illustrate the difference we use the abstract notion of a flow chart rather than basic blocks, jumps, methods, and method invocations. Consider the flow chart in Figure 7.3(a). The speculative loop is detected at `x`. If the abstract interpretation is continued at the present point (at the test), then the flow of control in the abstract interpreter would equal that of interpreting a program abstracted by the flow chart in Figure 7.3(b). The program state is generalized at `x1`, and the interpreter can proceed with handling the lower part of the flow chart.

If the abstract interpretation is continued at the point where the speculative loop was first entered, then the flow of control can not be expressed by a simple flow chart as the abstract interpretation performs backtracking with information on how to try again. We have added some lines to the flow chart in Figure 7.3(c) to illustrate this. The thick line represents the flow of control during the abstract interpreters first attempt at interpreting the program fragment. The dotted line represents the backtracking performed. After backtracking and generalizing the program state using the backward propagated information, the abstract interpreter is ready to try again.

Our experience from writing programs implementing different program analyses tells us that the abstract interpretation will finish in shortest time if we restart the abstract interpretation at the point where we first entered the speculative loop. This because the program might contain nested loops. Considering the impact on code generation, continuing the abstract interpretation (and thus later also the code generation) from the present point will lead to code duplication.

Fuse continues the computations from the test. Using the terminology from [Weise 91b] the expansion can be regarded as a *preample* to the real loop. This preample can easily

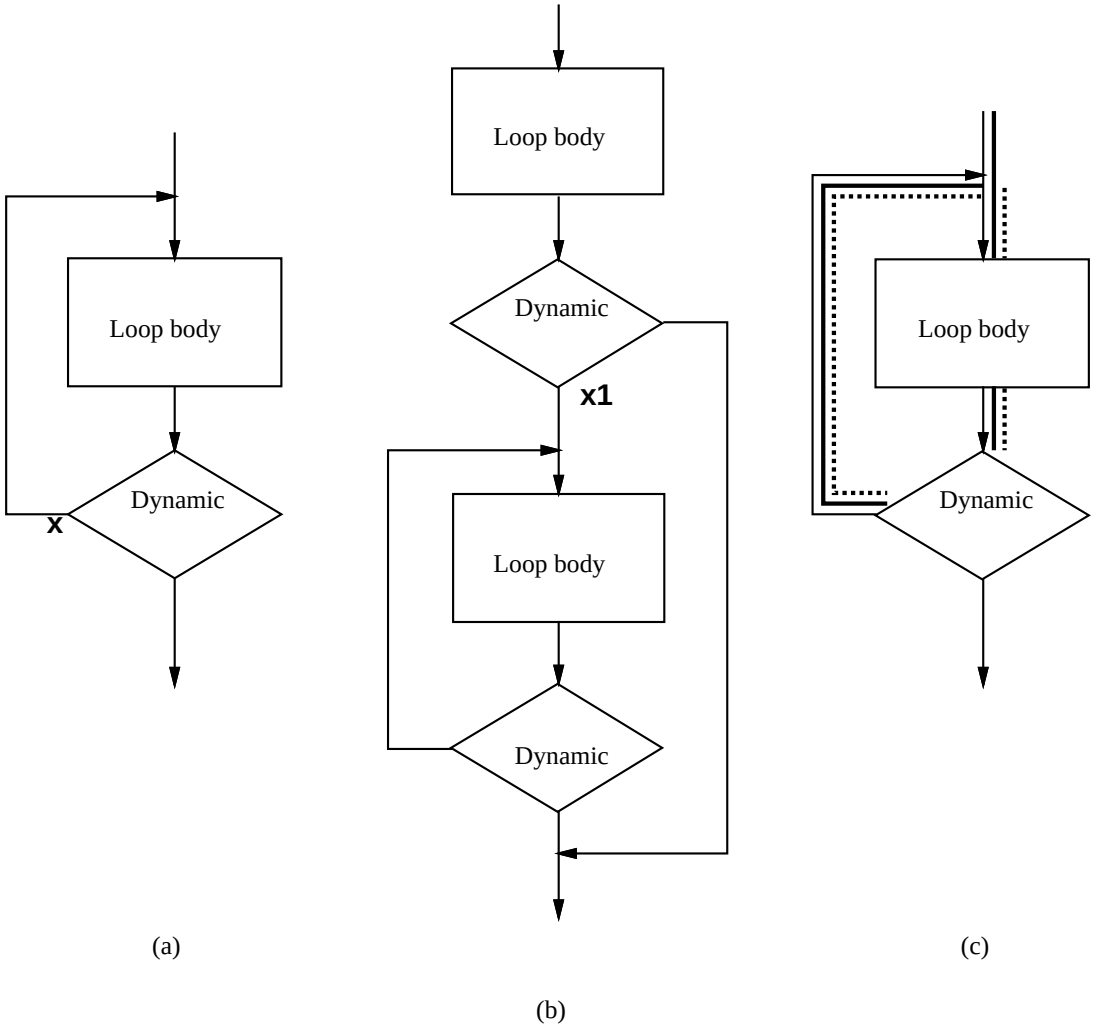

Figure 7.3: Flow charts illustrating a loop

be removed but the specialized methods generated for use by the preample are not so easily removed.

We will restart the abstract interpretation at the entrance point of the speculative loop. To do this we use a set of information to return to the entrance point of the loop. The information should include the abstract program state(s) at the point(s) where the speculative loop is closed. The generalized program state should be created from the program states from the set of backward propagated information. The generalization of program states and redoing the abstract interpretation from the generalization point is repeated until the program state at the entrance of the loop is at least as general as the program state(s) at the point(s) where the loop is detected/closed. For recursive loops the program state at the exit of the loop must also reach a fixpoint before the repeated abstract interpretation is stopped.

7.2.2 How to handle nested speculative loops

Speculative loops may of course be nested. Because we may detect speculative loops in all taken branches, the set of backward propagated information can contain information on several loops. We want the abstract interpreter to handle the outermost loops first. The backtracking should therefore continue to the outermost entrance point.

To help identify the outermost entrance point, information should be removed form the set of backward propagated information during backtracking. At each possible entrance point, information on this point is removed from the set. If the set only contains information on the current point during backtracking, then the outermost entrance point has been reached.

In the case of nested loops where interpretation of the outer loop has been iterated to a fixpoint, interpretation of the inner loop(s) might still need to be iterated. We let the decision of whether or not it is necessary to perform an(-other) iteration be taken at the point where the speculative loop is detected. This requires information on program states in the set of information used to detect speculative loops. If an(-other) iteration is needed, information indicating this is added to the set of backward propagated information. If not, the behavior depends on whether an iterative of recursive loop has been identified. If it is an iterative loop, then the set is left untouched. If is a recursive loop then information is added to the set that the result of the method is used. This will make it possible to ensure the the loop exit program state also reaches a fixpoint before the iterated abstract interpretation of the body of the speculative loop is stopped.

The set of backward propagated information will only contain *redo* information on the loops that we know for sure need iteration of the abstract interpretation. When an outer loop has been iterated to a fixpoint of program state at the loop entrance, the inner loop(s) will then be iterated using exactly the same backtracking mechanism. Only when iteration of abstract interpretation of all inner loops has finished can it be positively be determined that the program state at the exit of an outer recursive loop has reached a fixed point.

We will illustrate the technique of backtracking and removing information by going through the abstract interpretation of the program fragment in Figure 7.4(b). The program fragment in Figure 7.4(b) is a desugared version of the program fragment in Figure 7.4(a). The program fragment contains two nested loops. We will go through the iterations needed to compute the fixpoint of the abstract program state in the loops. The variable *DynObject* is supposed to be bound to a symbolic value containing only the object value denoting unknown input values. The symbolic values bound to the other occurring variables are irrelevant for the example.

The first undecidable branch encountered is the conditional jump in the basic block with label *label3*. The relevant part of the program state at this point is the following part of an environment (using an informal notation):

$$\{i : 1;\ j : 0;\ x : 2\}.$$

The program state and an indication of the basic block with the undecidable branch (*label3*) is added to the set of forward propagated information. For the moment we ignore what happens when taking the branch jumping to label *label4*. In the branch jumping to

```
i ← 0
j ← 0
repeat
    x ← 2
    i ← i.plus[1]
    if x.lt[i] then
        repeat
            x ← x.plus[1]
            j ← j.plus[1]
        until x.gt[i]
    end if
until i.gt[Dyn]
```

(a)

```
label0: i ← 0
        j ← 0
        goto label1
label1: x ← 2
        i ← i.plus[1]
        if x.lt[i] then label2 else label3
label2: x ← x.plus[1]
        j ← j.plus[1]
        if x.gt[i] then label3 else label2
label3: skip
        if i.gt[Dyn] then label4 else label1
label4:
```

(b)

Figure 7.4: A program fragment with two nested loops

label *label1* there are no more undecidable branches before we again encounter the basic block with label *label3*. The relevant part of the program state is then:

$$\{i : 2;\ j : 0;\ x : 2\}.$$

The abstract interpreter now identifies a speculative loop. It does this by comparing the label of the basic block we are jumping to (*label3*) with the labels in the set of propagated information.

The interpreter is to backtrack to the start of the speculative loop. The current program state and an indication of the basic block is added to the set of backward propagated information (that was empty before the addition). The backtracking stops at the basic block where we first encountered the undecidable branch because this is the only basic block indicated in the set of backward propagated information. The abstract program state is then generalized. The relevant environment part can be illustrated by the following:

$$\{i : Integer;\ j : 0;\ x : 2\}.$$

The interpretation is restarted with the generalized program state and *label3* in the set of forward propagated information. The next undecidable branch encountered is the conditional jump in the basic block with label *label1*. The relevant part of the program state at the entrance of the basic block is then

$$\{i : Integer;\ j : 0;\ x : 2\}.$$

This program state and *label1* is added to the set of forward propagated information. Both branches are taken. In the branch jumping to label *label3*, a speculative loop is quickly identified. The program state at the entrance of the loop and the current program state are identical so a fixpoint for this part of the loop has been found. In the other branch (the jump to label *label2*) a new undecidable branch is encountered in the basic block with label *label2*. The program state at the *entrance* of the basic block is

$\{i : Integer;\ j : 0;\ x : 2\}$.

Both branches close speculative loops. In the branch jumping to *label3* the relevant part of the program state at the entrance to the basic block where the speculative loop is identified (*label3*) is

$\{i : Integer;\ j : 1;\ x : 3\}$.

The program state and *label3* is added to the set of backward propagated information and the interpreter must backtrack out of this branch. The jump to *label2* also closes a speculative loop. The program state is the same as above. This program state and *label2* is added to the set of backpropagated information and the interpreter backtracks out of this branch too. The combination of the two sets of backpropagated information contains information on entrance points of two speculative loops (*label2* and *label3*). Having backtracked to the basic block with the label *label2*, the interpreter observes that it has not backtracked to the outermost loop since the set contains information on two speculative loops. The information on the present loop (*label2*) is removed and the backtracking continued until reaching the basic block with label *label3* is reached. Since the set of backward propagated information now only contains information on one basic block (the current one), the outermost loop has been identified. The program state is then generalized. The relevant part of the generalized program state is

$\{i : Integer;\ j : Integer;\ x : Integer\}$.

Computations are restarted with this program state. In the speculative loops, the program state is now the same at the places where the speculative loops are identified as they are at the entrance of the loops, so no more iterations are needed.

The branch jumping to *label4* is also interpreted with the generalized program states. This is the observable effect of the speculative loops.

The treatment of recursive loops has to be a little different. During the fixpoint iteration the effect of a recursive invocation has to be approximated. The effect of invoking a functional method is the value returned. The effect of invoking a procedural method can be described by the program state after the invocation. To compute the side effects of invoking a recursive procedural method it is necessary to iterate the interpretation of the method body until the program states at both the entrance and the exit of the method has reached a fixpoint. For functional methods the program state at the entrance of the loop and the value returned should reach a fixpoint before the iteration is stopped.

7.2.3 Objects created in speculative loops

In speculative loops we must handle creation of objects in a special way as described earlier. We introduce a special object value for each object constructor in the program. We call these object values *quantified* object values. Quantified object values represent an unbounded number of objects created with the same object constructor. We call object values denoting single run-time objects *singular* object values.

The symbolic value returned by abstract interpretation of an object constructor in a loop contains only a quantified object value. The same object value is used in all iterations of a loop. This ensures that it is possible to reach a fixpoint.

All runtime objects represented by a quantified object value will have been created in speculative loops. It is thus possible to reflect the creation of a specific number of objects created outside speculative loops and an unspecified number of objects created inside speculative loops — all created by the same object constructor. There are no dependencies between objects created by the same object constructor inside and outside speculative loops. State changes in the quantified objects do not change the state of any of the singular objects and vice versa.

Quantified object values represent an unbounded number of runtime objects. The abstract interpreter must maintain an instance variable environment for quantified objects that represent the state of all the possible runtime objects the object value represents. Assignments to instance variables in quantified objects cannot be treated the same way as assignments to instance variables in singular objects. Only the state of one runtime object will be modified. The environment must therefore only be updated by generalizing.

In the abstract object store there is to be one entrance for each quantified object. The environment of the quantified object is a generalization of all the possible environments for all the run-time objects possibly denoted by the quantified abstract object. An assignment statement assigning a value to a variable in a quantified object's environment will only modify the state of one run-time object. The state of the other run-time objects is unchanged. During abstract interpretation an assignment statement may thus not change the environment to be less general. The value assigned should thus be the least upper bound of the old value of the variable and the computed value.

7.3 Semantics of the New Algorithm

7.3.1 Termination Properties

The termination properties of this new abstract interpretation are quite good. The abstract interpretation will only fail to terminate if the source program would not terminate for any instantiation of the unknown input parameters that satisfy the minimal interaction criterion, or if the source program contains a nonterminating region independent of such instantiations. These regions may never be entered by any possible execution path, but the mere existence may cause nontermination of the abstract interpretation.

For this to have any meaning we have to specify what we mean by "a nonterminating region independent of such instantiations". To this purpose we imagine the existence of a *almost ideal partial evaluator* that will terminate and generate a residual program for any program that will execute and terminate for at least one instantiation of the unknown input values. If when applied like our abstract interpreter to the source program and the known input values, this almost ideal partial evaluator does not terminate, then our abstract interpretation algorithm should not terminate either. If the almost ideal partial evaluator terminates, then our algorithm is still not required to terminate.

We say that the program contains a nonterminating region independent of the input values if the residual program contains a point such that if the flow of control ever reaches this point then the program does not terminate. If the generated residual program contains a nonterminating region independent of the input values, then we say that the source program contains a nonterminating region independent of any instantiation of the unknown input values. If this is the case, then we do not require our abstract interpreter to terminate. It should terminate in all other situations.

These termination properties can not be proved. We are however rather convinced that they are correct.

7.3.2 An Operational Semantics for the Abstract Interpretation

The semantics of the abstract interpretation with the termination properties we want is much more complicated than the semantics of the abstract interpretation described in the previous chapter. Sets of information are passed forward to identify possible speculative loops, and sets of information are returned during backtracking to guide the generalization of program states. Also two different kinds of generalization of program states are used. One for generalizing program states in loops, and one for generalizing program states from several possible branches. The description of the semantics is going to be more algorithmic than in the previous chapters because of the backtracking. The description is still done using deduction rules. The rules should show the close relationship to the abstract interpretation given in the previous chapter.

We will not formally define the partial orders (there are two) on program states and abstract values. We just note that the partial orders on program states are defined in terms of the partial orders on abstract values as in the previous chapter. The partial orders on abstract values are both defined so that an abstract value is less than or equal to another abstract value is the second abstract value represents at least all the concrete values the first abstract value represents. This means that an abstract value containing only the object value denoting the integer value 1 is less than the abstract value containing only the object value representing all integer values.

We will describe the difference between the two partial orders in terms of the different methods for computing the least upper bound of two values. The partial order we use to generalize program states in undecidable branches is the *if* partial order. We use the symbol $\sqcup_{\text{if}}$ to denote the least upper bound operator of this partial order. The least upper bound of two abstract values is the union of the two sets of object values. A minor optimization that will improve the termination properties slightly and otherwise not affect the results of the abstract interpretation is removing object values denoting singular builtin objects from an abstract value if the abstract value contains the object value denoting all the builtin values of the kind of the singular object values.

The other partial order is the *loop* partial order. We use the symbol $\sqcup_{\text{loop}}$ to denote the least upper bound operator of this partial order. The least upper bound of two abstract values can be computed by first computing the union of the two sets of object values. If the resulting set contains two or more builtin object values of the same kind, then

these object values are replaced by the object value representing all builtin objects of this kind. There might not be a partial order fulfilling these requirements. If the object values denoting builtin values only are replaced by the quantified object value representing them all if the object values are from different abstract values, then there does exist such a partial order. Choosing between these two strategies for combining program states in loops does not influence the results of the abstract interpretation in other ways than the minor optimization we have described for the *if* partial order.

The basic semantic objects listed in Figure 7.5 are mostly the same as the basic semantic objects listed in Figure 6.1. The semantic object "$\mathcal{D}$" is new. It is used to indicate whether or not the body of a speculative loop is being interpreted. The semantic domain "Abstract_Object" is on the other hand no longer a basic domain. The semantic objects "Ω", "nil", "x", "true", and "false" are therefore no longer basic semantic objects but are compound semantic objects.

FormalPar	$\in$	FORMALPAR
LVarName	$\in$	LVARNAME
ResultName	$\in$	RESULTNAME
LocalName	$\in$	LOCALNAME = FORMALPAR $\cup$ LVARNAME $\cup$ RESULTNAME
IVarName	$\in$	IVARNAME
ConstName	$\in$	CONSTNAME
FnctName	$\in$	FNCTNAME
ProcName	$\in$	PROCNAME
MethodName	$\in$	METHODNAME = PROCNAME + FNCTNAME
oc	$\in$	Object_Constructor_Name
		ObjectTag = {Single, Quantified}
bv	$\in$	Builtin_Object
Arg	$\in$	ARG = CONSTNAME + BaseValueName + `nil` + `Dynamic`
InputDynamic	$\in$	INPUTDYNAMIC = {InputDynamic}
label	$\in$	Label
$\mathcal{D}$	$\in$	Boolean
$\mathcal{B}$	$\in$	Body = BBlist
		MethodTag = {***use***, ***redo***}

Figure 7.5: Basic Semantic Objects for Nonstandard interpretation With Termination

The compound semantic objects are listen in Figure 7.6. A number of new semantic objects have been introduced. The semantic objects "MF" and "LF" are the sets used to detect recursive and iterative loops. The two first elements of the MF tuples identify the method and the object in which it occurs of a possible entrance point of a speculative loop. The other four elements of the tuples describe the program state at the entrance and at the exit of the method. The first element of the LF tuples describes the label with the undecidable branch instruction. The two last elements describe the current approximation to the program state at the entrance/closing point of the loop.

The semantic objects "MB" and "LB" are the sets of backward propagated information used when generalizing program states in loops and also used to detect when reaching the outermost speculative loop that still requires iterated interpretation. The second and the third element of the MB tuples identify the method that is regarded as the entrance of

$$
\begin{array}{rcl}
\Omega & \in & \text{Abstract\_Object} = (\text{ObjectTag} \times \text{Program\_Object}) + \mathsf{nil} \\
x & \in & \text{OID} = \text{Builtin\_Object} + \text{Abstract\_Object} + \text{INPUTDYNAMIC} \\
\mathsf{true} & \in & \text{OID} \\
\mathsf{false} & \in & \text{OID} \\
\beta & \in & \text{AbsVal} = \mathcal{P}(\text{OID}) \\
\widehat{\beta} & \in & \text{Arglist} = \text{AbsVal list} \\
\text{Code} & \in & \text{CODE} = \\
 & & \quad \text{PROCNAME} \stackrel{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{LVARNAME list} \times \text{Body} + \\
 & & \quad \text{FNCTNAME} \stackrel{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{RESULTNAME} \times \text{LVARNAME list} \times \text{Body} \\
\tau & \in & \text{LocalEnv} = \text{LOCALNAME} \stackrel{\text{fin}}{\mapsto} \text{AbsVal} \\
\imath & \in & \text{InstanceEnv} = \text{IVARNAME} \stackrel{\text{fin}}{\mapsto} \text{AbsVal} \\
\rho & \in & \text{Object\_Store} = \text{Abstract\_Object} \stackrel{\text{fin}}{\mapsto} \\
 & & \quad (\text{Object\_Constructor\_Name} \times \text{CODE} \times \text{InstanceEnv}) \\
\text{MF} & \in & \text{MethodForward} = \mathcal{P}(\text{Abstract\_Object} \times \text{METHODNAME} \times \\
 & & \quad \text{Object\_Store} \times \text{Arglist} \times \text{Object\_Store} \times \text{AbsVal}) \\
\text{MB} & \in & \text{MethodBackward} = \mathcal{P}(\text{MethodTag} \times \text{METHODNAME} \times \text{Abstract\_Object} \times \\
 & & \quad \text{Object\_Store} \times \text{Arglist}) \\
\text{LF} & \in & \text{LabelForward} = \mathcal{P}(\text{Label} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\text{LB} & \in & \text{LabelBackward} = \mathcal{P}(\text{Label} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\mathcal{C} & \in & \text{METHODNAME} \times \text{Object\_Store} \times \text{Arglist} \\
\phi & \in & \text{Quantified\_Map} = \text{Object\_Constructor\_Name} \stackrel{\text{fin}}{\mapsto} \text{Abstract\_Object}_{\perp} \\
\pi & \in & \text{Problematic} = \mathcal{P}(\text{Abstract\_Object} \times \text{METHODNAME} \times \text{Abstract\_Object})
\end{array}
$$

Figure 7.6: Semantic Objects for Nonstandard interpretation with termination

the speculative loop. The first element indicates if the program state at the closing of the loop was more (***redo***) or less (***use***) general than the program state in the current iteration of interpretation of the loop. The two last elements describe the program state at the closing of the loop. These values are only used when the first element is ***redo***. The structure of the LB tuples is similar with the exception that only one element is needed to identify the entrance of the loop.

The semantic object "$\mathcal{C}$" is used to describe the invocation of the current method. This is used to update the MFsemantic object when encountering an undecidable branch in the body of the method.

The semantic object "ϕ" is a mapping used when creating quantified object values. The range (codomain) of the mapping is the quantified object values. We have to combine elements of the Quantified_Map semantic domain. Because the abstract interpreter takes all possible execution paths and quantified objects may be created in more than one path, we might have to combine two semanic objects with overlapping domains. We assume that there is some kind of superficial order on the Abstract_Object semantic domain so we deterministically can choose one of the possible (codomain) values if the domains overlap.

The last of the new semantic objects is "π" that is used to identify the invocations that are dangerous because they might be part of object creation loops as described in Section 7.1.4. The first element of the π tuples is an indication of the object we should be careful about invoking methods in. The second element is the name of the method whose invocation is dangerous. A tuple is added to the set because we in MF found

information about an invocation of the same method in another object. This object is indicated in the third element of the tuple. When the invocation of the method in this second object is finished, then it is no longer dangerous to invoke the named method in the object indicated by the first element of the tuple.

The signatures of the semantic functions used in the semantic rules are given in Figure 7.7. Apart from the two last functions, the semantic functions are the same as shown in Figure 6.3. The "Singular" function is a predicate on abstract values. If yields true iff the abstract value only has a single element and that single element is a singular object value. The predicate "NonSingular" is the logical complement to "Singular".

$$
\begin{array}{rcl}
\gamma & \in & \text{BaseValueName} \mapsto \text{Builtin\_Object} \\
\epsilon & \in & \text{MethodDef*} \mapsto \text{CODE} \\
\textbf{DoBaseFnctCall} & \in & \text{Builtin\_Object} \times \text{FNCTNAME} \times \text{Arglist} \mapsto \text{AbsVal} \\
\text{Manifest} & \in & \text{AbsVal} \mapsto \text{Boolean} \\
\text{NonManifest} & \in & \text{AbsVal} \mapsto \text{Boolean} \\
\text{Singular} & \in & \text{AbsVal} \mapsto \text{Boolean} \\
\text{NonSingular} & \in & \text{AbsVal} \mapsto \text{Boolean} \\
\text{Disjoint} & \in & \text{AbsVal} \times \text{AbsVal} \mapsto \text{Boolean} \\
\text{NonDisjoint} & \in & \text{AbsVal} \times \text{AbsVal} \mapsto \text{Boolean}
\end{array}
$$

Figure 7.7: Semantic functions for Nonstandard interpretation With Termination

The semantic rule in Figure 7.8 is a slight modification of the rule in Figure 6.4. The difference is the addition of the semantic object ϕ. This is a general difference between the formulas in this chapter and the previous chapter. Whenever there is the possibility of a speculative loop the ϕ semantic object can be updated. The mapping is not precomputed from the program text before the abstract interpretation is commenced because modifying the mapping during the abstract interpretation makes an extra improvement possible.

$$
\frac{\begin{array}{c}
\vdash [\![\mathit{Program}]\!] \Rightarrow \rho \\
\rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \Omega \\
\rho \vdash [\![\mathit{Arg*}]\!] \Rightarrow \widehat{\beta} \\
\vdash \textbf{ProcCall}(\Omega,\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_{\mathrm{none}} : \rho,\ \phi \rightarrow \rho',\ \phi' \\
\rho' \vdash \textbf{FnctCall}(\Omega,\ \mathit{FnctName},\ [\,]) \Rightarrow \beta,\ \mathrm{MB}_{\mathrm{null}} : \rho',\ \phi' \rightarrow \rho'',\ \phi''
\end{array}}{\vdash [\![\mathit{Program\ ConstName\ ProcName\ Arg*\ FnctName}]\!] \Rightarrow \beta}
$$

Figure 7.8: Semantic rules for the input–output function of a program.

The semantic rules in Figure 7.9 describe the semantics of constant declarations. The interesting part is the last rule. All the semantic objects used to taming the loops are here set to their bottom values. It is not possible that there are any quantified object values at this point of the abstract interpretation.

The semantic rules in Figure 7.10 describe the semantics of the simple expressions. They are slightly modified versions of the rules in Figure 6.5. The expressions cannot lead to the closing of any loops, so the MB returned is the minimal value (the empty set.) The rules for basic expressions are textually equal to the rules in Chapter 4 (Figure 4.9).

$$\frac{\text{Program} = \text{ConstDecl*} \quad \vdash [\![ConstDecl*]\!] : \rho_0 \rightarrow \rho}{\vdash [\![Program]\!] \Rightarrow \rho}$$

$$\frac{\vdash [\![ConstDecl*]\!] : \rho \rightarrow \rho' \quad \vdash [\![ConstDecl]\!] : \rho' \rightarrow \rho''}{\vdash [\![ConstDecl*\ ConstDecl]\!] : \rho \rightarrow \rho''}$$

$$\frac{\Omega = \mathsf{nil} \quad \mathcal{D} = \textbf{false} \quad \mathcal{D}, \mathcal{C}_0, \mathrm{MF}_0, \Omega \vdash [\![Exp]\!] \Rightarrow \beta : \rho, \tau_0, \phi_0, \pi_0 \rightarrow \rho', \tau', \phi, \pi \quad \rho'' = \rho'[\Omega, ConstName \mapsto \beta]}{\vdash [\![\textbf{const}\ ConstName == Exp]\!] : \rho \rightarrow \rho''}$$

Figure 7.9: Semantic rules for nonstandard interpretation of constant declarations with termination

$$\frac{\rho, \tau, \Omega \vdash [\![BExp]\!] \Rightarrow \beta}{\rho, \tau, \Omega \vdash [\![BExp]\!] \Rightarrow \beta, \mathrm{MB}_0}$$

$$\frac{\rho, \tau, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta \quad \rho, \tau, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta \quad \mathrm{Singular}(\beta)}{\rho, \tau, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{true}\}, \mathrm{MB}_0}$$

$$\frac{\rho, \tau, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta_1 \quad \rho, \tau, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta_2 \quad \mathrm{Disjoint}(\beta_1, \beta_2)}{\rho, \tau, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{false}\}, \mathrm{MB}_0}$$

$$\frac{\rho, \tau, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta_1 \quad \rho, \tau, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta_2 \quad \mathrm{NonDisjoint}(\beta_1, \beta_2) \vee \mathrm{NonSingular}(\beta_1) \vee \mathrm{NonSingular}(\beta_2)}{\rho, \tau, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{true}, \mathsf{false}\}, \mathrm{MB}_0}$$

Figure 7.10: Semantic rules for simple and equality expressions.

The semantic rules in Figure 7.11 describe the semantics of invocations of functional methods. They replace the fourth rule in Figure 6.5. The first rule describes manifest invocations and the second rule describes nonmanifest invocations. A nonmanifest invocation is an undecidable branch. Information on the current method thus has to be added to the set of information used to detect speculative loops. If the set already contains information on the current method, then the set should be left untouched. This is implemented by the $\uplus$ operator.

$$\frac{\begin{array}{c}\Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \{x\} \\ \text{Manifest}(\{x\}) \\ \Omega,\ \tau,\ \rho \vdash [\![BExp_i]\!] \Rightarrow \beta_i, \quad \text{for } i \in \{1, \ldots, n\} \\ \mathcal{D},\ \text{MF} \vdash \mathbf{FnctCall}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi'\end{array}}{\mathcal{D},\ \text{MF},\ \Omega,\ \tau \vdash [\![BExp.FnctName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi}$$

$$\frac{\begin{array}{c}\Omega,\ \tau,\ \rho \vdash [\![BExp]\!] \Rightarrow \beta \\ \text{NonManifest}(\beta) \\ \Omega,\ \tau,\ \rho \vdash [\![BExp_i]\!] \Rightarrow \beta_i, \quad \text{for } i \in \{1, \ldots, n\} \\ \mathcal{C} = <MethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}}> \\ \text{MF}' = \text{MF} \uplus \{<\Omega,\ MethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}},\ \rho_0,\ \emptyset>\} \\ \mathcal{D},\ \text{MF}' \vdash \mathbf{FnctCall}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \beta_x,\ \text{MB}_x : \rho,\ \phi,\ \pi \rightarrow \rho_x,\ \phi_x,\ \pi_x, \quad \text{for } x \in \beta \\ \beta' = \sqcup_{\text{if}} \{\beta_x\}, \quad \text{for } x \in \beta \\ \text{MB}' = \cup \{\text{MB}_x\}, \quad \text{for } x \in \beta \\ \rho' = \sqcup_{\text{if}} \{\rho_x\}, \quad \text{for } x \in \beta \\ \phi' = \sqcup \{\phi_x\}, \quad \text{for } x \in \beta \\ \pi' = \cup \{\pi_x\}, \quad \text{for } x \in \beta\end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega,\ \tau \vdash [\![BExp.FnctName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \beta',\ \text{MB}' : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi'}$$

Figure 7.11: Semantic rules for function call expressions.

The semantic rules in Figure 7.12 describe the semantics of object constructor expressions. The first rule describes the semantics of object constructors outside (identified) speculative loops. This is characterized by the semantic object $\mathcal{D}$ being false. In the third line we pick out the MF-tuples describing invocations of methods in objects created by the object constructor we are about to evaluate. This information is used in the fourth line to update the π semantic object. This semantic object is used to tame object creation loops as described in Section 7.1.4.

The two last rules describe the semantics of object constructor expressions inside (detected) speculative loops. The first of the two rules describes the first creation of a quantified object. The second of the two describes creation of a quantified object where the object value to use is forced. In this rule, the first modification of the ρ semantic object may seem superfluous. At least we thought so until we tried implementing the algorithm. The modification is necessary because some of the semantic rules may return the empty store. Some of the elements of the store tuples have to plugged in for the semantics to be well-defined for the whole class of programs we want to handle.

The semantic rules in Figure 7.13 describe the semantics of statements. The rules are

$$\frac{\begin{array}{c}\mathcal{D} = \textbf{false} \\ (\Omega' \in \text{Abstract\_Object}) \wedge (\Omega' = \text{<Single, \_>}) \\ \text{MF}' = \{\text{<}\Omega''\text{, \_, \_, \_, \_>} \in \text{MF} \mid \mathsf{oc} = \rho_{\text{oc}}(\Omega'')\} \\ \pi' = \pi \cup (\bigcup \{\text{<}\Omega'\text{, } \mathit{MethodName}\text{, } x\text{>}\}), \quad \text{for } \text{<}x\text{, } \mathit{MethodName}\text{, \_, \_, \_, \_>} \in \text{MF}' \\ \rho' = \rho[\Omega' \mapsto \text{<oc, } \epsilon(\text{MethodDef*}), \imath_0\text{>}] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![\mathit{IBody}]\!] \Rightarrow \text{MB} : \rho', \tau, \phi, \pi' \rightarrow \rho'', \tau', \phi', \pi'' \\ \beta = \{\Omega'\}\end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau \vdash [\![\textbf{object } \mathsf{oc} \ (\mathit{IVarName}*)(\mathit{MethodDef}*)\mathit{IBody}]\!] \Rightarrow \beta, \text{MB} : \rho, \phi, \pi \rightarrow \rho'', \phi', \pi''}$$

$$\frac{\begin{array}{c}\mathcal{D} = \textbf{true} \\ \bot = \phi(\mathsf{oc}) \\ (\Omega' \in \text{Abstract\_Object}) \wedge (\Omega' = \text{<Quantified, \_>}) \\ \rho' = \rho[\Omega' \mapsto \text{<oc, } \epsilon(\text{MethodDef*}), \imath_0\text{>}] \\ \phi' = \phi[\mathsf{oc} \mapsto \Omega'] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![\mathit{IBody}]\!] \Rightarrow \text{MB} : \rho', \tau, \phi', \pi \rightarrow \rho'', \tau', \phi'', \pi' \\ \beta = \{\Omega'\}\end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau, \pi \vdash [\![\textbf{object } \mathsf{oc} \ (\mathit{IVarName}*)(\mathit{MethodDef}*)\mathit{IBody}]\!] \Rightarrow \beta, \text{MB} : \rho, \phi \rightarrow \rho'', \phi''}$$

$$\frac{\begin{array}{c}\mathcal{D} = \textbf{true} \\ \Omega' = \phi(\mathsf{oc}) \\ \rho' = \rho[\Omega' \mapsto \text{<oc, } \epsilon(\text{MethodDef*}), \rho_{\text{env}}(\Omega')\text{>}] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![\mathit{IBody}]\!] \Rightarrow \text{MB} : \rho', \tau, \phi, \pi \rightarrow \rho'', \tau', \phi', \pi' \\ \beta = \{\Omega'\}\end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau, \pi \vdash [\![\textbf{object } \mathsf{oc} \ (\mathit{IVarName}*)(\mathit{MethodDef}*)\mathit{IBody}]\!] \Rightarrow \beta, \text{MB} : \rho, \phi \rightarrow \rho'', \phi'}$$

Figure 7.12: Semantic equations for object constructor expressions.

based on the semantic rules in Figure 6.6. The major differences are the semantics of assignments to instance variables and invocation of procedural methods.

The two first rules describe the semantics of assignments to instance variables. If the current object is a singular object, then the semantics is just as in Figure 6.6. If the current object is a quantified object value, then the modification of program state is performed by generalization.

The two last rules describe the semantics of procedure call statements. The choice of rule depends on whether the invocation is manifest or not. We subtract the nil object value before testing if the target object value is manifest because an invocation of a method in nil will always fail. It is just a minor optimization.

If the target object value is manifest, then the semantics is equivalent to the semantics described in Figure 6.6. If the target object value is nonmanifest, then we have an undecidable branch. The MF semantic object is accordingly modified. Otherwise this last rule also resembles the rule in Figure 6.6 apart from the addition of the new semantic objects.

The semantic rule in Figure 7.14 describes the semantics of a body of a method or initialization part. The thing to note is that the semantic object LF is set to the bottom element of its domain.

The semantic rules in Figure 7.15 describe how iterative loops are detected. The first two rules describe the semantics of closing a speculative loop. If the program state fixpoint has not been reached yet, then information is stored and returned in the LB semantic object.

The last four rules describe the semantics of basic blocks that do not close a speculative iterative loop. The first of the four rules covers the case where there is no information in the LB semantic object about the current basic block. The second of the four rules describe the case where there is information about the current basic block, but where there is also information about other basic blocks. This means that the interpretation algorithm has not backtracked to the outermost speculative iterative loop yet. The last two rules describe the case where the outermost iterative loop has been reached. If the semantic object $\mathcal{D}$ was false during the just finished interpretation of the body of the loop, then we set it to true during the further computations and use the bottom element of the "Quantified_Map" domain for the ϕ semantic object. This introduces the optimization that different sets of quantified values will be used in most speculative loops. The last two rules use the **IterBB** semantic rules to describe the iterated interpretation of the body of the speculative loop.

The **IterBB** semantic rules listed in Figure 7.16 describe the interpretation of the body of speculative iterative loops. The first rule describes when the fixpoint iteration is finished either because the fixpoint has been reached or because it is discovered that an outer loop needed an(-other) fixpoint iteration. The second rule describes the iteration details.

The **BB** semantic rules listed in Figure 7.17 and Figure 7.18 describe the semantics of basic blocks without all the stuff related to detecting and handling speculative loops. They are based on semantic rules in Figure 6.7. The differences are only the addition of the new semantic objects and the required combination of these.

$$\frac{\Omega = \text{<Single, \_>} \quad \mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \text{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \quad \rho'' = \rho'[\Omega, \mathit{IVarName} \mapsto \beta]}{\mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![\mathit{IVarName} \leftarrow \mathit{Exp}]\!] \Rightarrow \text{MB} : \rho, \phi, \pi \rightarrow \rho'', \phi', \pi'}$$

$$\frac{\Omega = \text{<Quantified, \_>} \quad \mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \text{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \quad \beta' = \beta \sqcup_{\text{if}} \rho_{\text{env}}(\Omega)(\mathit{IVarName}) \quad \rho'' = \rho'[\Omega, \mathit{IVarName} \mapsto \beta']}{\mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![\mathit{IVarName} \leftarrow \mathit{Exp}]\!] \Rightarrow \text{MB} : \rho, \phi, \pi \rightarrow \rho'', \phi', \pi'}$$

$$\frac{\mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \text{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \quad \tau' = \tau[\mathit{LocalName} \mapsto \beta]}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{LocalName} \leftarrow \mathit{Exp}]\!] \Rightarrow \text{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi'}$$

$$\frac{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{Stmnt1}]\!] \Rightarrow \text{MB}_1 : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi' \quad \mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{Stmnt2}]\!] \Rightarrow \text{MB}_2 : \rho', \tau', \phi', \pi' \rightarrow \rho'', \tau'', \phi'', \pi'' \quad \text{MB}' = \text{MB}_1 \cup \text{MB}_2}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{Stmnt1}\ \mathit{Stmnt2}]\!] \Rightarrow \text{MB}' : \rho, \tau, \phi, \pi \rightarrow \rho'', \tau'', \phi'', \pi''}$$

$$\frac{\rho, \tau, \Omega \vdash [\![\mathit{BExp}]\!] \Rightarrow \beta \quad \beta' = (\beta \setminus \{\mathsf{nil}\}) \quad \text{Manifest}(\beta') \quad \Omega, \tau, \rho \vdash [\![\mathit{BExp}_i]\!] \Rightarrow \beta_i, \quad \text{for } i \in \{1, \ldots, n\} \quad (\beta' = \{x\}) \quad \mathcal{D}, \text{MF} \vdash \mathbf{ProcCall}(x, \mathit{ProcName}, \widehat{\beta}) \Rightarrow \text{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi'}{\mathcal{D}, \text{MF}, \tau, \Omega \vdash [\![\mathit{BExp.ProcName}[\mathit{BExp}_1 \ \ldots \ \mathit{BExp}_n]]\!] \Rightarrow \text{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi'}$$

$$\frac{\begin{array}{c}\rho, \tau, \Omega \vdash [\![\mathit{BExp}]\!] \Rightarrow \beta \quad \beta' = (\beta \setminus \{\mathsf{nil}\}) \quad \text{NonManifest}(\beta') \\ \Omega, \tau, \rho \vdash [\![\mathit{BExp}_i]\!] \Rightarrow \beta_i, \quad \text{for } i \in \{1, \ldots, n\} \\ \mathcal{C} = \text{<}\mathit{MethodName}, \rho_{\text{call}}, \widehat{\beta}_{\text{call}}\text{>} \\ \text{MF}' = \text{MF} \uplus \{\text{<}\Omega, \mathit{MethodName}, \rho_{\text{call}}, \widehat{\beta}_{\text{call}}, \rho_0, \emptyset\text{>}\} \\ \mathcal{D}, \text{MF}' \vdash \mathbf{ProcCall}(x, \mathit{ProcName}, \widehat{\beta}) \Rightarrow \text{MB}_x : \rho, \phi, \pi \rightarrow \rho_x, \phi_x, \pi_x, \quad \text{for } x \in \beta' \\ \text{MB}' = \cup \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}} \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup \{\pi_x\}, \quad \text{for } x \in \beta'\end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \tau, \Omega \vdash [\![\mathit{BExp.ProcName}[\mathit{BExp}_1 \ \ldots \ \mathit{BExp}_n]]\!] \Rightarrow \text{MB}' : \rho, \phi, \pi \rightarrow \rho', \phi', \pi'}$$

Figure 7.13: Semantic rules for statements.

The **ProcCall** semantic rules listed in Figure 7.19 describe how speculative recursive procedure invocation loops are detected. The first three rules describe the semantics of

$$\frac{\mathcal{B} = Body \qquad \mathcal{B} = [\![BB]\!] \ldots \qquad \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \mathrm{LF}_0, \Omega \vdash [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi'}{\mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Body]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi'}$$

Figure 7.14: Semantic rules for nonstandard interpretation of basic blocks

closing a speculative loop. The first two of these describe the detection of the speculative loop by inspection of the MF semantic object. If the loop entrance program state fixpoint has not been reached yet, then information is stored and returned in the MB semantic object that guides the iteration controlling rules that interpretation of the body of the loop should be redone. If the loop entrance program state fixpoint has been reached, then information is stored and returned in the MB semantic object informing the iteration controlling rules that the "result" program state has been used. The third rule describes the detection of the speculative loop by inspection of the π semantic object. This the semantic object introduced to help detect object creation loops as described in Section 7.1.4. In this case interpretation of the body of the loop should be redone.

The last three rules describe the semantics of method invocations that do not close a speculative recursive loop. The first of the three rules covers the case where the MB semantic object does not contain information on the current method invocation or the current method invocation is not the entrance point of the outermost speculative recursive loop. The last two rules describe the case where the outermost recursive loop has been reached. If the semantic object $\mathcal{D}$ was false during the just finished interpretation of the body of the loop, then we set it to true during the further computations and use the bottom element of the "Quantified_Map" domain for the ϕ semantic object. This is the same optimization as used for speculative iterative loops (Figure 7.15. The last two rules use the **ProcIter** semantic rules to describe the iterated interpretation of the body of the body of the speculative loop.

The **ProcIter** semantic rules listed in Figure 7.20 describe the interpretation of the body of speculative recursive loops. The first rule describes when the fixpoint iteration is finished either because the recursion is no longer there[2] or because it is discovered that an outer loop needed an(-other) fixpoint iteration.

The second rule describes when the fixpoint iteration is finished because both the loop entrance program state and the loop exit program state have reached their fixpoints. The program state at the exit of the loop is a monotonic (increasing) function of the number of iterations of abstract interpretation of the body of the loop. The "$\sqsubseteq$" operator could thus be replaced by the equality operator without changing the semantics. The "$\sqsubseteq$" operator is used to allow future experiments with the semantics.

The third and the fourth rule both describe the case when the outermost loop has been reached and the fixpoint has not been reached yet. The third rule describes when another iteration of interpretation of the loop body is needed because the loop exit program state has not yet reached a fixpoint. The fourth rule describes when another iteration of

[2]This should never happen

$$\frac{\begin{array}{c}<label,\ \rho',\ \tau'> \ \in\ \mathrm{LF} \\ (\rho \sqsubseteq \rho') \wedge (\tau \sqsubseteq \tau')\end{array}}{\mathrm{LF} \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}_0,\ \mathrm{MB}_0 : \rho,\ \tau \rightarrow \rho_0,\ \tau_0}$$

$$\frac{\begin{array}{c}<label,\ \rho',\ \tau'> \ \in\ \mathrm{LF} \\ (\rho \not\sqsubseteq \rho') \vee (\tau \not\sqsubseteq \tau') \\ \mathrm{LB} = \{<label,\ \rho,\ \tau>\}\end{array}}{\mathrm{LF} \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB}_0 : \rho,\ \tau \rightarrow \rho_0,\ \tau_0}$$

$$\frac{\begin{array}{c}<label,\ \_,\ \_> \ \notin\ \mathrm{LF} \\ \mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ <label,\ \_,\ \_> \ \notin\ \mathrm{LB}\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi'}$$

$$\frac{\begin{array}{c}<label,\ \_,\ \_> \ \notin\ \mathrm{LF} \\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ <label,\ \_,\ \_> \ \in\ \mathrm{LB} \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label,\ \_,\ \_>\} \\ \mathrm{LB}' \neq \emptyset\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi'}$$

$$\frac{\begin{array}{c}<label,\ \_,\ \_> \ \notin\ \mathrm{LF} \\ \mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ <label,\ \_,\ \_> \ \in\ \mathrm{LB} \\ (\mathrm{LB} \setminus \{<label,\ \_,\ \_>\}) = \emptyset \\ \mathrm{LF}' = \mathrm{LF} \cup \{<label,\ \rho,\ \tau>\} \\ \mathcal{D} = \mathbf{false} \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{B},\ \mathcal{D}',\ \mathcal{C},\ \mathrm{LF}',\ \mathrm{MF},\ \Omega,\ \pi \vdash \mathbf{IterBB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB}' : \rho,\ \tau,\ \phi_0 \rightarrow \rho'',\ \tau'',\ \phi''\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega,\ \phi,\ \pi \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB}' : \rho,\ \tau \rightarrow \rho'',\ \tau''}$$

$$\frac{\begin{array}{c}<label,\ \_,\ \_> \ \notin\ \mathrm{LF} \\ \mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ <label,\ \_,\ \_> \ \in\ \mathrm{LB} \\ (\mathrm{LB} \setminus \{<label,\ \_,\ \_>\}) = \emptyset \\ \mathrm{LF}' = \mathrm{LF} \cup \{<label,\ \rho,\ \tau>\} \\ \mathcal{D} = \mathbf{true} \\ \mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF}',\ \mathrm{MF},\ \Omega,\ \pi \vdash \mathbf{IterBB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB}' : \rho,\ \tau,\ \phi \rightarrow \rho'',\ \tau'',\ \phi''\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega,\ \pi \vdash [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB}' : \rho,\ \tau,\ \phi \rightarrow \rho'',\ \tau'',\ \phi''}$$

Figure 7.15: Main semantic equations for basic blocks

$$\frac{\begin{array}{c}\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![\mathit{label\ Stmnt\ Jump}]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<\mathit{label},\ \_,\ \_>\} \\ (\mathrm{LB}' \neq \emptyset) \vee (<\mathit{label},\ \_,\ \_> \notin \mathrm{LB})\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega,\ \pi \vdash \mathbf{IterBB}\ [\![\mathit{label\ Stmnt\ Jump}]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB} : \rho,\ \tau,\ \phi \rightarrow \rho',\ \tau',\ \phi'}$$

$$\frac{\begin{array}{c}\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \mathbf{BB}\ [\![\mathit{label\ Stmnt\ Jump}]\!] \Rightarrow \mathrm{LB},\ \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<\mathit{label},\ \_,\ \_>\} \\ (\mathrm{LB}' = \emptyset) \wedge (<\mathit{label},\ \_,\ \_> \in \mathrm{LB}) \\ \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_{\mathrm{back}}\}),\quad \text{for } <\mathit{label},\ \rho_{\mathrm{back}},\ \_> \in \mathrm{LB} \\ \tau'' = \tau \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\tau_{\mathrm{back}}\}),\quad \text{for } <\mathit{label},\ \_,\ \tau_{\mathrm{back}}> \in \mathrm{LB} \\ \mathrm{LF}' = (\mathrm{LF} \setminus \{<\mathit{label},\ \_,\ \_>\}) \cup \{<\mathit{label},\ \rho'',\ \tau''>\}\end{array}}{\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF}',\ \mathrm{MF},\ \Omega,\ \pi \vdash \mathbf{IterBB}\ [\![\mathit{label\ Stmnt\ Jump}]\!] \Rightarrow \mathrm{LB}'',\ \mathrm{MB}' : \rho'',\ \tau'',\ \phi' \rightarrow \rho''',\ \tau''',\ \phi''}$$

$$\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega,\ \pi \vdash \mathbf{IterBB}\ [\![\mathit{label\ Stmnt\ Jump}]\!] \Rightarrow \mathrm{LB}'',\ \mathrm{MB}' : \rho,\ \tau,\ \phi \rightarrow \rho''',\ \tau''',\ \phi''$$

Figure 7.16: The **IterBB** semantic equations for basic blocks.

interpretation of the loop body is needed because the loop entrance program state has not yet reached a fixpoint.

The **ProcBody** semantic rules listed in Figure 7.21 are equal to the **ProcCall** semantic rules listed in Figure 6.8 in the previous chapter except for the addition of the new semantic objects.

The **FnctCall** semantic rules listed in Figure 7.22 describe how speculative recursive functional loops are detected. The rules are very much like the rules in Figure 7.19. The only noteworthy difference is in the first and second rule where the partial order requirement between the two object stores has been replaced with an equality requirement.

The **FnctIter** semantic rules listed in Figure 7.23 describe the interpretation of the body of speculative recursive loops. The rules are very much like the rules in Figure 7.20. Again, the only noteworthy difference is the replacing of the partial order requirement between object stores has been replaced by an equality requirement.

The **FnctBody** semantic rules listed in Figure 7.24 are equal to the **FnctCall** semantic rules listed in Figure 6.8 in the previous chapter except for the addition of the new semantic objects.

$$\frac{Jump = [\![\textbf{return}]\!] \quad \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi'}{\mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}_0, \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi'}$$

$$\frac{\begin{array}{c} Jump = [\![\textbf{goto}\ label1]\!] \quad \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi' \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jump1]\!]\ \ldots \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash [\![label1\ Stmnt1\ Jump1]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}' : \rho', \tau', \phi', \pi' \rightarrow \rho'', \tau'', \phi'', \pi'' \\ \mathrm{MB}'' = \mathrm{MB} \cup \mathrm{MB}' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}'' : \rho, \tau, \phi, \pi \rightarrow \rho'', \tau'', \phi'', \pi''}$$

$$\frac{\begin{array}{c} Jump = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi' \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}' : \rho', \phi', \pi' \rightarrow \rho_{\mathrm{none}}, \phi'', \pi_{\mathrm{none}} \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{false} \notin \beta) \wedge (\mathsf{true} \in \beta) \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jump1]\!]\ \ldots \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash [\![label1\ Stmnt1\ Jump1]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}'' : \rho', \tau', \phi'', \pi' \rightarrow \rho'', \tau'', \phi''', \pi'' \\ \mathrm{MB}''' = \mathrm{MB} \cup \mathrm{MB}' \cup \mathrm{MB}'' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}''' : \rho, \tau, \phi, \pi \rightarrow \rho'', \tau'', \phi''', \pi''}$$

$$\frac{\begin{array}{c} Jump = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi' \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}' : \rho', \phi', \pi' \rightarrow \rho_{\mathrm{none}}, \phi'', \pi_{\mathrm{none}} \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \in \beta) \\ \mathcal{B} = \ldots\ [\![label2\ Stmnt2\ Jump2]\!]\ \ldots \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash [\![label2\ Stmnt2\ Jump2]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}'' : \rho', \tau', \phi'', \pi' \rightarrow \rho'', \tau'', \phi''', \pi'' \\ \mathrm{MB}''' = \mathrm{MB} \cup \mathrm{MB}' \cup \mathrm{MB}'' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}''' : \rho, \tau, \phi, \pi \rightarrow \rho'', \tau'', \phi''', \pi''}$$

$$\frac{\begin{array}{c} Jump = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho, \tau, \phi, \pi \rightarrow \rho', \tau', \phi', \pi' \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}' : \rho', \phi', \pi' \rightarrow \rho_{\mathrm{none}}, \phi'', \pi_{\mathrm{none}} \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \notin \beta) \end{array}}{\mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}_0, \mathrm{MB}_0 : \rho, \tau, \phi, \pi \rightarrow \rho_0, \tau_0, \phi'', \pi'}$$

Figure 7.17: Semantic equations for the static branches of the **BB** basic blocks.

$$Jump = [\![\textbf{if } Exp \textbf{ then goto } label1 \textbf{ else goto } label2]\!]$$
$$\mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi'$$
$$\mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \tau',\ \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\ \mathrm{MB}' : \rho',\ \phi',\ \pi' \rightarrow \rho_{\mathrm{none}},\ \phi'',\ \pi_{\mathrm{none}}$$
$$(\mathsf{InputDynamic} \in \beta) \vee ((\mathsf{true} \in \beta) \wedge (\mathsf{false} \in \beta))$$
$$\mathcal{C} = <MethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}}>$$
$$\mathrm{MF}' = \mathrm{MF} \uplus <\Omega,\ MethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}},\ \rho_0,\ \emptyset>$$
$$\mathrm{LF}' = \mathrm{LF} \uplus <label,\ \rho,\ \tau>$$
$$\mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jump1]\!]\ \ldots$$
$$\mathcal{B} = \ldots\ [\![label2\ Stmnt2\ Jump2]\!]\ \ldots$$
$$\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF}',\ \mathrm{MF}',\ \Omega \vdash [\![label1\ Stmnt1\ Jump1]\!] \Rightarrow \mathrm{LB}_1,\ \mathrm{MB}_1 : \rho',\ \tau',\ \phi'',\ \pi' \rightarrow \rho_1,\ \tau_1,\ \phi_1,\ \pi_1$$
$$\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF}',\ \mathrm{MF}',\ \Omega \vdash [\![label2\ Stmnt2\ Jump2]\!] \Rightarrow \mathrm{LB}_2,\ \mathrm{MB}_2 : \rho',\ \tau',\ \phi'',\ \pi' \rightarrow \rho_2,\ \tau_2,\ \phi_2,\ \pi_2$$
$$\mathrm{MB}'' = \mathrm{MB} \cup \mathrm{MB}' \cup \mathrm{MB}_1 \cup \mathrm{MB}_2$$
$$\mathrm{LB}' = \mathrm{LB}_1 \cup \mathrm{LB}_2$$
$$\rho'' = \rho_1 \sqcup_{\mathrm{if}} \rho_2$$
$$\tau'' = \tau_1 \sqcup_{\mathrm{if}} \tau_2$$
$$\phi''' = \phi_1 \sqcup \phi_2$$
$$\pi'' = \pi_1 \cup \pi_2$$

$$\mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \mathrm{LF},\ \mathrm{MF},\ \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jump]\!] \Rightarrow \mathrm{LB}',\ \mathrm{MB}'' : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho'',\ \tau'',\ \phi''',\ \pi''$$

Figure 7.18: Semantic deduction rule for the dynamic branch of **BB** basic blocks.

$$\frac{<\Omega', \mathit{ProcName}, \rho', \widehat{\beta'}, \rho'', \_> \in \mathrm{MF} \quad (\rho \sqsubseteq \rho') \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta'}) \quad \mathrm{MB} = \{<\textbf{*use*}, \mathit{ProcName}, \Omega', \_, \_>\}}{\mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{<\Omega', \mathit{ProcName}, \rho', \widehat{\beta'}, \rho'', \_> \in \mathrm{MF} \quad (\rho \not\sqsubseteq \rho') \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta'}) \quad \mathrm{MB} = \{<\textbf{*redo*}, \mathit{ProcName}, \Omega', \rho, \widehat{\beta}>\}}{\mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{(<\Omega', \mathit{ProcName}, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{ProcName}, \_> \in \pi) \quad \mathrm{MB} = \{<\textbf{*redo*}, \mathit{ProcName}, \Omega', \rho_0, \widehat{\emptyset}_j>\}}{\pi, \mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}_j) \Rightarrow \mathrm{MB} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{ProcName}, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{ProcName}, \_> \notin \pi) \\ \mathcal{D}, \mathrm{MF} \vdash \textbf{ProcBody}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{ProcName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_, \_> \in \mathrm{MB}') \vee (<\_, \mathit{ProcName}, \Omega', \_, \_> \notin \mathrm{MB}) \\ \pi'' = \pi' \setminus \{<\_, \mathit{ProcName}, \Omega'>\}\end{array}}{\mathcal{D}, \mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB}' : \rho, \phi, \pi \rightarrow \rho''', \phi', \pi''}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{ProcName}, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{ProcName}, \_> \notin \pi) \\ \mathcal{D}, \mathrm{MF} \vdash \textbf{ProcBody}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{ProcName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_, \_> \notin \mathrm{MB}') \wedge (<\_, \mathit{ProcName}, \Omega', \_, \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega', \mathit{ProcName}, \rho, \widehat{\beta}, \rho_0, \emptyset>\} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}' = \textbf{true} \\ \mathcal{D}', \mathrm{MF}', \pi \vdash \textbf{ProcIter}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB}'' : \rho, \phi_0 \rightarrow \rho'', \phi''\end{array}}{\mathcal{D}, \mathrm{MF}, \phi, \pi \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB}'' : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{ProcName}, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{ProcName}, \_> \notin \pi) \\ \mathcal{D}, \mathrm{MF} \vdash \textbf{ProcBody}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho, \phi, \pi \rightarrow \rho', \phi', \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{ProcName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_> \notin \mathrm{MB}') \wedge (<\_, \mathit{ProcName}, \Omega', \_, \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega', \mathit{ProcName}, \rho, \widehat{\beta}, \rho_0, \emptyset>\} \\ \mathcal{D} = \textbf{true} \\ \mathcal{D}, \mathrm{MF}', \pi \vdash \textbf{ProcIter}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB}'' : \rho, \phi \rightarrow \rho'', \phi''\end{array}}{\mathcal{D}, \mathrm{MF}, \pi \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}) \Rightarrow \mathrm{MB}'' : \rho, \phi \rightarrow \rho'', \phi''}$$

Figure 7.19: Semantic rules for **ProcCall**.

$$\frac{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \in \mathrm{MB}') \vee (<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \notin \mathrm{MB})}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}' : \rho,\ \phi \rightarrow \rho',\ \phi'}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ <\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \\ \rho' \sqsubseteq \rho_{\mathrm{old}} \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\}\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}' : \rho,\ \phi \rightarrow \rho_{\mathrm{old}},\ \phi'}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ <\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \\ \rho' \not\sqsubseteq \rho_{\mathrm{old}} \\ \mathrm{MF}' = (\mathrm{MF} \setminus \{<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_>\}) \cup \{<\Omega',\ \mathit{ProcName},\ \rho,\ \widehat{\beta},\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \emptyset>\} \\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}' : \rho,\ \phi' \rightarrow \rho'',\ \phi''\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}' : \rho,\ \phi \rightarrow \rho'',\ \phi''}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}), \quad \text{for } <\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \rho_x,\ \_> \in \mathrm{MB} \\ \widehat{\beta}'' = \widehat{\beta} \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\widehat{\beta}_x\}), \quad \text{for } <\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \_,\ \widehat{\beta}_x> \in \mathrm{MB} \\ <\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \\ \mathrm{MF}' = (\mathrm{MF} \setminus \{<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_>\}) \cup \{<\Omega',\ \mathit{ProcName},\ \rho'',\ \widehat{\beta}'',\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \emptyset>\} \\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}'') \Rightarrow \mathrm{MB}'' : \rho'',\ \phi' \rightarrow \rho''',\ \phi''\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}'' : \rho,\ \phi \rightarrow \rho''',\ \phi''}$$

Figure 7.20: Semantic rules for **ProcIter**.

$$\frac{}{\vdash \mathbf{ProcBody}(\mathsf{bv},\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \mathbf{ProcBody}(\mathsf{InputDynamic},\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 :}$$

$$\frac{}{\vdash \mathbf{ProcBody}(\mathsf{nil},\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName})\end{array}}{\vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}}_i,\ \widehat{\mathit{LVarName}}_k,\ \mathit{PBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName}) \\ i \neq j\end{array}}{\vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}_j) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}}_j,\ \widehat{\mathit{LVarName}}_k,\ \mathit{PBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{ProcName}) \\ \tau = [\mathit{FormalPar}_i \mapsto \beta_i]\ , \quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <\mathit{ProcName},\ \rho,\ \widehat{\beta}_j> \\ \mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \Omega' \vdash [\![\mathit{PBody}]\!] \Rightarrow \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi'\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}_j) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi'}$$

Figure 7.21: Semantic rules for **ProcBody**.

$$\frac{<\Omega',\ FnctName,\ \rho',\ \widehat{\beta}',\ \rho'',\ \beta> \ \in \ \mathrm{MF} \quad (\rho = \rho') \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta}') \quad \mathrm{MB} = \{<\textbf{*use*},\ FnctName,\ \Omega',\ \_,\ \_>\}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{<\Omega',\ FnctName,\ \rho',\ \widehat{\beta}',\ \rho'',\ \beta> \ \in \ \mathrm{MF} \quad (\rho \neq \rho') \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta}') \quad \mathrm{MB} = \{<\textbf{*redo*},\ FnctName,\ \Omega',\ \rho,\ \widehat{\beta}>\}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{(<\Omega',\ FnctName,\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MF}) \wedge (<\Omega',\ FnctName,\ \_> \ \in \ \pi) \quad \mathrm{MB} = \{<\textbf{*redo*},\ FnctName,\ \Omega',\ \rho_0,\ \widehat{\emptyset}_j>\}}{\pi,\ \mathrm{MF} \vdash \textbf{FnctCall}(\Omega',\ ProcName,\ \widehat{\beta}_j) \Rightarrow \mathrm{MB} : \rho \rightarrow \rho'}$$

$$\frac{\begin{array}{c}(<\Omega',\ FnctName,\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MF}) \wedge (<\Omega',\ FnctName,\ \_> \ \in \ \pi) \\ \mathcal{D},\ \mathrm{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ FnctName,\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \in \ \mathrm{MB}') \vee (<\_,\ FnctName,\ \Omega',\ \_,\ \_> \ \notin \ \mathrm{MB}) \\ \pi'' = \pi' \setminus \{<\_,\ FnctName,\ \Omega'>\}\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \textbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB}' : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi''}$$

$$\frac{\begin{array}{c}(<\Omega',\ FnctName,\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MF}) \wedge (<\Omega',\ FnctName,\ \_> \ \in \ \pi) \\ \mathcal{D},\ \mathrm{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ FnctName,\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MB}') \wedge (<\_,\ FnctName,\ \Omega',\ \_,\ \_> \ \in \ \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega',\ FnctName,\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}' = \textbf{true}\end{array}}{\mathcal{D}',\ \mathrm{MF}',\ \pi \vdash \textbf{FnctIter}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}'' : \rho,\ \phi_0 \rightarrow \rho'',\ \phi''}$$

$$\frac{\begin{array}{c}(<\Omega',\ FnctName,\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MF}) \wedge (<\Omega',\ FnctName,\ \_> \ \in \ \pi) \\ \mathcal{D},\ \mathrm{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ FnctName,\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \notin \ \mathrm{MB}') \wedge (<\_,\ FnctName,\ \Omega',\ \_,\ \_> \ \in \ \mathrm{MB}) \\ \mathrm{MF}= \mathrm{MF} \cup \{<\Omega',\ FnctName,\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \textbf{true}\end{array}}{\mathcal{D},\ \mathrm{MF}',\ \pi \vdash \textbf{FnctIter}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}'' : \rho,\ \phi \rightarrow \rho'',\ \phi''}$$

$$\frac{}{\mathcal{D},\ \mathrm{MF},\ \phi,\ \pi \vdash \textbf{FnctCall}(\Omega',\ FnctName,\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}'' : \rho \rightarrow \rho'',\ \phi''}$$

Figure 7.22: Semantic rules for **FnctCall**.

$$\frac{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi{\rightarrow}\rho',\ \phi',\ \pi' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \in \mathrm{MB}') \vee (<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \notin \mathrm{MB})}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB}' : \rho,\ \phi{\rightarrow}\rho',\ \phi'}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi{\rightarrow}\rho',\ \phi',\ \pi' \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ <\Omega',\ \mathit{FnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \\ (\rho' = \rho_{\mathrm{old}}) \wedge (\beta \sqsubseteq \beta_{\mathrm{old}}) \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\}\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta_{\mathrm{old}},\ \mathrm{MB}' : \rho,\ \phi{\rightarrow}\rho_{\mathrm{old}},\ \phi'}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi{\rightarrow}\rho',\ \phi',\ \pi' \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ <\Omega',\ \mathit{FnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \\ (\rho' \neq \rho_{\mathrm{old}}) \vee (\beta \not\sqsubseteq \beta_{\mathrm{old}}) \\ \mathrm{MF}' = (\mathrm{MF} \setminus \{<\Omega',\ \mathit{FnctName},\ \_,\ \_,\ \_,\ \_>\}) \cup \{<\Omega',\ \mathit{FnctName},\ \rho'',\ \widehat{\beta}'',\ \rho',\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\} \\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}' : \rho,\ \phi'{\rightarrow}\rho'',\ \phi''\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}' : \rho,\ \phi{\rightarrow}\rho'',\ \phi''}$$

$$\frac{\begin{array}{c}\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi{\rightarrow}\rho',\ \phi',\ \pi' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\textbf{*redo*},\ \mathit{Fnctname},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \\ \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}) \ , \quad \text{for } <\textbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \rho_x,\ \_> \in \mathrm{MB} \\ \widehat{\beta}' = \widehat{\beta} \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\widehat{\beta}_x\}) \ , \quad \text{for } <\textbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \_,\ \widehat{\beta}_x> \in \mathrm{MB} \\ <\Omega',\ \mathit{FnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \\ \mathrm{MF}' = (\mathrm{MF} \setminus \{<\Omega',\ \mathit{FnctName},\ \_,\ \_,\ \_,\ \_>\}) \cup \{<\Omega',\ \mathit{FnctName},\ \rho'',\ \widehat{\beta}',\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\} \\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}') \Rightarrow \beta',\ \mathrm{MB}'' : \rho'',\ \phi'{\rightarrow}\rho''',\ \phi''\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta',\ \mathrm{MB}'' : \rho,\ \phi{\rightarrow}\rho''',\ \phi''}$$

Figure 7.23: Semantic rules for **FnctIter**.

$$\frac{\vdash \mathbf{DoBaseFnctCall}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta_i}) \Rightarrow \beta :}{\vdash \mathbf{FnctBody}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB}_0 :}$$

$$\frac{}{\vdash \mathbf{FnctBody}(\mathsf{InputDynamic},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \{\mathsf{InputDynamic}\},\ \mathrm{MB}_0 :}$$

$$\frac{}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\mathsf{nil},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \{\ \},\ \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName})\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \{\ \},\ \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_i},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ i \neq j\end{array}}{\vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_j}) \Rightarrow \{\ \},\ \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_j},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ \tau = [\mathit{FormalPar}_i \mapsto \beta_i]\ ,\quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <\mathit{FnctName},\ \rho,\ \widehat{\beta_j}> \\ \mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \Omega' \vdash [\![\mathit{FBody}]\!] \Rightarrow \mathrm{MB} : \rho,\ \tau,\ \phi,\ \pi \rightarrow \rho',\ \tau',\ \phi',\ \pi'\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta_j}) \Rightarrow \tau'(\mathit{ResultName}),\ \mathrm{MB} : \rho,\ \phi,\ \pi \rightarrow \rho',\ \phi',\ \pi'}$$

Figure 7.24: Semantic rules for **FnctBody**.

7.3.3 How to prove correctness

There are two thing that we would like to be able to prove about our abstract interpretation algorithm. The first being that the abstract interpretation should terminate for any legal source program. The second being that the during the abstract interpretation, the abstract program states are always safe approximations to the possible real program states occurring during actual interpretation of the source program with real values substituted for the unknown ones.

The first thing is impossible to prove. It is on the contrary possible to disprove it by trying the algorithm on a program that will not terminate for any input. Since our specification of when the abstract interpretation will actually terminate is too vague, it will not be possible to prove this either.

The second thing should be possible to prove using induction. The proof is somewhat complicated by the fact that one execution path in the abstract interpretation will match several execution paths during real interpretation because of the recursion. To do the proofs clearly requires some work on specifying what real execution paths are matched by an execution path during abstract interpretation and on defining miscellaneous invariants. We believe that we will be able to do the proofs with only a little more practice. It will however take a long time for us if we ever do it.

Chapter 8

Code Generation During Abstract Interpretation

Partial evaluation is as mentioned in Chapter 2 based upon the following basic principles:

- Symbolic computation
- Unfolding
- Program point specialization

In this chapter we describe how it is possible to extend the abstract interpretation (also called symbolic computation) to generate code for specialized methods (program point specialization). By code we here mean the program fragments that the final residual program is going to consist of. The actual program generation is left to a subsequent phase of the partial evaluator.

The important point of the code generation is the strategy of when to specialize (*reduce*) computations and when to generate residual code for them. Computations are here either: computing the value of expressions, or assigning values to variables. Invocations of methods can be eliminated by *unfolding*, but this will not be addressed in this chapter.

The residual program should consist of the computations that it was not possible to perform during partial evaluation. For assignments this is however not the whole truth. Some assignments of globally accessible values to variables may have to occur in the residual program even though such assignments apparently can be performed symbolically. This happens if a variable is assigned a new value in (at least) one branch in an undecidable branch situation and not in the other(s).

Consider the case where the value was assigned a globally accessible value before the undecidable branch and it was assumed at the time of performing the assignment symbolically that the assignment could be performed completely at specialization time. There is therefore not an assignment statement in the already generated program fragments that will assign the variable the value it has before the undecidable branch. We must consequently generate an assignment statement that ensures the variable has the correct value independent of what branch is taken at runtime. Following [Meyer 91] we use the

term *explicators* for assignment statements that must be generated because of undecidable branches.

In Section 8.1 we will describe some of the problems related to the code generation. For the minor problems we will only describe the solution we are going to use. For the major problems we will list the possible solutions and choose among them. The problem of when to reduce or residualize an assignment statement and how to generate the necessary explicators are the only problems we consider being major problems. In Section 8.2 we describe how we intend to collect the generated program fragments and combine them to larger fragments with the granularity of residual methods. Finally in Section 8.3 we give an operational semantics using the same style as in the previous chapters.

8.1 Problem Analysis for the Code Generation

In this section we will describe some of the problems related to the generation of code during the abstract interpretation. The problems we will discuss are, (1) when an assignment statement should be specialized and when it should be residualized, (2) making objects globally accessible by having them created at load time rather than runtime, (3) name clashes in speculative recursive loops, (4) forced use of source program method names, generating explicators, and (5) various optimizations possible during code generation.

8.1.1 Specializing Assignment Statements

If we generate residual assignment statements for all assignment statements that are performed symbolically during the abstract interpretation, then we would in most cases generate a lot of unnecessary assignments. Ideally we only want to generate residual assignment statements for those assignments that affect the semantics of the residual program. Assignments to variables that are never used should not be generated. This does however require *lazy* generation of assignment statements.

We do not intend to use lazy code generation. Instead we want to decide during the abstract interpretation which statements to specialize and which to residualize. We assume that assignment of nonmanifest values and manifest values consisting of a quantified object value must be residualized. This leaves us with the problem of whether or not to residualize assignment of manifest values consisting of a singular object value.

If the value assigned to the variable is accessible for the lifetime of the assignment, then we do not have to generate the assignment because we instead can use the constant/variable that references this value. If the value is not globally accessible, then determining of the value is accessible for the lifetime of the assignment requires an analysis of the program. We do not want to perform such an analysis. Consequently we will only specialize an assignment if the value assigned is globally accessible. The globally accessible values are of course all the named builtin objects (like the integers) and the constants of the residual program.

8.1.2 Globalizing Object Creation

We only specialize assignments if the value assigned is a manifest value consisting of a singular globally accessible value. We are therefore interested in making as many objects as possible global in the residual program. Creating the objects at load time rather than at runtime also reduces the runtime of the residual program. This may be said to be cheating but then again we have almost never seen anybody count the time for loading the program when listing speedup achieved by partial evaluation.

An object may be globalized if the object is created outside speculative loops and none of the instance variables after initialization of the object references objects that are not globally accessible.

Due to the evaluation order of constant declarations we cannot globalize two objects that mutually references each other after initialization and otherwise fulfill the requirements for globalization. The problem is illustrated by Example 8.1.

Example 8.1 Two objects that cannot both be global in the residual program
Consider the program fragment:

```
object a
   var x var y
   function create[] → [result]
      var x1
      x1 ← x
      result ← object b
         var z
         initially
            z ← x1
         end initially
      end b
   end create
   initially
      x ← self
      y ← self.create[]
   end initially
end a
```

The object created by the object constructor b fulfills the requirements of being global in the residual program if the object created by the object constructor a is global. If they are both created by constant declarations of the form "**const** *name* == **object** ..." then their mutual referencing is impossible because of the evaluation order of constant declarations. In this case only the object created by the object constructor a can be made global.

A fact also illustrated by Example 8.1 is that not only objects whose instance instance variables after initialization only references globally accessible values may be globalized. Determining what objects may be globalized is quite complicated. This is to some extent exemplified by Example 8.2.

Example 8.2 Some cases where globalization is not allowed
Inspired by Example 8.1 one could suggest that if an object's instance variables after initialization references only globally accessible value *or* singular object values created during initialization, then the object could be globalized. This is however not always correct as the instance variables of the object created during initialization of the "outer" object may reference objects that are not globally accessible.

If an object after initialization references only globally accessible values besides a quantified object created during initialization of the object, then globalization is also impossible as the residual initialization part of the "outer" object would have to contain a speculative loop.

We suggest that only objects satisfying the above mentioned criteria should be globalized. The residual initialization part generated during abstract interpretation of the source program initialization part should be discarded and replaced by a sequence of assignment statements performing the initialization. This because the generated initialization part may contain references to objects that are not accessible at the load time of the residual program.

8.1.3 Name Clashes in Speculative Recursive Loops

Consider a nonmanifest invocation where the different possible branches close different speculative loops. As we perform a polyvariant specialization with respect to methods, the already generated residual names for the methods that are the entrance points of the closed loops do not necessarily have to be the same. It may also be impossible to perform renaming of the methods that will make the names the same.

We suggest solving this problem by generating new method names for all invocations. When detecting a speculative recursive loop, we generate an *alias* between the generated name and the method being the entrance point of the loop. The postphase printing the program is then left with the problem of handling these aliases. In most cases proper renaming can solve the problem. In some cases the postphase has to generate a new method that does nothing but invoke another method.

8.1.4 Forced use of Method Names

Consider the abstract interpretation of a nonmanifest invocation, where one of the objects that can be invoked is either an unknown value or a builtin value. We will call such invocations *dynamic invocations*. Adding the invocation to the trace of computation to perform in the residual program, the method name *must* be the same as in the source program. We can not define aliases between methods in builtin or unknown objects as it may be impossible to resolve the aliases without adding methods that invokes the method that really should have been invoked in the first place (rephrased version of the above). The forced use of method names is illustrated by example 8.3.

Example 8.3 Residual method names may be forced
Consider the following program fragment:

```
function dummy[a] → [result]
   if a then
      result ← 4
   else
      result ← object withplus
         function plus[b] → [some]
            some ← self
         end plus
      end withplus
   end if
end dummy
⋮
temp ← self.dummy[dynamic]
temp ← temp.plus[4]
```

In the residual program, the object created by the object constructor *withplus* must have a function method with the name *plus*. Using any other name for the method could lead to runtime errors in the residual program.

If the target of a dynamic invocation is a symbolic value containing object values that denote program objects, then we may run into some nasty problems. If there are several such dynamic invocations, then we might end up trying to generate two specialized methods that have to have the same name in the same object.

We are of course only allowed to have one method (or rather one functional and one procedural method) with a given name in an object. We must therefore generate one method that performs all the computations that would have been performed by all the specialized methods we would have liked to generate. This method can be generated by performing an abstract interpretation of the source program method with a program state that is a generalization of all the program states we would have used generating the multiple methods with the same name.

8.1.5 Generating explicators

Assignment of globally accessible values are as mentioned reduced so they do not appear in the trace of generated residual code. At some point after the assignment is reduced it may however turn out that the assignment must be left residual anyway. A simple example of when this can happen is given in Example 8.4.

Example 8.4 A reduced assignment may have to be residualized anyway (local)

Consider the following fragment of a program, where <*dyn*> represent any expression that will cause both branches to be taken by the abstract interpretation:

```
var i
...
procedure modify[]
    i ← 3
    if <dyn> then
        i ← 2
    else
        skip
    end if
end modify
```

When encountering the first assignment to i during abstract interpretation it is easily seen that the assignment should be specialized. After abstract interpretation of the two branches it becomes clear that the assignment should appear in the residual program anyway in case the "false" branch of the conditional jump is taken at runtime.

The problem illustrated by Example 8.4 is not that severe since we can add the assignments to the program fragments we are currently generating. We can identify the need for inserting the assignment statements when we generalize the program states created by taking the two branches. Assignment statements inserted before the points of program state generalization to ensure that variables are set in the residual program is called *explicators* [Meyer 91]. This technique is general enough for ordinary imperative programs where the program state is determined by the values assigned to global variables. For object-oriented programming languages where the program state is determined by both the values bound to the global constants and the internal state of objects, the problem can not be solved so easily. The point where it is detected, that a reduced assignment really had to be performed anyway, can be in another object. In this case the variable cannot be assigned a value at the point of generalization. This is illustrated by Example 8.5.

Example 8.5 A reduced assignment may have to be residualized anyway (global)
Consider the following fragment of a program:

```
var i
...
procedure modify[]
    i ← 2
end modify
procedure change[]
    i ← 3
end change
```

Suppose the method *modify* is invoked in a manifest invocation. Since the object representing the integer 2 is globally accessible the assignment is specialized. Then suppose that the method *change* subsequently is invoked in a nonmanifest invocation. After symbolic execution of all the possible branches it becomes clear that the assignment in the body of the method *modify* must appear in the residual program. If the nonmanifest invocation of the method *change* is in another object, then we cannot use the simple technique of inserting explicator assignments before the nonmanifest invocations.

We propose a novel (at least to our knowledge) technique for generating explicator assignments for variables possibly out of the current scope. We will keep track of where variables referencing globally accessible values are assigned their present value. This may

be in more than one place if the variable is assigned the same value in all branches of an undecidable branch. This information can be used to add the assignment statement to the residual program in the place it would have been if the decision to specialize instead had been a decision to residualize.

Suppose we generate new unique labels for all residual basic blocks. We can then use these labels as almost complete specifications of where a variable was assigned a value. In fact the labels can be used to identify the place an explicator assignment should be inserted if two simple rules are obeyed. The first rule is that explicator statements should be inserted at the end of the basic block. The second rule is that there must not be a residual method invocation later in the basic block than where the reduced assignment would have been had it instead been residualized.

The first requirement is caused by the fact that a variable may be assigned a value twice in the same basic block. The first assignment can be residualized and the second specialized. It is the last of these, we have to generate an explicator assignment for. The second requirement is necessary because it may be the method invocation that caused the need for the explicator assignment statement. This is illustrated by Example 8.6.

Example 8.6 Problems with inserting explicators
Consider the following basic block:

```
label:  a ← 4
        self.SomeProcedure[]
        goto nextlabel
```

The assignment to the variable *a* should be reduced. Suppose the body of the method *SomeProcedure* modifies the value of the variable *a* in one part of an undecidable branch but not in the other. This would require the generation of an explicator assignment that performs the just reduced assignment. This explicator assignment should be inserted before the method invocation. If explicators should always be inserted at the end of a basic block, then the method invocation should be in another basic block.

Suppose we extend the environments mapping variable names to symbolic values. The extended environments should map variables to symbolic values *and* if the symbolic value is a manifest value denoting a globally accessible object, then also a set of labels indicating all the residual basic blocks where the variable should have been assigned the present value had the assignment not been reduced. When comparing the program states after symbolic execution of the branches of an undecidable branch construction we have to generate explicators for all variables referencing manifest values denoting globally accessible objects that will reference nonmanifest values in the generalization of the program states. We will postpone the actual insertion of the explicators to the postphase and instead build a mapping from residual labels into sets of explicator assignment statements. The explicator assignment statements are very simple to generate as the name of the variable and the assigned value is readily available in the environments.

8.1.6 Various Optimizations possible during Code Generation

In method invocations where some of the arguments are global values, these arguments can be removed. If the invoked method is recursive, the arguments that are global values

at the invocation point may be generalized so the body of the method does not see them as global values. It is thus problematic to remove the global arguments indiscriminately. There are two solutions to this problem. One is to allow aliases between methods, where the different method names assume a different number of arguments. That would require an implicit binding of formal parameter names to global values when using the shorter of the two argument lists. Another solution is to remove only the arguments that are global values in the final fixpoint of program states. We will use this last solution. The first solution is too much work for no real benefit.

Functional methods that return global values can also be treated in a special way. For a manifest invocation of such a function, the function call expression is replaced by the constant name denoting the global value. For nonmanifest invocations, the function body is replaced by a single assignment statement assigning the global value to the result variable. This means that our specializer is going to be nonstrict [Bondorf 90b]. A function in the source program that may never terminate its computations but will return a specific global value if it terminates will be specialized to a method that always terminate. This means that the residual program has radically different termination properties than the original program. This is the same behavior as found in Fuse [Weise 91a].

8.2 Collecting the trace (code)

The interpretation rules for expressions and statements should be modified to generate new expressions and statements that are as reduced as possible. The routines for interpreting basic blocks should be extended to generate new basic blocks. The same routines can appropriately be used to generate the list of basic blocks making a method body or initialization part. The program fragments built by these routines are building blocks that are immediately used to build other blocks.

The labels of basic blocks must be correct in the sense that it must be possible to interpret the list according to the semantics given in Chapter 4. There should only be jumps to basic blocks in the same body. This can be ensured by having the rules for handling basic blocks return the residual label of the generated basic block to the rule that should generate the previous basic block.

Methods are the units of code fragments that should later be glued together to make the final residual program. Methods should be associated with an object, be named, have a list of formal parameters (and for functional methods a result variable), and have a body. The residual method body is to be generated by the routines performing the abstract interpretation of the method body of the source program.

The residual list of formal parameters can be created by filtering the list of formal parameters in the source program according to what actual parameters will be global objects. This can be deducted from the program state fixpoint at the time of method invocation. The actual list of parameters in a method invocation should of course be reduced to match the formal parameter list. Enough information to perform the reduction must consequently also be computed by the rule for abstract interpretation of the invocation.

The names of a method in an object must of course be equal to the name of the method

that should be invoked. This can be ensured by forcing the interpretation routine for a method invocation to generate a method with a given name. Forcing the interpretation routine to use a given name in the residual program gives problems with recursion. The problem is easily overcome by allowing methods to have more than one name. This is what we use aliases for.

The residual program should contain object constructor expressions for each program object in the object store. While performing the symbolic execution of a method a specialized method is generated. In this way we generate a set of specialized methods for each object. These are the methods that will make out the body of the object constructor. These are the only methods that are needed as no other methods can be invoked (except by the unknown input values that are not allowed to invoke methods in program objects).

The object constructor cannot be built before the abstract interpretation has finished because new residual methods may be added at any time. The result of the abstract interpretation should therefore be a single return value, a set of residual methods for each object in the object store, a list of residual constant declarations that does not just create an object, and some information that enables the postphase to generate constant declarations for all the program objects that are globally accessible.

8.3 An Operational semantics

Once again the semantics of the abstract interpretation has become much more complicated than in the previous chapter. New semantic objects are introduced and now also syntactic objects are manipulated during the abstract interpretation.

The semantics of the abstract interpretation that generates code is given using the same notation as in the previous chapters. We have tried to prevent the introduction of even more semantic objects by stuffing more information inside the already existing ones. This does make some of the rules a bit clumsy but saves the other rules the addition of new semantic objects.

To the tuples in the object store we have added information that is to be used to build the residual object constructors. We call this part "ResCode". This part is itself a tuple. One of the elements of the ResCode tuples is a constant name. We have used this element to have the semantics that if the name is not the (artificial) bottom element of the constant name domain, then the residual object will be globally accessible in the residual program.

The basic semantic and syntactic objects used in the specification of the semantics is listed in Figure 8.1. Only two new semantic objects have been added compared to the previous chapter. All labels from the source program are indicated by syntactic objects called "label", whereas labels in the residual program to be are indicated syntactic objects called "lbl". The semantic object $\mathcal{M}$ is a flag used in the rules for method invocations and describes whether the current invocation is manifest or not.

The compound semantic objects used in the specification of the semantics is listed in Figure 8.2. Only a single semantic object ($\mathcal{M}$) has been introduced since the previous chapter, but the structure of some of the others have been altered. The environments no longer map names into elements of the "AbsVal" domain but maps names into elements of

FormalPar	$\in$	FORMALPAR
LVarName	$\in$	LVARNAME
ResultName	$\in$	RESULTNAME
LocalName	$\in$	LOCALNAME = FORMALPAR $\cup$ LVARNAME $\cup$ RESULTNAME
IVarName	$\in$	IVARNAME
ConstName	$\in$	CONSTNAME
FnctName	$\in$	FNCTNAME
ProcName	$\in$	PROCNAME
MethodName	$\in$	METHODNAME = PROCNAME + FNCTNAME
ObjectName, `oc`	$\in$	Object_Constructor_Name
		ObjectTag = {Single, Quantified}
`bv`	$\in$	Builtin_Object
Arg	$\in$	ARG = CONSTNAME + Builtin_Object + `nil` + `Dynamic`
InputDynamic	$\in$	INPUTDYNAMIC = {InputDynamic}
label	$\in$	Label
lbl	$\in$	ResLabel = Label
$\mathcal{B}$	$\in$	Body = BBlist
$\mathcal{M}$	$\in$	Manifest = Boolean
$\mathcal{D}$	$\in$	Dynamic = Boolean
		MethodTag = {***redo***, ***use***}

Figure 8.1: Basic Semantic Objects for Nonstandard interpretation with Code generation

the "SomeVal" domain. An element of "SomeVal" domain is an element of the "AbsVal" domain plus a set of residual labels. The Object_Store domain has also been changed as described above. The "ObjectName" part of the "ResCode" elements is the name to be of the residual object if the object is to be globally accessible in the residual program.

To the sets of forward propagated information we have also added extra elements. In addition to the source program methodnames and labels, the tuples now also contain residual method names and labels. This is required to generate residual code that actually closes the loops in the residual program.

The new semantic object is "δ" that is a finite mapping from residual labels into sets of explicator assignment statements.

The signatures of the semantic functions used in the description of the semantics is listed in Figure 8.3. The predicates "Global" and "NonGlobal" are to be used to determine if a program object will be globally accessible in the residual program or not. The predicates "Globalizable" and "NonGlobalizable" are to be used to determine if a just created object can be made global in the residual program.

The function "MakeExp" will map a object value denoting a global program object or a builtin object to an expression. The function "**MakeBody**" creates a new IBody from an instance environment. All variables in the instance environment must denote global values. The new IBody performs the assignments that when executed will create lead to the same instance environment. The function "**FilterOutGlobals**" takes two lists of the same length of which the first is a list of abstract values and returns a list of the same type. The second list will either be a list of formal parameters or a list of actual parameters, and the result is a list where those parameters are removed that correspond to manifest abstract values denoting globally accessible objects in the list of abstract values.

$$
\begin{array}{rcl}
\Omega & \in & \text{Abstract\_Object} = (\text{ObjectTag} \times \text{Program\_Object}) + \mathsf{nil} \\
\mathsf{nil} & \in & \text{Abstract\_Object} \\
x & \in & \text{OID} = \text{Builtin\_Object} + \text{Abstract\_Object} + \text{INPUTDYNAMIC} \\
\mathsf{true} & \in & \text{OID} \\
\mathsf{false} & \in & \text{OID} \\
\beta & \in & \text{AbsVal} = \mathcal{P}(\text{OID}) \\
\widehat{\beta} & \in & \text{Arglist} = \text{AbsVal list} \\
\text{Code} & \in & \text{CODE} = \\
 & & \quad \text{PROCNAME} \overset{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{LVARNAME list} \times \text{Body} + \\
 & & \quad \text{FNCTNAME} \overset{\text{fin}}{\mapsto} \text{FORMALPAR list} \times \text{RESULTNAME} \times \text{LVARNAME list} \times \text{Body} \\
 & & \text{SomeVal} = \mathcal{P}(\text{ResLabel}) \times \text{AbsVal} \\
\tau & \in & \text{LocalEnv} = \text{LOCALNAME} \overset{\text{fin}}{\mapsto} \text{SomeVal} \\
\imath & \in & \text{InstanceEnv} = \text{IVARNAME} \overset{\text{fin}}{\mapsto} \text{SomeVal} \\
 & & \text{ResCode} = \text{ObjectName}_{\perp} \times \text{CODE} \times \text{Body} \\
\rho & \in & \text{Object\_Store} = \text{Abstract\_Object} \overset{\text{fin}}{\mapsto} \\
 & & \quad (\text{Object\_Constructor\_Name} \times \text{CODE} \times \text{InstanceEnv} \times \text{ResCode}_{\perp}) \\
\text{MF} & \in & \text{MethodForward} = \mathcal{P}(\text{Abstract\_Object} \times \text{METHODNAME} \times \text{METHODNAME} \times \\
 & & \quad \text{Object\_Store} \times \text{Arglist} \times \text{Object\_Store} \times \text{AbsVal}) \\
\text{MB} & \in & \text{MethodBackward} = \mathcal{P}(\text{MethodTag} \times \text{METHODNAME} \times \text{Abstract\_Object} \times \\
 & & \quad \text{Object\_Store} \times \text{Arglist}) \\
\text{LF} & \in & \text{LabelForward} = \mathcal{P}(\text{Label} \times \text{ResLabel} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\text{LB} & \in & \text{LabelBackward} = \mathcal{P}(\text{Label} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\mathcal{C} & \in & \text{METHODNAME} \times \text{METHODNAME} \times \text{Object\_Store} \times \text{Arglist} \\
\phi & \in & \text{Quantified\_Map} = \text{Object\_Constructor\_Name} \overset{\text{fin}}{\mapsto} (\text{Program\_Object} \times \text{LocalEnv}) \\
\pi & \in & \text{Problematic} = \mathcal{P}(\text{Abstract\_Object} \times \text{METHODNAME} \times \text{Abstract\_Object}) \\
\delta & \in & \text{ExplicatorMap} = \text{ResLabel} \overset{\text{fin}}{\mapsto} \mathcal{P}(\text{Stmnt})
\end{array}
$$

Figure 8.2: Compound Semantic Objects for Nonstandard interpretation with Code generation

The semantic rule in Figure 8.4 describes the semantics of the specializer. The values returned are all that is needed to build the residual program. The invocations are manifest causing the formal parameter list to be reduced so the constant arguments does not have to be given as arguments in the residual program.

We have to invent new names for the used procedural and functional method. If we do not, then we are forced to use the **HardProcCall** and **HardFnctCall** semantic rules[1] and it may not be possible to remove the static arguments to the invoked procedural method. If no methods are generated in the target object with source program method names, then the methods can be renamed in the postphase. The set inclusion is assumed to be generative as well for syntactic objects as for semantic objects. In this way we ensure that we do not generate two methods with the same name.

The semantic rules in Figure 8.5 describe the semantics of constant declarations. At the time of evaluating an object constructor it is decided if the object is to be globally accessible in the residual program based on an inspection of the objects instance environment after initialization. If the expression part of a constant declaration evaluates to an

[1] will be explained later

γ	$\in$	BaseValueName $\mapsto$ Builtin_Object
ϵ	$\in$	MethodDef* $\mapsto$ CODE
DoBaseFnctCall	$\in$	Builtin_Object $\times$ fnctname $\times$ AbsVal list $\mapsto$ AbsVal
Explicator	$\in$	Object_Store $\times$ LocalEnv $\times$ Object_Store $\times$ LocalEnv $\times$ ExplicatorMap $\mapsto$ ExplicatorMap
		Env $\times$ EnvExplicatorMap $\mapsto$ ExplicatorMap
		SomeVal $\times$ SomeVal $\times$ ExplicatorMap $\mapsto$ ExplicatorMap
Alias	$\in$	Abstract_Object $\times$ MethodName $\times$ MethodName $\times$ Object_Store $\mapsto$ Object_Store
Manifest	$\in$	AbsVal $\mapsto$ Boolean
NonManifest	$\in$	AbsVal $\mapsto$ Boolean
Singular	$\in$	AbsVal $\mapsto$ Boolean
NonSingular	$\in$	AbsVal $\mapsto$ Boolean
Disjoint	$\in$	AbsVal $\times$ AbsVal $\mapsto$ Boolean
NonDisjoint	$\in$	AbsVal $\times$ AbsVal $\mapsto$ Boolean
Global	$\in$	AbsVal $\mapsto$ Boolean
NonGlobal	$\in$	AbsVal $\mapsto$ Boolean
Globalizable	$\in$	InstanceEnv $\mapsto$ Boolean
NonGlobalizable	$\in$	InstanceEnv $\mapsto$ Boolean
MakeExp	$\in$	OID $\mapsto$ Expression$_\perp$
MakeBody	$\in$	InstanceEnv $\mapsto$ IBody
FilterOutGlobals	$\in$	Object_Store $\times$ SomeList $\mapsto$ SomeList

Figure 8.3: Functions on semantic objects

$$\vdash [\![\mathit{Program}]\!] \Rightarrow \mathit{NewProgram},\ \rho$$
$$\rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \Omega$$
$$\rho \vdash [\![\mathit{Arg}*]\!] \Rightarrow \widehat{\beta}$$
$$\mathcal{D} = \mathbf{false}$$
$$\mathcal{M} = \mathbf{true}$$
$$\mathit{NewProcName} \in \text{PROCNAME}$$
$$\mathit{NewFnctName} \in \text{FNCTNAME}$$
$$\mathcal{M},\mathcal{D},\mathrm{MF}_0 \vdash \mathbf{ProcCall}(\Omega,\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}_{\mathrm{none}},\mathrm{MB}_{\mathrm{none}} : \rho,\phi_0,\pi_0,\delta_0 \rightarrow \rho',\phi',\pi',\delta'$$
$$\mathcal{M},\mathcal{D},\mathrm{MF}_0 \vdash \mathbf{FnctCall}(\Omega,\ \mathit{FnctName},\ [\,],\ \mathit{NewFnctName}) \Rightarrow \beta,\widehat{\beta}_{\mathrm{null}},\mathrm{MB}_{\mathrm{null}} : \rho',\phi_0,\pi_0,\delta' \rightarrow \rho'',\phi'',\pi'',\delta''$$
$$\vdash [\![\mathit{Program\ ConstName\ ProcName\ Arg}*\ \mathit{FnctName}]\!] \Rightarrow \mathit{NewProgram},\ \beta,\ \rho'',\ \delta''$$

Figure 8.4: Semantic rules for the input–output function of a program.

abstract value with a single object value that is already marked as being global in the residual program, then there is no need to generate an extra constant declaration for this value. If the object value is not marked as being global in the residual program, then it is marked as being so and a new constant declaration is generated.

$$\frac{Program = ConstDecl^* \quad \vdash [\![ConstDecl^*]\!] \Rightarrow NewProgram : \rho_0 \rightarrow \rho}{\vdash [\![Program]\!] \Rightarrow NewProgram : \rho_0 \rightarrow \rho}$$

$$\frac{\begin{array}{c}\vdash [\![ConstDecl^*]\!] \Rightarrow NewConstDecls : \rho \rightarrow \rho' \\ \vdash [\![ConstDecl]\!] \Rightarrow NewConstDecl : \rho' \rightarrow \rho'' \\ NewList = NewConstDecl :: NewConstDecls\end{array}}{\vdash [\![ConstDecl^*\ ConstDecl]\!] \Rightarrow NewList : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}\Omega = \mathsf{nil} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}, \mathcal{C}_0, \mathrm{MF}_0, \Omega \vdash [\![Exp]\!] \Rightarrow \{x\}, \mathrm{MB}, NewExp : \rho, \pi_0, \phi_0, \delta_0 \rightarrow \rho', \pi', \phi', \delta' \\ \mathrm{Global}(x) \\ \rho'' = \rho'[\Omega, ConstName \mapsto \{x\}] \\ NewConstDecl = [\,]\end{array}}{\vdash [\![\textbf{const}\ ConstName == Exp]\!] \Rightarrow NewConstDecl : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}\Omega = \mathsf{nil} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}, \mathcal{C}_0, \mathrm{MF}_0, \Omega \vdash [\![Exp]\!] \Rightarrow \{x\}, \mathrm{MB}, NewExp : \rho, \tau_0, \phi_0, \delta_0 \rightarrow \rho', \tau', \phi', \delta' \\ \mathrm{NonGlobal}(x) \\ <\_, MDef, NewIBody> = \rho_{\mathrm{rescode}}(x) \\ \rho'' = \rho'[x \overset{\mathrm{rescode}}{\mapsto} <ConstName, MDef, NewIBody>] \\ \rho'' = \rho'[\Omega, ConstName \mapsto \{x\}] \\ NewDecl = [\,[\![\textbf{const}\ ConstName == NewExp]\!]\,]\end{array}}{\vdash [\![\textbf{const}\ ConstName == Exp]\!] \Rightarrow NewDecl : \rho \rightarrow \rho''}$$

Figure 8.5: Static semantic rules for nonstandard interpretation of a program and constant declarations
f

The semantic rules in Figure 8.6 describe the abstract interpretation and code generation for basic expressions. The basic principle behind the code generation is that we wish to avoid using the local and instance environments. Besides possibly increasing program speed if the residual program is compiled rather than interpreted, this strategy may also make merging of objects easier and allow some methods to be unfolded.

The semantic rules listed in Figure 8.7 describe the the semantics of the simple expressions. The rules are trivial extensions of the rules in Figure 7.10.

The semantic rules listed in Figure 8.8 describe the semantics of invocations of functional methods. These rules together with the **EvalFnct** semantic rules in Figure 8.9 replace the semantic rules in Figure 7.11. An extra "layer" of semantic rules has been

$$\frac{<\beta, \_> = \rho_{\text{env}}(\mathsf{nil})(\mathit{ConstName}) \quad \beta \neq \{\Omega\}}{\rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \beta,\ \mathit{ConstName}}$$

$$\frac{<\beta, \_> = \rho_{\text{env}}(\mathsf{nil})(\mathit{ConstName}) \quad \beta = \{\Omega\}}{\rho \vdash [\![\mathit{ConstName}]\!] \Rightarrow \beta,\ \mathbf{self}}$$

$$\frac{<\beta, \_> = \rho(\Omega)(\mathit{IVarName}) \quad (\beta = \{\Omega'\}) \wedge \text{Global}(\Omega') \wedge (\Omega' \neq \Omega) \quad \mathit{ResExp} = \text{MakeExp}(\Omega')}{\Omega, \rho \vdash [\![\mathit{IVarName}]\!] \Rightarrow \beta,\ \mathit{ResExp}}$$

$$\frac{<\beta, \_> = \rho(\Omega)(\mathit{IVarName}) \quad \beta = \{\Omega\}}{\Omega, \rho \vdash [\![\mathit{IVarName}]\!] \Rightarrow \beta,\ \mathbf{self}}$$

$$\frac{<\beta, \_> = \rho(\Omega)(\mathit{IVarName}) \quad (\text{Arity}(\beta) \neq 1) \vee ((\beta = \{\Omega'\}) \wedge \text{NonGlobal}(\Omega') \wedge (\Omega' \neq \Omega))}{\Omega, \rho \vdash [\![\mathit{IVarName}]\!] \Rightarrow \beta,\ \mathit{IVarName}}$$

$$\frac{<\beta, \_> = \tau(\mathit{LocalName}) \quad (\beta = \{\Omega'\}) \wedge \text{Global}(\Omega') \wedge (\Omega' \neq \Omega) \quad \mathit{ResExp} = \text{MakeExp}(\Omega')}{\tau \vdash [\![\mathit{LocalName}]\!] \Rightarrow \beta,\ \mathit{ResExp}}$$

$$\frac{<\beta, \_> = \tau(\mathit{LocalName}) \quad \beta = \{\Omega\}}{\tau \vdash [\![\mathit{LocalName}]\!] \Rightarrow \beta,\ \mathbf{self}}$$

$$\frac{<\beta, \_> = \tau(\mathit{LocalName}) \quad (\text{Arity}(\beta) \neq 1) \vee ((\beta = \{\Omega'\}) \wedge \text{NonGlobal}(\Omega') \wedge (\Omega' \neq \Omega))}{\tau \vdash [\![\mathit{LocalName}]\!] \Rightarrow \beta,\ \mathit{LocalName}}$$

$$\frac{}{\Omega \vdash [\![\mathbf{self}]\!] \Rightarrow \{\Omega\},\ \mathbf{self}}$$

$$\frac{\mathsf{bv} = \gamma(\mathit{BaseValueName})}{\rho \vdash [\![\mathit{BaseValueName}]\!] \Rightarrow \{\mathsf{bv}\},\ \mathit{BaseValueName}}$$

Figure 8.6: Semantic deduction rules for basic expressions.

$$\frac{\begin{array}{c} Exp = BExp \\ \rho,\, \tau,\, \Omega \vdash [\![BExp]\!] \Rightarrow \{\Omega'\},\, NewBExp \end{array}}{\rho,\, \tau,\, \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\, NewBExp,\, \mathrm{MB}_0}$$

$$\frac{\begin{array}{c} \rho,\, \tau,\, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta,\, NewExp1 \\ \rho,\, \tau,\, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta,\, NewExp2 \\ \mathrm{Singular}(\beta) \\ ResExp = \mathrm{MakeExp}(\mathsf{true}) \end{array}}{\rho,\, \tau,\, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{true}\},\, ResExp,\, \mathrm{MB}_0}$$

$$\frac{\begin{array}{c} \rho,\, \tau,\, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta_1,\, NewExp1 \\ \rho,\, \tau,\, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta_2,\, NewExp2 \\ \mathrm{Disjoint}(\beta_1, \beta_2) \\ ResExp = \mathrm{MakeExp}(\mathsf{false}) \end{array}}{\rho,\, \tau,\, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{false}\},\, ResExp,\, \mathrm{MB}_0}$$

$$\frac{\begin{array}{c} \rho,\, \tau,\, \Omega \vdash [\![BExp1]\!] \Rightarrow \beta_1,\, NewExp1 \\ \rho,\, \tau,\, \Omega \vdash [\![BExp2]\!] \Rightarrow \beta_2,\, NewExp2 \\ \mathrm{NonDisjoint}(\beta_1, \beta_2) \wedge (\mathrm{NonSingular}(\beta_1) \vee \mathrm{NonSingular}(\beta_2)) \\ NewExp = [\![NewExp1 == NewExp2]\!] \end{array}}{\rho,\, \tau,\, \Omega \vdash [\![BExp1 == BExp2]\!] \Rightarrow \{\mathsf{true}, \mathsf{false}\},\, NewExp,\, \mathrm{MB}_0}$$

Figure 8.7: Semantic rules for equality and simple (basic) expressions.

added to make at simple to replace the method invocation with a simpler expression if the value returned by the method invocation is a globally accessible value.

The structure of the semantic rules in Figure 8.9 is completely similar to the structure of the semantic rules in Figure 8.12. The **HardFnctCall** semantic rules are listed in Figure 8.25.

$$\frac{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash \textbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \{x\},NExp,\mathrm{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta' \quad \mathrm{Global}(x) \quad ResExp = \mathrm{MakeExp}(x)}{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \{x\},ResExp,\mathrm{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

$$\frac{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash \textbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta,NExp,\mathrm{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta' \quad (\mathrm{Arity}(\beta) \neq 1) \vee ((\beta = \{x\}) \wedge \mathrm{NonGlobal}(x))}{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta,NExp,\mathrm{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

Figure 8.8: Semantic rules for function call expressions.

The semantic rules listed in Figure 8.10 defines the semantics of abstract interpretation and code generation for object constructor expressions. These rules correspond to the semantic rules in Figure 7.12.

The first rule describes the creation of a globally accessible object. The residual expression generated is just the constant name that will reference the object in the residual program. The object value is inserted in the object store before abstract interpretation of the body of the initialization part. If the instance variables of the object after initialization body only references globally accessible object it is possible to create the object at load time in the residual program. The generated body of the initialization part is discarded and replaced by a new body that performs exactly the initialization of the object and nothing more. The objects residual name is inserted in the "ResCode" tuple to indicate that the object will be globally accessible in the residual program. The residual methods in the object that was created during abstract interpretation of the initialization part of the object is discarded as these will never be used.

The second rule describes the creation of a singular object that will not be globally accessible in the residual program. In this rule the generated body of the initialization part is to be used in the residual program.

The third rule describes the creation of a quantified object that has not been created before in the current speculative loop. This rule is very similar to the rule describing creation of a singular object that will not be globally accessible.

The fourth rule describes the creation of a quantified object that has been created before. To make sure that the residual initialization part has not been generated using a too specialized local environment, a new residual initialization part is generated using a local environment that is the generalization of the current local environment and the previously used local environment. The generalization of the two local environments may require generation of explicators. The generalized local environment is preserved to serve as the next "previously used local environment". The previously generated initialization

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \mathrm{Manifest}(\beta') \wedge (\beta' \subset \mathrm{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ NewFnctName \ \in \mathrm{FNCTNAME} \\ \mathcal{M} = \mathbf{true} \\ \beta' = \{x\} \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctCall}(x,\ FnctName,\ \widehat{\beta},\ NewFnctName) \Rightarrow \beta'',\ \widehat{\beta}',\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \rho \vdash \mathbf{FilterOutGlobals}(\widehat{\beta}',\ \widehat{NewBExp}) \Rightarrow NewBExp^* \\ ResExp = [\![NewBExp.NewFnctName[NewBExp^*]]\!]\end{array}}{\mathcal{D},\mathrm{MF},\tau,\Omega \vdash \mathbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \mathrm{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = \beta \setminus \{\mathsf{nil}\} \\ \mathrm{NonManifest}(\beta') \wedge (\beta' \subset \mathrm{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ < MethodName,\ ResMethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}} > = \mathcal{C} \\ \mathrm{MF}' = \mathrm{MF} \uplus \{< \Omega,\ MethodName,\ ResMethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}},\ \rho_0,\ \emptyset >\} \\ NFN \ \in \mathrm{FNCTNAME} \\ \mathcal{M} = \mathbf{false} \\ \mathcal{M},\mathcal{D},\mathrm{MF}' \vdash \mathbf{FnctCall}(x,\ FnctName,\ \widehat{\beta},\ NFN) \Rightarrow \beta_x,\widehat{\beta}_x,\mathrm{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \beta'' = \sqcup_{\mathrm{if}}\ \{\beta_x\}, \quad \text{for } x \in \beta' \\ \mathrm{MB}' = \cup\ \{\mathrm{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\mathrm{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \mathbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ ResExp = [\![NewBExp.NFN[NewBExp_1\ \ldots\ NewBExp_n]]\!]\end{array}}{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash \mathbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \mathrm{MB}' : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = \beta \setminus \{\mathsf{nil}\} \\ \mathrm{NonManifest}(\beta') \wedge (\beta' \not\subset \mathrm{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ < MethodName,\ ResMethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}} > = \mathcal{C} \\ \mathrm{MF}' = \mathrm{MF} \uplus \{< \Omega,\ MethodName,\ ResMethodName,\ \rho_{\mathrm{call}},\ \widehat{\beta}_{\mathrm{call}},\ \rho_0,\ \emptyset >\} \\ \mathcal{D},\mathrm{MF}' \vdash \mathbf{HardFnctCall}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \beta_x,\ \mathrm{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \beta'' = \sqcup_{\mathrm{if}}\ \{\beta_x\}, \quad \text{for } x \in \beta' \\ \mathrm{MB}' = \cup\ \{\mathrm{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\mathrm{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \mathbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ ResExp = [\![NewBExp.FnctName[NewBExp_1\ \ldots\ NewBExp_n]]\!]\end{array}}{\mathcal{D},\mathcal{C},\mathrm{MF},\tau,\Omega \vdash \mathbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \mathrm{MB}' : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

Figure 8.9: Semantic rules for **EvalFnct** expressions.

part for the quantified object is discarded.

The semantic rules in Figure 8.11 describe the abstract interpretation and code generation for assignment statements and composite statements. The first and the fourth rule describe assignment of a manifest value denoting a globally accessible object. These assignments are specialized and replaced by a **skip** statement. The value is of course also assigned symbolically. The label of the residual basic block that we are in the process of generating is associated with the abstract value in the proper environment to make generation of explicator assignments possible.

The three other rules describing assignment statements describes generation of residual assignment statements. Because a generalization is performed in the third rule it may be necessary to generate an explicator assignment statement. The last rule describes abstract interpretation and code generation for composite statements.

The semantic rules in Figure 8.12 describe the semantics of abstract interpretation and code generation for invocations of procedural methods. If the target of the invocation only can be program objects, then the code generation is fairly straightforward. This is described by the two first rules. Both rules force the generation of a method with a specific name.

If the invocation also is manifest then we wish to remove the globally accessible parameters to the method. If the method is recursive then the parameters may be part of the program state that is generalized and we therefore only remove the globally accessible parameters in the fixpoint of program state. The parameter list part of the program state fixpoint is returned by the **ProcCall** semantic rule.

If the invocation is not manifest, then we do not want to try to remove the globally accessible parameters. This would require that the methods be generated first and later stripping of the unnecessary parameters. We cannot tell which parameters are unnecessary before all the required methods are generated and possible program state fixpoints have been found. Because we generalize program state we might have to generate explicators. The local environment cannot be modified by the invocation so only the program stores are compared.

If the target of the invocation may be either a builtin object or an unknown input value, then we have to use the method name from the source program as described in Section 8.1.4. The semantics of the actual abstract interpretation and code generation for the possibly invoked methods is described by the **HardProcCall** semantic rules listed in Figure 8.21.

The semantic rules in Figure 8.13 describes the semantics of a body of a method or initialization part. There are not significant changes from the rules in Figure 7.14.

The semantic rules in Figure 8.14 describe interpretation and code generation for basic blocks. The rules all return a list of already generated basic blocks and the residual label of the specialized version of the current basic block. This is of course used to be used by the rule that specialized the basic block that ended with a (possible) jump to the current basic block.

In the first rule we have used four projection functions on the object store and combined the four projections again. The purpose of this is that we want to preserve possible generated residual program fragments since the entrance of the speculative loop and

$$\frac{\begin{array}{c} Exp = [\![\textbf{object oc } (IVarName^*)(MethodDef^*)\ IBody]\!] \\ \mathcal{D} = \textbf{false} \\ (\Omega' \in \text{Abstract\_Object}) \wedge (\Omega' = <\mathsf{Single}, \_>) \\ \text{MF}' = \{<\Omega'', \_, \_, \_, \_, \_, \_> \in \text{MF} \mid \mathsf{oc} = \rho_{\mathsf{oc}}(\Omega'')\} \\ \pi' = \pi \cup (\bigcup \{<\Omega', Name, x>\}), \quad \text{for } <x, Name, \_, \_, \_, \_, \_> \in \text{MF'} \\ \rho' = \rho[\Omega' \mapsto <\mathsf{oc}, \epsilon(\text{MethodDef}^*), \imath_0, \bot>] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![IBody]\!] \Rightarrow \text{MB}, NewIBody : \rho', \tau, \phi, \pi', \delta{\rightarrow}\rho'', \tau', \phi', \pi'', \delta' \\ \text{Globalizable}(\rho''_{\text{env}}(\Omega')) \\ \vdash \textbf{MakeBody}(\rho''_{\text{env}}(\Omega')) \Rightarrow MadeIBody : \\ ObjectName \in \text{Object\_Constructor\_Name} \\ \rho''' = \rho''[\Omega' \overset{\text{rescode}}{\mapsto} <ObjectName, \bot, MadeIBody>] \end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau \vdash [\![Exp]\!] \Rightarrow \{\Omega'\}, ObjectName, \text{MB} : \rho, \phi, \pi, \delta{\rightarrow}\rho''', \phi, \pi'', \delta'}$$

$$\frac{\begin{array}{c} Exp = [\![\textbf{object oc } (IVarName^*)(MethodDef^*)\ IBody]\!] \\ \mathcal{D} = \textbf{false} \\ (\Omega' \in \text{Abstract\_Object}) \wedge (\Omega' = <\mathsf{Single}, \_>) \\ \text{MF}' = \{<\Omega'', \_, \_, \_, \_, \_, \_> \in \text{MF} \mid \mathsf{oc} = \rho_{\mathsf{oc}}(\Omega'')\} \\ \pi' = \pi \cup (\bigcup \{<\Omega', Name, x>\}), \quad \text{for } <x, Name, \_, \_, \_, \_, \_> \in \text{MF'} \\ \rho' = \rho[\Omega' \mapsto <\mathsf{oc}, \epsilon(\text{MethodDef}^*), \imath_0, \bot>] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![IBody]\!] \Rightarrow \text{MB}, NewIBody : \rho', \tau, \phi, \pi', \delta{\rightarrow}\rho'', \tau', \phi', \pi'', \delta' \\ \text{NonGlobalizable}(\rho''_{\text{env}}(\Omega')) \\ <\_, code, \bot> = \rho''_{\text{rescode}}(\Omega') \\ \rho''' = \rho''[\Omega' \overset{\text{rescode}}{\mapsto} <\bot, code, NewIBody>] \end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau \vdash [\![Exp]\!] \Rightarrow \{\Omega'\}, \textbf{object } \Omega', \text{MB} : \rho, \phi, \pi, \delta{\rightarrow}\rho''', \phi', \pi'', \delta'}$$

$$\frac{\begin{array}{c} Exp = [\![\textbf{object oc } (IVarName^*)(MethodDef^*)\ IBody]\!] \\ \mathcal{D} = \textbf{true} \\ \bot = \phi(\mathsf{oc}) \\ (\Omega' \in \text{Abstract\_Object}) \wedge (\Omega' = <\mathsf{Quantified}, \_>) \\ \rho' = \rho[\Omega' \mapsto <\mathsf{oc}, \epsilon(\text{MethodDef}^*), \imath_0, \bot>] \\ \phi' = \phi[\mathsf{oc} \mapsto <\Omega', \tau>] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![IBody]\!] \Rightarrow \text{MB}, NewIBody : \rho', \tau, \phi, \pi, \delta{\rightarrow}\rho'', \tau', \phi', \pi', \delta' \\ <\_, code, \bot> = \rho''_{\text{rescode}}(\Omega') \\ \rho''' = \rho''[\Omega' \overset{\text{rescode}}{\mapsto} <\bot, code, NewIBody>] \end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau \vdash [\![Exp]\!] \Rightarrow \{\Omega'\}, \textbf{object } \Omega', \text{MB} : \rho, \phi, \pi, \delta{\rightarrow}\rho''', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c} Exp = [\![\textbf{object oc } (IVarName^*)(MethodDef^*)\ IBody]\!] \\ \mathcal{D} = \textbf{true} \\ <\Omega', \tau_{\text{last}}> = \phi(\mathsf{oc}) \\ \rho' = \rho[\Omega' \mapsto <\mathsf{oc}, \epsilon(\text{MethodDef}^*), \rho_{\text{env}}(\Omega')>] \\ \rho' = \rho[\Omega' \mapsto <\mathsf{oc}, \epsilon(\text{MethodDef}^*), \rho_{\text{env}}(\Omega'), \rho_{\text{rescode}}(\Omega')>] \\ \vdash \textbf{Explicator}(\tau_{\text{last}}, \tau) : \delta{\rightarrow}\delta' \\ \tau' = \tau_{\text{last}} \sqcup_{\text{if}} \tau \\ \phi' = \phi[\mathsf{oc} \mapsto <\Omega', \tau'>] \\ \mathcal{D}, \mathcal{C}, \text{MF}, \Omega' \vdash [\![IBody]\!] \Rightarrow \text{MB}, NewIBody : \rho', \tau', \phi, \pi, \delta'{\rightarrow}\rho'', \tau'', \phi', \pi', \delta'' \\ <\_, code, \bot> = \rho''_{\text{rescode}}(\Omega') \\ \rho''' = \rho''[\Omega' \overset{\text{rescode}}{\mapsto} <\bot, code, NewIBody>] \end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega, \tau \vdash [\![Exp]\!] \Rightarrow \{\Omega'\}, \textbf{object } \Omega', \text{MB} : \rho, \phi, \pi, \delta{\rightarrow}\rho''', \phi', \pi', \delta''}$$

Figure 8.10: Semantic equations for object constructor expressions.

$$\frac{\begin{array}{c}\Omega = \langle \mathsf{Single}, \_ \rangle \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \{x\}, \mathrm{MB}, NewExp : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta' \\ \mathrm{Global}(x) \\ \rho'' = \rho'[\Omega, IVarName \mapsto \langle \{x\}, \{lbl\} \rangle] \\ NewStmnt = [\![\mathbf{skip}]\!]\end{array}}{lbl, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![IVarName \leftarrow Exp]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \phi, \pi, \delta \rightarrow \rho'', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}\Omega = \langle \mathsf{Single}, \_ \rangle \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}, NewExp : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta' \\ (\mathrm{Arity}(\beta) \neq 1) \vee ((\beta = \{x\}) \wedge (\mathrm{NonGlobal}(x))) \\ \rho'' = \rho'[\Omega, IVarName \mapsto \langle \beta, \{\ \} \rangle] \\ NewStmnt = [\![IVarName \leftarrow NewExp]\!]\end{array}}{\mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![IVarName \leftarrow Exp]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \phi, \pi, \delta \rightarrow \rho'', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}\Omega = \langle \mathsf{Quantified}, \_ \rangle \\ \mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}, NewExp : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta' \\ \beta' = \beta \sqcup_{\mathrm{if}} \rho_{\mathrm{env}}(\Omega)(IVarName) \\ \rho'' = \rho'[\Omega, IVarName \mapsto \langle \beta', \{\ \} \rangle] \\ \vdash \mathbf{Explicator}(\rho_{\mathrm{env}}(\Omega)(IVarName), \beta') : \delta_x \rightarrow \delta'_x \\ NewStmnt = [\![IVarName \leftarrow NewExp]\!]\end{array}}{\mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![IVarName \leftarrow Exp]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \phi, \pi, \delta \rightarrow \rho'', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}\mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \{x\}, \mathrm{MB}, NewExp : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta' \\ \mathrm{Global}(x) \\ \tau' = \tau[LocalName \mapsto \langle \{x\}, \{lbl\} \rangle] \\ NewStmnt = [\![\mathbf{skip}]\!]\end{array}}{lbl, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![LocalName \leftarrow Exp]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}\mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau, \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}, NewExp : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta' \\ (\mathrm{Arity}(\beta) \neq 1) \vee ((\beta = \{x\}) \wedge (\mathrm{NonGlobal}(x))) \\ \tau' = \tau[LocalName \mapsto \langle \beta, \{\ \} \rangle] \\ NewStmnt = [\![LocalName \leftarrow NewExp]\!]\end{array}}{\mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![LocalName \leftarrow Exp]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}lbl, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt1]\!] \Rightarrow \mathrm{MB}_1, NewStmnt_1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ lbl, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt2]\!] \Rightarrow \mathrm{MB}_2, NewStmnt_2 : \rho', \tau', \phi', \pi', \delta' \rightarrow \rho'', \tau'', \phi'', \pi'', \delta'' \\ \mathrm{MB}' = \mathrm{MB}_1 \cup \mathrm{MB}_2 \\ NewStmnt = [\![NewStmnt_1\ NewStmnt_2]\!]\end{array}}{lbl, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt1\ Stmnt2]\!] \Rightarrow \mathrm{MB}', NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho'', \tau'', \phi'', \pi'', \delta''}$$

Figure 8.11: Semantic rule for composite statements.

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \text{Manifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ NewProcName \in \text{PROCNAME} \\ \mathcal{M} = \textbf{true} \\ \beta' = \{x\} \\ \mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{ProcCall}(x,\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \widehat{\beta'},\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \rho \vdash \textbf{FilterOutGlobals}(\widehat{\beta'},\ \widehat{NewBExp}) \Rightarrow NewBExp* \\ Stmnt = [\![NewBExp.NewProcName[NewBExp*]]\!] \end{array}}{\mathcal{D},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB},\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \text{NonManifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ <MethodName,\ ResMethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}}> = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\Omega,\ MethodName,\ ResMethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}},\ \rho_0,\ \emptyset>\} \\ NPN \in \text{PROCNAME} \\ \mathcal{M} = \textbf{false} \\ \mathcal{M},\mathcal{D},\text{MF}',\Omega \vdash \textbf{ProcCall}(x,\ ProcName,\ \widehat{\beta},\ NPN) \Rightarrow \widehat{\beta}_x,\ \text{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \text{MB}' = \cup\ \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ Stmnt = [\![NewBExp.NPN[NewBExp_1\ \ldots\ NewBExp_n]]\!] \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB}',\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \beta' \not\subset \text{Program\_Object} \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ <MethodName,\ ResMethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}}> = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\Omega,\ MethodName,\ ResMethodName,\ \rho_{\text{call}},\ \widehat{\beta}_{\text{call}},\ \rho_0,\ \emptyset>\} \\ \mathcal{D},\ \text{MF}' \vdash \textbf{HardProcCall}(x,\ ProcName,\ \widehat{\beta}) \Rightarrow \text{MB}_x : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho_x,\ \phi_x,\ \pi_x,\ \delta_x, \quad \text{for } x \in \beta' \\ \text{MB}' = \cup\ \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ Stmnt = [\![NewBExp.ProcName[NewBExp_1\ \ldots\ NewBExp_n]]\!] \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB}',\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

Figure 8.12: Semantic rules for procedure call statements.

$$\frac{\mathcal{B} = \text{Body} \qquad \mathcal{B} = [\![BB]\!] \ \ldots \qquad \mathcal{B},\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \text{LF}_0,\ \Omega \vdash [\![BB]\!] \Rightarrow \text{LB},\ \text{MB},\ lbl,\ list : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta'}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Body]\!] \Rightarrow \text{MB},\ list : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta'}$$

Figure 8.13: Semantic rules for nonstandard interpretation of basic blocks

throw away the environment parts. In Figure 7.15 the corresponding rule just returns the bottom element of the Object_Store domain.

The last two rules use the **IterBB** semantic rules because the current basic block is the entrance of a speculative loop that requires generalization of the program state. After the generalization of the program state it must be made sure that the necessary explicators are generated.

The **IterBB** semantic rules listed in Figure 8.15 are very similar to the rules in Figure 7.16. The problem with the methodnames that made aliases necessary also exists for basic blocks. In this case we have chosen to implement the alias directly by an empty basic block. This is what's done in the last line in the first rule. The forced label of the basic block marking the entrance of the speculative loop appears on the left hand side of the turnstile.

The **BB** semantic rules listed in Figure 8.16 and Figure 8.17 describe the describe abstract interpretation and code generation for basic blocks disregarding everything about detecting speculative loops, etc. The rule in Figure 8.17 describes the generation of code if the test expression in the conditional jump does not evaluate to a boolean value. The specialized expression is used as the expression part of a new conditional jump to make sure the error remains in the residual program. The two possible branches could be any label, and we have just used those closest at hand.

The **BB** semantic rule in Figure 8.18 describes the abstract interpretation and code generation for a basic block with a conditional jump where the decision of which branch to jump to must be determined at runtime. The only interesting difference between this rule and the rule in Figure 7.18 is the generation of explicators. The object stores resulting from abstract interpretation of both branches are used to possibly generate explicators for instance variables in the current and other objects. The two local environments are used to possibly generate an explicator assignment for the result variable if the basic block is in the body of a functional method. There is no need to generate explicators for the other local variables as they are anyway discarded when execution of the body of the method is finished.

The **ProcCall** semantic rules listed in Figure 8.19 are very similar to the **ProcCall** semantic rules listed in Figure 7.19 that they are derived from. A thing to note is that use of these rules result in an argument list. If the method invoked is the entrance point of a recursive speculative loop, then this argument list is the argument list of the current approximation to the program state fixpoint. Otherwise it is just the used argument list.

The first rule describes a method invocation that closes a speculative loop. Because the program state fixpoint at the entrance of the loop may be greater than the current

$$\frac{\begin{array}{c}<label, reslbl, \rho', \tau'> \in \text{LF} \\ (\rho \sqsubseteq \rho') \wedge (\tau \sqsubseteq \tau') \\ \vdash \textbf{Explicator}(\rho, \tau, \rho', \tau') : \delta \rightarrow \delta' \\ \rho'' = <\text{Name}(\rho_0), \text{Code}(\rho_0), \text{Env}(\rho_0), \text{Rescode}(\rho)>\end{array}}{\text{LF} \vdash [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}_0, \text{MB}_0, reslbl, [\,] : \rho, \tau, \delta \rightarrow \rho'', \tau_0, \delta'}$$

$$\frac{\begin{array}{c}<label, reslbl, \rho', \tau'> \in \text{LF} \\ (\rho \not\sqsubseteq \rho') \vee (\tau \not\sqsubseteq \tau') \\ \text{LB} = \{<label, \rho, \tau>\}\end{array}}{\text{LF} \vdash [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}, \text{MB}_0, reslbl, [\,] : \rho, \tau \rightarrow \rho_0, \tau_0}$$

$$\frac{\begin{array}{c}<label, \_, \_, \_> \notin \text{LF} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB}\,[\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}, \text{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ <label, \_, \_> \notin \text{LB}\end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}, \text{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi, \delta'}$$

$$\frac{\begin{array}{c}<label, \_, \_, \_> \notin \text{LF} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB}\,[\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}, \text{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ <label, \_, \_> \in \text{LB} \\ \text{LB}' = \text{LB} \setminus \{<label, \_, \_>\} \\ \text{LB}' \neq \emptyset\end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}', \text{MB}, lbl, list : \rho, \tau, \phi, \delta \rightarrow \rho', \tau', \phi', \delta'}$$

$$\frac{\begin{array}{c}BB = [\![label\ Stmnt\ Jmp]\!] \\ <label, \_, \_, \_> \notin \text{LF} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB}\,[\![BB]\!] \Rightarrow \text{LB}, \text{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ (<label, \_, \_> \in \text{LB}) \\ (\text{LB} \setminus \{<label, \_, \_>\}) = \emptyset \\ lbl2 \in \text{Label} \\ \text{LF}' = \text{LF} \cup \{<label, lbl2, \rho, \tau>\} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}' = \textbf{true} \\ \mathcal{B}, \mathcal{D}', \mathcal{C}, \text{LF}', \text{MF}, \Omega, \pi, lbl2 \vdash \textbf{IterBB}\,[\![BB]\!] \Rightarrow \text{LB}', \text{MB}', list2, \rho'', \tau'' : \rho, \tau, \phi_0, \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \\ \vdash \textbf{Explicator}(\rho, \tau, \rho'', \tau'') : \delta'' \rightarrow \delta'''\end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega, \phi, \pi \vdash [\![BB]\!] \Rightarrow \text{LB}', \text{MB}', lbl2, list2 : \rho, \tau, \delta \rightarrow \rho''', \tau''', \delta'''}$$

$$\frac{\begin{array}{c}BB = [\![label\ Stmnt\ Jmp]\!] \\ <label, \_, \_, \_> \notin \text{LF} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB}\,[\![BB]\!] \Rightarrow \text{LB}, \text{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ <label, \_, \_> \in \text{LB} \\ (\text{LB} \setminus \{<label, \_, \_>\}) = \emptyset \\ lbl2 \in \text{Label} \\ \text{LF}' = \text{LF} \cup \{<label, lbl2, \rho, \tau>\} \\ \mathcal{D} = \textbf{true} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}', \text{MF}, \Omega, \pi, lbl2 \vdash \textbf{IterBB}\,[\![BB]\!] \Rightarrow \text{LB}', \text{MB}', list2, \rho'', \tau'' : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \\ \vdash \textbf{Explicator}(\rho, \tau, \rho'', \tau'') : \delta'' \rightarrow \delta'''\end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega, \pi \vdash [\![BB]\!] \Rightarrow \text{LB}', \text{MB}', lbl2, list2 : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta'''}$$

Figure 8.14: Main semantic equations for basic blocks

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jmp]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl1, list1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi, \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, \_, \_>\} \\ (\mathrm{LB}' \neq \emptyset) \vee (<label, \_, \_> \notin \mathrm{LB}) \\ list = [\![lbl\ \mathbf{skip}\ \text{'}\mathbf{goto}\ lbl1\text{'}]\!] :: list1 \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}, list, \rho, \tau : \rho, \tau, \phi, \delta \rightarrow \rho', \tau', \phi', \delta'}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jmp]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl1, list1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, \_, \_>\} \\ (\mathrm{LB}' = \emptyset) \wedge (<label, \_, \_> \in \mathrm{LB}) \\ \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_{\mathrm{back}}\}), \quad \text{for } <label, \rho_{back}, \_> \in \mathrm{LB} \\ \tau'' = \tau \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\tau_{\mathrm{back}}\}), \quad \text{for } <label, \_, \tau_{back}> \in \mathrm{LB} \\ \mathrm{LF}' = (\mathrm{LF} \setminus \{<label, lbl, \_, \_>\}) \cup \{<label, lbl, \rho'', \tau''>\} \\ \mathcal{B},\mathcal{D},\mathcal{C},\mathrm{LF}',\mathrm{MF},\Omega,\pi,lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', list2, \rho_{\mathrm{fin}}, \tau_{\mathrm{fin}} : \rho'', \tau'', \phi', \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\mathrm{LF},\mathrm{MF},\Omega,\pi,lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', list2, \rho_{\mathrm{fin}}, \tau_{\mathrm{fin}} : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta''}$$

Figure 8.15: The **IterBB** semantic equations for basic blocks.

fixpoint an implicit generalization of program state is performed and it may be necessary to generate explicators. Since the residual method name is forced upon this rule, an alias is generated. The fifth line of the rule describes how the generated residual program fragments generated in the branch closing the loop are preserved while the program state after the method invocation is the current approximation to the fixpoint for the loop exit program state.

The two last rules use the **ProcIter** semantic rules because the method invoked is the entrance point of a speculative loop that requires generalization of the program state. After the generalization of the program state it must be made sure that the necessary explicators are generated.

The **ProcIter** semantic rules listed in Figure 8.20 are very similar to the rules in Figure 7.20. In the last line of the second rule it is defined that the just generated residual program fragments are preserved while the "old" program state at the exit of the loop is preserved. The "old" and the present program state are equal as described on in Chapter 7 on page 103 so this is really unnecessary. It may however ease future experiments with the semantics.

The **HardProcCall** semantic rules listed in Figure 8.21 define the semantics of the branches of invocations of procedural methods if the target can be anything else than a program object. This forces the residual method name to be the same as the source program method name. If the target in the current branch is not a program object, then the semantics is defined by the first three rules. We have not defined the semantics yet for the case where the target in the current branch is a program object. The actions that must be taken resemble what must be done if reusing an already generated method. Also the flow of semantic objects during abstract interpretation must be altered in the same way as required for achieving proper reuse of methods. Reuse of methods will be

$$\frac{\begin{array}{c} Jmp = [\![\textbf{return}]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ list = [\ [\![lbl2\ NewStmnt\ \text{'}\textbf{return}\text{'}]\!]\ ] \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}_0,\ \text{MB},\ lbl2,\ list : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{goto}\ label1]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jmp1]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label1\ Stmnt1\ Jmp1]\!] \Rightarrow \text{LB},\ \text{MB}',\ lbl,\ list : \rho',\ \tau',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \tau'',\ \phi'',\ \pi'',\ \delta'' \\ \text{MB}'' = \text{MB} \cup \text{MB}' \\ list2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: list \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}'',\ lbl2,\ list2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \tau'',\ \phi'',\ \pi'',\ \delta''}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau',\ \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\ \text{MB}',\ NewExp : \rho',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \phi'',\ \pi_{\text{none}},\ \delta'' \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{false} \in \beta) \wedge (\mathsf{true} \in \beta) \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jmp1]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label1\ Stmnt1\ Jmp1]\!] \Rightarrow \text{LB},\text{MB}'',lbl,lst : \rho'',\ \tau',\ \phi'',\ \pi',\ \delta'' \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta''' \\ \text{MB}''' = \text{MB} \cup \text{MB}' \cup \text{MB}'' \\ lst2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: lst \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}''',\ lbl2,\ lst2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta'''}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau',\ \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\ \text{MB}',\ NewExp : \rho',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \phi'',\ \pi_{\text{none}},\ \delta' \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \in \beta) \\ \mathcal{B} = \ldots\ [\![label2\ Stmnt2\ Jmp2]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label2\ Stmnt2\ Jmp2]\!] \Rightarrow \text{LB},\text{MB}'',lbl,lst : \rho'',\ \tau',\ \phi'',\ \pi',\ \delta'' \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta''' \\ \text{MB}''' = \text{MB} \cup \text{MB}' \cup \text{MB}'' \\ lst2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: lst \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}''',\ lbl2,\ lst2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta'''}$$

Figure 8.16: Semantic equations for the static branches of the **BB** basic blocks.

$$Jmp = [\![\textbf{if } Exp \textbf{ then goto } label1 \textbf{ else goto } label2]\!]$$
$$lbl2 \in \text{Label}$$
$$lbl2, \mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB}, NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'$$
$$\mathcal{D}, \mathcal{C}, \text{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \text{MB}', NewExp : \rho', \phi', \pi', \delta' \rightarrow \rho'', \phi'', \pi_{\text{none}}, \delta''$$
$$(\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{false} \notin \beta) \wedge (\mathsf{true} \notin \beta)$$
$$list = [\,[\![lbl2\ NewStmnt\ \text{'}\textbf{if } NewExp \textbf{ then goto } lbl2 \textbf{ else goto } lbl2\text{'}]\!]\,]$$
$$\mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash \textbf{BB } [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB}_0, \text{MB}_0, lbl2, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho_0, \tau_0, \phi'', \pi', \delta''$$

Figure 8.17: Semantic equation for the static branch of an if-statement, where none of the branches contain a boolean value.

$$BB = [\![label\ Stmnt\ Jmp]\!]$$
$$Jmp = [\![\textbf{if } Exp \textbf{ then goto } label1 \textbf{ else goto } label2]\!]$$
$$lbl2 \in \text{Label}$$
$$lbl2, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB}, NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'$$
$$\mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \text{MB}', NewExp : \rho', \phi', \pi', \delta' \rightarrow \rho'', \phi'', \pi_{\text{none}}, \delta''$$
$$(\mathsf{InputDynamic} \in \beta) \vee ((\mathsf{true} \in \beta) \wedge (\mathsf{false} \in \beta))$$
$$\mathcal{C} = <MethodName, ResMethodName, \rho_{\text{call}}, \widehat{\beta}_{\text{call}}>$$
$$\text{MF}' = \text{MF} \uplus \{<\Omega, MethodName, ResMethodName, \rho_{\text{call}}, \widehat{\beta}_{\text{call}}, \rho_0, \emptyset>\}$$
$$\text{LF}' = \text{LF} \uplus \{<label, lbl2, \rho, \tau>\}$$
$$(\mathcal{B} = \ldots\ BB1\ \ldots) \wedge (BB1 = [\![label1\ Stmnt1\ Jmp1]\!])$$
$$(\mathcal{B} = \ldots\ BB2\ \ldots) \wedge (BB2 = [\![label2\ Stmnt2\ Jmp2]\!])$$
$$\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}', \text{MF}', \Omega \vdash [\![BB1]\!] \Rightarrow \text{LB}_1, \text{MB}_1, lbl_1, list_1 : \rho'', \tau', \phi'', \pi', \delta'' \rightarrow \rho_1, \tau_1, \phi_1, \pi_1, \delta_1$$
$$\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}', \text{MF}', \Omega \vdash [\![BB2]\!] \Rightarrow \text{LB}_2, \text{MB}_2, lbl_2, list_2 : \rho'', \tau', \phi'', \pi', \delta'' \rightarrow \rho_2, \tau_2, \phi_2, \pi_2, \delta_2$$
$$\text{MB}'' = \text{MB} \cup \text{MB}' \cup \text{MB}_1 \cup \text{MB}_2$$
$$\text{LB}' = \text{LB}_1 \cup \text{LB}_2$$
$$\rho''' = \rho_1 \sqcup_{\text{if}} \rho_2$$
$$\tau'' = \tau_1 \sqcup_{\text{if}} \tau_2$$
$$\phi''' = \phi_1 \sqcup \phi_2$$
$$\pi'' = \pi_1 \cup \pi_2$$
$$\delta''' = \delta_1 \sqcup \delta_2$$
$$\vdash \textbf{Explicator}(\rho_1, \tau_1, \rho_2, \tau_2) : \delta''' \rightarrow \delta^{(4)}$$
$$list = [\![lbl2\ NewStmnt\ \text{'}\textbf{if } NewExp \textbf{ then goto } lbl_1 \textbf{ else goto } lbl_2\text{'}]\!] :: list_1 :: list_2$$
$$\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB } [\![BB]\!] \Rightarrow \text{LB}', \text{MB}'', lbl2, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho''', \tau'', \phi''', \pi'', \delta^{(4)}$$

Figure 8.18: Semantic deduction rule for the dynamic branch of **BB** basic blocks.

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \mathit{ResProcName},\ \rho',\ \widehat{\beta}',\ \rho'',\ \_> \ \in\ \mathrm{MF} \\ (\rho \sqsubseteq \rho') \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta}') \\ \vdash \mathbf{Explicator}(\rho,\ \rho') : \delta \rightarrow \delta' \\ \mathrm{MB} = \{<\textbf{*use*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \\ \rho''' = <\mathrm{Name}(\rho''),\ \mathrm{Code}(\rho''),\ \mathrm{Env}(\rho''),\ \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \mathbf{Alias}(\mathit{ResProcName},\ \mathit{NewProcName}) : \rho''' \rightarrow \rho^{(4)}\end{array}}{\mathrm{MF} \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB} : \rho,\ \delta \rightarrow \rho^{(4)},\ \delta'}$$

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \mathit{ResProcName},\ \rho',\ \widehat{\beta}',\ \rho'',\ \_> \ \in\ \mathrm{MF} \\ (\rho \not\sqsubseteq \rho') \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \rho,\ \widehat{\beta}>\}\end{array}}{\mathrm{MF} \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MF} \\ <\Omega',\ \mathit{ProcName},\ \_> \ \in\ \pi \\ \mathrm{MB} = \{<\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \rho_0,\ \widehat{\emptyset}>\}\end{array}}{\mathrm{MF},\ \pi \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MF} \\ <\Omega',\ \mathit{ProcName},\ \_> \ \notin\ \pi \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \in\ \mathrm{MB}') \vee (<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \ \notin\ \mathrm{MB}) \\ \pi'' = \pi' \setminus \{<\_,\ \mathit{ProcName},\ \Omega'>\}\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB}' : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi'',\ \delta'}$$

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MF} <\Omega',\ \mathit{ProcName},\ \_> \ \notin\ \pi \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MB}') \wedge (<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \ \in\ \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}' = \textbf{true} \\ \mathcal{M},\ \mathcal{D}',\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi_0,\ \delta \rightarrow \rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta'' \rightarrow \delta'''\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \phi,\ \pi \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \delta \rightarrow \rho''',\ \delta'''}$$

$$\frac{\begin{array}{c}<\Omega',\ \mathit{ProcName},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MF} \\ <\Omega',\ \mathit{ProcName},\ \_> \ \notin\ \pi \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \ \notin\ \mathrm{MB}') \wedge (<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \ \in\ \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \textbf{true} \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi',\ \delta \rightarrow \rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta'' \rightarrow \delta'''\end{array}}{\mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \phi,\ \delta \rightarrow \rho''',\ \phi'',\ \delta'''}$$

Figure 8.19: Semantic rules for **ProcCall**.

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \in \mathrm{MB}') \vee (<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \notin \mathrm{MB})}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB}',\ \rho : \rho,\ \phi,\ \delta{\rightarrow}\rho',\ \phi',\ \delta'}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad <\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \quad \rho' \sqsubseteq \rho_{\mathrm{old}} \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \quad \rho'' = <\mathrm{Name}(\rho'),\ \mathrm{Code}(\rho'),\ \mathrm{Env}(\rho_{\mathrm{old}}),\ \mathrm{ResCode}(\rho')>}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB}',\ \rho : \rho,\ \phi,\ \delta{\rightarrow}\rho'',\ \phi',\ \delta'}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad <\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \quad \rho' \not\sqsubseteq \rho_{\mathrm{old}} \quad \mathrm{MF}' = \mathrm{MF} \setminus \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \_,\ \_,\ \_,\ \_>\} \quad \mathrm{MF}'' = \mathrm{MF}' \cup \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho,\ \widehat{\beta},\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \emptyset>\} \quad \mathcal{M},\ \mathcal{D},\ \mathrm{MF}'',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}',\ \rho'' : \rho,\ \phi',\ \delta'{\rightarrow}\rho''',\ \phi'',\ \delta''}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}',\ \rho'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta''}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}),\quad \text{for } <\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \rho_x,\ \_> \in \mathrm{MB} \quad \widehat{\beta}' = \widehat{\beta} \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\widehat{\beta}_x\}),\quad \text{for } <\textbf{*redo*},\ \mathit{ProcName},\ \Omega',\ \_,\ \widehat{\beta}_x> \in \mathrm{MB} \quad <\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \_> \in \mathrm{MF} \quad \mathrm{MF}' = \mathrm{MF} \setminus \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \_,\ \_,\ \_,\ \_>\} \quad \mathrm{MF}'' = \mathrm{MF}' \cup \{<\Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho'',\ \widehat{\beta}',\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \emptyset>\} \quad \mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}',\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}'',\ \mathrm{MB}'',\ \rho''' : \rho'',\ \phi',\ \delta'{\rightarrow}\rho^{(4)},\ \phi'',\ \delta''}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}'',\ \mathrm{MB}'',\ \rho''' : \rho,\ \phi,\ \delta{\rightarrow}\rho^{(4)},\ \phi'',\ \delta''}$$

Figure 8.20: Semantic rules for **ProcIter**.

described in Chapter 10. We will postpone defining the rule describing the semantics for this special case till Chapter 10. Specialization of most programs does not require this semantic rule to be defined.

$$\frac{}{\vdash \textbf{HardProcCall}(\mathsf{bv},\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \textbf{HardProcCall}(\mathsf{InputDynamic},\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 :}$$

$$\frac{\Omega' = \mathsf{nil}}{\vdash \textbf{HardProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \%\ \text{issue a warning}\ \% \end{array}}{\vdash \textbf{HardProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta}) \Rightarrow \mathrm{MB}_0 :}$$

Figure 8.21: Semantic rules for **HardProcCall**.

The semantic rules listed in Figure 8.22 describe abstract interpretation and code generation for individual branches of a procedure invocation. The last two correspond to the last rule in Figure 7.21. The first five lines of these rules almost identical to the five lines of the last rule in Figure 7.21. The following lines describe the generation of the residual methods and how the generated methods are stored in the "ρ" semantic object.

The remaining semantic rules in Figure 8.23 to Figure 8.28 have many similarities with the semantic rules in Figure 8.19 to Figure 8.22.

$$\frac{}{\vdash \mathbf{ProcBody}(\mathsf{bv},\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{}{\vdash \mathbf{ProcBody}(\mathsf{InputDynamic},\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \mathrm{MB}_0 :}$$

$$\frac{}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\mathsf{nil},\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(ProcName)\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_i},\ \widehat{LVarName_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(ProcName) \\ i \neq j\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ ProcName,\ \widehat{\beta}_j,\ NewProcName) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ \widehat{LVarName_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(ProcName) \\ \tau = [FormalPar_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <ProcName,\ NewProcName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \Omega' \vdash [\![PBody]\!] \Rightarrow \mathrm{MB},\ NewPBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \mathbf{true} \\ \rho \vdash \mathbf{FilterOutGlobals}(\widehat{\beta},\ \widehat{FormalPar_j}) \Rightarrow \widehat{NewPar} \\ NewProc = <\widehat{NewPar},\ \widehat{LVarName_k},\ NewPBody> \\ <name,\ MethodMap,\ IBody> = \rho'_{\mathrm{rescode}}(\Omega') \\ NewResCode = <name,\ MethodMap[NewProcName \mapsto NewProc],\ IBody> \\ \rho'' = \rho'[\Omega' \overset{\mathrm{rescode}}{\mapsto} NewResCode]\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ ProcName,\ \widehat{\beta}_j,\ NewProcName) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ \widehat{LVarName_k},\ PBody> = \rho_{\mathrm{code}}(\Omega')(ProcName) \\ \tau = [FormalPar_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <ProcName,\ NewProcName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \mathrm{MF},\ \Omega' \vdash [\![PBody]\!] \Rightarrow \mathrm{MB},\ NewPBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \mathbf{false} \\ NewProc = <\widehat{FormalPar_j},\ \widehat{LVarName_k},\ NewPBody> \\ <name,\ MethodMap,\ IBody> = \rho'_{\mathrm{rescode}}(\Omega') \\ NewResCode = <name,\ MethodMap[NewProcName \mapsto NewProc],\ IBody> \\ \rho'' = \rho'[\Omega' \overset{\mathrm{rescode}}{\mapsto} NewResCode]\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ ProcName,\ \widehat{\beta}_j,\ NewProcName) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

Figure 8.22: Semantic rules for **ProcBody**.

$$\frac{\begin{array}{c}<\Omega', \mathit{FnctName}, \mathit{ResFnctName}, \rho', \widehat{\beta}', \rho'', \beta> \in \mathrm{MF} \\ (\mathrm{Env}(\rho) = \mathrm{Env}(\rho')) \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*use*}, \mathit{FnctName}, \Omega', \_, \_>\} \\ \rho''' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho''), \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \textbf{Alias}(\mathit{ResFnctName}, \mathit{NewFnctName}) : \rho'''\rightarrow\rho^{(4)}\end{array}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}', \mathrm{MB} : \rho\rightarrow\rho^{(4)}}$$

$$\frac{\begin{array}{c}<\Omega', \mathit{FnctName}, \mathit{ResFnctName}, \rho', \widehat{\beta}', \rho'', \beta> \in \mathrm{MF} \\ (\mathrm{Env}(\rho) \neq \mathrm{Env}(\rho')) \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{FnctName}, \Omega', \rho, \widehat{\beta}>\}\end{array}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}, \mathrm{MB} : \rho\rightarrow\rho''}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{FnctName}, \_> \in \pi) \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{FnctName}, \Omega', \rho_0, \widehat{\emptyset}>\}\end{array}}{\mathrm{MF}, \pi \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \widehat{\beta}, \mathrm{MB} : \rho\rightarrow\rho_0}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{FnctName}, \_> \in \pi) \\ \mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{FnctBody}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \mathrm{MB} : \rho, \phi, \pi, \delta\rightarrow\rho', \phi', \pi', \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{FnctName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_, \_> \in \mathrm{MB}') \vee (<\_, \mathit{FnctName}, \Omega', \_, \_> \notin \mathrm{MB}) \\ \pi'' = \pi \setminus \{<\_, \mathit{FnctName}, \Omega'>\}\end{array}}{\mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}, \mathrm{MB}' : \rho, \phi, \pi, \delta\rightarrow\rho', \phi', \pi'', \delta'}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{FnctName}, \_> \in \pi) \\ \mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{FnctBody}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \mathrm{MB} : \rho, \phi, \pi, \delta\rightarrow\rho', \phi', \pi', \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{FnctName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_, \_> \notin \mathrm{MB}') \wedge (<\_, \mathit{FnctName}, \Omega', \_, \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega', \mathit{FnctName}, \mathit{NewFnctName}, \rho, \widehat{\beta}, \rho_0, \emptyset>\} \\ \mathcal{D} = \textbf{false} \\ \mathcal{D}' = \textbf{true} \\ \mathcal{D}', \mathrm{MF}', \pi \vdash \textbf{FnctIter}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta', \widehat{\beta}', \mathrm{MB}'', \rho'' : \rho, \phi_0, \delta\rightarrow\rho''', \phi'', \delta'' \\ \vdash \textbf{Explicator}(\rho, \rho'') : \delta''\rightarrow\delta'''\end{array}}{\mathcal{M}, \mathcal{D}, \mathrm{MF}, \phi, \pi \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta', \widehat{\beta}', \mathrm{MB}'' : \rho, \delta\rightarrow\rho''', \delta'''}$$

$$\frac{\begin{array}{c}(<\Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF}) \wedge (<\Omega', \mathit{FnctName}, \_> \in \pi) \\ \mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{FnctBody}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \mathrm{MB} : \rho, \phi, \pi, \delta\rightarrow\rho', \phi', \pi', \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_, \mathit{FnctName}, \Omega', \_, \_>\} \\ (<\textbf{*redo*}, \_, \_, \_, \_> \notin \mathrm{MB}') \wedge (<\_, \mathit{FnctName}, \Omega', \_, \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\Omega', \mathit{FnctName}, \mathit{NewFnctName}, \rho, \widehat{\beta}, \rho_0, \emptyset>\} \\ \mathcal{D} = \textbf{true} \\ \mathcal{M},\mathcal{D},\mathrm{MF}',\pi \vdash \textbf{FnctIter}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta', \widehat{\beta}', \mathrm{MB}'', \rho'' : \rho, \phi', \delta\rightarrow\rho''', \phi'', \delta'' \\ \vdash \textbf{Explicator}(\rho, \rho'') : \delta''\rightarrow\delta'''\end{array}}{\mathcal{M}, \mathcal{D}, \mathrm{MF}, \pi \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta', \widehat{\beta}', \mathrm{MB}'' : \rho, \phi, \delta\rightarrow\rho''', \phi'', \delta'''}$$

Figure 8.23: Semantic rules for **FnctCall**.

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \in \mathrm{MB}') \vee (<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \notin \mathrm{MB})}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \widehat{\beta},\ \mathrm{MB}',\ \rho : \rho,\ \phi,\ \delta{\rightarrow}\rho',\ \phi',\ \delta'}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad <\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \quad (\mathrm{Env}(\rho') = \mathrm{Env}(\rho_{\mathrm{old}})) \wedge (\beta \sqsubseteq \beta_{\mathrm{old}}) \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta_{\mathrm{old}},\ \widehat{\beta},\ \mathrm{MB}',\ \rho : \rho,\ \phi,\ \delta{\rightarrow}\rho',\ \phi',\ \delta'}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}) \wedge (<\textbf{*use*},\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad <\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \quad (\mathrm{Env}(\rho') \neq \mathrm{Env}(\rho_{\mathrm{old}})) \vee (\beta \not\sqsubseteq \beta_{\mathrm{old}}) \quad \mathrm{MF}' = \mathrm{MF} \setminus \{<\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \_>\} \quad \mathrm{MF}'' = \mathrm{MF}' \cup \{<\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho,\ \widehat{\beta},\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\} \quad \mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}',\ \rho'' : \rho,\ \phi',\ \delta'{\rightarrow}\rho''',\ \phi'',\ \delta''}{\mathcal{M},\mathcal{D},\mathrm{MF},\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}',\ \rho'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta''}$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \delta{\rightarrow}\rho',\ \phi',\ \delta' \quad \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \_,\ \_>\} \quad (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\textbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \_,\ \_> \in \mathrm{MB}) \quad \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}),\ \text{ for } <\textbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \rho_x,\ \_> \in \mathrm{MB} \quad \widehat{\beta}' = \widehat{\beta} \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\widehat{\beta}_x\}),\ \text{ for } <\textbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \_,\ \widehat{\beta}_x> \in \mathrm{MB} \quad <\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF} \quad \mathrm{MF}' = \mathrm{MF} \setminus \{<\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \_>\} \quad \mathrm{MF}'' = \mathrm{MF}' \cup \{<\Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho'',\ \widehat{\beta}',\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\} \quad \mathcal{M},\mathcal{D},\mathrm{MF}'' \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}',\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}'',\ \mathrm{MB}',\ \rho''' : \rho'',\ \phi',\ \delta'{\rightarrow}\rho^{(4)},\ \phi'',\ \delta''}{\mathcal{M},\mathcal{D},\mathrm{MF} \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}'',\ \mathrm{MB}',\ \rho''' : \rho,\ \phi,\ \delta{\rightarrow}\rho^{(4)},\ \phi'',\ \delta''}$$

Figure 8.24: Semantic rules for **FnctIter**.

$$\frac{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{BaseFnctCall}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta :}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{HardFnctCall}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB}_0}$$

$$\frac{\beta = \{\mathsf{InputDynamic}\}}{\mathrm{MF} \vdash \mathbf{HardFnctCall}(\mathsf{InputDynamic},\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \beta,\ \mathrm{MB}_0}$$

$$\frac{\Omega' = \mathsf{nil}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{HardFnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \emptyset,\ \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \%\ \text{issue a warning}\ \%\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{HardFnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}) \Rightarrow \emptyset,\ \mathrm{MB}_0}$$

Figure 8.25: Semantic rules for **HardFnctCall**.

$$\frac{\%\ \textit{Will never happen}\ \%}{\vdash \mathbf{FnctBody}(\mathsf{bv},\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \{\ \},\ \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\%\ \textit{Will never happen}\ \%}{\vdash \mathbf{FnctBody}(\mathsf{InputDynamic},\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \{\ \},\ \mathrm{MB}_0 :}$$

$$\frac{\%\ \textit{Will never happen}\ \%}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\mathsf{nil},\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \bot = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName})\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{\mathit{FormalPar}_i},\ \mathit{ResultName},\ \widehat{\mathit{LVarName}_k},\ \mathit{FBody}> = \rho_{\mathrm{code}}(\Omega')(\mathit{FnctName}) \\ i \neq j\end{array}}{\mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}_j,\ \mathit{NewFnctName}) \Rightarrow \mathrm{MB}_0 : \rho \rightarrow \rho_0}$$

Figure 8.26: **FnctBody** semantic rules – Part 1.

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ FBody> = \rho_{\text{code}}(\Omega')(FnctName) \\ \tau = [FormalPar_i \mapsto \beta_i],\quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <FnctName,\ NewFnctName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega' \vdash [\![FBody]\!] \Rightarrow \text{MB},\ NewFBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \textbf{true} \\ \rho \vdash \textbf{FilterOutGlobals}(\widehat{\beta}_j,\ \widehat{FormalPar_j}) \Rightarrow \widehat{NewPar} \\ \beta = \tau'(ResultName) \\ (\beta = \{x\}) \wedge \text{Global}(x) \\ ResExp = \text{MakeExp}(x) \\ lbl \in \text{ResLabel} \\ NewFnct = <\widehat{NewPar},\ ResultName,\ \widehat{LVarName_k},\ [\,[\![lbl\ \text{'}ResultName \leftarrow ResExp\text{'}\ \textbf{'return'}]\!]\,]> \\ <name,\ MethodMap,\ IBody> = \rho'_{\text{rescode}}(\Omega') \\ \rho'' = \rho'[\Omega' \overset{\text{rescode}}{\mapsto} <name,\ MethodMap[NewFnctName \mapsto NewFnct],\ IBody>] \end{array}}{\mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}_j,\ NewFnctName) \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ FBody> = \rho_{\text{code}}(\Omega')(FnctName) \\ \tau = [FormalPar_i \mapsto \beta_i],\quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <FnctName,\ NewFnctName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega' \vdash [\![FBody]\!] \Rightarrow \text{MB},\ NewFBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \textbf{true} \\ \rho \vdash \textbf{FilterOutGlobals}(\widehat{\beta}_j,\ \widehat{FormalPar_j}) \Rightarrow \widehat{NewPar} \\ \beta = \tau'(ResultName) \\ (\text{Arity}(\beta) \neq 1) \vee (\beta = \{x\} \wedge \text{NonGlobal}(x)) \\ NewFnct = <\widehat{NewPar},\ ResultName,\ \widehat{LVarName_k},\ NewFBody> \\ <name,\ MethodMap,\ IBody> = \rho'_{\text{rescode}}(\Omega') \\ \rho'' = \rho'[\Omega' \overset{\text{rescode}}{\mapsto} <name,\ MethodMap[NewFnctName \mapsto NewFnct],\ IBody>] \end{array}}{\mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}_j,\ NewFnctName) \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

Figure 8.27: **FnctBody** semantic rules – Part 2.

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ FBody> = \rho_{\text{code}}(\Omega')(FnctName) \\ \tau = [FormalPar_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <FnctName,\ NewFnctName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega' \vdash [\![FBody]\!] \Rightarrow \text{MB},\ NewFBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \textbf{false} \\ \beta = \tau'(ResultName) \\ (\beta = \{x\}) \wedge \text{Global}(x) \\ ResExp = \text{MakeExp}(\Omega'') \\ lbl \in \text{ResLabel} \\ NewFnct = <FormalPar*,\ ResultName,\ \widehat{LVarName_k},\ [\,[\![lbl\ \text{'}ResultName \leftarrow ResExp\text{'}\ \textbf{'return'}]\!]\,]> \\ <name,\ MethodMap,\ IBody> = \rho'_{\text{rescode}}(\Omega') \\ \rho'' = \rho'[\Omega' \overset{\text{rescode}}{\mapsto} <name,\ MethodMap[NewFnctName \mapsto NewFnct],\ IBody>] \end{array}}{\mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}_j,\ NewFnctName) \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ <\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ FBody> = \rho_{\text{code}}(\Omega')(FnctName) \\ \tau = [FormalPar_i \mapsto \beta_i], \quad \text{for } i \in \{1, \ldots, j\} \\ \mathcal{C} = <FnctName,\ NewFnctName,\ \rho,\ \widehat{\beta}> \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega' \vdash [\![FBody]\!] \Rightarrow \text{MB},\ NewFBody : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{M} = \textbf{false} \\ \beta = \tau'(ResultName) \\ (\text{Arity}(\beta) \neq 1) \vee (\beta = \{x\} \wedge \text{NonGlobal}(x)) \\ NewFnct = <\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ NewFBody> \\ <name,\ MethodMap,\ IBody> = \rho'_{\text{rescode}}(\Omega') \\ \rho'' = \rho'[\Omega' \overset{\text{rescode}}{\mapsto} <name,\ MethodMap[NewFnctName \mapsto NewFnct],\ IBody>] \end{array}}{\mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{FnctBody}(\Omega',\ FnctName,\ \widehat{\beta}_j,\ NewFnctName) \Rightarrow \beta,\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \phi',\ \pi',\ \delta'}$$

Figure 8.28: Semantic rules for **FnctBody** – Part 3.

This page intentionally left blank.

Chapter 9

Unrolling Loops

The abstract interpretation algorithm developed so far is really too restrictive in many cases to be used as the basis of code generation. The problem is that the algorithm is designed to generate approximations to the program state. It is not designed to unroll loops as you want specializers to do. In this chapter we describe how the abstract interpretation can be modified to investigate some loops to some extent. When the code generation rules are adapted to the abstract interpretation described in this chapter, the residual program will have methods unrolled in the cases where the algorithm investigates loops. Due to this, we will call the investigation of loops for *unrolling*.

The basic unrolling strategy is very simple. It can informally be explained as: if none of the explored execution paths in a speculative loop exit the loop, then we unfold the loop once. The loop can then be viewed as a prefix and a loop. The possible execution paths in the "new" loop is then explored. The process is repeated until at least one possible execution path exits the loop.

This strategy is to our knowledge a completely novel strategy. We do not now if the strategy leads to nontermination of the partial evaluation in too many cases. We do however believe in its usability. Practical use of the technique will have to show whether or not this is the case.

9.1 The unrolling strategy

Consider the definition in Figure 9.1 of an object with a functional method that computes Ackermann's function. Consider an invocation of this method, where the first argument is a known integer value and the second argument is an unknown (integer) value. During abstract interpretation of the method body we discover an undecidable conditional jump instruction. The recursive invocations are classified as speculative and we have to generalize the arguments using the described algorithm. The generalized arguments are symbolic values containing only the object value representing all integers. We are then able to compute that the method returns an integer value.

This is the correct result but since the code generation is based on the abstract interpretation no specialization is performed. We would like the loop to be unrolled. Looking at the two branches of the undecidable conditional jump instruction we observe that none

```
const Ackermann == object ack
   function compute[m, n] → [result]
      var newm   var newn
      newm ← m.minus[1]
      if m.equal[0] then
         result ← n.plus[1]
      elseif n.equal[0] then
         result ← self.compute[newm, 1]
      else
         newn ← n.minus[1]
         newn ← self.compute[m, newn]
         result ← self.compute[newm, newn]
      end if
   end compute
end ack
```

Figure 9.1: Object with a method to compute Ackermann's function

of them exit the loop. We could thus try unrolling the loop once. If this does not lead to exiting the loop, then we unroll the loop once more. We continue this procedure until one of the branches of an undecidable conditional jump in the loop lead to the exit of the loop.

The decision on whether or not to unroll a loop must be taken at the entrance of the loop since this is the only place in the loop where we can identify whether all the possible execution paths lead to this point.

This unrolling of loops does of course affect the termination properties of the abstract interpreter. One can just imagine the first argument to the Ackermann function being a negative integer. If such an invocation occurs during abstract interpretation, then the program does contain a "nonterminating loop regardless of input". Using the strategy described in the previous chapters the abstract interpretation would however terminate anyway. In other words, there is a trade-off between being able to unroll loops and having the abstract interpretation process terminate. We believe that our strategy for unrolling has an acceptable impact on the termination properties of the abstract interpretation and thus on the partial evaluator.

Consider an invocation of the above function with the first argument being the integer value 2. Using the just described strategy for unrolling loops we will get a residual object definition similar to the one listed in Figure 9.2.

This result cannot be achieved by performing the unrolling simply by not performing the generalization at the entrance point of the loop and removing the loop detection information from the forward propagated set of information before redoing the abstract interpretation of the body of the loop. This will behave correctly in many cases, but there are some important exceptions that will loop forever (including the unrolling of the Ackermann function). Consider the invocation of Ackermann's function as defined in Figure 9.1 with the first argument being the integer value 2. After abstract interpretation

```
const Ackermann == object ack
   function compute-2[n] → [result]
      var newn
      if n.equal[0] then
         result ← 3
      else
         newn ← n.minus[1]
         newn ← self.compute-2[newn]
         result ← self.compute-1[newn]
      end if
   end compute
   function compute-1[n] → [result]
      var newn
      if n.equal[0] then
         result ← 2
      else
         newn ← n.minus[1]
         newn ← self.compute-1[newn]
         result ← self.compute-0[newn]
      end if
   end compute
   function compute-0[n] → [result]
      var newn
      result ← n.plus[1]
   end compute
end ack
```

Figure 9.2: Object with unfolded methods to compute Ackermann's function.

of the method body for the first time we find that the loop should be unrolled once. During the following abstract interpretation of the method body we encounter applications of *compute* with the first argument being respectively the integer objects 1 and 2. If we do not somehow keep the information that the *compute* has been invoked with the first argument being the integer object 2 and detect that the invocation of *compute* with the same arguments should not be symbolically performed once more, then our algorithm would loop infinitely.

Unfolding should take place while keeping information on the program states in unfolded parts of the loop so the unfolded loop "entrances" can be used to close the loops in the residual program.

9.2 Semantics of the algorithm

The semantics of the specialization algorithm that performs unrolling of loops can be described by deduction rules as in the previous chapters. We will only describe the changes

relative to the deduction rules given in Chapter 8. The changes amount to changes in the domain of the MF, MB, LF, LB, and π semantic objects and modifications to the deduction rules taming speculative loops.

The semantic object whose domain has been changed is listed in Figure 9.3.

$$
\begin{array}{rcl}
\mathit{flag} & \in & \text{ForwardTag} = \{\textbf{*iter*}, \textbf{*unfold*}\} \\
 & & \text{MethodTag} = \{\textbf{*redo*}, \textbf{*use*}\} \\
\text{MF} & \in & \text{MethodForward} = \mathcal{P}(\text{ForwardTag} \times \text{Abstract\_Object} \times \text{METHODNAME} \times \\
 & & \quad \text{METHODNAME} \times \text{Object\_Store} \times \text{Arglist} \times \text{Object\_Store} \times \text{AbsVal}) \\
\text{MB} & \in & \text{MethodBackward} = \mathcal{P}(\textbf{*exit*} + \text{MethodTag} \times \text{METHODNAME} \\
 & & \quad \text{Abstract\_Object} \times \text{METHODNAME} \times \text{Object\_Store} \times \text{Arglist}) \\
\text{LF} & \in & \text{LabelForward} = \mathcal{P}(\text{ForwardTag} \times \text{Label} \times \text{ResLbl} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\text{LB} & \in & \text{LabelBackward} = \mathcal{P}(\textbf{*exit*} + \text{Label} \times \text{ResLabel} \times \text{Object\_Store} \times \text{LocalEnv}) \\
\pi & \in & \text{Problematic} = \\
 & & \quad \mathcal{P}(\text{Abstract\_Object} \times \text{METHODNAME} \times \text{METHODNAME} \times \text{Abstract\_Object})
\end{array}
$$

Figure 9.3: Changed Compound Semantic Objects.

The structure of the tuples in the domain of the MF and LF semantic objects is modified. A tag is added to all the tuples telling whether the tuple is used for unfolding purposes or used for controlling iterated abstract interpretation of the body of a speculative loop.

The structure of the tuples in the domain of the MB, LB, and π semantic object is also modified. Because of unfolding there may be several "entrance" points to a speculative loop during abstract interpretation. These entrance points can be positively identified by using residual labels/methodnames in addition to the source program labels/methodnames. We have therefore added an extra field to all these kinds of tuples to accommodate residual labels/methodnames.

The sets in the MethodBackward and LabelBackward domain may also include a ***exit*** flag. The flag carries the information that it is possible to exit the loop. The flag is added to the LB semantic object when symbolically executing a **return** instruction. Combination of LB semantic objects shall still be done by computing the union of the sets. For MB semantic objects the situation is somewhat more difficult.

When abstract interpretation of a program fragment (expression or statement) returns a MB semantic object the set should contain the ***exit*** flag if that program fragment *can* be executed without closing a recursive loop. Combination of MB sets of information has in Chapter 7 and 8 been performed using the union operator. This method of combining the sets cannot be used any longer when combining MB semantic objects from a sequence of operations. This is illustrated by Example 9.1.

Example 9.1 Combining MB semantic objects from a sequence
Consider the abstract interpretation of a basic block. The interpretation of the statement part of the basic block results in one instance of a MB. The jump part results in another instance of a MB. If the jump part MB includes the ***exit*** tuple and the statement part MB indicates that the statement always closes a loop, then the MB resulting from interpretation of the basic block should not contain the ***exit*** flag as execution of the basic block will always lead to a loop.

We use the operator $\sqcup_{\text{if}}$ to combine two MB semantic objects from a choice of operations. The choice being either one of the two branches in an undecidable jump or the branches of a nonmanifest invocation. We define the *if* partial order on MB semantic objects to be subset inclusion so the $\sqcup_{\text{if}}$ operator is equal to the union operator.

To combine two MB semantic objects from a sequence of operations we use the $\sqcup_{\text{seq}}$ operator. A useful *seq* partial order on MB semantic objects and the one we will use is defined below. The *seq* least upper bound of two values can be computed by first computing the union of the values. If both values do not contain the ***exit*** flag, then an eventual ***exit*** flag from the union should be removed. If a ***nonexit*** flag was used instead of the ***exit*** flag (in the complementary situations), then the $\sqcup_{\text{seq}}$ operator would be equal to the union operator.

(MethodBackward) $x \sqsubseteq y \Leftrightarrow ((x \setminus \{\textbf{*exit*}\}) \subseteq y) \wedge ((*\textbf{exit}* \in x) \vee (*\textbf{exit}* \notin y))$

We still use the name MB_0 for the empty MB semantic object. The ***exit*** flag is not a member of the MB_0 semantic object.

The **EvalFnct** semantic rules listed in Figure 9.4 replace the semantic rules listed in Figure 8.9. The only differences between the two sets of semantic rules are the number of elements in the MF tuples and the operator used to combine MB semantic objects.

The semantic rules listed in Figure 9.5 define the abstract interpretation and specialization of invocations of procedural methods. These semantic rules replace the semantic rules listed in Figure 8.12. The only differences between the two sets of semantic rules are the number of elements in the MF tuples and the operator used to combine MB semantic objects.

The semantic rules listed in Figure 9.6 describe a small part of how unrolling iterative speculative loops are performed. The first rule describes how a loop is closed by inserting a jump to an unrolled entrance point to a loop. Failing to do this may lead to nontermination of the specializer on some wellbehaved source programs.

The next two rules match the two first rules of Figure 8.14. The ***exit*** flag is added to the LB semantic semantic object in the first of these rules to prevent infinite unrolling of the inner of two nested loops. This should really be handled as it is to be done for recursive loops. To avoid extra complexity we have chosen this rather "ugly" solution for iterative loops.

The last rule matches the last 4 rules of Figure 8.14. The **DoBB** semantic rules have been introduced to keep the size of the rules down.

The semantic rules listed in Figure 9.7 describe the semantics of specialization/abstract interpretation of basic blocks that are not entrance points of speculative iterative loops. The only interesting changes relative to the rules listed in Figure 8.14 is that we must remove the ***exit*** flag from the LB semantic objects before testing if the sets are empty.

The **IterBB** semantic rules listed in Figure 9.8 defines how unrolling of speculative iterative loops is controlled. They replace the semantic rules listed in Figure 8.15. The first rule describes the criteria for unrolling. An important requirement is that there is no outer loop that still needs to be redone as this would mean that the eventual unrolling of the loop would be wasted. The unrolling of the loop is ensured by removing the iter-tuple we know must be in the forward propagated set of information and inserting an

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \text{Manifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ NewFnctName \ \in \text{FNCTNAME} \\ \mathcal{M} = \textbf{true} \\ \beta' = \{x\} \\ \mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{FnctCall}(x,\ FnctName,\ \widehat{\beta},\ NewFnctName) \Rightarrow \beta'',\ \widehat{\beta}',\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \rho \vdash \textbf{FilterOutGlobals}(\widehat{\beta}',\ \widehat{NewBExp}) \Rightarrow NewBExp^* \\ ResExp = [\![NewBExp.NewFnctName[NewBExp^*]]\!]\end{array}}{\mathcal{D},\text{MF},\tau,\Omega \vdash \textbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \text{MB} : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = \beta \setminus \{\mathsf{nil}\} \\ \text{NonManifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ < MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}} > = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\textbf{*iter*},\ \Omega,\ MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}},\ \rho_0,\ \emptyset>\} \\ NFN \ \in \text{FNCTNAME} \\ \mathcal{M} = \textbf{false} \\ \mathcal{M},\mathcal{D},\text{MF}' \vdash \textbf{FnctCall}(x,\ FnctName,\ \widehat{\beta},\ NFN) \Rightarrow \beta_x,\widehat{\beta}_x,\text{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \beta'' = \sqcup_{\text{if}}\ \{\beta_x\}, \quad \text{for } x \in \beta' \\ \text{MB}' = \sqcup_{\text{if}}\ \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ ResExp = [\![NewBExp.NFN[NewBExp_1\ \ldots\ NewBExp_n]]\!]\end{array}}{\mathcal{D},\mathcal{C},\text{MF},\tau,\Omega \vdash \textbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \text{MB}' : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash\ [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = \beta \setminus \{\mathsf{nil}\} \\ \beta' \not\subset \text{Program\_Object} \\ \rho,\ \tau,\ \Omega \vdash\ [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ < MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}} > = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\textbf{*iter*},\ \Omega,\ MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}},\ \rho_0,\ \emptyset>\} \\ \mathcal{D},\text{MF}' \vdash \textbf{HardFnctCall}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \beta_x,\ \text{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \beta'' = \sqcup_{\text{if}}\ \{\beta_x\}, \quad \text{for } x \in \beta' \\ \text{MB}' = \sqcup_{\text{if}}\ \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}}\ \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup\ \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup\ \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup\ \{\delta'_x\}, \quad \text{for } x \in \beta' \\ ResExp = [\![NewBExp.FnctName[NewBExp_1\ \ldots\ NewBExp_n]]\!]\end{array}}{\mathcal{D},\mathcal{C},\text{MF},\tau,\Omega \vdash \textbf{EvalFnct}\ [\![BExp.FnctName[BExp_1 \ldots BExp_n]]\!] \Rightarrow \beta'', ResExp, \text{MB}' : \rho,\phi,\pi,\delta \rightarrow \rho',\phi',\pi',\delta'}$$

Figure 9.4: New semantic rules for **EvalFnct** expressions (8.9).

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \text{Manifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ NewProcName \in \text{PROCNAME} \\ \mathcal{M} = \textbf{true} \\ \beta' = \{x\} \\ \mathcal{M},\ \mathcal{D},\ \text{MF} \vdash \textbf{ProcCall}(x,\ ProcName,\ \widehat{\beta},\ NewProcName) \Rightarrow \widehat{\beta}',\ \text{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta' \\ \rho \vdash \textbf{FilterOutGlobals}(\widehat{\beta}',\ \widehat{NewBExp}) \Rightarrow NewBExp* \\ Stmnt = [\![NewBExp.NewProcName[NewBExp*]]\!] \end{array}}{\mathcal{D},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB},\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \text{NonManifest}(\beta') \wedge (\beta' \subset \text{Program\_Object}) \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ <MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}}> = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\textbf{*iter*},\ \Omega,\ MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}},\ \rho_0,\ \emptyset>\} \\ NPN \in \text{PROCNAME} \\ \mathcal{M} = \textbf{false} \\ \mathcal{M},\mathcal{D},\text{MF}',\Omega \vdash \textbf{ProcCall}(x,\ ProcName,\ \widehat{\beta},\ NPN) \Rightarrow \widehat{\beta}_x,\ \text{MB}_x : \rho,\phi,\pi,\delta \rightarrow \rho_x,\phi_x,\pi_x,\delta_x, \quad \text{for } x \in \beta' \\ \text{MB}' = \sqcup_{\text{if}} \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}} \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup \{\delta'_x\}, \quad \text{for } x \in \beta' \\ Stmnt = [\![NewBExp.NPN[NewBExp_1\ \ldots\ NewBExp_n]]\!] \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB}',\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c}\rho,\ \tau,\ \Omega \vdash [\![BExp]\!] \Rightarrow \beta,\ NewBExp \\ \beta' = (\beta \setminus \{\mathsf{nil}\}) \\ \beta' \not\subset \text{Program\_Object} \\ \rho,\ \tau,\ \Omega \vdash [\![BExp_i]\!] \Rightarrow \beta_i,\ NewBExp_i, \quad \text{for } i \in \{1, \ldots, n\} \\ <MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}}> = \mathcal{C} \\ \text{MF}' = \text{MF} \uplus \{<\textbf{*iter*},\ \Omega,\ MethodName,\ ResMethodName,\ \rho^{\text{call}},\ \widehat{\beta}^{\text{call}},\ \rho_0,\ \emptyset>\} \\ \mathcal{D},\ \text{MF}' \vdash \textbf{HardProcCall}(x,\ ProcName,\ \widehat{\beta}) \Rightarrow \text{MB}_x : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho_x,\ \phi_x,\ \pi_x,\ \delta_x, \quad \text{for } x \in \beta' \\ \text{MB}' = \sqcup_{\text{if}} \{\text{MB}_x\}, \quad \text{for } x \in \beta' \\ \rho' = \sqcup_{\text{if}} \{\rho_x\}, \quad \text{for } x \in \beta' \\ \phi' = \sqcup \{\phi_x\}, \quad \text{for } x \in \beta' \\ \pi' = \cup \{\pi_x\}, \quad \text{for } x \in \beta' \\ \vdash \textbf{Explicator}(\rho_x,\ \rho') : \delta_x \rightarrow \delta'_x, \quad \text{for } x \in \beta' \\ \delta' = \sqcup \{\delta'_x\}, \quad \text{for } x \in \beta' \\ Stmnt = [\![NewBExp.ProcName[NewBExp_1\ \ldots\ NewBExp_n]]\!] \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau,\ \Omega \vdash [\![BExp.ProcName[BExp_1\ \ldots\ BExp_n]]\!] \Rightarrow \text{MB}',\ Stmnt : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}$$

Figure 9.5: New semantic rules for procedure call statements (8.12).

$$\frac{<\textbf{*unfold*}, \mathit{label}, \mathit{reslbl}, \rho', \tau> \in \mathrm{LF} \quad \mathrm{Env}(\rho') = \mathrm{Env}(\rho)}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}_0, \mathrm{MB}_0, \mathit{reslbl}, [\,] : \rho, \tau \rightarrow \rho_0, \tau_0}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \mathit{label}, \_, \rho_x, \tau> \in \mathrm{LF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \mathit{label}, \mathit{reslbl}, \rho', \tau'> \in \mathrm{LF} \\ (\rho \sqsubseteq \rho') \wedge (\tau \sqsubseteq \tau') \\ \vdash \textbf{Explicator}(\rho, \tau, \rho', \tau') : \delta \rightarrow \delta' \\ \rho'' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho_0), \mathrm{Rescode}(\rho)> \\ \mathrm{LB} = \{\textbf{*exit*}\}\end{array}}{\mathrm{LF} \vdash [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}_0, \mathit{reslbl}, [\,] : \rho, \tau, \delta \rightarrow \rho'', \tau_0, \delta'}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \mathit{label}, \_, \rho_x, \tau> \in \mathrm{LF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \mathit{label}, \mathit{reslbl}, \rho', \tau'> \in \mathrm{LF} \\ (\rho \not\sqsubseteq \rho') \vee (\tau \not\sqsubseteq \tau') \\ \mathrm{LB} = \{<\mathit{label}, \mathit{reslbl}, \rho, \tau>\}\end{array}}{\mathrm{LF} \vdash [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}_0, \mathit{reslbl}, [\,] : \rho, \tau \rightarrow \rho_0, \tau_0}$$

$$\frac{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{DoBB}\ [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, \mathit{lbl}, \mathit{list} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, \mathit{lbl}, \mathit{list} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \mathit{label}, \_, \rho_x, \tau> \in \mathrm{LF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ (<\textbf{*iter*}, \mathit{label}, \_, \_, \_> \notin \mathrm{LF})\end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{DoBB}\ [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, \mathit{lbl}, \mathit{list} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

Figure 9.6: Main semantic equations for basic blocks (8.14).

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ <label, lbl, \_, \_> \notin \mathrm{LB} \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{DoBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, lbl, \_, \_>\} \\ (\mathrm{LB}' \setminus \{\mathbf{*exit*}\}) \neq \emptyset \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{DoBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ <label, lbl, \_, \_> \in \mathrm{LB} \\ (\mathrm{LB} \setminus \{<label, lbl, \_, \_>, \mathbf{*exit*}\}) = \emptyset \\ lbl2 \in \mathrm{Label} \\ \mathrm{LF}' = \mathrm{LF} \cup \{<\mathbf{*iter*}, label, lbl2, \rho, \tau>\} \\ \mathcal{D} = \mathbf{false} \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{B}, \mathcal{D}', \mathcal{C}, \mathrm{LF}', \mathrm{MF}, \Omega, \pi, lbl2 \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}', list2, \rho'', \tau'' : \rho, \tau, \phi_0, \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \\ \vdash \mathbf{Explicator}(\rho, \tau, \rho'', \tau'') : \delta'' \rightarrow \delta''' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \phi, \pi \vdash \mathbf{DoBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}', lbl2, list2 : \rho, \tau, \delta \rightarrow \rho''', \tau''', \delta'''}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl, list : \rho, \tau, \phi, \delta \rightarrow \rho', \tau', \phi', \delta' \\ <label, lbl, \_, \_> \in \mathrm{LB} \\ (\mathrm{LB} \setminus \{<label, lbl, \_, \_>, \mathbf{*exit*}\}) = \emptyset \\ lbl2 \in \mathrm{Label} \\ \mathrm{LF}' = \mathrm{LF} \cup \{<\mathbf{*iter*}, label, lbl2, \rho, \tau>\} \\ \mathcal{D} = \mathbf{true} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}', \mathrm{MF}, \Omega, \pi, lbl2 \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}', list2, \rho'', \tau'' : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \\ \vdash \mathbf{Explicator}(\rho, \tau, \rho'', \tau'') : \delta'' \rightarrow \delta''' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \pi \vdash \mathbf{DoBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}', lbl2, list2 : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta'''}$$

Figure 9.7: Semantic equations for **DoBB**.

unfold-tuple instead before the next fixpoint iteration.

The two other rules match the two rules in Figure 8.15 with only minor modifications.

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl1, list1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, lbl, \_, \_>\} \\ (*\mathbf{exit}* \notin \mathrm{LB}) \wedge (\mathrm{LB}' = \emptyset) \\ \mathrm{LF}' = (\mathrm{LF} \setminus \{<\mathbf{*iter*}, label, lbl, \_, \_>\}) \cup \{<\mathbf{*unfold*}, label, lbl, \rho, \tau>\} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}', \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', list, \rho^{\mathrm{fin}}, \tau^{\mathrm{fin}} : \rho, \tau, \phi, \delta \rightarrow \rho'', \tau'', \phi'', \delta'' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', list, \rho^{\mathrm{fin}}, \tau^{\mathrm{fin}} : \rho, \tau, \phi, \delta \rightarrow \rho'', \tau'', \phi'', \delta''}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl1, list1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, lbl, \_, \_>\} \\ (*\mathbf{exit}* \in \mathrm{LB}) \vee (\mathrm{LB}' \neq \emptyset) \\ ((\mathrm{LB}' \setminus \{\mathbf{*exit*}\}) \neq \emptyset) \vee (<label, lbl, \_, \_> \notin \mathrm{LB}) \\ list = [\![lbl\ \mathbf{skip}\ \text{'}\mathbf{goto}\ lbl1\text{'}]\!] :: list1 \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}, list, \rho, \tau : \rho, \tau, \phi, \delta \rightarrow \rho', \tau', \phi', \delta'}$$

$$\frac{\begin{array}{c} BB = [\![label\ Stmnt\ Jump]\!] \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \mathbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}, \mathrm{MB}, lbl1, list1 : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathrm{LB}' = \mathrm{LB} \setminus \{<label, lbl, \_, \_>\} \\ (*\mathbf{exit}* \in \mathrm{LB}) \vee (\mathrm{LB}' \neq \emptyset) \\ ((\mathrm{LB}' \setminus \{\mathbf{*exit*}\}) = \emptyset) \wedge (<label, lbl, \_, \_> \in \mathrm{LB}) \\ \rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}), \quad \text{for } <label, lbl, \rho_x, \_> \in \mathrm{LB} \\ \tau'' = \tau \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\tau_x\}), \quad \text{for } <label, lbl, \_, \tau_x> \in \mathrm{LB} \\ \mathrm{LF}' = (\mathrm{LF} \setminus \{<\mathbf{*iter*}, label, lbl, \_, \_>\}) \cup \{<\mathbf{*iter*}, label, lbl, \rho'', \tau''>\} \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}', \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', lst2, \rho^{\mathrm{fin}}, \tau^{\mathrm{fin}} : \rho'', \tau'', \phi', \delta \rightarrow \rho''', \tau''', \phi'', \delta'' \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega, \pi, lbl \vdash \mathbf{IterBB}\ [\![BB]\!] \Rightarrow \mathrm{LB}'', \mathrm{MB}', lst2, \rho^{\mathrm{fin}}, \tau^{\mathrm{fin}} : \rho, \tau, \phi, \delta \rightarrow \rho''', \tau''', \phi'', \delta''}$$

Figure 9.8: The new **IterBB** semantic equations for basic blocks (8.15).

The four semantic rules listed in Figure 9.9 replaces the semantic rules listed in Figure 8.16. The only modifications are the new LB semantic object returned in the first rule and the use of the $\sqcup_{\mathrm{seq}}$ operator rather than the union operator when combining MB semantic objects.

The semantic rule listed in Figure 9.10 is textually identical to the semantic rule listed in Figure 8.17. It is only listed here for the sake of completeness of the **BB** semantic rules.

The **BB** semantic rule listed in Figure 9.11 replaces the semantic rule listed in Figure 8.18. The important difference is the way that the MB semantic objects are combined.

The **ProcCall** semantic rules listed in Figure 9.12 and the **DoProcCall** semantic rules listed in Figure 9.13 replace the semantic rules in Figure 8.19. The splitting of the

$$\frac{\begin{array}{c} Jmp = [\![\textbf{return}]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \text{list} = [\ [\![lbl2\ NewStmnt\ \text{'}\textbf{return}\text{'}]\!]\ ] \\ \text{LB} = \{\textbf{*exit*}\} \\ \text{MB}' = \text{MB} \sqcup_{\text{seq}} \{\textbf{*exit*}\} \end{array}}{\mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}',\ lbl2,\ list : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta'}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{goto}\ label1]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jmp1]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label1\ Stmnt1\ Jmp1]\!] \Rightarrow \text{LB},\ \text{MB}',\ lbl,\ lst : \rho',\ \tau',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \tau'',\ \phi'',\ \pi'',\ \delta'' \\ \text{MB}'' = \text{MB} \sqcup_{\text{seq}} \text{MB}' \\ lst2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: lst \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}'',\ lbl2,\ lst2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho'',\ \tau'',\ \phi'',\ \pi'',\ \delta''}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau',\ \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\ \text{MB}',\ NewExp : \rho',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \phi'',\ \pi_{\text{none}},\ \delta'' \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{false} \in \beta) \wedge (\mathsf{true} \in \beta) \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jmp1]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label1\ Stmnt1\ Jmp1]\!] \Rightarrow \text{LB},\text{MB}'',lbl,lst : \rho'',\tau',\phi'',\pi',\delta'' \rightarrow \rho''',\tau'',\phi''',\pi'',\delta''' \\ \text{MB}''' = \text{MB} \sqcup_{\text{seq}} \text{MB}' \sqcup_{\text{seq}} \text{MB}'' \\ lst2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: lst \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}''',\ lbl2,\ lst2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta'''}$$

$$\frac{\begin{array}{c} Jmp = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!] \\ lbl2 \in \text{Label} \\ lbl2,\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \Omega \vdash [\![Stmnt]\!] \Rightarrow \text{MB},\ NewStmnt : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \tau',\ \phi',\ \pi',\ \delta' \\ \mathcal{D},\ \mathcal{C},\ \text{MF},\ \tau',\ \Omega \vdash [\![Exp]\!] \Rightarrow \beta,\ \text{MB}',\ NewExp : \rho',\ \phi',\ \pi',\ \delta' \rightarrow \rho'',\ \phi'',\ \pi_{\text{none}},\ \delta' \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{true} \notin \beta) \wedge (\mathsf{false} \in \beta) \\ \mathcal{B} = \ldots\ [\![label2\ Stmnt2\ Jmp2]\!]\ \ldots \\ \mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash [\![label2\ Stmnt2\ Jmp2]\!] \Rightarrow \text{LB},\text{MB}'',lbl,lst : \rho'',\ \tau',\ \phi'',\ \pi',\ \delta'' \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta''' \\ \text{MB}''' = \text{MB} \sqcup_{\text{seq}} \text{MB}' \sqcup_{\text{seq}} \text{MB}'' \\ lst2 = [\![lbl2\ NewStmnt\ \text{'}\textbf{goto}\ lbl\text{'}]\!] :: lst \end{array}}{\mathcal{B},\mathcal{D},\mathcal{C},\text{LF},\text{MF},\Omega \vdash \textbf{BB}\ [\![label\ Stmnt\ Jmp]\!] \Rightarrow \text{LB},\ \text{MB}''',\ lbl2,\ lst2 : \rho,\ \tau,\ \phi,\ \pi,\ \delta \rightarrow \rho''',\ \tau'',\ \phi''',\ \pi'',\ \delta'''}$$

Figure 9.9: New semantic equations for the static branches of the **BB** basic blocks (8.16).

$$\frac{\begin{array}{c} \mathit{Jmp} = [\![\textbf{if } \mathit{Exp} \textbf{ then goto } \mathit{label1} \textbf{ else goto } \mathit{label2}]\!] \\ \mathit{lbl2} \in \text{Label} \\ \mathit{lbl2}, \mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{Stmnt}]\!] \Rightarrow \text{MB}, \mathit{NewStmnt} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathcal{D}, \mathcal{C}, \text{MF}, \tau', \Omega \vdash [\![\mathit{Exp}]\!] \Rightarrow \beta, \text{MB}', \mathit{NewExp} : \rho', \phi', \pi', \delta' \rightarrow \rho'', \phi'', \pi_{\text{none}}, \delta'' \\ (\mathsf{InputDynamic} \notin \beta) \wedge (\mathsf{false} \notin \beta) \wedge (\mathsf{true} \notin \beta) \\ \mathit{list} = [\ [\![\mathit{lbl2}\ \mathit{NewStmnt}\ \textbf{'if } \mathit{NewExp} \textbf{ then goto } \mathit{lbl2} \textbf{ else goto } \mathit{lbl2'}]\!]\ ] \end{array}}{\mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash \textbf{BB}\ [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jump}]\!] \Rightarrow \text{LB}_0, \text{MB}_0, \mathit{lbl2}, \mathit{list} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho_0, \tau_0, \phi'', \pi', \delta''}$$

Figure 9.10: New semantic equations for an if-statement, where none of the branches contain a boolean value.

$$\frac{\begin{array}{c} \mathit{BB} = [\![\mathit{label}\ \mathit{Stmnt}\ \mathit{Jmp}]\!] \\ \mathit{Jmp} = [\![\textbf{if } \mathit{Exp} \textbf{ then goto } \mathit{label1} \textbf{ else goto } \mathit{label2}]\!] \\ \mathit{lbl2} \in \text{Label} \\ \mathit{lbl2}, \mathcal{D}, \mathcal{C}, \text{MF}, \Omega \vdash [\![\mathit{Stmnt}]\!] \Rightarrow \text{MB}, \mathit{NewStmnt} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta' \\ \mathcal{D}, \mathcal{C}, \text{MF}, \tau', \Omega \vdash [\![\mathit{Exp}]\!] \Rightarrow \beta, \text{MB}', \mathit{NewExp} : \rho', \phi', \pi', \delta' \rightarrow \rho'', \phi'', \pi_{\text{none}}, \delta'' \\ (\mathsf{InputDynamic} \in \beta) \vee ((\mathsf{true} \in \beta) \wedge (\mathsf{false} \in \beta)) \\ \mathcal{C} = <\mathit{MethodName}, \mathit{ResMethodName}, \rho^{\text{call}}, \widehat{\beta}^{\text{call}}> \\ \text{MF}' = \text{MF} \uplus \{<\textbf{*iter*}, \Omega, \mathit{MethodName}, \mathit{ResMethodName}, \rho^{\text{call}}, \widehat{\beta}^{\text{call}}, \rho_0, \emptyset>\} \\ \text{LF}' = \text{LF} \uplus \{<\textbf{*iter*}, \mathit{label}, \mathit{lbl2}, \rho, \tau>\} \\ \mathcal{B} = (\ldots\ \mathit{BB1}\ \ldots) \wedge (\mathit{BB1} = [\![\mathit{label1}\ \mathit{Stmnt1}\ \mathit{Jmp1}]\!]) \\ \mathcal{B} = (\ldots\ \mathit{BB2}\ \ldots) \wedge (\mathit{BB2} = [\![\mathit{label2}\ \mathit{Stmnt2}\ \mathit{Jmp2}]\!]) \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}', \text{MF}', \Omega \vdash [\![\mathit{BB1}]\!] \Rightarrow \text{LB}_1, \text{MB}_1, \mathit{lbl}_1, \mathit{list}_1 : \rho'', \tau', \phi'', \pi', \delta'' \rightarrow \rho_1, \tau_1, \phi_1, \pi_1, \delta_1 \\ \mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}', \text{MF}', \Omega \vdash [\![\mathit{BB2}]\!] \Rightarrow \text{LB}_2, \text{MB}_2, \mathit{lbl}_2, \mathit{list}_2 : \rho'', \tau', \phi'', \pi', \delta'' \rightarrow \rho_2, \tau_2, \phi_2, \pi_2, \delta_2 \\ \text{MB}'' = \text{MB} \sqcup_{\text{seq}} \text{MB}' \sqcup_{\text{seq}} (\text{MB}_1 \sqcup_{\text{if}} \text{MB}_2) \\ \text{LB}' = \text{LB}_1 \cup \text{LB}_2 \\ \rho''' = \rho_1 \sqcup_{\text{if}} \rho_2 \\ \tau'' = \tau_1 \sqcup_{\text{if}} \tau_2 \\ \phi''' = \phi_1 \sqcup \phi_2 \\ \pi'' = \pi_1 \cup \pi_2 \\ \delta''' = \delta_1 \sqcup \delta_2 \\ \vdash \textbf{Explicator}(\rho_1, \tau_1, \rho_2, \tau_2) : \delta''' \rightarrow \delta^{(4)} \\ \mathit{list} = [\![\mathit{lbl2}\ \mathit{NewStmnt}\ \textbf{'if } \mathit{NewExp} \textbf{ then goto } \mathit{lbl}_1 \textbf{ else goto } \mathit{lbl}_2\textbf{'}]\!] :: \mathit{list}_1 :: \mathit{list}_2 \end{array}}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \text{LF}, \text{MF}, \Omega \vdash \textbf{BB}\ [\![\mathit{BB}]\!] \Rightarrow \text{LB}', \text{MB}'', \mathit{lbl2}, \mathit{list} : \rho, \tau, \phi, \pi, \delta \rightarrow \rho''', \tau'', \phi''', \pi'', \delta^{(4)}}$$

Figure 9.11: New semantic deduction rule for the dynamic branch of **BB** basic blocks (8.18).

rules into two sets of rules is similar to the splitting of the semantic rules as in Figure 9.6 and Figure 9.7.

The problem with possible infinite unrolling of nested loops is not handled by an ugly trick in this set of semantic rules. The proper countermeasures are defined as part of the **ProcIter** semantic rules.

$$\frac{\begin{array}{c}<\textbf{*unfold*}, \Omega', \mathit{ProcName}, \mathit{ResProcName}, \rho_x, \widehat{\beta}, \rho', \_> \in \mathrm{MF} \\ \mathrm{Env}(\rho_x) = \mathrm{Env}(\rho) \\ \mathrm{MB} = \{<\textbf{*use*}, \mathit{ProcName}, \Omega', \mathit{ResProcName}, \_, \_>\} \\ \rho'' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho'), \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \textbf{Alias}(\mathit{ResProcName}, \mathit{NewProcName}) : \rho'' \rightarrow \rho''' \end{array}}{\mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}', \mathrm{MB} : \rho \rightarrow \rho'''}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{ProcName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{ProcName}, \mathit{ResProcName}, \rho', \widehat{\beta}', \rho'', \_> \in \mathrm{MF} \\ (\rho \sqsubseteq \rho') \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta}') \\ \vdash \textbf{Explicator}(\rho, \rho') : \delta \rightarrow \delta' \\ \mathrm{MB} = \{<\textbf{*use*}, \mathit{ProcName}, \Omega', \mathit{ResProcName}, \_, \_>\} \\ \rho''' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho''), \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \textbf{Alias}(\mathit{ResProcName}, \mathit{NewProcName}) : \rho''' \rightarrow \rho^{(4)} \end{array}}{\mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}', \mathrm{MB}_0 : \rho, \delta \rightarrow \rho^{(4)}, \delta'}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{ProcName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{ProcName}, \mathit{ResProcName}, \rho', \widehat{\beta}', \rho'', \_> \in \mathrm{MF} \\ (\rho \not\sqsubseteq \rho') \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{ProcName}, \Omega', \mathit{ResProcName}, \rho, \widehat{\beta}>\} \end{array}}{\mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}, \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{ProcName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{ProcName}, \_, \_, \_, \_, \_> \notin \mathrm{MF} \\ <\Omega', \mathit{ProcName}, \_, \_> \in \pi \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{ProcName}, \Omega', \mathit{ResProcName}, \rho_0, \widehat{\emptyset}>\} \end{array}}{\mathrm{MF}, \pi \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}, \mathrm{MB} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{ProcName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{ProcName}, \_, \_, \_, \_, \_> \notin \mathrm{MF} \\ <\Omega', \mathit{ProcName}, \_, \_> \notin \pi \end{array}}{\mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{DoProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}', \mathrm{MB} : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta'}$$
$$\mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \widehat{\beta}', \mathrm{MB} : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta'$$

Figure 9.12: Semantic rules for New **ProcCall** (8.19).

The **ProcIter** semantic rules listed in Figure reffig:ProcIter:unroll replace the semantic rules listed in Figure 8.20. The first rule describes when to unfold a recursive speculative loop. If none of the possible execution paths reach a **exit** instruction before a loop is closed or close an outer loop (that is, all possible paths close the current loop), then the loop should be unfolded one step.

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \ \in \mathrm{MB}') \vee (<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_> \notin \mathrm{MB}) \\ \pi'' = \pi' \setminus \{<\_,\ \mathit{ProcName},\ \mathit{NewProcName},\ \Omega'>\}\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{DoProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta},\ \mathrm{MB}' : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi'',\ \delta'}$$

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin \mathrm{MB}') \wedge (<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\textbf{*iter*},\ \Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \mathbf{false} \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{M},\ \mathcal{D}',\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi_0,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta''{\rightarrow}\delta'''\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \phi,\ \pi \vdash \mathbf{DoProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \delta{\rightarrow}\rho''',\ \delta'''}$$

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{ProcBody}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \ \notin \mathrm{MB}') \wedge (<\_,\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\textbf{*iter*},\ \Omega',\ \mathit{ProcName},\ \mathit{NewProcName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \mathbf{true} \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{ProcIter}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta''{\rightarrow}\delta'''\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{DoProcCall}(\Omega',\ \mathit{ProcName},\ \widehat{\beta},\ \mathit{NewProcName}) \Rightarrow \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'''}$$

Figure 9.13: Semantic rules for **DoProcCall**.

Apart from this, the semantic rules are just a mixture of the semantic rules listed in Figure 8.20 and Figure 9.8.

The **FnctCall** and **DoFnctCall** semantic rules listed in Figure 9.15 and Figure reffig:DoFnctCall:unroll are a mixture of the semantic rules listed in Figure 8.23, Figure 9.12, and Figure 9.13.

The **FnctIter** semantic rules listed in Figure 9.17 are a mixture of the semantic rules listed in Figure 8.24 and Figure 9.14.

$$\frac{\begin{array}{c}\mathcal{M},\,\mathcal{D},\,\mathrm{MF}\vdash\mathbf{ProcBody}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\mathrm{MB}:\rho,\,\phi,\,\pi,\,\delta{\rightarrow}\rho',\,\phi',\,\pi',\,\delta'\\ \mathrm{MB}'=\mathrm{MB}\setminus\{<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\}\\ (*\mathbf{exit}*\;\notin\;\mathrm{MB})\wedge(\mathrm{MB}'=\emptyset)\\ \mathrm{MF}'=\mathrm{MF}\setminus\{<\mathbf{*iter*},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\_,\,\_>\}\\ \mathrm{MF}''=\mathrm{MF}'\cup\{<\mathbf{*unfold*},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\rho,\,\widehat{\beta},\,\rho_0,\,\emptyset>\}\\ \mathcal{M},\,\mathcal{D},\,\mathrm{MF}'',\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta}',\,\mathrm{MB}'',\,\rho'''':\rho,\,\phi',\,\delta{\rightarrow}\rho'',\,\phi'',\,\delta''\end{array}}{\mathcal{M},\,\mathcal{D},\,\mathrm{MF},\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta}',\,\mathrm{MB}'',\,\rho'''':\rho,\,\phi,\,\delta{\rightarrow}\rho'',\,\phi'',\,\delta''}$$

$$\frac{\begin{array}{c}\mathcal{M},\,\mathcal{D},\,\mathrm{MF}\vdash\mathbf{ProcBody}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\mathrm{MB}:\rho,\,\phi,\,\pi,\,\delta{\rightarrow}\rho',\,\phi',\,\pi',\,\delta'\\ \mathrm{MB}'=\mathrm{MB}\setminus\{<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\}\\ (*\mathbf{exit}*\;\in\;\mathrm{MB})\vee(\mathrm{MB}'\neq\emptyset)\\ (<\mathbf{*redo*},\,\_,\,\_,\,\_,\,\_,\,\_>\;\in\;\mathrm{MB}')\vee(<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\notin\mathrm{MB})\end{array}}{\mathcal{M},\,\mathcal{D},\,\mathrm{MF},\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta},\,\mathrm{MB}',\,\rho:\rho,\,\phi,\,\delta{\rightarrow}\rho',\,\phi',\,\delta'}$$

$$\frac{\begin{array}{c}\mathcal{M},\,\mathcal{D},\,\mathrm{MF}\vdash\mathbf{ProcBody}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\mathrm{MB}:\rho,\,\phi,\,\pi,\,\delta{\rightarrow}\rho',\,\phi',\,\pi',\,\delta'\\ \mathrm{MB}'=\mathrm{MB}\setminus\{<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\}\\ (*\mathbf{exit}*\;\in\;\mathrm{MB})\vee(\mathrm{MB}'\neq\emptyset)\\ <\mathbf{*redo*},\,\_,\,\_,\,\_,\,\_,\,\_>\;\notin\;\mathrm{MB}\\ <\mathbf{*use*},\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\;\in\;\mathrm{MB}\\ <\_,\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\rho_{\mathrm{old}},\,\_>\;\in\;\mathrm{MF}\\ \rho'\;\sqsubseteq\;\rho_{\mathrm{old}}\\ \rho''=<\mathrm{Name}(\rho'),\,\mathrm{Code}(\rho'),\,\mathrm{Env}(\rho_{\mathrm{old}}),\,\mathrm{Rescode}(\rho')>\end{array}}{\mathcal{M},\,\mathcal{D},\,\mathrm{MF},\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta},\,\mathrm{MB}',\,\rho:\rho,\,\phi,\,\delta{\rightarrow}\rho'',\,\phi',\,\delta'}$$

$$\frac{\begin{array}{c}\mathcal{M},\,\mathcal{D},\,\mathrm{MF}\vdash\mathbf{ProcBody}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\mathrm{MB}:\rho,\,\phi,\,\pi,\,\delta{\rightarrow}\rho',\,\phi',\,\pi',\,\delta'\\ \mathrm{MB}'=\mathrm{MB}\setminus\{<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\}\\ (*\mathbf{exit}*\;\in\;\mathrm{MB})\vee(\mathrm{MB}'\neq\emptyset)\\ <\mathbf{*redo*},\,\_,\,\_,\,\_,\,\_,\,\_>\;\notin\;\mathrm{MB}\\ <\mathbf{*use*},\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\;\in\;\mathrm{MB}\\ <\mathit{flag},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\rho_{\mathrm{old}},\,\_>\;\in\;\mathrm{MF}\\ \rho'\not\sqsubseteq\rho_{\mathrm{old}}\\ \mathrm{MF}'=\mathrm{MF}\setminus\{<\_,\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\_,\,\_>\}\\ \mathrm{MF}''=\mathrm{MF}'\cup\{<\mathit{flag},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\rho,\,\widehat{\beta},\,\rho'\sqcup_{\mathrm{loop}}\rho_{\mathrm{old}},\,\emptyset>\}\\ \mathcal{M},\,\mathcal{D},\,\mathrm{MF}'',\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta}',\,\mathrm{MB}'',\,\rho'':\rho,\,\phi',\,\delta'{\rightarrow}\rho''',\,\phi'',\,\delta''\end{array}}{\mathcal{M},\,\mathcal{D},\,\mathrm{MF},\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta}',\,\mathrm{MB}'',\,\rho'':\rho,\,\phi,\,\delta{\rightarrow}\rho''',\,\phi'',\,\delta''}$$

$$\frac{\begin{array}{c}\mathcal{M},\,\mathcal{D},\,\mathrm{MF}\vdash\mathbf{ProcBody}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\mathrm{MB}:\rho,\,\phi,\,\pi,\,\delta{\rightarrow}\rho',\,\phi',\,\pi',\,\delta'\\ \mathrm{MB}'=\mathrm{MB}\setminus\{<\_,\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\}\\ (*\mathbf{exit}*\;\in\;\mathrm{MB})\vee(\mathrm{MB}'\neq\emptyset)\\ (<\mathbf{*redo*},\,\_,\,\_,\,\_,\,\_,\,\_>\;\notin\;\mathrm{MB}')\wedge(<\mathbf{*redo*},\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\_>\in\mathrm{MB})\\ \rho''=\rho\sqcup_{\mathrm{loop}}(\bigsqcup_{\mathrm{loop}}\{\rho_x\}),\quad\text{for }<\mathbf{*redo*},\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\rho_x,\,\_>\in\mathrm{MB}\\ \widehat{\beta}'=\widehat{\beta}\sqcup_{\mathrm{loop}}(\bigsqcup_{\mathrm{loop}}\{\widehat{\beta}_x\}),\quad\text{for }<\mathbf{*redo*},\,\mathit{ProcName},\,\Omega',\,\mathit{NewProcName},\,\_,\,\widehat{\beta}_x>\in\mathrm{MB}\\ <\_,\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\rho_{\mathrm{old}},\,\_>\;\in\;\mathrm{MF}\\ \mathrm{MF}'=\mathrm{MF}\setminus\{<\mathit{flag},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\_,\,\_,\,\_,\,\_>\}\\ \mathrm{MF}''=\mathrm{MF}'\cup\{<\mathit{flag},\,\Omega',\,\mathit{ProcName},\,\mathit{NewProcName},\,\rho'',\,\widehat{\beta}',\,\rho'\sqcup_{\mathrm{loop}}\rho_{\mathrm{old}},\,\emptyset>\}\\ \mathcal{M},\mathcal{D},\mathrm{MF}'',\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta}',\,\mathit{NewProcName})\Rightarrow\widehat{\beta}'',\,\mathrm{MB}'',\,\rho''':\rho'',\,\phi',\,\delta'{\rightarrow}\rho^{(4)},\,\phi'',\,\delta''\end{array}}{\mathcal{M},\,\mathcal{D},\,\mathrm{MF},\,\pi\vdash\mathbf{ProcIter}(\Omega',\,\mathit{ProcName},\,\widehat{\beta},\,\mathit{NewProcName})\Rightarrow\widehat{\beta}'',\,\mathrm{MB}'',\,\rho''':\rho,\,\phi,\,\delta{\rightarrow}\rho^{(4)},\,\phi'',\,\delta''}$$

Figure 9.14: New semantic rules for **ProcIter** (8.20).

$$\frac{\begin{array}{c}<\textbf{*unfold*}, \Omega', \mathit{FnctName}, \mathit{ResFnctName}, \rho', \widehat{\beta}, \rho'', \beta> \in \mathrm{MF} \\ \mathrm{Env}(\rho) = \mathrm{Env}(\rho') \\ \mathrm{MB} = \{<\textbf{*use*}, \mathit{FnctName}, \Omega', \mathit{ResFnctName}, \rho, \widehat{\beta}>\} \\ \rho''' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho''), \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \textbf{Alias}(\mathit{ResFnctName}, \mathit{NewFnctName}) : \rho''' \rightarrow \rho^{(4)}\end{array}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \beta, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}', \mathrm{MB} : \rho \rightarrow \rho^{(4)}}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{FnctName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{FnctName}, \mathit{ResFnctName}, \rho', \widehat{\beta}', \rho'', \beta> \in \mathrm{MF} \\ (\mathrm{Env}(\rho) = \mathrm{Env}(\rho')) \wedge (\widehat{\beta} \sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*use*}, \mathit{FnctName}, \Omega', \mathit{ResFnctName}, \rho, \widehat{\beta}>\} \\ \rho''' = <\mathrm{Name}(\rho), \mathrm{Code}(\rho), \mathrm{Env}(\rho''), \mathrm{Rescode}(\rho)> \\ \Omega' \vdash \textbf{Alias}(\mathit{ResFnctName}, \mathit{NewFnctName}) : \rho''' \rightarrow \rho^{(4)}\end{array}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}', \mathrm{MB} : \rho \rightarrow \rho^{(4)}}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{FnctName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{FnctName}, \mathit{ResFnctName}, \rho', \widehat{\beta}', \rho'', \beta> \in \mathrm{MF} \\ (\mathrm{Env}(\rho) \neq \mathrm{Env}(\rho')) \vee (\widehat{\beta} \not\sqsubseteq \widehat{\beta}') \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{FnctName}, \Omega', \mathit{ResFnctName}, \rho, \widehat{\beta}>\}\end{array}}{\mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}, \mathrm{MB} : \rho \rightarrow \rho''}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{FnctName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF} \\ <\Omega', \mathit{FnctName}, \mathit{ResName}, \Omega''> \in \pi \\ \mathrm{MB} = \{<\textbf{*redo*}, \mathit{FnctName}, \Omega', \mathit{ResName}, \rho_0, \widehat{\emptyset}>\}\end{array}}{\mathrm{MF}, \pi \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \mathit{OBJvector}, \mathit{NewFnctName}) \Rightarrow \{\ \}, \widehat{\beta}, \mathrm{MB} : \rho \rightarrow \rho_0}$$

$$\frac{\begin{array}{c}\forall <\textbf{*unfold*}, \Omega', \mathit{FnctName}, \_, \rho_x, \widehat{\beta}, \_, \_> \in \mathrm{MF} : \mathrm{Env}(\rho_x) \neq \mathrm{Env}(\rho) \\ <\textbf{*iter*}, \Omega', \mathit{FnctName}, \_, \_, \_, \_, \_> \notin \mathrm{MF} \\ <\Omega', \mathit{FnctName}, \_, \_> \notin \pi \\ \mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{DoFnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}', \mathrm{MB} : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta'\end{array}}{\mathcal{M}, \mathcal{D}, \mathrm{MF} \vdash \textbf{FnctCall}(\Omega', \mathit{FnctName}, \widehat{\beta}, \mathit{NewFnctName}) \Rightarrow \beta, \widehat{\beta}', \mathrm{MB} : \rho, \phi, \pi, \delta \rightarrow \rho', \phi', \pi', \delta'}$$

Figure 9.15: Semantic rules for **FnctCall** (8.23).

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \ \in \mathrm{MB}') \vee (<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_> \notin \mathrm{MB}) \\ \pi'' = \pi' \setminus \{<\_,\ \mathit{ProcName},\ \mathit{NewProcName},\ \Omega'>\}\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{DoFnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \widehat{\beta},\ \mathrm{MB}' : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi'',\ \delta'}$$

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\textbf{*iter*},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \mathbf{false} \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{D}',\ \mathrm{MF}',\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi_0,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta''{\rightarrow}\delta'''\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \phi,\ \pi \vdash \mathbf{DoFnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \delta{\rightarrow}\rho''',\ \delta'''}$$

$$\frac{\begin{array}{c}\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta{\rightarrow}\rho',\ \phi',\ \pi',\ \delta' \\ \mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\} \\ (<\textbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_> \in \mathrm{MB}) \\ \mathrm{MF}' = \mathrm{MF} \cup \{<\textbf{*iter*},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\} \\ \mathcal{D} = \mathbf{true} \\ \mathcal{M},\ \mathcal{D},\ \mathrm{MF}',\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'' \\ \vdash \mathbf{Explicator}(\rho,\ \rho'') : \delta''{\rightarrow}\delta'''\end{array}}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{DoFnctCall}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'' : \rho,\ \phi,\ \delta{\rightarrow}\rho''',\ \phi'',\ \delta'''}$$

Figure 9.16: Semantic rules for **DoFnctCall**.

$$\frac{\mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho''' : \rho,\ \phi',\ \delta \rightarrow \rho'',\ \phi'',\ \delta''}{\mathcal{M},\mathcal{D},\mathrm{MF},\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho''' : \rho,\ \phi,\ \delta \rightarrow \rho'',\ \phi'',\ \delta''}$$

where

$$\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'$$
$$\mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\}$$
$$(*\mathbf{exit}* \notin \mathrm{MB}) \wedge (\mathrm{MB}' = \emptyset)$$
$$\mathrm{MF}' = \mathrm{MF} \setminus \{<\mathbf{*iter*},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \_,\ \_>\}$$
$$\mathrm{MF}'' = \mathrm{MF}' \cup \{<\mathbf{*unfold*},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho,\ \widehat{\beta},\ \rho_0,\ \emptyset>\}$$

$$\frac{\mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho''' : \rho,\ \phi',\ \delta \rightarrow \rho'',\ \phi'',\ \delta''}{\mathcal{M},\mathcal{D},\mathrm{MF},\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho''' : \rho,\ \phi,\ \delta \rightarrow \rho'',\ \phi'',\ \delta''}$$

where

$$\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'$$
$$\mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\}$$
$$(*\mathbf{exit}* \in \mathrm{MB}) \vee (\mathrm{MB}' \neq \emptyset)$$
$$(<\mathbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \in \mathrm{MB}') \vee (<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_> \notin \mathrm{MB})$$

$$\frac{\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \widehat{\beta},\ \mathrm{MB}',\ \rho : \rho,\ \phi,\ \delta \rightarrow \rho',\ \phi',\ \delta'}$$

where

$$\mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\}$$
$$(*\mathbf{exit}* \in \mathrm{MB}) \vee (\mathrm{MB}' \neq \emptyset)$$
$$<\mathbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}$$
$$<\mathbf{*use*},\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_> \in \mathrm{MB}$$
$$<\mathit{flag},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF}$$
$$(\mathrm{Env}(\rho') = \mathrm{Env}(\rho_{\mathrm{old}})) \vee (\beta \not\sqsubseteq \beta_{\mathrm{old}})$$
$$\mathrm{MF}' = \mathrm{MF} \setminus \{<\_,\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \_,\ \_>\}$$
$$\mathrm{MF}'' = \mathrm{MF}' \cup \{<\mathit{flag},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho,\ \widehat{\beta},\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\}$$

$$\frac{\mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi',\ \delta' \rightarrow \rho''',\ \phi'',\ \delta''}{\mathcal{M},\ \mathcal{D},\ \mathrm{MF},\ \pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\ \widehat{\beta}',\ \mathrm{MB}'',\ \rho'' : \rho,\ \phi,\ \delta \rightarrow \rho''',\ \phi'',\ \delta''}$$

where

$$\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'$$
$$\mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\}$$
$$(*\mathbf{exit}* \in \mathrm{MB}) \vee (\mathrm{MB}' \neq \emptyset)$$
$$<\mathbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}$$
$$<\mathbf{*use*},\ \mathit{ProcName},\ \Omega',\ \mathit{NewProcName},\ \_,\ \_> \in \mathrm{MB}$$
$$<\_,\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF}$$
$$(\mathrm{Env}(\rho') = \mathrm{Env}(\rho_{\mathrm{old}})) \wedge (\beta' \sqsubseteq \beta_{\mathrm{old}})$$

$$\frac{\mathcal{M},\mathcal{D},\mathrm{MF}'',\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta}',\ \mathit{NewFnctName}) \Rightarrow \beta',\widehat{\beta}'',\mathrm{MB}'',\rho''' : \rho'',\phi',\delta' \rightarrow \rho^{(4)},\phi'',\delta''}{\mathcal{M},\mathcal{D},\mathrm{MF},\pi \vdash \mathbf{FnctIter}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta',\widehat{\beta}'',\mathrm{MB}'',\ \rho''' : \rho,\phi,\delta \rightarrow \rho^{(4)},\phi'',\delta''}$$

where

$$\mathcal{M},\ \mathcal{D},\ \mathrm{MF} \vdash \mathbf{FnctBody}(\Omega',\ \mathit{FnctName},\ \widehat{\beta},\ \mathit{NewFnctName}) \Rightarrow \beta,\ \mathrm{MB} : \rho,\ \phi,\ \pi,\ \delta \rightarrow \rho',\ \phi',\ \pi',\ \delta'$$
$$\mathrm{MB}' = \mathrm{MB} \setminus \{<\_,\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_>\}$$
$$(*\mathbf{exit}* \in \mathrm{MB}) \vee (\mathrm{MB}' \neq \emptyset)$$
$$(<\mathbf{*redo*},\ \_,\ \_,\ \_,\ \_,\ \_> \notin \mathrm{MB}') \wedge (<\mathbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \_> \in \mathrm{MB})$$
$$\rho'' = \rho \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\rho_x\}),\quad \text{for } <\mathbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \rho_x,\ \_> \in \mathrm{MB}$$
$$\widehat{\beta}' = \widehat{\beta} \sqcup_{\mathrm{loop}} (\bigsqcup_{\mathrm{loop}} \{\widehat{\beta}_x\}),\quad \text{for } <\mathbf{*redo*},\ \mathit{FnctName},\ \Omega',\ \mathit{NewFnctName},\ \_,\ \widehat{\beta}_x> \in \mathrm{MB}$$
$$<\mathit{flag},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \rho_{\mathrm{old}},\ \beta_{\mathrm{old}}> \in \mathrm{MF}$$
$$\mathrm{MF}' = \mathrm{MF} \setminus \{<\_,\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \_,\ \_,\ \_,\ \_>\}$$
$$\mathrm{MF}'' = \mathrm{MF}' \cup \{<\mathit{flag},\ \Omega',\ \mathit{FnctName},\ \mathit{NewFnctName},\ \rho'',\ \widehat{\beta}',\ \rho' \sqcup_{\mathrm{loop}} \rho_{\mathrm{old}},\ \beta \sqcup_{\mathrm{loop}} \beta_{\mathrm{old}}>\}$$

Figure 9.17: Semantic rules for **FnctIter** (8.24).

This page intentionally left blank.

Chapter 10

Reusing Computations

The algorithm developed so far will perform the desired specialization. A lot of redundant computations is however performed. In this chapter we will try to find remedies for that. We recognize three cases where the algorithm performs redundant computations.

The first case is where a speculative loop has been identified. The **IterBB**, **ProcIter**, or **FnctIter** semantic rules are then to be used. The first thing specified by these rules is that the just finished computations are redone with the semantic object $\mathcal{D}$ being true. If $\mathcal{D}$ was true already or no objects were created during the just finished computations, then these computations will be redone exactly. Instead of repeating the computations we can use the results of the first computations instead of just throwing the results away. The modifications to the semantic rules are very simple. We will therefore not list the new semantic rules.

The second case of performing redundant computations is repeated generation of methods where the program state (or just the used part of the program state) is the same. We would like to reuse already generated residual methods. This is a technique often used in partial evaluators. Reusing methods is however complicated by the presence of side effects in the language. In Section 10.1 we describe how we reuse methods.

The third case where we perform (at least partially) redundant computations is the fixpoint iterations of nested speculative loops. Each time the algorithm starts an iteration of symbolic execution of an outer loop, the fixpoint iteration of the inner loops is redone from scratch. The program state fixpoint of inner loops is a monotone (increasing) function of the number of iterations of the outer loops. The program state fixpoint in the inner loops will therefore be reached quicker if the results of all fixpoint iterations of symbolic execution of inner loops is preserved across the iterative symbolic execution of the outer loops. In Section 10.2 we describe how to reuse fixpoint computations of inner loops.

10.1 Reusing methods

The described algorithm generates a specialized method for each symbolic execution of a method invocation. In many cases this means that the residual objects will have a number of methods performing exactly the same actions. We want to prevent the generation of

redundant methods.

Instead of comparing generated residual methods for semantic equality we want to detect possible reuse of a method *before* performing the symbolic execution of the method. We also want to reuse the computations of the effect of symbolic execution of the method. In other words we want to be able to perform the modifications to the program state without doing the symbolic execution of the method.

In order to reuse a method at least a part of the current program state and the program state used when generating the method we want to reuse must be equal. Requiring the program states to be exactly identical would be to strong a requirement as the behavior of a method often depends on only one or two variables. We must perform an analysis revealing what parts of the program states that must be identical in order to reuse a method. The analysis should also reveal what parts of the program state may be modified by the method. The analysis, which we call a *utilization analysis*, is described in Section 10.1.1. In Section 10.1 we describe how the utilization information is used to reuse methods and list the rules defining the algorithm for reuse.

The technique of reusing already generated methods resembles the technique we use when redefining methods as we sometimes have to do when generating code for dynamic invocations where the target object also may be a program object. In Section 10.1.3 we give the missing **HardProcCall** and **HardFnctCall** semantic rules describing the code generation for these invocations.

10.1.1 Generation of utilization information

When having to decide if a previously generated residual method can be reused we need to know what variables may be used during execution of the method. If we decide to reuse a method, then we want to perform the modifications to the object store that would be performed during symbolic execution method without actually performing the symbolic execution. To do this we need to know what variables may be assigned a value during the symbolic execution of the method.

These two kinds of information are not used before generation of the method we want to reuse is finished. The required information can be computed while generating the method we eventually want to reuse later. The utilization analysis can thus be performed *online*.

The utilization analysis can be performed by abstract interpretation using abstract values from a four-point domain. A variable can be either Unused, Used, Defined, or Use&Def[1]. When commencing abstract interpretation of a method body, all variables are Unused. All variables assigned a value are said to be Defined. A variable that is first assigned a value and then used are still only Defined. A variable whose value is first used and then assigned a value is Use&Def. All variables only used in expressions are Used. The utilization analysis should only reveal which instance variables are used or defined.

We will define the semantics of the utilization analysis using deduction rules as in the previous chapters. The analysis is an add-on to an abstract interpretation similar to the one described in Chapter 7 or later. We will only list the important additions to the

[1]We have used *defined* rather than *modified* to follow the terminology used in e.g. [Aho 86].

deduction rules. Combining the rules is left to the reader. In the rules presented in this section some semantic objects may seemingly pop up out of the blue but are really just "imported" from the abstract interpretation rules.

Only a single semantic object is added to implement the analysis. The domain of this semantic object is defined in Figure 10.1.

$$\begin{array}{lll} & \text{UTIL} = \{\text{Unused}, \text{Used}, \text{Defined}, \text{Use\&Def}\} \\ \nu \in & \text{Util\_Store} = \text{Program\_Object} \overset{\text{fin}}{\mapsto} (\text{IVARNAME} \overset{\text{fin}}{\mapsto} \text{UTIL}) \end{array}$$

Figure 10.1: Semantic objects — utilization analysis

To combine two ν semantic objects from a sequence of computations we use the $\oplus$ operator. This is an asymmetric operator. To combine two ν semantic objects from two different possible branches of either a conditional jump or a nonmanifest invocation we use the $\sqcup_{\text{if}}$ operator. As usual the operators are defined in terms of similar operators on environments that in turn are defined in terms of similar operators on values in the UTIL domain. The operators in values in the UTIL domain are defined by the tables in Figure 10.2. The $\sqcup_{\text{if}}$ operator is the least upper bound operator for the partial order illustrated in Figure 10.3.

$\oplus$	Used	Defined
Unused	Used	Defined
Used	Used	Use&Def
Defined	Defined	Defined
Use&Def	Use&Def	Use&Def

$\sqcup_{\text{if}}$	Unused	Used	Defined	Use&Def
Unused	Unused	Used	Defined	Use&Def
Used	Used	Used	Use&Def	Use&Def
Defined	Defined	Use&Def	Defined	Use&Def
Use&Def	Use&Def	Use&Def	Use&Def	Use&Def

Figure 10.2: The definition of the $\oplus$ and $\sqcup_{\text{if}}$ operators on Util objects

The only additions to the semantic rules for basic expressions are additions to the semantic rule describing the semantics of an *IVarName* as an expression. This addition is defined in Figure 10.4.

The deduction rules in Figure 10.5 define the generation of utilization information for expressions. Only the deduction rules for object constructor expressions and invocation of functional methods are given as the other deduction rules are trivial.

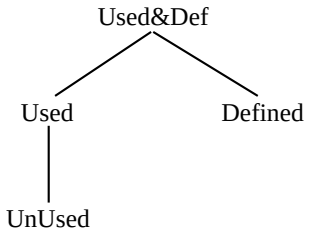

Figure 10.3: The *if* partial order on values in the UTIL domain

$$\frac{\Omega \vdash [\![Exp]\!] \Rightarrow \beta : \nu \rightarrow \nu' \qquad \nu'' = \nu'[IVarName \mapsto (\nu'(\Omega)(IVarName) \oplus \mathsf{Used})]}{\Omega \vdash [\![IVarName]\!] : \nu \rightarrow \nu''}$$

Figure 10.4: Semantic deduction rules for basic expressions.

The generation of utilization information for statements is described by the deduction rules listed in Figure 10.6. We have only given the additions to the semantic rules describing assignment to instance variables and invocations of procedural methods as the additions to the other deduction rules are trivial.

The deduction rule describing assignments to instance variables in singular objects is quite natural. If the assignment is performed in a quantified object, then the value before the assignment is both defined and used.

The deduction rule defining generation of utilization information for a method body or initialization part body is given in Figure 10.7. The semantic object ν_0 is the utilization store that maps all variable names in all objects to the abstract value Unused.

Generation of utilization information for basic blocks is trivial for all the static branches. The deduction rule in Figure 10.8 defines the additions to the deduction rule for the undecidable conditional branch. The only interesting thing is that the $\sqcup_{\text{if}}$ operator is used for combining the utilization information from the two branches.

The semantic rules listed in Figure 10.9 define how to generate the utilization infor-

$$\frac{\begin{array}{c}\Omega \vdash [\![BExp]\!] \Rightarrow \beta : \nu \rightarrow \nu' \\ \Omega \vdash [\![\widehat{BExp}]\!] \Rightarrow \widehat{\beta} : \nu' \rightarrow \nu'' \\ \vdash \mathbf{FnctBody}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \nu_x, \quad \text{for } x \in \beta \\ \nu''' = \nu'' \oplus (\sqcup_{\text{if}} \{\nu_x\}), \quad \text{for } x \in \beta\end{array}}{\Omega \vdash [\![BExp.FnctName[BExp_1 \ \ldots \ BExp_n]]\!] : \nu \rightarrow \nu'''}$$

$$\frac{\vdash [\![IBody]\!] \Rightarrow \nu'}{\Omega \vdash [\![\mathbf{object}\ ObjectName\ (IVarName*)(MethodDef*)IBody]\!] : \nu \rightarrow \nu''}$$

Figure 10.5: Semantic rules for expressions

$$\frac{\Omega = \langle \mathsf{Single}, \_ \rangle \quad \Omega \vdash [\![Exp]\!] \Rightarrow \beta : \nu \rightarrow \nu' \quad \nu'' = \nu'[\Omega,\ IVarName \mapsto (\nu'(\Omega)(IVarName) \oplus \mathsf{Defined})]}{\Omega \vdash [\![IVarName \leftarrow Exp]\!] : \nu \rightarrow \nu''}$$

$$\frac{\Omega = \langle \mathsf{Quantified}, \_ \rangle \quad \Omega \vdash [\![Exp]\!] \Rightarrow \beta : \nu \rightarrow \nu' \quad \nu'' = \nu'[\Omega,\ IVarName \mapsto (\nu'(\Omega)(IVarName) \oplus \mathsf{Use\&Def})]}{\Omega \vdash [\![IVarName \leftarrow Exp]\!] : \nu \rightarrow \nu''}$$

$$\frac{\Omega \vdash [\![BExp]\!] \Rightarrow \beta : \nu \rightarrow \nu' \quad \Omega \vdash [\![\widehat{BExp}]\!] \Rightarrow \widehat{\beta} : \nu' \rightarrow \nu'' \quad \vdash \mathbf{ProcBody}(x,\ FnctName,\ \widehat{\beta}) \Rightarrow \nu_x, \quad \text{for } x \in \beta \quad \nu''' = \nu'' \oplus (\sqcup_{\text{if}} \{\nu_x\}), \quad \text{for } x \in \beta}{\Omega \vdash [\![BExp.ProcName[BExp_1 \ \ldots \ BExp_n]]\!] : \nu \rightarrow \nu'''}$$

Figure 10.6: Semantic rules for statements.

$$\frac{\mathcal{B} = Body \quad \mathcal{B} = [\![BB]\!] \ \ldots \quad \mathcal{B}, \Omega \vdash [\![BB]\!] : \nu_0 \rightarrow \nu'}{\Omega \vdash [\![Body]\!] \Rightarrow \nu'}$$

Figure 10.7: Semantic rules for nonstandard interpretation of basic blocks

$$\frac{\begin{array}{c}\text{Jump} = [\![\mathbf{if}\ Exp\ \mathbf{then\ goto}\ label1\ \mathbf{else\ goto}\ label2]\!] \quad \Omega \vdash [\![Stmnt]\!] : \nu \rightarrow \nu' \quad \Omega \vdash [\![Exp]\!] \Rightarrow \beta : \nu' \rightarrow \nu'' \\ (\mathsf{InputDynamic} \in \beta) \vee ((\mathsf{true} \in \beta) \wedge (\mathsf{true} \in \beta)) \vee ((\mathsf{true} \notin \beta) \wedge (\mathsf{true} \notin \beta)) \\ \mathcal{B} = \ldots\ [\![label1\ Stmnt1\ Jump1]\!]\ \ldots \quad \mathcal{B} = \ldots\ [\![label2\ Stmnt2\ Jump2]\!]\ \ldots \\ \mathcal{B}, \Omega \vdash [\![label2\ Stmnt2\ Jump2]\!] : \nu'' \rightarrow \nu_1 \quad \mathcal{B}, \Omega \vdash [\![label2\ Stmnt2\ Jump2]\!] : \nu'' \rightarrow \nu_2 \\ \nu''' = \nu_1 \sqcup_{\text{if}} \nu_2\end{array}}{\mathcal{B}, \Omega \vdash [\![label\ Stmnt\ Jump]\!] : \nu \rightarrow \nu'''}$$

Figure 10.8: Generation of utilization analysis for an undecidable conditional jump

mation for the **ProcBody** and **FnctBody** semantic rules.

$$\frac{\begin{array}{c}<\widehat{FormalPar_j},\ \widehat{LVarName_k},\ PBody> = \rho_{\text{code}}(\Omega) \\ \vdash [\![PBody]\!] \Rightarrow \nu' \\ \nu'' = \nu \oplus \nu'\end{array}}{\rho \vdash \mathbf{ProcBody}(\Omega,\ ProcName,\ \widehat{\beta_i}) : \nu \rightarrow \nu''}$$

$$\frac{\begin{array}{c}<\widehat{FormalPar_j},\ ResultName,\ \widehat{LVarName_k},\ FBody> = \rho_{\text{code}}(\Omega) \\ \vdash [\![FBody]\!] \Rightarrow \nu' \\ \nu'' = \nu \oplus \nu'\end{array}}{\rho \vdash \mathbf{FnctBody}(\Omega,\ ProcName,\ \widehat{\beta_i}) : \nu \rightarrow \nu''}$$

Figure 10.9: Utilization analysis for **ProcBody** and **FnctBody**

10.1.2 How to obtain code reuse

When the specializer is about to commence abstract interpretation of the body of a method it should be determined if it is possible to reuse a previously generated residual method and to reuse the computations of the abstract interpretation. Reusing a method reduces the size of the residual program and may reduce the amount of computations needed during specialization.

Suppose the method that the specializer is about to generate a specialized version of is a procedural method with the name *asString*. The methods that may possibly be reused are the already generated specialized versions of this method. We will however only consider reusing specialized methods in the *current* object because we want to avoid problems with methods using **self** or invoking other methods in **self**. Additionally, we will not consider reusing methods that create singular objects.

A method may be reused if all variables that may be *used* in the method are identical in the current program state and the program state as it was when commencing generation of the method we now are considering to reuse. It is also required that reusing the method does not lead to inconsistencies like wrong number of parameters because we remove globally accessible parameters at one invocation site and not at another.

If there is a method that may be reused, then we may also reuse the computations of the abstract interpretation of the method. For procedural methods, all variables that may be *changed* in the method should be modified in the current program state so they have the value they had in the program state as it was immediately after finishing the generation of the reused method. For functional methods, the relevant part of the program state is only the return parameter.

To obtain reuse in situations where a method is generated in one branch of an undecidable jump or nonmanifest invocation and may be reused in another, we also have to change the flow of data in the specializer.

We observe that the information needed to decide if a method can be reused is the utilization information for the method, the $\mathcal{M}$ and $\mathcal{D}$ semantic objects, the object store and argument list when commencing the generation of the method. The parts of the

$$
\begin{array}{l}
\text{Prestate} = \text{Object\_Store} \times \text{Arglist} \times \text{Dynamic} \times \text{Manifest} \\
\text{Poststate} = \\
\text{Object\_Store} \times \text{Quantified\_Map} \times \text{Util\_Store} \times \text{AbsVal} \\
\text{ResFnct} = \\
\quad \text{FUNCTIONNAME} \times \text{FORMALPAR list} \times \text{RESULTNAME} \times \text{LVARNAME list} \times \text{BODY} \\
\text{ResProc} = \text{PROCNAME} \times \text{FORMALPAR list} \times \text{LVARNAME list} \times \text{BODY} \\
\text{MethodMap} = \\
\quad \text{PROCNAME} \xrightarrow{\text{fin}} \mathcal{P}(\text{Prestate} \times \text{ResProc} \times \text{Poststate}) + \\
\quad \text{FNCTNAME} \xrightarrow{\text{fin}} \mathcal{P}(\text{Prestate} \times \text{ResFnct} \times \text{Poststate}) \\
\text{ResCode} = \text{ObjectName} \times \text{MethodMap} \times \text{Body}
\end{array}
$$

Figure 10.10: Semantic object used to obtain code reuse.

program state after the generation is finished is needed to avoid redoing the abstract interpretation of the method.

The semantic objects that are required to reuse methods and are available before generating the residual method are the object store, the argument list, and the $\mathcal{D}$ and $\mathcal{M}$ flags. These semantic objects are combined to a *prestate* for the method. The semantic objects that is also required for reusing and are available after generation of the residual method are the object store, the utilization information, and for functional method also a return value. These semantic objects are combined to a *poststate* for the method. The prestate and the poststate are stored along with the generated residual methods to enable reusing.

The "ResCode" semantic domain is modified so it is possible to to find all the specialized versions of a method given the source program method name (cf. MethodMap). The new definition of the ResCode semantic domain is given in Figure 10.10.

The changes to the "ResCode" semantic domain must of course be followed by appropriate changes in the deduction rules manipulating semantic objects of this domain.

Detecting whether or not there exists a procedural method that can be reused can only be done at the time of symbolic execution of the method invocation. This has previously been described by the **ProcCall** semantic rules. If the **ProcCall** semantic rules are renamed to be e.g. **AProcCall** semantic rules, then reusing can be described by adding a new set of **ProcCall** semantic rules as listed in Figure 10.11.

The **ReuseProc** semantic rules are presumed to return an explicit bottom value if there is no method that can be reused. If there is a method that can be reused, then the **ReuseProc** semantic rules are presumed to all the necessary code generation, etc.

The only "positive" **ReuseProc** semantic rule is listed in Figure 10.12. The "negative" rules describing when reusing is not possible are not listed. They just describe that the explicit bottom element is returned when the listed rule cannot be used.

The flow of data should be changed so a method generated in one branch of an undecidable jump or nonmanifest invocation can be reused in another. The semantic rule in Figure 10.13 shows how the flow of data is changed in the specializer when handling an undecidable conditional jump. The residual code generated in one branch is plugged into the object store used in the other branch. Similar changes have to be made to the semantic rules specifying the semantics of nonmanifest invocations. We will however not

$$\frac{\mathcal{M}, \mathcal{D}, \rho \vdash \mathbf{ReuseProc}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \bot \quad \mathcal{M}, \mathcal{D} \vdash \mathbf{AProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \nu : \rho \rightarrow \rho'}{\mathcal{M}, \mathcal{D} \vdash \mathbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \nu : \rho \rightarrow \rho'}$$

$$\frac{\mathcal{M}, \mathcal{D} \vdash \mathbf{ReuseProc}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \nu : \rho \rightarrow \rho'}{\mathcal{M}, \mathcal{D} \vdash \mathbf{ProcCall}(\Omega', \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \nu : \rho \rightarrow \rho'}$$

Figure 10.11: Code reuse — rules for **ProcCall**

$$\frac{\begin{array}{c} <\_, \mathit{MMap}, \_> \ \in \ \rho_{\text{rescode}}(\Omega) \\ <\mathit{pre}, \mathit{ResidProc}, \mathit{post}> \ \in \ \mathit{MMap}(\mathit{ProcName}) \\ <\rho_{\text{b4}}, \widehat{\beta}_{\text{b4}}, \mathcal{D}_{\text{b4}}, \mathcal{M}_{\text{b4}}> = \mathit{pre} \\ <\rho_{\text{a}}, \nu, \_> = \mathit{post} \\ (\text{Dom}(\rho_{\text{b4}}) = \text{Dom}(\rho)) \vee ((\mathcal{D} = \mathcal{D}_{\text{b4}}) \wedge (\mathcal{D} = \mathbf{true})) \\ \mathcal{M} = \mathcal{M}_{\text{b4}} \\ \widehat{\beta} = \widehat{\beta}_{\text{b4}} \\ \rho_{\text{b4}} \stackrel{\nu}{=} \rho \\ <\mathit{ResProcName}, \_, \_, \_> = \mathit{ResidProc} \\ \Omega' \vdash \mathbf{Alias}(\mathit{ResProcName}, \mathit{NewProcName}) : \rho \rightarrow \rho' \\ \vdash \mathbf{Update}(\ \nu, \rho_{\text{a}}) : \rho' \rightarrow \rho'' \end{array}}{\mathcal{M}, \mathcal{D} \vdash \mathbf{ReuseProc}(\Omega, \mathit{ProcName}, \widehat{\beta}, \mathit{NewProcName}) \Rightarrow \nu : \rho \rightarrow \rho''}$$

Figure 10.12: Semantic rule for **ReuseProc**.

list these changed rules.

10.1.3 Generating code for dynamic invocations

We are finally in a position to describe how to generate code for dynamic invocations and how to redefine already generated methods because the choice of method name is forced. This is the case if the target of a nonmanifest invocation may be both a program object and a nonprogram object. The definition of the **HardProcCall** and **HardFnctCall** semantic rules describing this has been postponed till now.

The problems occur if we have generated a specialized residual method using one program state and subsequently want to generate another specialized residual method using a program state that is not less general than the original program state. We have chosen the solution of replacing the already generated residual method by a new method generated using a program state that is the generalization of the two program states.

The reason that we could not do this before is that we did not preserve the program state at time of commencing the generation of the specialized method along with the generated method. Additionally, the flow of data in conditional jumps and nonmanifest invocations now permits redefining the same method in the same object in more than one of the possible branches.

Below we will list the necessary extensions to the rules given in Chapter 8 and modified as sketched in the previous sections of this chapter. We only have to change the semantic

$$BB = [\![label\ Stmnt\ Jmp]\!]$$
$$\mathrm{Jmp} = [\![\textbf{if}\ Exp\ \textbf{then goto}\ label1\ \textbf{else goto}\ label2]\!]$$
$$lbl2 \in \mathrm{Label}$$
$$lbl2, \mathcal{D}, \mathcal{C}, \mathrm{MF}, \Omega \vdash [\![Stmnt]\!] \Rightarrow \mathrm{MB}, NewStmnt : \rho, \tau, \phi, \pi, \delta \rightarrow \rho', \tau', \phi', \pi', \delta'$$
$$\mathcal{D}, \mathcal{C}, \mathrm{MF}, \tau', \Omega \vdash [\![Exp]\!] \Rightarrow \beta, \mathrm{MB}', NewExp : \rho', \phi', \pi', \delta' \rightarrow \rho'', \phi'', \pi_{\mathrm{none}}, \delta''$$
$$(\mathsf{InputDynamic} \in \beta) \vee ((\mathsf{true} \in \beta) \wedge (\mathsf{false} \in \beta))$$
$$\mathcal{C} = \langle MethodName, ResMethodName, \rho^{\mathrm{call}}, \widehat{\beta}^{\mathrm{call}} \rangle$$
$$\mathrm{MF}' = \mathrm{MF} \uplus \{\langle \textbf{*iter*}, \Omega, MethodName, ResMethodName, \rho^{\mathrm{call}}, \widehat{\beta}^{\mathrm{call}}, \rho_0, \emptyset \rangle\}$$
$$\mathrm{LF}' = \mathrm{LF} \uplus \{\langle \textbf{*iter*}, label, lbl2, \rho, \tau \rangle\}$$
$$\mathcal{B} = (\ldots\ BB1\ \ldots) \wedge (BB1 = [\![label1\ Stmnt1\ Jmp1]\!])$$
$$\mathcal{B} = (\ldots\ BB2\ \ldots) \wedge (BB2 = [\![label2\ Stmnt2\ Jmp2]\!])$$
$$\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}', \mathrm{MF}', \Omega \vdash [\![BB1]\!] \Rightarrow \mathrm{LB}_1, \mathrm{MB}_1, lbl_1, list_1 : \rho'', \tau', \phi'', \pi', \delta'' \rightarrow \rho_1, \tau_1, \phi_1, \pi_1, \delta_1$$
$$\rho''' = \langle \mathrm{Name}(\rho''), \mathrm{Code}(\rho''), \mathrm{Env}(\rho''), \mathrm{Rescode}(\rho_1) \rangle$$
$$\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}', \mathrm{MF}', \Omega \vdash [\![BB2]\!] \Rightarrow \mathrm{LB}_2, \mathrm{MB}_2, lbl_2, list_2 : \rho''', \tau', \phi'', \pi', \delta'' \rightarrow \rho_2, \tau_2, \phi_2, \pi_2, \delta_2$$
$$\mathrm{MB}'' = \mathrm{MB} \sqcup_{\mathrm{seq}} \mathrm{MB}' \sqcup_{\mathrm{seq}} (\mathrm{MB}_1 \sqcup_{\mathrm{if}} \mathrm{MB}_2)$$
$$\mathrm{LB}' = \mathrm{LB}_1 \cup \mathrm{LB}_2$$
$$\rho^{(4)} = \rho_1 \sqcup_{\mathrm{if}} \rho_2$$
$$\tau'' = \tau_1 \sqcup_{\mathrm{if}} \tau_2$$
$$\phi''' = \phi_1 \sqcup_{\mathrm{if}} \phi_2$$
$$\pi'' = \pi_1 \cup \pi_2$$
$$\delta''' = \delta_1 \sqcup_{\mathrm{if}} \delta_2$$
$$\vdash \textbf{Explicator}(\rho_1, \tau_1, \rho_2, \tau_2) : \delta''' \rightarrow \delta^{(4)}$$

$$\frac{list = [\![lbl2\ NewStmnt\ \text{'}\textbf{if}\ NewExp\ \textbf{then goto}\ lbl_1\ \textbf{else goto}\ lbl_2\text{'}]\!] :: list_1 :: list_2}{\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathrm{LF}, \mathrm{MF}, \Omega \vdash \textbf{BB}\ [\![BB]\!] \Rightarrow \mathrm{LB}', \mathrm{MB}'', lbl2, list : \rho, \tau, \phi, \pi, \delta \rightarrow \rho^{(4)}, \tau'', \phi''', \pi'', \delta^{(4)}}$$

Figure 10.13: New semantic deduction rule for the dynamic branch of **BB** basic blocks (8.18).

rules for **HardProcCall** (Figure 8.21) and **HardFnctCall** (Figure 8.25). We will only list the new semantic rules for **HardProcCall** because the rules for **HardFnctCall** are very similar. The semantic rules are listed in Figure 10.15.

We assume that the **ReuseProc** semantic rules work nearly as described above. The explicit bottom value, $\perp$, is returned if there has not been generated any specialized versions of the current method that can be reused *and* it is allowed to generate a specialized residual method with the suggested name. If there is a method that can be reused, then the behavior is unchanged. If it is not allowed to generate a method with the suggested name because there exists a generated residual method, then a tuple of information is returned. This information consists of the program state at the time of commencing generation of the existing method. To ensure that the same quantified objects are created (if any) in the method we are about to generate as in the method the new method should replace, we have chosen to use the same ϕ semantic object when generating the new method as was used when generating the old method. This is therefore also included in the tuple of information.

We observe that the poststate tuples must be extended to include elements of the Quantified_Map semantic domain. The definition of the new semantic domain is listed in Figure 10.14.

```
poststate = Object_Store × Util_Store × Quantified_Map
```

Figure 10.14: The required change to the poststate semantic domain

The **HardProcCall** semantic rules listed in Figure 10.15 define the semantics of dynamic invocations where the target is a program object. These rules replace the improper semantic rule listed in Figure 8.21 (the last rule). The rules are quite straightforward now when all the prerequisites are taken care of. The only interesting thing is that the a generalized program state is computed before the **AProcCall** semantic rules are used to generate and also save the generated method for future use. The **AProcCall** semantic rules automatically do this without any modifications.

As mentioned above, the **ReuseProc** semantic rules must be modified. The semantic rule listed in Figure 10.12 only needs a slight modifications. An additional semantic rule is required to define how and when to return the required tuples of information. Both the old "positive" rule and the extra rule are listed in Figure 10.16. The "negative" rules have to be modified to describe the complementary cases to both "positive" rules.

10.2 Reusing computations in nested loops

Consider two nested speculative loops. Each time the algorithm starts an iteration of abstract interpretation of the outer loop, the fixpoint iteration of the inner loop is redone from scratch. All previously performed generalization of the program state at the entrance of the loops is therefore thrown away. For methods, the computed approximation to the fixpoint of program state at the exit of the loop is also thrown away. We would like

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \mathcal{M},\, \mathcal{D},\, \rho \vdash \mathbf{ReuseProc}(\Omega',\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{ProcName}) \Rightarrow \bot : \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{D}',\, \mathrm{MF} \vdash \mathbf{AProcCall}(\Omega',\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{ProcName}) \Rightarrow \widehat{\beta}',\, \mathrm{MB},\, \nu : \rho,\, \phi,\, \pi,\, \delta{\rightarrow}\rho',\, \phi',\, \pi',\, \delta'\end{array}}{\mathcal{M},\, \mathcal{D},\, \mathrm{MF} \vdash \mathbf{HardProcCall}(\Omega',\, \mathit{ProcName},\, \widehat{\beta}) \Rightarrow \mathrm{MB},\, \nu : \rho,\, \phi,\, \pi,\, \delta{\rightarrow}\rho',\, \phi',\, \pi',\, \delta}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \mathcal{M},\, \mathcal{D} \vdash \mathbf{ReuseProc}(\Omega',\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{ProcName}) \Rightarrow \nu : \rho{\rightarrow}\rho'\end{array}}{\mathcal{M},\, \mathcal{D},\, \mathrm{MF} \vdash \mathbf{HardProcCall}(\Omega',\, \mathit{ProcName},\, \widehat{\beta}) \Rightarrow \mathrm{MB}_0,\, \nu : \rho{\rightarrow}\rho''}$$

$$\frac{\begin{array}{c}\Omega' \neq \mathsf{nil} \\ \mathcal{M},\, \mathcal{D},\, \rho \vdash \mathbf{ReuseProc}(\Omega',\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{ProcName}) \Rightarrow <\rho_{\mathrm{b4}},\, \widehat{\beta}_{\mathrm{b4}},\, \phi_{\mathrm{b4}}> : \\ \rho_{\mathrm{new}} = \rho_{\mathrm{b4}} \sqcup_{\mathrm{if}} \rho \\ \widehat{\beta}_{\mathrm{new}} = \widehat{\beta}_{\mathrm{b4}} \sqcup_{\mathrm{if}} \widehat{\beta} \\ \vdash \mathbf{Explicator}(\rho,\, \tau,\, \rho_{\mathrm{b4}},\, \tau_{\mathrm{b4}}) : \delta{\rightarrow}\delta' \\ \mathcal{D}' = \mathbf{true} \\ \mathcal{M},\, \mathcal{D}',\, \mathrm{MF} \vdash \mathbf{AProcCall}(\Omega',\, \mathit{ProcName},\, \widehat{\beta}_{\mathrm{new}},\, \mathit{ProcName}) \Rightarrow \widehat{\beta}',\, \mathrm{MB},\, \nu : \rho_{\mathrm{new}},\phi_{\mathrm{b4}},\pi,\delta'{\rightarrow}\rho',\phi',\pi',\delta''\end{array}}{\mathcal{M},\, \mathcal{D},\, \mathrm{MF},\, \phi \vdash \mathbf{HardProcCall}(\Omega',\, \mathit{ProcName},\, \widehat{\beta}) \Rightarrow \mathrm{MB},\, \nu : \rho,\, \pi,\, \delta{\rightarrow}\rho',\, \pi',\, \delta''}$$

Figure 10.15: The missing **HardProcCall** semantic rules.

$$\frac{\begin{array}{c}<\_,\, \mathit{MMap},\, \_> \;\in\; \rho_{\mathrm{rescode}}(\Omega) \\ <\mathit{pre},\, \mathit{ResidProc},\, \mathit{post}> \;\in\; \mathit{MMap}(\mathit{ProcName}) \\ <\mathit{ProcName},\, \_,\, \_,\, \_> \;\notin\; \mathit{ResidProc} \\ <\rho_{\mathrm{b4}},\, \widehat{\beta}_{\mathrm{b4}},\, \mathcal{D}_{\mathrm{b4}},\, \mathcal{M}_{\mathrm{b4}}> = \mathit{pre} \\ <\rho_{\mathrm{a}},\, \nu,\, \_> = \mathit{post} \\ (\mathrm{Dom}(\rho_{\mathrm{b4}}) = \mathrm{Dom}(\rho)) \vee ((\mathcal{D} = \mathcal{D}_{\mathrm{b4}}) \wedge (\mathcal{D} = \mathbf{true})) \\ \mathcal{M} = \mathcal{M}_{\mathrm{b4}} \\ \widehat{\beta} = \widehat{\beta}_{\mathrm{b4}} \\ (\rho_{\mathrm{b4}} \overset{\nu}{=} \rho) \\ <\mathit{ResProcName},\, \_,\, \_,\, \_> = \mathit{ResidProc} \\ \Omega' \vdash \mathbf{Alias}(\mathit{ResProcName},\, \mathit{NewProcName}) : \rho{\rightarrow}\rho' \\ \vdash \mathbf{Update}(\, \nu,\, \rho_{\mathrm{a}}) : \rho'{\rightarrow}\rho''\end{array}}{\mathcal{M},\, \mathcal{D} \vdash \mathbf{ReuseProc}(\Omega,\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{NewProcName}) \Rightarrow \nu : \rho{\rightarrow}\rho''}$$

$$\frac{\begin{array}{c}<\_,\, \mathit{MMap},\, \_> \;\in\; \rho_{\mathrm{resproc}}(\Omega) \\ <\mathit{pre},\, \mathit{ResidProc},\, \mathit{post}> \;\in\; \mathit{MMap}(\mathit{ProcName}) \\ <\mathit{ProcName},\, \_,\, \_,\, \_> \;\in\; \mathit{ResidProc} \\ <\rho_{\mathrm{b4}},\, \widehat{\beta}_{\mathrm{b4}},\, \mathcal{D}_{\mathrm{b4}},\, \mathcal{M}_{\mathrm{b4}}> = \mathit{pre} \\ <\rho_{\mathrm{a}},\, \phi,\, \nu,\, \_> = \mathit{post}\end{array}}{\mathcal{M},\, \mathcal{D},\, \rho \vdash \mathbf{ReuseProc}(\Omega,\, \mathit{ProcName},\, \widehat{\beta},\, \mathit{NewProcName}) \Rightarrow (\rho_{\mathrm{b4}}, \widehat{\beta}_{\mathrm{b4}}, \phi) :}$$

Figure 10.16: Changes to **ReuseProc** to handle Dynamic invocation.

to use this information instead of throwing it away. Instead of throwing the fixpoint approximations away we will preserve them across iterations of the outer loop. The fixpoints may be reached much faster in this way.

For iterative loops the program state fixpoint of inner loops is always a monotone (increasing) function of the number of iterations of the outer loop[2]. For recursive loops the same is true if the body of the outer loop does not contain two recursive inner loops involving the same methods. Reusing the fixpoint approximations in the inner loops can in these cases be performed without loss of accuracy.

If the outer loop does contain two recursive loops involving the same methods, then we will have two approximations to the fixpoint of these loops. As we are not able to distinguish between these loops we are left with only less than optimal solutions. One possibility is to avoid reusing the already generated approximations to the fixpoint. Other solutions are reusing some least upper bound of the approximations to the fixpoints, or just using an arbitrary of the two approximations. The last two may lead to a loss of accuracy in the abstract interpretation while the first does not save any computations.

The current approximations to the program state fixpoints are all kept in the LF and MF semantic objects during the abstract interpretation. We suggest adding new semantic objects that propagate this information *both* backward and forward in the semantic rules. The LF and MF tuples that should be propagated backwards in these semantic objects are those that describe the approximations to the program state fixpoints of the loops that are "inner" to the current loop and that we have finished abstract interpretation of in the current iteration of the current loop. In the semantic rules that control the fixpoint iteration, the backward propagated information is turned into forward propagated information in the new semantic objects. Whenever a tuple should be added to the LF and MF semantic objects, the new sets of forward propagated information is first consulted. If these sets contain a tuple with information on an approximation to the program state fixpoint from a previous iteration of the body of the current loop, then we instead add a tuple that is a generalization of the tuple we normally would have added and the tuple found in the set.

[2]This is actually only true if we do not reuse methods that creates quantified objects other than those specified by the current ϕ semantic object. The impact of this "impurity" is so little that we will ignore it.

Chapter 11

Program Generation

Generating program fragments while performing an abstract interpretation does not generate a program. The program fragments have to be collected and put together in a form that obeys the syntax rules of the target language. This is performed in a postphase to the abstract interpretation phase. During this postphase a lot of other last-minute optimizations can be made. These include *unfolding* or *inlining* methods, removal of all unused variables and methods, etc. In this chapter we describe the activities performed in the postphase.

We will not give the algorithms for all the transformations described in this chapter. Most are more or less trivial and thus of no real interest.

11.1 Collecting the Trace

All the generated program fragments are located in the object store. The object store contains information on all abstract objects created during the symbolic execution of the program. For every such abstract object there should be an object constructor in the residual program. For the globalizable abstract objects the object constructor should be the expression of a constant declaration.

The object construct expressions are easily built from the information in the object store as the method definitions and the initialization part are there and the declaration of the instance variables can be taken from the source program.

When creating static or quantified abstract objects during the symbolic execution of the program it was not possible to generate the necessary object constructor expressions. Instead pseudo-expressions were generated. These should now be replaced by real object constructor expressions.

When printing a basic block, eventual explicator assignment statements must be inserted at the end of the sequence of statements. The order of the assignment statements is irrelevant.

11.2 Garbage Collection

Sometimes the generated residual program will contain constant declarations where the constant is never used. If these are removed from the residual program there might also be methods that are never invoked. These should also be removed from the program. Both superfluous constant declarations and method definitions can be removed using a simple mark and sweep garbage collection algorithm using the method invoked from the outside as the only member of the root set.

As variable declarations are copied from the source program into the residual program the latter may contain declarations of unused variables. In the postphase of the partial evaluator we will remove such declarations. We classify a variable as unused if it never occurs in its scope. One could argue that if a variable only occurs on the left-hand-side of assignments then it is unused, but then again we could also implement all different kinds of other optimizations.

11.3 Merging Objects

Many programs need to use datastructures of varying sizes. Such datastructures may be lists, trees, mappings, sets, etc. Since the source language does not have any builtin data structures of varying length, the data structures must be implemented using a varying number of objects. When the programs are specialized, the size of the needed data structures may be fixed in the residual program. The number of objects used to implement the datastructure will then also be fixed. In most cases these objects could be merged into one object implementing the same data structure. This kind of merging objects is often required if the residual programs are to have a natural structure.

As stated in Section 5.5 we will only merge objects that are globally accessible in the residual program. The algorithm may always be improved later to include merging of other objects. We will not limit merging of objects to objects implementing a datastructure but try to merge as many objects as possible. Our only criteria for merging is that merging is possible and that merging two objects will change at least one invocation from an invocation in another object to an invocation in the current object (**self**). This makes the needed analysis simple and will yield faster residual programs[1].

There are some restrictions to what objects may be merged. If two objects each have a method with the same name, then the two objects cannot be merged. Since we perform a polyvariant specialization of methods and always generate new methodnames, then objects will only share methodnames if it was not possible at an invocation point decide which of the objects were the target of the invocation. That automatically excludes merging.

Two objects may not be merged if they are ever compared using the equality (==) operator. This can only be detected by abstract interpretation of the program. We do perform such an abstract interpretation during the specialization. We have changed the

[1] We assume that an invocation of a method in **self** is faster than an invocation of a method in another object. We know this to be true for Emerald programs compiled with the current version of the Emerald compiler.

specialization algorithm to mark as "dirty" all those global objects that have ever been compared by the equality operator. This is not the optimal degree of information but will suffice for most purposes. We will not merge two global objects if they are both dirty. If neither or only one object is dirty, then we can say for certain that they have not been compared for equality.

Objects may be initialized as part of their creation. The initialization part of the object constructor expression may depend on the existence of other objects. These dependencies define a partial order on the constant declarations: $x \sqsubseteq y$ iff x *must* appear before y in the program. The second and final restriction on what objects can be merged can be phrased using this partial order: the two global objects created by the constant declarations x and y may *not* be merged if there exists a constant declaration z so the following is true:

$$(x \sqsubseteq z \wedge z \sqsubseteq y) \vee (y \sqsubseteq z \wedge z \sqsubseteq x)$$

The merging is simply performed by generating a new object constructor expression with all the method definitions and variable declarations from both original object constructor expressions. Name clashes between instance variables must of course be resolved first. A new constant declaration must also be constructed. All occurrences of the two old constant names must be replaced by the new constant name. In methods in the new object constructor the name **self** may be used instead.

After the merging of objects the constant declarations must be sorted topologically according to the partial order on them. It is not always the case that the new constant declarations can replace one of the old constant declarations without introducing errors in the program.

11.4 Unfolding/Inlining Methods

By unfolding or inlining a method we mean replacing a method invocation by the computational contents of the method invoked. Inlining a method saves the cost of the overhead connected with a method invocation at the cost of possible code duplication. Non manifest invocations can never be inlined. We consider inlining of two different kinds of manifest invocations: invocations of methods in the current object (**self**), and invocations of methods in other objects.

Invocations of methods in the current object that are not directly recursive can always be unfolded because the set of instance variables is the same at the invocation site and in the method body. Directly recursive methods can of course not be unfolded. Recursive, but not directly recursive, methods may be unfolded until they become directly recursive.

Manifest invocations of methods in other objects can only sometimes be unfolded. If instance variables are used inside the body of the invoked method, the method cannot be unfolded. Invocations of directly recursive methods may not be unfolded either. Otherwise the invoked method can be unfolded.

We should be able to unfold a lot of methods using this technique. Especially all procedural methods with an empty body can be unfolded. Because we at the time of performing abstract interpretation of a procedural method do not know if we have to insert explicators, we cannot simply remove methods with an empty body at that time.

In the postphase where all program fragments are generated we know that we will not insert any more explicators, and methods with empty bodies can be removed.

Having only the program text of the residual program it can be hard to judge if a given invocation is manifest. Invocations where the target is **self** or a constant name can easily be seen to be manifest. It requires an abstract interpretation of the program to determine if the other invocations are manifest. We could of course mark all manifest invocations during the generation of the program but we will suffice with unfolding invocations of methods in global objects and in the current object.

Inlining in our language may be a lot simpler than in functional languages. In most cases the actual parameters can directly replace the formal parameters in the method body. Only if an instance variable is passed as parameter to the invoked method and the same instance variable is modified in the body of the method should an extra variable be introduced to preserve the call-by-value semantics. There is no need to count number of uses of formal parameters to prevent redoing computations since the actual parameters all are basic expressions.

Besides replacing formal parameters with actual parameters in the body of the method to be unfolded, the name **self** should be replaced by the constant name denoting the target object if the method invoked is in another object. The only remaining problem with inlining is name clashes introduced by the extra variables and the combination of two sets of declarations of local variables. It is well known how to solve such problems. The possibility of name clashes between labels in basic blocks is nonexisting because of the way new labels are generated.

11.5 Transition Compressing

If the lists of basic blocks are printed as they were generated, there will be a lot of lists similar to the following:

label1: **skip**
goto *label2*
label2: $a \leftarrow a.plus[1]$
goto *label3*
label3: **skip**
if $a.gt[7]$ **then goto** *label4* **else goto** *label5*

The code contains unnecessary basic blocks and jumps between basic blocks. This is neither nice to read or fast to interpret. We would like to perform *transition compression* of the basic blocks. The above sequence of basic blocks can by transition compression be converted to something like:

label1: $a \leftarrow a.plus[1]$
if $a.gt[7]$ **then goto** *label4* **else goto** *label5*

Judging from a few generated programs the program size may shrink up to 90% due to transition compressing. There is often a direct correlation between the number of basic blocks that can be eliminated and the number of computations that it was possible to specialize.

11.6 Summary

In the post phase the residual program is pieced together from the generated program fragments. During the postphase a number of optimizations can be made. We have implemented some optimizations that largely reduce program size and increase readability of the residual programs. Superfluous constant and variable declarations are removed along with superfluous method definitions. Global objects are merged to reduce the number of objects in the residual program. This improves speed and readability of programs where datastructures of possibly varying sizes are being specialized to fixed sizes. Unfolding/inlining of method invocations improves speed, size and readability of most of our test programs. Finally transition compression is used to reduce the size (and increase readability and speed) of method bodies and the initialization parts. None of these optimizations could have been left out if the residual programs should be reasonable readable.

This page intentionally left blank.

Chapter 12

The implementation

This chapter describes what we have actually implemented and how it is done. Further we briefly describe how to use the developed programs. Finally we give some examples used to test our partial evaluator.

We have implemented an interpreter and a partial evaluator for our language Simili in ML. Further a self-interpreter and some test programs have been implemented in Simili. We believe that the partial evaluator works as described in the formulas in the preceding chapters, and all tests we have performed support this belief.

In Section 12.1 we explain why we have chosen ML as implementation language, and explain some common guidelines for the implementation in ML. Further we describe the amount of code written, where it can be found, etc. In Section 12.2 we describe an implementation of a lexer and a parser in ML using the ML-version of *lex* and *yacc*. This section further describes how to represent the abstract syntax using ML *datatypes*. Section 12.3 describes the implementation of the interpreter in ML, and Section 12.4 describe the implementation of the partial evaluator. Given the abstract syntax, semantic objects and inference rules it actually turns out to be an easy task. In Section 12.5 we describe how to implement the self-interpreter in our object-oriented language using what we have called the *active syntax* technique, as described in Section 3.5. Section 12.6 describes the test programs we have used to test the partial evaluator.

12.1 Introduction

In the actual implementation we have used the ML-compiler of New Jersey, version 0.75, the Edinburgh Library written by Dave Berry and the ML-version of lex and yacc. We have also tried using the PolyML compiler but the attempt failed due to lack of documentation.

The Edinburgh Library states some conventions for writing modules and uses the make system within the library itself [Berry 91, Chapter 3 and 4]. We have adopted these conventions and added a few more.

12.1.1 ML as implementation language

We have chosen to implement our partial evaluator in ML because it has given us the possibility to implement different parts of the system separately, and combine these parts into a big system. The flexible module system and the strong type system that ensures consistency across modules is what makes this style possible and desirable.

We have used the execution time profiler delivered with the New Jersey ML compiler [Appel 88]. With this system, we found that we had to implement a new abstract data type and reimplement others from the library. These simple changes cut down execution times approximately 80%.

Unfortunately there is no usable trace or debugging facilities with the New Jersey ML compiler. We have therefore implemented a debug facility, that allows debugging of programs. This facility is used in nearly every implemented module to print out debugging information depending on the debugging level.

12.1.2 A lot of code

We have written more than 1 MB source code. The source code of the partial evaluator amounts to a little less than half of this. The implementation can be divided into five different parts. Table 12.1 shows the division into these five parts and in which section each part is described.

Module	Lines	Bytes	Section
Parser	2.442	65.525	12.2
Interpreter	7.603	257.898	12.3
Partial Evaluator	13.870	423.474	12.4
Self-interpreter	2.900	84.888	12.5
Test examples	1.939	47.273	12.6

Table 12.1: The code in the system

12.2 The extended parser

We have used the ML-version of *lex* [Appel 89] and *yacc* [Tarditi 90] to implement a parser to convert a string into the sugared or unsugared abstract syntax. After parsing the basic blocks are split as described in Section 8.1.5. The result is a syntax tree.

The abstract syntax of the sugared and unsugared program is represented by a list of datatypes found in the two ML-signatures SUGAR and SIMILI[1]. SIMILI is listed in appendix Appendix C.1.16.

[1] Simili is used as the name for the unsugared language throughout the implementation.

The SUGAR and SIMILI signatures also describe functions that, given an abstract syntax tree return a string corresponding to the syntax. This function is very important and easy to implement.

From the grammar it cannot be deduced whether a name is a constant name, an instance name, a local name, a formal parameter, or a result name. Therefore, the abstract syntax tree is analyzed to assign name types to the different names used, and to make sure that the contexts of variable names are correct (so assignment to variables only occur in the proper places). The signature `TRANS` describes a function to make this conversion. For practical reasons this module is also used to separate basic blocks, to make sure that a method invocation is always the first statement in a basic block. This is important due to the generation of explicators as described in Chapter 8.

12.3 The interpreter implemented in ML

The semantics of our language is described in Section 4.5. It is fairly easy to convert the description of semantic objects and inference rules into ML code. When reading this section refer to Figures 4.4 and 4.5 on page 40 listing semantic objects. Table 12.2 illustrates what semantic objects are implemented in which ML modules. The inference

Semantic Object	Element	Module	Comment
Program_Object	Ω	Object	Functions on builtin
Builtin_Object	`bv`	Object	objects are implemented
			in the BASEVALUE module.
OID	β		OBJECT + BASEVALUE
CODE	Code, $\mathcal{B}$	ProgramState	The programstate
LocalEnv	τ	ProgramState	implements the current
Object_Store	ρ	ProgramState	state of a program.

Table 12.2: The semantic object found in the interpreter

rules found in Section 4.5 are implemented in the `Interpreter` module.

Finally, in the `Command` module, we have implemented a command interpreter for our language, similar to the command interpreter of ML. This command interpreter is useful when testing programs and measuring runtimes. We will not go into detail on how to use it. The command interpreter have functions to parse programs, desugar them, etc.

We will now take a closer look at (1) how to implement the semantic objects and (2) how to implement the inference rules. In Figure 12.1 we show the signature of the `Object` module. This signature should be compared with the definition of the semantic object as found in Figure 4.4 and 4.5 on page 40. The type `object` is the implementation of the semantic domain, Program_Object. An object is either represented by an object created by evaluating an object constructor, `AnObject`, or a builtin object, `ABaseValue`. The functions `initOID` and `nextOID` is used to give a sequence of integers starting with 2. It is possible to test whether two objects are equal by calling the `eq` function, and also

```
signature OBJECT =
  sig
    exception Object of string;

    (* TYPES *)
    eqtype OID sharing type OID = int
    eqtype basevalue          (* Simili basevalue ! *)
    eqtype OC                 (* = string *)

    datatype BaseValue =
      Integer of int
    | Boolean of bool
    | String of string
    | Char of string
    | Nil
    | Unix
    | InStream of instream
    | OutStream of outstream

    datatype object =
      AnObject of OID * OC
    | ABaseValue of BaseValue

    (* CONSTANTS *)
    val nilOID : OID
    val True   : object
    val False  : object
    val aNil   : object

    (* OBSERVERS *)
    val eq : object -> object -> bool
    val isBaseValue      : object -> bool

    (* CREATORS *)
    val initOID : unit -> unit
    val nextOID : unit -> OID

    (* CONVERTERS *)
    val oid2object       : OID * OC -> object
    val int2oid          : int -> OID
    val basevalue2object : basevalue -> object
    val abasevalue2object: BaseValue -> object
    val object2basevalue : object list -> BaseValue list
    val string           : object -> string
    val OID2string       : OID -> string
    val OIDOC2string     : OID * OC -> string
    val BaseValue2string : BaseValue -> string
  end;
```

Figure 12.1: The signature of the Object module

to test whether an object is a builtin object by calling the function `isBaseValue`. The function `basevalue2object` is used to convert syntactic representation of builtin objects to their semantic representation. Functions to print out objects are given, i.e. `string`, `OID2string`, `OIDOC2string`, and `BaseValue2string`. Finally, a function to convert a list of objects into a list of builtin objects is provided, `object2basevalue`. This function is used in the implementation of the function invocations on builtin objects.

In Figure 4.10 on page 46 the inference rules for expressions are listed, and in Figure 12.2 the corresponding ML code is listed. We will shortly explain similarities and differences between the inference rules and the ML code.

We first observe that there are five inference rules, and that the second and third rule both handles equality expressions, where the first covers the case of equality. In the ML code we observe four different rules, one for each syntactic category of an expression. This is done by pattern matching.

The first inference rules is covered by the first rule in the ML program, the line matching `Simili.SBASICEXPR(be)`. "Simili" is the structure name of the structure containing the datatype of the abstract syntax tree representing programs. The function `evalbasicexpr` is called yielding the result. The second and third inference rule is covered by the second rule in the ML program.

The fourth inference rule is covered by the third rule in the ML program. Evaluating a function call expression, we first evaluate the basic expression to get the object to be invoked, and secondly we evaluate the arguments to the invocation. In the inference rules we then find the semantics of **FnctCall** (see Figure 8.23), but the code listed implements also this rule.

The fifth inference rule is covered by the fourth rule in the ML code. First, we make a new object by calling the function, `ProgramState.MakeNewObject`. This function returns the newly created object, a new programstate, and the first label of the initially part in the object constructor. We then execute the initially part, and return the object created together with the programstate, found after calling the function `ProgramState.ReturnFromMethodInvocation`.

This should give an impression of how we have implemented the interpreter.

12.4 The partial evaluator

In this section we will describe the most important part of the implementation, the partial evaluator itself. As the interpreter, it is written in ML. In principle we have simply converted the inference rules and semantic objects to ML code as described earlier. However, during the development we have made changes to the code several times, simply to get better results and better performance.

Performance improvements has been very important in order to get reasonable response times. Our biggest problem related to performance has been storage despite the fact that we have used a Sun Sparc-2 with 64 MB internal storage. We have had huge problems with thrashing as the system had to swap memory pages in and out all the time. It is our impression that this problem is caused by (1) the way the New Jersey ML compiler generates code, and (2) the use of storage in our algorithms.

```
fun evalexpr (Simili.SBASICEXPR(be), ps) = (evalbasicexpr (be, ps), ps)
  | evalexpr (Simili.SEQUALEXPR(be1, be2), ps) =
    let val value1 = evalbasicexp (be1, ps)
      val value2 = evalbasicexp (be2, ps)
    in
      if (Object.eq value1 value2) then
        (Object.True, ps)
      else
        (Object.False, ps)
    end
  | evalexpr (Simili.SFNCTCALLEXPR(be, name, bes), ps) =
    let val value = evalbasicexpr (be, ps)
      val values =
        List.map (fn (be) => evalbasicexpr (be, ps)) bes
    in
      (case value of
         Object.AnObject(oid) =>
           let val (newps, lbl, rn) =
             ProgramState.DoFnctCall (ps, oid, name, values)
         in
           let val bbps =
             executebb (ProgramState.lookupBB (newps, lbl), newps)
           in
             (ProgramState.lookup (bbps, rn),
              ProgramState.ReturnFromMethodInvocation (bbps, ps))
           end
         end
       | Object.ABaseValue(bv) =>
         let val result = BaseValue.DoFnctCall(bv, name, values)
         in
           (result, ps)
         end)
  | evalexpr (Simili.SOBJECTEXPR(oc, names, mdefs, body), ps) =
    (let val (obj, newps, lbl) =
       ProgramState.MakeNewObject (ps, oc, mdefs, body)
     in (* Execute initially part *)
       let val bbps =
         executebb (ProgramState.lookupBB (newps, lbl), newps)
       in
         (obj, ProgramState.ReturnFromMethodInvocation (bbps, ps))
       end
     end
```

Figure 12.2: ML code for the inference rules

We have therefore worked a lot to save heap space, by (1) writing the program so it saves heap space, and (2) changing our algorithms so we use less memory. This has resulted in reasonable response times for our system. Further, we have made improvements not documented in the inference rules and we have considered making even more. These improvements have primarily been made to get improved execution speed and getting residual programs with a more natural structure.

The partial evaluator consist of the following parts:

- A prephase
  The prephase is described in Section 12.2.
- The specializer
  The implementation of the inference rules described in the previous chapters.
- A postphase
  The postphase will not be described into further detail.

12.4.1 Semantics objects and inference rules vs. ML-code

The semantic objects described in the earlier chapters is implemented in different modules as listed in Table 12.3. These modules are then combined with the inference rules to form the partial evaluator. The inference rules are implemented in the "PE" module. The ML modules implementing the semantic functions are indicated in Table 12.4.

Figure 9.11 on page 162 shows the inference rule of a dynamic conditional, and Figure 12.3 shows the corresponding ML code. Note, that the inference rule does not include all aspects of the generated ML code, because the figure does not include utilization information.

The executing of the statement is not shown in the ML code. This is due to the way the program is written. Executing the statement is shared by all basic block rules, and therefore not included in this part of the ML code.

The first part of the ML code recognizes a conditional, `Simili.SIF`, and we then evaluate the expression of the conditional. When evaluating an expression we use a new semantic object, `eqmap`. This semantic object is very important for the postprocessing of the residual program. Every time we generate a residual equality expression, all object values that may be compared with other object values to this semantic object[2], to make sure that we, in the postphase, does not merge two objects together, that can be compared. We have not included the code fragment for the static branching.

First, we update method forward and label forward. Secondly we execute the first branch of the conditional jump, by calling the function `PeBB`.

A new object store is created (as described in Figure 10.13 on page 179) before we execute the second branch of the conditional jump. After having executed both branches we combine the information retained from both branches of the conditional.

Finally, we generate a list of basic blocks, and return all the generated information. Generating explicator information requires a short explanation. Explicators are generated

[2]The semantic objects is a set despite the name.

Semantic Object	**Element**	**Module**	**Chapter**
Builtin_Object	bv	AbsVal	
Abstract_Object	Ω	AbsVal	
OID	x	AbsVal	
AbsVal	β	AbsVal	
INPUTDYNAMIC	InputDynamic	AbsVal	6
Arglist	$\widehat{\beta}$	Arg	
CODE	Code, $\mathcal{B}$	Rho	
Dynamic	$\mathcal{D}$	Pe	7
Manifest	$\mathcal{M}$	Pe	7
LocalEnv	τ	Tau	
InstanceEnv	ι	Rho	
Object_Store	ρ	Rho	
MethodForward	MF	MethodForward	7
MethodBackward	MB	MethodBackward	7
LabelForward	LF	LabelForward	7
LabelBackward	LB	LabelBackward	7
CurrentCall	Ω	Pe	7
Quantified_Map	ϕ	Qmap	7
Problematic	π	Pi	7
ExplicatorSet	δ	Delta	8
Precondition		Rho	10
FunctionPostCondition		Rho	10
ProcedurePostCondition		Rho	10
ResidualFunction		Rho	10
ResidualProcedure		Rho	10
FunctionTuple		Rho	10
ProcedureTuple		Rho	10
Util_Store	ν	Util	10

Table 12.3: Semantic objects and their implementation.

Semantic function	**Name**	**Module**
DoBaseFnctCall	DoFnctCall	PeBaseValue
γ	fromBaseValue	AbsVal
ϵ		Rho
Manifest		
NonManifest		
ρ_{name}	getname	Rho
ρ_{code}	code	Rho
ρ_{env}	ienv	Rho
ρ_{rescode}	rescode	Rho
Explicator	Explicator	Delta
Alias	procedurealias	Rho
	functionalias	
Singular		AbsVal
NonSingular		AbsVal
Global	Globalizable	Rho
NonGlobal	Globalizable	Rho
Disjoint	disjoint	AbsVal
NonDisjoint	disjoint	AbsVal
FilterOutGlobals	filteroutglobals	Pe
ρ_{resfnct}	ReuseFnct	Rho
ρ_{resproc}	ReuseProc	Rho

Table 12.4: Semantic functions and their implementation

```
| Simili.SIF(expr, label1, label2) =>
    let val (beta, NewExp, mb1, rho2, util2, qmap2, delta2, eqmap2, _) =
      PeExpr (expr, dyn, call, mf, oid, rho1, tau1,
              util1, qmap1, delta1, eqmap1, pi1)
    in
...
      else (* DYNAMIC TEST *)
        let val CALL(MethodName, ResMethodName, rhocall, betascall, rn) = call
          val mf1 =
            MF.adddependent (mf, MF.Iter, oid, MethodName, ResMethodName,
                             rhocall, betascall, None, AbsVal.empty)
          val lf1 = LF.adddependent (lf, LF.Iter, label, lbl2, rho, tau)
          val (lbu1, mbu1, lblu1, listu1, rhou1, tauu1,
               utilu1, qmapu1, deltau1, eqmap3, piu1) =
            PeBB (lookupBB (label1, bblist), bblist, dyn, call, mf1,
                  lf1, oid, rho2, tau1, util2, qmap2, delta2, eqmap2, pi1)
          (* reuse code generated in the first branch *)
          val newrho = Rho.CombRho(rho2, rhou1)
          val (lbu2, mbu2, lblu2, listu2, rhou2, tauu2,
               utilu2, qmapu2, deltau2, eqmap4, piu2) =
            PeBB (lookupBB (label2, bblist), bblist, dyn, call, mf1, lf1,
                  oid, newrho, tau1, util2, qmap2, delta2, eqmap3, pi1)
          val mb2 = MB.lubseq (mb, MB.lubseq (mb1, MB.lubif (mbu1, mbu2)))
          val lb1 = LB.union (lbu1, lbu2)
          val rho3 = Rho.lubif (rhou1, rhou2)
          val tau2 = Tau.lubif (tauu1, tauu2)
          val qmap3 = Qmap.lubif (qmapu1, qmapu2)
          val util3 = Util.lubif utilu1 utilu2
          val pi2 = Pi.lub (piu1, piu2)
          val (tau11, tau22) =
            case rn of
              None => (Tau.empty, Tau.empty)
            | Some(rn) =>
                ((case Tau.getvaluesafe (tauu1, rn) of
                    Some(value) => Tau.update (Tau.empty, rn, value)
                  | None => Tau.empty),
                  (case Tau.getvaluesafe (tauu2, rn) of
                     Some(value) => Tau.update (Tau.empty, rn, value)
                   | None => Tau.empty))
          val delta3 = Delta.lubif deltau1 deltau2
          val delta4 = Delta.Explicator4(rhou1, tau11,
                                         Rho.rho2simple rhou2, tau22, delta3)
          val list = Simili.SBB(lbl2, NewStmnt,
                                Simili.SIF(NewExp, lblu1, lblu2)) ::
            (listu1 @ listu2)
        in
          (lb1, mb2, lbl2, list, rho3, tau2,
           util3, qmap3, delta4, eqmap4, pi2)
        end
```

Figure 12.3: The ML code handling the dynamic branch of basic blocks.

as expected for instance variables. The only local variables that we generate explicators for is an eventual result name as the other variables are dead.

This brief description of how to implement the inference rules of the partial evaluator hopefully gives the reader an impression of how we have implemented the partial evaluator. In Appendix C.2 the source code for the `PE` module is listed.

12.4.2 The different modules — a short description

The partial evaluator is implemented in three different modules, each of which consist of a collection of other modules. Below, we will give a short description of the modules used in the different phases.

The prephase

simili.lex and sugar.lex Describe the tokens of the languages. ML-lex uses these files to generate lexers that converts strings into a token streams.

simili.grm and sugar.grm Describes the grammar of the languages, and what code to generate for each rule. ML-yacc reads these files and generates a parser to convert the token stream, provided by the program generated by ML-lex, into an abstract syntax tree.

SIMILI.sml and SUGAR.sml Two signatures that describe the abstract syntax tree and some functions operating on the abstract syntax tree, e.g. printing a tree.

DESUGAR.sml Describes how a sugared abstract syntax tree is transformed into an unsugared.

TRANS.sml Describes how an unsugared abstract tree is analyzed, how to check that all names are used in a proper context, and how to assign the name type to the string. This module is used to prepare a program for partial evaluation.

The specializer

AbsVal Types representing builtin objects, abstract object, and abstract values, and functions on these types of objects, i.e. different lub operators.

Arg Implements the type corresponding to an argument list, i.e. a list of abstract objects, and the corresponding functions on these objects.

Delta Implements the semantic domain, Explicit_Map, as a type and functions operation on this type. Information is collected during partial evaluation, and when fetching the program from the object store, ρ, the explicator information is used to insert the required assignment statements. This is done in the Print module.

EQMap Describe which objects that are compared during partial evaluation. The information is used in the postphase to ensure that two objects are never merged together, if they can be compared to each other.

LabelBackward Implement the semantic domain, LabelBackward, as a type and functions operating on this type.

LabelForward Implement the semantic domain, LabelForward, as a type and functions operating on this type.

MethodBackward Implement the semantic domain, MethodBackward, as a type and functions operating on this type.

MethodForward Implement the semantic object, MethodForward, as a type and functions operating on this type.

Pe This is the most important module in the partial evaluator, as it implement the inference rules described in the previous chapters. A full listing of this module can be found in Appendix C.2.

PeBaseValue This module implements functions on built in objects.

Pi Implements the semantic domain, Problematic, as a type and functions operating on this type. This module is integrated with the MethodForward module.

Print Implements a function, `program`, to fetch the program from the Object_Store, ρ, and return an abstract syntax of the program.

Qmap Implements the semantic domain, Quantified_Map, as a type, and functions on this type.

Rho Implements the semantic domain, Object_Store, as a type. Further, a lot of local types are implemented to represent the different component of the elements in the Object_Store. Functions operating on the type is also included in this module.

Tau Implements the semantic domain, LocalEnv, as a type, and functions operating on this type, i.e. different lub operators, lookup and update functions etc.

Util Implements the semantic domain, Utilization_Store, as a type, and functions operating on this type.

The postphase

GarbageCollect Implements a routine that takes an abstract syntax tree, and remove unused constant declarations, methods, instance variables and local variables. This module uses a mark and sweep algorithm to remove the garbage from the program.

Merging Implements a function that first calculates which constant declarations can be merged together into one single constant declaration by merging object constructors on this information the constant declarations are merged and the resulting declarations sorted.

Postphase Implements a function to postprocess the program, i.e. a function that takes an abstract syntax tree and return an abstract syntax tree. The postphase includes four phases. The garbage collection phase collects unused constant declarations, objects, instance and local variables and methods. The next phase merge merge constant declarations together. In the third phase, method invocations within the same object are unfolded, and methods that do not perform any actions, i.e. does not change an instance variable, are removed. Finally, in the last phase basic blocks are combined, i.e. all transitions are compressed.

Specializer Implements some flags used to control how the postphase of the partial evaluator should work.

TransitionCompression Implements a function to compress transitions in a body with labels and basic blocks.

Unfold This module binds all unfolding functions found in the Postphase together. The unfolding functions can be found in three modules, `Merging`, `MethodUnfold` and `TransitionCompression`.

The reader should now be able to identify the different modules in the system.

12.4.3 Getting better performance

To achieve better performance we have implemented a mapping from integers into any type. The integer keys are sorted in a binary tree. The implementation of mappings in the library, EqFinMap, uses lists to implement mappings. This implementation is actually not even implemented using tail recursive functions, which gives bad response times and uses a lot of heap space. The new datatype is described by the signature "INT_BIN_MAP", listed in Appendix C.1.5.

This mapping is used in the `Rho` and `Util` modules, that maps OIDs into an instance environment and other components. This resulted in much better response times and reduced the use of heap tremendously relative to when the EqBinMap datatype was used.

12.4.4 Optimizations

This section will shortly describe what optimizations we have performed. They cover saving of heap space to decrease the number of page swaps and garbage collections, and reusing computations in different ways.

In MethodForward and LabelForward we save the object store when a loop is entered and the approximations to the resulting object store is also saved. We are only interested in the instance environment of the object store, so therefore the object store is converted to

a simple version of the object store, without the name, code and residual code components. This type is denoted by `rhosimple`. This saves a lot of heap space.

In MethodBackward and LabelBackward we do not save the whole object store either. Because we need the residual code we cannot use the technique described above for MethodForward and LabelForward. Instead we compute the difference between the current object store and the object store in the LF or MF semantic objects. The function to compute the difference is described by `Rho.difference` which take an object store and a simple object store (from Forward) and return an object store with all elements where the instance environments was not equal.

Finally, in MethodForward, we do not save an empty object store, ρ_0 as the result of the method invocation or loop, but just the $\perp$ value, represented in the ML program by `None`. If we invokes a method and recursion is identified, then if the result object store in MethodForward is $\perp$, then we know this is the first invocation of the method after the loop was entered, and we therefore know that it *must* be recalculated, i.e. we can add a **Redo** tuple in MethodBackward instead of a **Use** tuple.

When reusing methods we only have to save the object store before and after the method invocation, if it is possible to reuse the method. This is *only* possible if we have not created any static objects during the method invocation. If it is possible to reuse the method we only have to save the information in the object store about the value of the used respectively defined instance variables. This saved a lot of heap space.

12.5 The self-interpreter

This section will describe how to write an interpreter in our own language, that uses the technique *active syntax* described earlier in Section 3.5. To summarize the technique consist of two phases:

- Convert the program into an "active syntax"
  In the first phase the program is converted from a string into an "active syntax". This is very similar to converting a string into an abstract syntax tree, but the nodes in the "active syntax" are not just datarecords, but objects with methods that knows how to execute or evaluate itself. We must therefore describe these nodes, what method they must have, what other objects they must use etc.

- Tell the program to interpret itself.
  Given the program represented as an "active syntax" tree, we simply interpret the program by invoking the method, say "interpret", that interprets the program.

For each syntactic category of our language we have a constant declaration in the self-interpreter with two methods, `parse` and `new`. The `parse` method parses the its own syntactic category, and uses `new` to return a semantic object representing the syntactic object. The parse routines make a recursive descend until basic objects are found. The `parse` and `New` routines also exists for builtin objects.

Throughout this section we will use a notation of abstract types of objects, which is defined as the collection of method signatures, i.e. a method name, type of the methods

arguments, and eventually the type of the result. This is similar to the abstract types used in Emerald [Hutchinson 87]. Abstract types is useful when describing the self-interpreter. It can simply be seen which methods the different objects must have, and it is simple to read the code.

An object is said to conform to an abstract type, iff the object always can be regarded as an object of the type, i.e. in some sense is greater than or equal to the type. The term *conformance* is formally defined in [Raj 88].

We assume the existence of some builtin abstracts types that the builtin objects conform to.

12.5.1 Parsing into a token stream

Parsing a program is based on the simple syntax listed in Figure 12.4. The parsing is done in two phases, a lexical and a grammatical analysis. The conversion of a string into a token stream is done by a global object denoted by `token`. The objects is initialized by the invocation `token.setstring[string]` where `string` is the program to be parsed. The procedure `token.readnexttoken[]` reads the next token, which can be retrieved by the function `token.gettoken[]`.

The grammatical analysis is done by the different `parse` methods as mentioned above. The global object `program` is used to parse a program. The `parse` routine in the object uses recursive descend to parse a program. Each syntactic category is parsed by the `parse` method found in the global object with the same name as the syntactic category.

As can be seen by the simple syntactic object in Figure 12.4 a lot of lists occur in the syntax. All begins with the keyword `list` and ends with the keyword `end`. Therefore, it is possible to build a general list constructor, that, given a syntactic category object, returns an object used to parse a list of the corresponding syntactic objects.

The code for suck list constructor is listed in Appendix D.1

12.5.2 The semantic objects

The semantic objects used when interpreting the program is the local environment, τ, the object store, ρ, the current object, Ω, and the current instance environment, $\imath$. The instance environment is not used in the inference rules, but used in the self-interpreter to make it more efficient. These objects are specializations of environments, which are simple to implement in our language. The implementation of an environment is listed in Appendix D.2.

The abstract data type of an environment is listed in Figure 12.5. An *equal* type matches an object that has a function `equal` that, given an object of the same type returns a boolean object. An *outstream* type matches an object that has a procedure `putString`. The environment has the abstract data type shown in the figure.

12.5.3 The "active syntax" objects

This section shall describe how we represent a program as an *active syntax* tree, i.e. what abstract types the different nodes in the tree must have to be able to execute itself.

```
Program        ::= list ConstDecl* end.

ConstDecl      ::= const ConstName Exp.

BasicExp       ::= constname ConstName
                 | self
                 | ivarname IVarName
                 | lvarname LVarName
                 | formal FormalPar
                 | result ResultName
                 | basevalue BaseValueName.

BaseValueName  ::= integer integer
                 | string string
                 | char char
                 | true
                 | false
                 | nil
                 | unix.

BasicExps      ::= list BasicExp* end.

Exp            ::= basic BasicExp
                 | equal BasicExp BasicExp
                 | invoke BasicExp FnctName BasicExps
                 | object ObjectName IVarDecls ProcDefs FnctDefs Body.

ProcDefs       ::= list ProcDef* end.

FnctDefs       ::= list FnctDef* end.

ProcDef        ::= procedure ProcName FormalPars LVarDecls Body.

FnctDef        ::= function FnctName FormalPars ResultName LVarDecls Body.

FormalPars     ::= list FormalPar* end.

IVarDecls      ::= list IVarName* end.

LVarDecls      ::= list LVarName* end.

Body           ::= list BasicBlock* end.

BasicBlock     ::= bb label Statements Jump.

Statements     ::= list Statement* end.

Statement      ::= ivarassign IVarName Exp
                 | lvarassign LVarName Exp
                 | resultassign ResultName Exp
                 | invoke BasicExp ProcName BasicExps
                 | skip.

Jump           ::= goto label
                 | if Exp label label
                 | return.
```

Figure 12.4: Special syntax used by the selfinterpreter

```
type Any
   function asString[] → [string]
end Any

type equal
   function equal[equal] → [bool]
end equal

type outstream
   procedure putString[string]
end outstream

type environment
   function lookup[equal] → [Any]
   function asString[] → [string]
   function copy[] → [environment]
   procedure setvalue[equal, Any]
   procedure introduce[equal]
   procedure settail[environment]
   procedure PrettyPrint[outstream]
end environment

type objectstore
   function lookup[equal] → [environment]
   function asString[] → [string]
   function copy[] → [environment]
   procedure setvalue[equal, environment]
   procedure introduce[equal]
   procedure settail[objectstore]
   procedure PrettyPrint[outstream]
end objectstore
```

Figure 12.5: Abstract data type for an environment object

type *load*
  procedure *load*[*objectstore*]
end *load*

Figure 12.6: Abstract data type for a load method

Interpreting a program can be separated into three different parts, loading, executing and evaluating a node. Below we will explain what actions these nodes must perform, to interpret a program, and what their abstract data type will look like.

Constant declarations are *loaded*, basic expressions, argument lists, i.e. a list of basic expressions, expressions and functions are *evaluated*, and finally statement, statement lists, basic blocks, and procedures are *executed*.

Loading a node

Loading a node is used when the constant declarations are loaded to give the initially object store. Loading a constant declarations consist of two parts, first the expressions must be *evaluated*, as described below, then, the value must be assigned to the constant name in the object store.

In our selfinterpreter a program is represented as a list of constant declarations, and loading a program, is done by loading all constant declarations in the list. The list itself, simply traverses all elements and load the constant declaration, so the new object store is returned. The implementation of the list is listed in Appendix D.1. Loading the single constant declaration is defined in the constant declaration itself, *load*. The initial object store is simply an empty environment, and each constant declaration adds a new element to this environment.

Loading a constant declaration is very simple. The methods is given the current object store, ρ, as an argument. The object store is implemented as an environment as described above in Figure 12.5. The `load` method then evaluates its expressions. It then introduce the constant name in the environment, add set the value of the constant name to the evaluated value. All constant declaration objects conform to the abstract type shown in Figure 12.6. The environment given to the method is updated during the load as a side-effect, so no new environments must be returned, as if we had implemented this function in a functional language, say ML.

Evaluating a node

All evaluate methods take the object store, ρ, the instance environment of the current object, $\imath$, the current local environment, τ, and the current object, Ω. The evaluate method is implemented using both a procedure and a function. The procedure, named `evaluate`, is used to evaluate the expression and thereby calling a procedure to add new objects if necessary, and the function, named `evaluateresult`, is used to return the result of evaluating the expression. The function must always be invoked immediately after we have

invoked the `evaluate` procedure. Evaluating a node is easy. The procedure `evaluate`

type *anobject*
  function *objecttag*[] → [*integer*]
  procedure *fnctcall*[*objectstore, string, anobject* **list**]
  function *fnctcallresult*[] → [*anobject*]
  procedure *proctcall*[*objectstore, string, anobject* **list**]
end *anobject*

type *evaluate*
  function *evaluateresult*[] → [*Any*]
  function *asString*[] → [*string*]
  procedure *evaluate*[*objectstore, environment, environment, anobject*]
  procedure *PrettyPrint*[*outstream*]
end *evaluate*

Figure 12.7: Abstract data type for an evaluate method

is passed the object store, current instance environment, current local environment and current object. The procedure `evaluate` can create new object, and will therefore have a side effect on the object store. The result of evaluating can be accessed by calling the function `evaluateresult`.

In Figure 12.7 we show the abstract type of all objects that can be evaluated must conform to. The abstract type `anobject` represents any object handled by the self-interpreter.

The abstract type of an object requires a short explanation. The function `objecttag` is used to get the "type" of the object, i.e. a built in or a program created. Two things can be done with an objects. A procedure or a function can be invoked. Invoking a procedure is implemented by the `proccall` procedure. The `string` given is the name of the procedure to invoke.

In Figure 12.8 we give an example of how to evaluate an equality expression. We show a constant declaration, `equalityexpression` that refers to an object with a method, `new`, used to return a semantic object with the abstract type, evaluate.

The semantic object has three instance variables, `firstexp`, `secondexp`, and `thevalue`. The two first variables each refers to a semantic object of a basic expression. When evaluating an equality expression, we first evaluate the first basic expression, and then the second. Then we compare the two values returned. If they are the same object, we return true, `truthobject`, else we return false. The result is assigned to the instance variable `thevalue`, and can be accessed by calling the function `evaluateresult`.

Executing a node

Executing a node is very similar to evaluating a node and we will therefore not comment further on this subject.

```
const equalityexpression == object equalityexpression
var parseresult
function new[first, second] → [result]
    result ← object anequalityexpression
        var firstexp var secondexp var thevalue
        procedure evaluate[thestore ienv, lenv, currentself]
            var thefirstvalue var thesecondvalue
            firstexp.evaluate[thestore, ienv, lenv, currentself]
            thefirstvalue ← firstexp.evaluateresult[]
            secondexp.evaluate[thestore, ienv, lenv, currentself]
            thesecondvalue ← secondexp.evaluateresult[]
            if thefirstvalue == thesecondvalue then thevalue ← truthobject
            else thevalue ← falseobject end if
        end evaluate
        function evaluateresult[] → [result]
            result ← thevalue
        end evaluateresult
        function asString[] → [result]
            var str
            result ← firstexp.asString[]
            result ← result.catenate[" == "]
            str ← secondexp.asString[]
            result ← result.catenate[str]
        end asString
        initially
            firstexp ← first  secondexp ← second
        end initially
    end anequalityexpression
end new
procedure parse[]
    var first var second var thetoken
    thetoken ← token.gettoken[]
    if thetoken.equal["equal"] then
        token.readnexttoken[]
        basicexpression.parse[]
        first ← basicexpression.parseresult[]
        basicexpression.parse[]
        second ← basicexpression.parseresult[]
        parsevalue ← self.new[first, second]
    else parsevalue ← nil.error["error in program (no equal keyword)"] end if
end parse
function parseresult[] → [result]
    result ← parsevalue
end parseresult
end equalityexpression
```

Figure 12.8: An equality expression.

12.5.4 Interpreting the program — a graph?

When a program is finally parsed, an "active syntax" tree is returned. This "active syntax" tree is *active* because it is capable of interpreting itself, and returning the result of executing the program.

Now, it is easy to interpret a program. The active syntax tree can be reached by calling the function `program.parseresult[ ]`, but interpreting the program is actually implemented another way! The constant declaration `program` refers to an object used to parse the program. When we want to interpret the program, we simply calls the routine `interpret` in the object denoted by the constant name `program`. We pass the program to be interpreted as a string, and further we pass three more strings, a constant name, a procedure name and a function name, and an argument list, containing the arguments of the program to be interpreted.

The `interpret` procedure, first invokes the token object to assign the string to the token, to prepare for parsing. It then parses the program, given an active syntax tree. Then the argumentlist is parsed, which give a list of objects to be passed to the procedure of the program to be interpreted. We first load the program, by calling the `load` procedure in the active syntax tree. Then, we fetch the constant declaration with the constant name passed to the interpreting routine, and then invoking the procedure with the argument list. Finally, the result of executing the program is reached by calling the function.

The argumentlist passed to the program to be interpreted uses the same syntax as the program to be parsed.

12.6 Testing the system

This section will briefly describe how we have tested our system. The test is not systematic and this description only covers the partial evaluator. We have not tested the parser, the interpreter written in ML, the postphase, or the selfinterpreter. We have only corrected errors as we encountered them. They have however been working faultless for quite some time now.

The specializer itself is tested more systematically. It is not tested in every little detail as this would require an enormous amount of work. We have written test programs testing different things we expect the partial evaluator to handle. These things can be divided into the following categories:

Static loops either iterative or recursive, must be fully specialized. The power function is the classical example of a static loops and specializing the power function should therefore by totally specialized. Another example of a static loops is found in an interpreter. In an interpreter loop all statements of the program are inspected. If the program is known the test of the loop is static and can therefore be fully specialized. Inside the static loops there will typically occur a dynamic conditional, that therefore should occur in the residual program.

Dynamic loops cowers two cases. First, information must be generalized to avoid infinite specialization, and second dynamic loops must be unfolded some steps, as

long all branches of the loop leads to itself. The tests must also cover nested loops. Examples of dynamic iterative loops is described in Section 7.2.

Object creation Objects created under static control should be static, and objects created under dynamic control should be quantified. It should be tested, that if a sequence of objects is created in a dynamic loop, the information should be generalized so it represent all objects.

Dynamic Object Creation Objects created in a speculative loop, where each creation step create a new object of the same kind, as illustrated in Example 7.2 and described in Section 7.1.4.

Globalization Objects created under static control should be globalized iff all instance variables defined in the initially body refers to global values, i.e. to object with a constant name or a built in object. Globalization of objects is described in Section 8.1.2.

Non-manifest invocation We must test that non-manifest method invocations works properly, that a method with the same name is generated in all objects, and that the program state after the invocation represent all possible invocations. Non manifest method invocations is described in Section 6.2.

Reusing method We must test the conditions for reusing methods. Reusing is very important to reduce the number of specialized methods.

Hard method invocation is a non-manifest invocation where one of the objects invoked is either input dynamic or a built in object. In this case we have to generate a residual method with the same name as the original method, and this can lead to problems, if a method is invoked more that once in a hard method invocation. In this case, the new residual method must be usable in all cases.

We have written test programs testing all of the above. It has not been possible to find errors in the specializer by specializing these test programs. Only a few of the test programs are applied in the appendices of this Thesis. In Table 12.5 we have indicated where to find the included test programs and what is tested by the programs.

We know of one error in the specializer, but we do not intend to correct this error. Suppose we have an object with two methods, `cons` and `cons_48`. If `cons_48` is invoked in a hard method invocation, then a residual method with the name `cons_48` is generated. If the method `cons` is invoked in a normal invocation, then we may generate a specialized version of this method with the name `cons_48`. We do not check if the generated method names clash with forced method names. We consider this an unimportant flaw of the implementation.

Program	**See**	**Testing**
Turing1	B.1	static iterative loops
Turing2	B.2	static recursive loops
Nested loops	B.3	dynamic iterative loops
Ackermann	B.4	dynamic recursive loops — functions
Hard methods	B.5	Hard method invocation
Devious	B.6	Dynamic object creation
Recursion	B.7	$[\![Mix]\!]$ selfint p

Table 12.5: Programs testing the self-interpreter

This page intentionally left blank.

Chapter 13

Performance Evaluation

The performance of a general tool can never be specified completely. There are no other tools that are supposed to fulfill the same goals as the partial evaluator we have developed so a relative comparison is not possible. In this section we shall describe how we evaluate the performance of the developed partial evaluator.

The usual yard stick to measure partial evaluators are the speedup gained on some more or less arbitrary set of test programs. Typical test programs are the power function, a function computing Ackermann's function, and a small interpreter. We will of course use this yard stick as part of the evaluation.

Another quantitative measurement we will perform is the time it takes to perform the specialization. This influences the usability of the specializer a lot. It does not help much if you have a fantastic algorithm that can make your "Hello World" program run 10% faster if it takes a couple of hours to achieve this result.

A criteria that is a little more "fuzzy" is the quality of the generated programs. Are there any obvious transformations that are not performed? Are the residual programs optimal or close to optimal? We will try to answer these vague questions as we feel able to.

13.1 Quantitative measures

The quantitative measures we will discuss are the speedup of the residual program relative to the source program, and the time it takes to generate the residual program.

Although the speedup of a specialized program relative to the source program can be measured quite accurately it does not necessarily say much about the quality of the specialization as the following holds:

> Name the speedup you want (it must be positive) and it is possible to construct a source program and partial input to this program so specialization of the program will show the desired speedup.

This fact makes it very hard to say anything absolute about the quality of any specializer in terms of speedup achieved by specialization.

Program Name	Execution Time	Speedup	# lines	Factor
L pow (5 x)	7.7 ms		28	
target = MIX pow 5	70.0 ms		15	1.9
L target x	1.7 ms	4.6		
L Ack (3 2)	879.9 ms		27	
target = MIX Ack 3	50.0 ms		40	-1.5
L target 2	28.0 ms	31.0		
L Tur_1 prg_T tape	280.0 ms		252	
target = MIX Tur_1 prg_T	5109.4 ms		31	8.1
L target tape	30.0 ms	9.3		
L Tur_2 prg_T tape	580.0 ms		448	
target = MIX Tur_2 prg_T	14798.5 ms		41	10.9
L target tape	40.0 ms	14.5		
L Neil prg_N 8	330.0 ms		141	
target = MIX Neil prg_N	2239.8 ms		19	7.4
L target 8	8.0 ms	41.3		

Table 13.1: A table showing speedup

There are a small set of programs that it is customary to try a new specializer on. These programs include the power function and Ackermann's function. In addition to this it is also normal to try the specializer on a small interpreter or a Turing machine. We have tried specializing the power function, Ackermann's function, and two different implementations of a Turing machine. One of the Turing machines is implemented using a case-construction as normally done. The other implementation of the Turing machine is atypical. A new implementation strategy has been used where each program statement is represented by one object, called "active syntax", see Section 3.5. This implementation strategy has been chosen to illustrate the specializers ability to deal with objects.

In Table 13.1 we have listed the execution times and number of lines of the source and residual programs along with the time it takes to generate the residual program for a number of test programs. All times are measured on a Sun Sparc-2 with 64 MB internal storage, using the ML compiler of New Jersey version 0.75. The time is the pure CPU time without garbage collection and time wasted on other processes. The garbage collection time was neglible in these tests.

We find the measured speedup factors satisfactory. They are of the same magnitude as the speedups achieved by all other specializers. The generated residual programs are close to optimal so eventual differences in speedup factors is due to differences in the language and the interpreter/compiler of the language.

A reduction in the number of lines from the source program to the residual program does not necessarily indicate good specialization. When specializing an interpreter, the size of the residual program depends as much on the static input to the interpreter as on the quality of the specialization. A reduction of the number of lines in the residual program relative to the source program does however indicate that specializing does not lead to a code blowup.

Specialization of all the test programs except the one computing Ackermann's function shows satisfactory reductions of the code size. Specialization of the program computing Ackermann's function leads to an increase in code size. The increase is caused only by the partial unfolding of the function. We find this an acceptable behavior.

The usability of the specializer really depends on the time it takes to specialize a program. The specialization times for the test programs are also listed in Table 13.1. The specialization time of the power function and Ackermann's functions are approximately as expected. The specialization times of the interpreters are somewhat higher than we expected. Part of the reason for this is that the parsing phase has to be specialized too. A major fraction of the specialization times for the interpreters is the time it takes to specialize the parsing phase.

We have tried specializing the self-interpreter (app. 3000 lines) with very simple programs as the static input. The specialization process only terminates for extremely simple programs. For more complicated programs the partial evaluator uses so much memory that page thrashing prevents effective use of the CPU. We infer that self application of a Simili implementation of the partial evaluator is not practical yet[1]. The SML/NJ is known to generate code that requires a lot of memory. Using another ML compiler might make it possible to specialize the self-interpreter with larger programs.

The memory requirements for the specializer are at the moment much too high. We have done some work trying to bring it down but there is still a lot that could be done. It seems that these improvements are necessary to make use of the partial evaluator realistic.

13.2 Quality of the specialized programs

Grading the quality of the generated specialized programs is difficult and subjective. It is however possible to determine if a specializer is not perfect. If the specializer is perfect, then the following would hold:

$$p = [\![MIX]\!]\, sintp$$

for all programs, p, and all self-interpreters, *sint*. We can show by counterexample that our specializer is not perfect.

The residual programs generated by specialization of the test programs have a quite natural structure. A few specialized programs have superflous parameters and/or methods that are identical apart from the names. Due to the way we have implemented methods unfolding, the residual programs will sometimes have a sequence of assignment statements where the first assignment statement assigns a value to a variable and the subsequent assignment statements assigns the value of the previous variable to a new variable. A simple change in the unfolding algorithm can remedy this. It looks like all the computations that it is possible to perform has been performed by the specializer.

[1] from conversations with technicians from companies delivering e.g. Sun computers we expect the workstations of the coming years equipped with a lot of random access memory in order to avoid having swap devices for each work station. We presume that self application of the partial evaluator should be possible when 1 GB random access memory is available.

The residual programs generated by specializing the self-interpreter have a far from natural structure, as can be seen in Appendix B.7. The self-interpreter represents both program objects and builtin objects by objects created by the interpreter. Integers, strings, etc. are encapsulated in objects with a certain interface. The encapsulation of the builtin objects is preserved in the residual programs. We cannot see any obvious ways to unpack these objects. Solving this problem is a prerequisite for the optimality of the specializer.

The quality of the generated programs can be said to be quite good for source programs that does not encapsulate builtin objects in program objects. There are minor flaws but these can very easily be removed by an optimizing compiler and a parameter splitter. The quality of residual programs generated by specialization of programs that encapsulate builtin object in program objects is not very good. The encapsulation is simply not removed.

Chapter 14

Conclusion

14.1 Summary

We have constructed, described and implemented an online partial evaluator for an object-oriented imperative programming language. The language is far from a toy language. It is a slightly simplified version of the Emerald programming language without types.

With the construction of the partial evaluator the first of the theses listed in Section 5.2 is proved. The second these cannot be said to be proved or disproved. Using objects it is possible to avoid a type analysis as required when performing partial evaluation of e.g. C [Andersen 91b]. The objects take the place of the types. On the other hand a number of other problems are introduced because objects are first class higher order values and because state is enclosed in objects.

During the work with constructing the partial evaluator we have developed a number of novel techniques:

- To achieve the desired termination properties for the abstract interpretation we use a call stack abstraction to detect speculative loops. The technique is slightly better than and developed independently from the similar technique developed at Stanford University for use in the partial evaluator Fuse.

- Unrolling of some speculative loops is made possible without affecting termination properties too much. The unrolling is stopped only when it is possible to exit the loop by taking one of the possible branches in undecidable conditional jumps and nonmanifest invocations.

- We can generate explicator assignment statements for variables that are not in the current scope. The assignent statements are inserted in the basic block they would have been in had the specializer decided to reduce rather than specialize in the first place. Abstract values are associated with basic blocks (labels) to make this possible.

- We have developed a partial evaluator using a very general strategy for stepwise development. The standard semantics is first changed to a nonstandard semantics using abstract values. The nonstandard semantics is then changed to ensure the

desired termination properties. Finally the code generation is added to the semantics. We have added more steps than these to improve the quality of the generated programs.

- We can reuse methods that have side effects. Only the used part of the program state at the time generation of the method was commenced has to be identical to the same part of the current program state in order to reuse a method. If we reuse a method then the effects of the symbolic execution of the method is also reused.

- We have developed a novel technique for identifying object creation loops. These are nonrecursive noniterative loops. The technique amounts to identifying when during execution of a method an object is created with exactly the same method (same object constructor) and this method is invoked in the new object.

The partial evaluator performs the desired specialization. The speedup achieved on miscellaneous test programs are satisfactory. The speed of the specialization process is however much slower than desired. The memory requirements during specialization are quite high and are to some extend the cause of the slow specialization.

14.2 Future Work

There are two major directions to follow after this project: refinement and application.

The partial evaluator could be refined in several ways. There are a lot of optimizations that still lack being done. To mention a few: aliasing should only be used when it is absolutely necessary, methods should be reused in more cases than presently done, unrolling loops should be cleaned up a bit, and many more methods should be unfolded in the postphase.

It could also be interesting to include types in the language. We have been toying around with this idea for some time during the project period and believe that it should be manageable.

Another direction that is not really a refinement but still involves developing rather than using is constructing an offline partial evaluator. An offline partial evaluator will probably be faster than our online partial evaluator but we do also believe that it will yield much worse results on many practical programs. The object-oriented programming techniques are the cause of the expected difference in quality of the specialization. The major factor is the intensive reuse of code in the form of classes (object constructing objects). Having an offline partial evaluator for comparison purposes could be quite educating.

The other direction to follow is application of the partial evaluator to real programs. We believe it could be interesting to try applying the partial evaluator to many of the programs that are written in the object-oriented style. This includes window systems, file managers, etc. This will however have to wait until the current efficiency problems have been solved.

14.3 Lessons learned

We have committed a lot of grave mistakes during this project. First of all, the project was too large. Second, we have failed completely to use our supervisor during our work. Third, we allowed the size of the project to grow during the project period due to our ambitions of wanting to solve almost all the problems we encountered.

The project was too large due to mainly two different reasons. First of all, the language was to big. It was a mistake to have both functional and procedural methods in the language. Having both functional and procedural method implies that we have to solve everything twice. In theory this should not matter much but in practice it does because there is so much more to keep track of. Having basic blocks and jumps in the language is also unnecessary as all loops could have been expressed by recursion. Next we know that *small is beautiful*.

The second reason for the project being to large is the choice of online techniques. There is no local expertise in online methods. The online techniques are only just now becoming mature enough to be used in realistic specializers. This on the other hand also one of the things that have made this an exciting project to work on.

Developing a partial evaluator for a simple object-oriented imperative programming language is complicated enough. Putting too many language constructs in the programming language *and* electing to use online techniques have made this project bigger than a normal Master's Thesis project. The challenge and satisfaction of mastering the complexity has also been so much greater.

Failing to use our supervisor may have been the gravest mistake we made. He is really a capacity in the area of partial evaluation and failing to draw from his great knowledge means that we have had to reinvent the wheel many times. A Danish phrase describing the situation is "*sanke ris til egen røv*"[1]. On the other hand it is nice to be able to say that the techniques invented has been invented by ourselves without any assistance.

That we allowed the size of the project to grow during the project can be seen both as a mistake and as a sign of how exciting the project has been to work on. There has been times where we have felt immense satisfaction after having solved yet another little problem. In the end we did however end up feeling more like finishing the project than solving even more problems.

14.4 Final words

It has been nice that the cooperation between us (the two authors) has worked so smoothly. We each have our strong and weak sides but the mutual respect for this and each other have prevented many problems. We want to end this Thesis with thanking each other for the good teamwork.

[1] An attempt at an English translation: cutting a rod for your own ass.

Bibliography

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools. Student Series.* Addison-Wesley, 1986. (The Dragon Book).

[Andersen 91a] Birger Andersen. *Ellie Language Definition Report.* Technical Report 91/3, DIKU, University of Copenhagen, Denmark, June 1991.

[Andersen 91b] L.O. Andersen. C Program Specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. DIKU Student Project 91-12-17, 128 pages.

[Andersen 92] L.O. Andersen and C.K. Gomard. Speedup Analysis in Partial Evaluation (Preliminary Results). In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University)*, pages 1–7. 1992.

[Appel 88] Andrew W. Appel, Bruce F. Duba, and David B. MacQueen. *Profiling in the Presence of Optimization and Garbage Collection.* As a part of the New Jersey compiler documentation., November 1988.

[Appel 89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer generator for Standard ML.* Princeton University, 1.3 edition, December 1989.

[Berlin 90] Andrew Berlin and Daniel Weise. *Compiling Scientific Code using Partial Evaluation.* Technical Report CSL-TR-90-422, MIT and Stanford, March 1990.

[Berry 91] Dave Berry. *The Edinburgh SML Library.* October 1991.

[Black 91] Andrew Black. Types and Polymorphism in Emerald. In *[?]*. Computer Science Department, Aarhus University, 1991.

[Blair 89] Gordon S. Blair, John J. Gallagher, and Javad Malik. Genericity vs Inheritance vs Delegation vs Conformance vs .... *Journal of Object-Oriented Programming* 11–17, September/October 1989.

[Bobrow 86] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification.* ANSI Report X3J13 Document 87-002, ANSI, 1986.

[Bondorf 88] A. Bondorf, N.D. Jones, T. Mogensen, and P. Sestoft. *Binding Time Analysis and the Taming of Self-Application.* Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[Bondorf 90a] A. Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. In Neil D. Jones (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Springer-Verlag, May 1990. Revised version in [**?**].

[Bondorf 90b] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[Bondorf 90c] A. Bondorf and T.Æ. Mogensen. Logimix: A Self-Applicable Partial Evaluator for Prolog. May 1990. 2 pages. DIKU, University of Copenhagen, Denmark.

[Booch 91] Grady Booch. *Object Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway, Redwood City, CA, 1991.

[Chambers 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In Andreas Paepcke (editor), *OOPSLA '91, Conference Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM/SIGPLAN, pages 1–15. ACM Press, October 1991.

[Consel 88] C. Consel. New Insights into Partial Evaluation: The Schism Experiment. In H. Ganzinger (editor), *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Springer-Verlag, 1988.

[Consel 90] C. Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, ACM, pages 264–272. 1990.

[DeMichiel 87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In *Proceedings of the 1987 European Conference on Object-Oriented Programming*, pages 151–170. 1987.

[Ershov 77] A.P. Ershov. On the Partial Computation Principle. *Information Processing Letters* 6(2):38–41, April 1977.

[Fischback 91] Rainer Fischback. Types and Class. In *[?]*. Computer Science Department, Aarhus University, 1991.

[Futamura 71] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 2(5):45–50, 1971.

[Gluck 91] R. Glück. Towards Multiple Self-Application. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, ACM, pages 309–320. 1991.

[Goad 82] C. Goad. Automatic Construction of Special Purpose Programs. In D.W. Loveland (editor), *6th Conference on Automated Deduction, New York, USA. (Lecture Notes in Computer Science, vol. 138)*, pages 194–208. Springer-Verlag, 1982.

[Goldberg 89] Adele Goldberg and David Robson. *Smalltalk-80: the Language.* Addison-Wesley, 1989.

[Gomard 91a] C.K. Gomard and N.D. Jones. Compiler Generation by Partial Evaluation: a Case Study. *Structured Programming* 12:123–144, 1991.

[Gomard 91b] C.K. Gomard and N.D. Jones. A Partial Evaluator for the Untyped Lambda-Calculus. *Journal of Functional Programming* 1(1):21–69, January 1991.

[Henglein 91] F. Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. In J. Hughes (editor), *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, ACM, pages 448–472. Springer-Verlag, 1991.

[Hutchinson 87] Norman C. Hutchinson. Emerald: A Language to Support Distributed Programming. In Mario R. Barbacci (editor), *Proceedings from the Second Workshop on Large-Grained Parallelism*, pages 45–47. Carnegie-Mellon University Software Engineering Institute, Pittsburgh, PA, 1987. This appears in Special Report CMU/SEI-87-SR-5.

[Hutchinson 91] Norman C. Hutchinson. Types vs. Classe, and Why We Need Both. In *[?]*. Computer Science Department, Aarhus University, 1991.

[Jacobsen 90] H.F. Jacobsen. Speeding Up the Back-Propagation Algorithm by Partial Evaluation. October 1990. DIKU Student Project 90-10-13, 32 pages. DIKU, University of Copenhagen. (In Danish).

[Jones 91] Neil D. Jones. personal communication. 1991. (during supervision of the project).

[Jones 92] Neil D. Jones. Partial Evaluation and the Generation of Program Generators. 1992. To appear in a 1992 issue of Communications of the ACM.

[Kahn 87] G. Kahn. Natural Semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing (editors), *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*, pages 22–39. Springer-Verlag, 1987.

[Kleene 52] S.C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[Meyer 90] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1990.

[Meyer 91] U. Meyer. Techniques for Partial Evaluation of Imperative Languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, ACM, pages 94–105. 1991.

[Milner 90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Milner 91] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[Mogensen 86] T. Mogensen. The Application of Partial Evaluation to Ray-Tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mogensen 88] T. Mogensen. Partially Static Structures in a Self-Applicable Partial Evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones (editors), *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–8. Springer Verlag, October 1986.

[Nielson 88] H.R. Nielson and F. Nielson. Automatic Binding Time Analysis for a Typed λ-Calculus. *Science of Computer Programming* 10:139–176, 1988.

[Nygaard 70] Kristen Nygaard. *System description by SIMULA An introduction.* Technical Report S-35, Norsk Regnesentral / Norwegian Computing Center, November 1970.

[OKeefe 91] Patrick O'Keefe. Partial evaluation in ICAD's programs. November 1991. (personal communication during his visit to DIKU).

[Palsberg 91a] Jens Palsberg. Object-Oriented Type Inference. In Andreas Paepcke (editor), *OOPSLA '91, Conference Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, pages 146–161. 1991.

[Palsberg 91b] Jens Palsberg and Michael I. Scwartzbach. Subclassing and Subtyping. In *[?]*. Computer Science Department, Aarhus University, 1991.

[Raj 88] Rajenda K. Raj, Ewan Tempero, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. *The Emerald Approach to Programming.* Technical Report 88-11-01, Department of Computer Science, University of Washington, Seattle, Washington, November 1988. Revised February 1989, later version published as [Raj 91].

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software — Practice and Experience* 21(1):91–118, January 1991.

[Reddy 88] Uday S. Reddy. Objects As Closures: Abstract Semantics of Object Oriented Languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, ACM, pages 289–297. July 25–27 1988.

[Romanenko 90] S.A. Romanenko. Arity Raiser and Its Use in Program Specialization. In N. Jones (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[Ruf 91] Erik Ruf and Daniel Weise. Using types to avoid Redundant Specialization. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantics-Directed Program Manipulation*, ACM SIGPLAN, pages 321–333. June 1991.

[Sahlin 91] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101, 170 pages.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In Norman Meyrowitz (editor), *OOPSLA '86, Conference Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM/SIGPLAN, pages 9–16. ACM Press, 1986.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Stroustrup 89] Bjarne Stroustrup. Multiple Inheritance for C++. *Computing Systems* 367–395, fall 1989.

[Tarditi 90] David R. Tarditi and Andrew W. Appel. *ML-Yacc, version 2.0, Documentation for Release Version.* April 1990.

[Ungar 87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87, Conference Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–241. December 1987. Published as SIGPLAN Notices 22(12).

[Wegner 86] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, pages 168–182. October 1986.

[Weise 90] Danial Weise and Erik Ruf. *Computing Types During Partial Evaluation.* Technical Report, Stanford University, December 1990. Fuse-Memo 91-3-revised.

[Weise 91a] Daniel Weise. Termination properties of FUSE. november 1991. personal communication.

[Weise 91b] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic Online Partial Evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ACM. 1991.

[Xerox 81] Xerox. The Smalltalk-80 System. *BYTE* 36–48, August 1981.

Appendix A

Simili **built-in objects**

The Simili language is provided with several *built-in* objects described on page 35. The methods for these built-ins are described below:

Boolean

The boolean objects true and false can be referred to in the program with the tokens **true** and **false**. The methods defined on these objects are described below:

function	*and*	[**Boolean**] → [**Boolean**]
function	*or*	[**Boolean**] → [**Boolean**]
function	*not*	[] → [**Boolean**]
function	*asString*	[] → [**String**]

The "nil" object

The nil object can be referred to in the program with the token **nil**. No methods are defined on the nil object.

Character

Character objects can be referred to in the program, if they are surrounded by single quotes, e.g. 'a', 'b', '!'. Methods defined on character objects are described below:

function	*ord*	[] →[**Integer**]
function	*lt*	[**Character**] →[**Boolean**]
function	*leq*	[**Character**] →[**Boolean**]
function	*gt*	[**Character**] →[**Boolean**]
function	*gte*	[**Character**] →[**Boolean**]
function	*eq*	[**Character**] →[**Boolean**]
function	*neq*	[**Character**] →[**Boolean**]
function	*asString*	[] →[**String**]

Integer

Integer objects can be referred to in the program in the usual way of writing integers, i.e. **-1**, **0**, **1**, **2**, **3**, ... The methods defined on integer objects are described below:

function	*plus*	[**Integer**] → [**Integer**]
function	*minus*	[**Integer**] → [**Integer**]
function	*times*	[**Integer**] → [**Integer**]
function	*div*	[**Integer**] → [**Integer**]
function	*sqr*	[**Integer**] → [**Integer**]
function	*even*	[] → [**Boolean**]
function	*lt*	[**Integer**] → [**Boolean**]
function	*leq*	[**Integer**] → [**Boolean**]
function	*gt*	[**Integer**] → [**Boolean**]
function	*gte*	[**Integer**] → [**Boolean**]
function	*eq*	[**Integer**] → [**Boolean**]
function	*neq*	[**Integer**] → [**Boolean**]
function	*asString*	[] → [**String**]
function	*chr*	[] → [**Character**]

String

String object can be referred to in the program if they are surrounded by double quotes, e.g. “a”, “b”, “!”, “abc”. Methods defined on string objects are described below:

function	*length*	[] → [**Integer**]
function	*getElement*	[**Integer**] → [**Character**]
function	*equal*	[**String**] → [**Boolean**]
function	*catenate*	[**String**] → [**String**]
function	*getslice*	[**Integer**, **Integer**] → [**String**]
function	*asInteger*	[] → [**Integer**]
function	*asString*	[] → [**String**]

Unix

This builtin object is not treated by the partial evaluator.

function	*stdin*	[] → [**InStream**]
function	*stdout*	[] → [**OutStream**]
function	*openin*	[**String**] → [**InStream**]
function	*openout*	[**String**] → [**OutStream**]

Input Stream

These builtin objects are not treated by the partial evaluator.

function	*eos*	[] → [**Boolean**]
function	*getchar*	[] → [**Character**]
procedure	*forwardchar*	[**Character**] → []
procedure	*backwardchar*	[**Character**] → []
procedure	*forwardstring*	[**Character**] → []
procedure	*close*	[] → []

Output Stream

These builtin objects are not treated by the partial evaluator.

procedure	*putChar*	[**Character**] → []
procedure	*putString*	[**String**] → []
procedure	*close*	[] → []

Appendix B

The tests

B.1 Turing1

The original program

```
(*************************************************************************)
(* turing1.sugar                                                         *)
(*                                                                       *)
(*************************************************************************)

const TuringMachine == object TuringMachine
   var result_tape
   procedure Execute[program, tape]
      var pc
      var instruction
      var temp var tempa var tempb
      var myout
      myout <- unix.stdout[]
      myout.putString["Executing turing program\n"]
      pc <- 0
      instruction <- program.getElement[pc]
      temp <- instruction.equal["halt"]
      while temp.not[] do
         temp <- instruction.getSlice[0, 3]
         tempa <- instruction.getSlice[0, 4]
         tempb <- instruction.getSlice[0, 1]
         if instruction.equal["left"] then
            tape.left[]
            pc <- pc.plus[1]
         elseif instruction.equal["right"] then
            tape.right[]
            pc <- pc.plus[1]
```

```
         elseif temp.equal["goto"] then
            temp <- instruction.getSlice[5, -1]
            pc <- temp.asInteger[]
         elseif tempa.equal["write"] then
            temp <- instruction.getSlice[6, -1]
            temp <- temp.asInteger[]
            tape.write[temp]
            pc <- pc.plus[1]
         elseif tempb.equal["if"] then
            temp <- tape.read[]
            tempa <- instruction.getSlice[3, 3]
            tempa <- tempa.asInteger[]
            if temp.eq[tempa] then
               temp <- instruction.getSlice[10, -1]
               pc <- temp.asInteger[]
            else
               pc <- pc.plus[1]
            end if
         else
            instruction <- "unknown instruction ".catenate[instruction]
            myout.putString[instruction]
         end if
         instruction <- program.getElement[pc]
         temp <- instruction.equal["halt"]
      end while
      tape.halt[]
      myout.putString["Finish executing turing program\n"]
      result_tape <- tape
   end Execute

   function ExecuteResult [] -> [result]
      result <- result_tape
   end ExecuteResult
end TuringMachine

const TapeNodeClass == object TapeNodeClass
   function New[] -> [result]
      result <- object aTapeNode
         var prev
         var next
         var value
         procedure setPrev[value]
            prev <- value
         end setPrev
```

```
         procedure setNext[value]
            next <- value
         end setNext
         procedure dowrite[v]
            value <- v
         end dowrite
         function doread[] -> [result]
            result <- value
         end doread
         function getPrev[] -> [result]
            result <- prev
         end getPrev
         function getNext[] -> [result]
            result <- next
         end getNext
         initially
            prev <- nil
            next <- nil
            value <- '*'
         end initially
      end aTapeNode
   end New
end TapeNodeClass

const Tape == object Tape
   var readhead
   function read[] -> [result]
      result <- readhead.doread[]
   end read
   procedure write[value]
      readhead.dowrite[value]
   end write
   procedure left[]
      var l var temp
      temp <- readhead.getPrev[]
      if temp == nil then
         l <- TapeNodeClass.New[]
         l.setNext[readhead]
         readhead.setPrev[l]
      else
         skip
      end if
      readhead <- readhead.getPrev[]
   end left
```

```
   procedure right[]
      var l var temp
      temp <- readhead.getNext[]
      if temp == nil then
         l <- TapeNodeClass.New[]
         l.setPrev[readhead]
         readhead.setNext[l]
      else
         skip
      end if
      readhead <- readhead.getNext[]
   end right
   procedure halt[]
      var temp var temp1 var myout
      myout <- unix.stdout[]
      myout.putString[" >> "]
      temp <- self.read[]
      temp1 <- temp == '*'
      while temp1.not[] do
         temp <- temp.asString[]
         myout.putString[temp]
         myout.putString[" "]
         self.left[]
         temp <- self.read[]
         temp1 <- temp == '*'
      end while
      myout.putString[" << "]
   end halt
   initially
      readhead <- TapeNodeClass.New[]
   end initially
end Tape

const temp1 == "[write 0,left,write 1,left,write 1,left,write 1,"
const temp2 == "if 0 goto 10,right,goto 7,write 1,right,write 0,halt]"
const aturingprogram == temp1.concat[temp2]

const consClass == object consClass
   function New[hdval, tlval] -> [result]
      result <- object consObject
         var hd
         var tl
         function hd[] -> [result]
            result <- hd
```

```
end hd
function tl[] -> [result]
   result <- tl
end tl
function getElement[i] -> [result]
   var i1
   if i.eq[0] then
      result <- hd
   else
      i1 <- i.minus[1]
      result <- tl.getElement[i1]
   end if
end getElement
procedure sethd[newhd]
   hd <- newhd
end sethd
procedure settl[newtl]
   tl <- newtl
end settl
function asString[] -> [result]
   var hds var tls
   hds <- hd.asString[]
   if tl == nil then
      tls <- " "
   else
      hds <- hds.catenate[", "]
      tls <- tl.asString[]
   end if
   result <- hds.catenate[tls]
end asString
(* changes the list to be in the opposite direction *)
procedure reverse[arg]
   if arg == nil then
      skip
   else
      arg.reverse[self]
   end if
   tl <- arg
end reverse
(* return a new list, but with the same elements *)
function reverse[a] -> [result]
   result <- consClass.New[hd, a]
   if tl == nil then
      skip
```

```
            else
               result <- tl.reverse[result]
            end if
         end reverse
         initially
            hd <- hdval
            tl <- tlval
         end initially
      end consObject
   end New
   (* test to see, if a list is empty *)
   function isEmpty[l] -> [result]
      result <- l == nil
   end isEmpty
   (* Parses a string of the form "[ <a> , <b>, <c> ]" *)
   (* and return the head of the list *)
   function fromString[s] -> [result]
      var i var j var jj
      var c var subs
      var b1 var b2
      result <- nil
      j <- 0
      c <- s.getElement[j]
      if c.eq['['] then
         i <- j.plus[1]
         c <- s.getElement[i]
         while c.neq[']'] do
            j <- i
            repeat
               j <- j.plus[1]
               c <- s.getElement[j]
               b1 <- c.eq[']']
               b2 <- c.eq[',']
            until b1.or[b2]
            jj <- j.minus[1]
            subs <- s.getSlice[i, jj]
            i <- j.plus[1]
            result <- consClass.New[subs, result]
         end while
         if result == nil then
            skip
         else
            result <- result.reverse[nil]
         end if
```

```
      end if
   end fromString
end consClass

const p == consClass.fromString[aturingprogram]
;
```

The residual program

```
const TuringMachine == object TuringMachine
  var result_tape
  function ExecuteResult[] -> [result]
    m2042:
      result <- result_tape
      return
  end ExecuteResult
  procedure Execute[tape]
    var myout var temp
    m923:
      myout <- unix.stdout[]
      myout.putString["Executing turing program\n"]
      tape.write[0]
      tape.left[]
      tape.write[1]
      tape.left[]
      tape.write[1]
      tape.left[]
      tape.write[1]
      temp <- tape.read[]
      goto l1322
    l1322:
      if temp.eq[0] then goto l1323 else goto l1782
    l1323:
      tape.write[1]
      tape.right[]
      tape.write[0]
      tape.halt[]
      myout.putString["Finish executing turing program\n"]
      result_tape <- tape
      return
    l1782:
      tape.right[]
      temp <- tape.read[]
      goto l1322
```

```
  end Execute
  initially
    l1:
      return
  end initially
end TuringMachine
;
```

B.2 Turing2

The original program

```
(*=============================================================================
  File: turing2.sugar
  This is an interpreter for a turing machine written in an object-oriented
  language. The interpreter is written using an object-oriented style of
  programming.
  The input program is represented as objects having references to each other.
  All statements of the input program is in a vector. When executing a program
  the interpreter first builds an "active syntax" of the program, and then
  executes it!
*)
const TuringMachine == object TuringMachine
   var statement (* First statement of the program! *)
   var result_tape
   function ExecuteResult[] -> [result]
      result <- result_tape
   end ExecuteResult

   procedure Execute[program, tape]
      self.Parse[program]   (* Build "active syntax" of the program! *)
      statement.Execute[tape] (* Execute the program! *)
      result_tape <- tape
   end Execute

(*   Parse takes an input string of the following format:
      <program> ::* <statement> "as a vector, i.e. #( ...)"
      <statement> ::= <instruction>
      <instruction> ::| "left"
                      | "right"
                      | "goto" <label>
                      | "write" <value>
                      | "if" <value> "goto" <label>

    It returns a vector of the statements, where each statement is an
    object of one of the classes listed below!
*)
   procedure Parse[program] (* Program must be a list of statements *)
      var counter
      var instruction
      var lc
      var temp var tempa var tempb
```

```
var ifinstruction var gotoinstruction var writeinstruction
var plength
var newprogram

(* First pass *)
counter <- 0
if program == nil then
   skip
else
   plength <- program.length[]
   newprogram <- consClass.create[plength]

   while counter.lt[plength] do
      instruction <- program.getElement[counter]
      ifinstruction <- instruction.getSlice[0,1]
      gotoinstruction <- instruction.getSlice[0,3]
      writeinstruction <- instruction.getSlice[0,4]

      if instruction.equal["left"] then
         temp <- leftClass.New[]
         newprogram.setElement[counter, temp]
      elseif instruction.equal["right"] then
         temp <- rightClass.New[]
         newprogram.setElement[counter, temp]
      elseif gotoinstruction.equal["goto"] then
         temp <- instruction.getSlice[5, -1]
         temp <- temp.asInteger[]
         temp <- gotoClass.New[temp]
         newprogram.setElement[counter, temp]
      elseif writeinstruction.equal["write"] then
         temp <- instruction.getSlice[6, -1]
         temp <- temp.asInteger[]
         temp <- writeClass.New[temp]
         newprogram.setElement[counter, temp]
      elseif ifinstruction.equal["if"] then
         (*           lc <- 4
         temp <- instruction.getElement[lc]
         while temp.neq['g'] do
            lc <- lc.plus[1]
            temp <- instruction.getElement[lc]
         end while
         tempa <- lc.minus[2]
         temp <- instruction.getSlice[4, tempa]
         temp <- temp.asInteger[]
```

```
               tempb <- lc.plus[4]
               tempa <- instruction.getSlice[tempb, -1]*)

               temp <- instruction.getSlice[3, 3]
               temp <- temp.asInteger[]

               tempa <- instruction.getSlice[10, -1]
               tempa <- tempa.asInteger[]
               temp <- ifClass.New[temp, tempa]
               newprogram.setElement[counter, temp]
            elseif instruction.equal["halt"] then
               temp <- haltClass.New[]
               newprogram.setElement[counter, temp]
            else
               temp <- errorClass.New[]
               newprogram.setElement[counter, temp]
            end if
            counter <- counter.plus[1]
         end while

         (* Second pass *)
         counter <- 0
         while counter.lt[plength] do
            tempa <- newprogram.getElement[counter]
            tempa.init[counter, newprogram]
            counter <- counter.plus[1]
         end while
         statement <- newprogram.hd[] (* getElement[0] *)
      end if
   end Parse
end TuringMachine

(*=============================================================================*)
(* Language classes!!! *)
const rightClass == object rightClass
   function New[] -> [result]
      result <- object aRightObject
         var NextStatement
         procedure init[label, statements]
            var llabel
            llabel <- label.plus[1]
            NextStatement <- statements.getElement[llabel]
         end init
         procedure Execute[tape]
```

```
            tape.right[]
            NextStatement.Execute[tape]
         end Execute
      end aRightObject
   end New
end rightClass

const leftClass == object leftClass
   function New[] -> [result]
      result <- object aLeftObject
         var NextStatement
         procedure init[label, statements]
            var llabel
            llabel <- label.plus[1]
            NextStatement <- statements.getElement[llabel]
         end init
         procedure Execute[tape]
            tape.left[]
            NextStatement.Execute[tape]
         end Execute
      end aLeftObject
   end New
end leftClass

const gotoClass == object gotoClass
   function New[lbl] -> [result]
      result <- object aGotoObject
         var mylabel
         var NextStatement
         procedure init[label, statements]
            NextStatement <- statements.getElement[mylabel]
         end init
         procedure Execute[tape]
            NextStatement.Execute[tape]
         end Execute
         initially
            mylabel <- lbl
         end initially
      end aGotoObject
   end New
end gotoClass

const writeClass == object writeClass
   function New[vl] -> [result]
```

```
      result <- object aWriteObject
         var NextStatement
         var value
         procedure init[label, statements]
            var temp
            temp <- label.plus[1]
            NextStatement <- statements.getElement[temp]
         end init
         procedure Execute[tape]
            tape.write[value]
            NextStatement.Execute[tape]
         end Execute
         initially
            value <- vl
         end initially
      end aWriteObject
   end New
end writeClass

const ifClass == object ifClass
   function New[vl, lbl] -> [result]
      result <- object aIfObject
         var NextStatement
         var JumpStatement
         var value
         var mylabel
         procedure init[label, statements]
            var temp
            temp <- label.plus[1]
            NextStatement <- statements.getElement[temp]
            JumpStatement <- statements.getElement[mylabel]
         end init
         procedure Execute[tape]
            var temp
            temp <- tape.read[]
            if value.eq[temp] then
               JumpStatement.Execute[tape]
            else
               NextStatement.Execute[tape]
            end if
         end Execute
         initially
            value <- vl
            mylabel <- lbl
```

```
         end initially
      end aIfObject
   end New
end ifClass

const haltClass == object haltClass
   function New[] -> [result]
      result <- object aHaltObject
         procedure init[label, statements]
            skip
         end init
         procedure Execute[tape]
            tape.halt[]
         end Execute
      end aHaltObject
   end New
end haltClass

const errorClass == object errorClass
   function New[] -> [result]
      result <- object anErrorObject
         procedure init[label, statements]
            skip
         end init
         procedure Execute[tape]
            skip
         end Execute
      end anErrorObject
   end New
end errorClass

(*=============================================================================*)
const consClass == object consClass
   function New[hdval, tlval] -> [result]
      result <- object consObject
         var hd
         var tl
         function hd[] -> [result]
            result <- hd
         end hd
         function tl[] -> [result]
            result <- tl
         end tl
         function getElement[i] -> [result]
```

```
   var i1
   if 0.eq[i] then
      result <- hd
   else
      i1 <- i.minus[1]
      result <- tl.getElement[i1]
   end if
end getElement
procedure sethd[newhd]
   hd <- newhd
end sethd
procedure settl[newtl]
   tl <- newtl
end settl
procedure setElement[i, value]
   var i1
   if 0.eq[i] then
      hd <- value
   else
      i1 <- i.minus[1]
      tl.setElement[i1, value]
   end if
end setElement
function length[] -> [result]
   if tl == nil then
      result <- 1
   else
      result <- tl.length[]
      result <- 1.plus[result]
   end if
end length
function asString[] -> [result]
   var hds var tls
   hds <- hd.asString[]
   if tl == nil then
      tls <- " "
   else
      hds <- hds.catenate[", "]
      tls <- tl.asString[]
   end if
   result <- hds.catenate[tls]
end asString
(* changes the list to be in the opposite direction *)
procedure reverse[arg]
```

```
            if arg == nil then
               skip
            else
               arg.reverse[self]
            end if
            tl <- arg
         end reverse
         (* return a new list, but with the same elements *)
         function reverse[a] -> [result]
            result <- consClass.New[hd, a]
            if tl == nil then
               skip
            else
               result <- tl.reverse[result]
            end if
         end reverse
         initially
            hd <- hdval
            tl <- tlval
         end initially
      end consObject
   end New
   (* test to see, if a list is empty *)
   function isEmpty[l] -> [result]
      result <- l == nil
   end isEmpty
   (* Parses a string of the form "[ <a> , <b>, <c> ]" *)
   (* and return the head of the list *)
   function fromString[s] -> [result]
      var i var j var jj
      var c var subs
      var b1 var b2
      result <- nil
      j <- 0
      c <- s.getElement[j]
      if c.eq['['] then
         i <- j.plus[1]
         c <- s.getElement[i]
         while c.neq[']'] do
            j <- i
            repeat
               j <- j.plus[1]
               c <- s.getElement[j]
               b1 <- c.eq[']']
```

```
            b2 <- c.eq[',']
         until b1.or[b2]
         jj <- j.minus[1]
         subs <- s.getSlice[i, jj]
         i <- j.plus[1]
         result <- consClass.New[subs, result]
      end while
      if result == nil then
         skip
      else
         result <- result.reverse[nil]
      end if
   end if
end fromString
function create[i] -> [result]
   var temp var i1
   if 0.eq[i] then
      result <- nil
   else
      i1 <- i.minus[1]
      temp <- self.create[i1]
      result <- self.New[nil, temp]
   end if
end create
end consClass

const temp1 == "[write 0,left,write 1,left,write 1,left,write 1,"
const temp2 == "if 0 goto 10,right,goto 7,write 1,right,write 0,halt]"
const aturingprogram == temp1.concat[temp2]

const p == consClass.fromString[aturingprogram]

const str == p.asString[]

(*****************************************************************************)
const TapeNodeClass == object TapeNodeClass
   function New[] -> [result]
      result <- object aTapeNode
         var prev
         var next
         var value
         procedure setPrev[value]
            prev <- value
         end setPrev
```

```
         procedure setNext[value]
            next <- value
         end setNext
         procedure dowrite[v]
            value <- v
         end dowrite
         function doread[] -> [result]
            result <- value
         end doread
         function getPrev[] -> [result]
            result <- prev
         end getPrev
         function getNext[] -> [result]
            result <- next
         end getNext
         initially
            prev <- nil
            next <- nil
            value <- '*'
         end initially
      end aTapeNode
   end New
end TapeNodeClass

const Tape == object Tape
   var readhead
   function read[] -> [result]
      result <- readhead.doread[]
   end read
   procedure write[value]
      readhead.dowrite[value]
   end write
   procedure left[]
      var l var temp
      temp <- readhead.getPrev[]
      if temp == nil then
         l <- TapeNodeClass.New[]
         l.setNext[readhead]
         readhead.setPrev[l]
      else
         skip
      end if
      readhead <- readhead.getPrev[]
   end left
```

```
   procedure right[]
      var l var temp
      temp <- readhead.getNext[]
      if temp == nil then
         l <- TapeNodeClass.New[]
         l.setPrev[readhead]
         readhead.setNext[l]
      else
         skip
      end if
      readhead <- readhead.getNext[]
   end right
   procedure halt[]
      var temp var temp1 var myout
      myout <- unix.stdout[]
      myout.putString[" >> "]
      temp <- self.read[]
      temp1 <- temp == '*'
      while temp1.not[] do
         temp <- temp.asString[]
         myout.putString[temp]
         myout.putString[" "]
         self.left[]
         temp <- self.read[]
         temp1 <- temp == '*'
      end while
      myout.putString[" << "]
   end halt
   initially
      readhead <- TapeNodeClass.New[]
   end initially
end Tape
;
```

The residual program

```
const TuringMachine == object TuringMachine
  var result_tape
  procedure Execute[tape]
    var temp
    m1075:
      tape.write[0]
      tape.left[]
      tape.write[1]
```

```
    tape.left[]
    tape.write[1]
    tape.left[]
    tape.write[1]
    temp <- tape.read[]
    if 0.eq[temp] then goto lbl194 else goto lbl195
  lbl194:
    tape.write[1]
    tape.right[]
    tape.write[0]
    tape.halt[]
    goto lbl196
  lbl195:
    tape.right[]
    self.Execute_128[tape]
    goto lbl196
  lbl196:
    result_tape <- tape
    return
end Execute
function ExecuteResult[] -> [result]
  m5190:
    result <- result_tape
    return
end ExecuteResult
procedure Execute_128[tape]
  var temp
  m5143:
    temp <- tape.read[]
    if 0.eq[temp] then goto l5146 else goto l5169
  l5146:
    tape.write[1]
    tape.right[]
    tape.write[0]
    tape.halt[]
    return
  l5169:
    tape.right[]
    self.Execute_128[tape]
    return
end Execute_128
initially
  l1:
    return
```

```
  end initially
end TuringMachine
;
```

B.3 Dynamic nested iterative loops

The original program

```
(*************************************************************************)
(* Iter.sugar                                                            *)
(* ==========                                                            *)
(* This example is used to tests that dynamic iterative loops works      *)
(* correctly. The example is taken from figure 7.4 in the report.        *)
(*************************************************************************)

const program == object program
   var i
   procedure execute[Dyn]
      var j
      var x
      i <- 0
      j <- 0
      repeat
         x <- 2
         i <- i.plus[1]
         if x.lt[i] then
            repeat
               x <- x.plus[1]
               j <- j.plus[1]
            until x.gt[i]
         end if
      until i.gt[Dyn]
   end execute
   function executeresult [] -> [result]
      result <- i
   end executeresult
end program
;
```

The residual program

```
const program == object program
  var i
  function executeresult[] -> [result]
    l46:
      result <- i
      return
  end executeresult
  procedure execute[Dyn]
```

```
    var x var j
    l3:
      j <- 0
      x <- 2
      i <- 1
      goto l10
    l10:
      if i.gt[Dyn] then goto l27 else goto l28
    l27:
      return
    l28:
      x <- 2
      i <- i.plus[1]
      if 2.lt[i] then goto l30 else goto l10
    l30:
      x <- 3
      j <- j.plus[1]
      goto l37
    l37:
      if x.gt[i] then goto l10 else goto l43
    l43:
      x <- x.plus[1]
      j <- j.plus[1]
      goto l37
  end execute
  initially
    l1:
      return
  end initially
end program
;
```

B.4 Ackermann's function

The original program

```
const Ackermann == object ack
   function compute[m, n] -> [result]
      var newm
      var newn
      newm <- m.minus[1]
      if m.eq[0] then
         result <- n.plus[1]
      elseif n.eq[0] then
         result <- self.compute[newm, 1]
      else
         newn <- n.minus[1]
         newn <- self.compute[m, newn]
         result <- self.compute[newm, newn]
      end if
   end compute
end ack
const comp == object comp
   var i
   procedure doit[m, n]
      i <- Ackermann.compute[m, n]
   end doit
   function getresult [] -> [result]
      result <- i
   end getresult
end comp
;
```

The residual program

```
const comp == object comp
  var i
  function compute_171[n2] -> [result]
    var n2 var newn
    m1724:
      if n2.eq[0] then goto l1728 else goto l1741
    l1728:
      result <- 2
      return
    l1741:
      newn <- n2.minus[1]
      newn <- self.compute_171[newn]
```

```
    result <- newn.plus[1]
    return
end compute_171
function compute_142[n2] -> [result]
  var n2 var newn
  m1593:
    if n2.eq[0] then goto l1597 else goto l1640
  l1597:
    result <- 3
    return
  l1640:
    newn <- n2.minus[1]
    newn <- self.compute_142[newn]
    result <- self.compute_171[newn]
    return
end compute_142
function compute_1[n2] -> [result]
  var n2 var newn
  m1235:
    if n2.eq[0] then goto l1239 else goto l1382
  l1239:
    result <- 5
    return
  l1382:
    newn <- n2.minus[1]
    newn <- self.compute_1[newn]
    result <- self.compute_142[newn]
    return
end compute_1
procedure doit[n2]
  var n2
  m5:
    i <- self.compute_1[n2]
    return
end doit
function getresult[] -> [result]
  m1762:
    result <- i
    return
end getresult
initially
  l3:
    return
end initially
```

```
end comp
;
```

B.5 Hard method invocations

The original program

```
(************************************************************************)
(* Hard2.sugar                                                          *)
(* ===========                                                          *)
(* This example is used to tests whether hard method invocations works  *)
(* correctly.  The hard method invocation created an objects, and thist  *)
(* must be reused in the second hardcall.                               *)
(* A function "geti" is reused in the second case.                      *)
(************************************************************************)
const program == object program
   var r var rr
   procedure execute[d]
      var l
      var j1
      var j2

      l <- self.get[d]
      (* l = {InputDynamic, v} *)

      (* HardProcCall *)
      l.seti[1]

      rr <- v.geti[]   (* manifest call, rr = {toQuantify} *)
      j1 <- rr.addj[d, 2] (* j1 = {3} *)

      l.seti[2]
      j2 <- v.geti[]   (* manifest call, j2 = {toQuantify} *)
      j2.add[9]
      j2 <- j2.getj[]
      (* j2 = {3, 11} *)

      r <- j1.plus[j2]
   end execute
   function get[d] -> [result]
      if d.eq[0] then
         result <- d
      else
         result <- v
      end if
   end get
   function executeresult [] -> [result]
```

```
      result <- r
   end executeresult
end program

const v == object v
   var i
   procedure seti [i1]
      i <- object toQuantify
         var j
         procedure add[j1]
            j <- j.add[j1]
         end add
         function getj[] -> [result]
            result <- j
         end getj
         function addj[d, i2] -> [result]
            if d.eq[0] then
               result <- 88
            else
               result <- i2.plus[j]
            end if
         end addj
         initially
            j <- i1
         end initially
      end toQuantify
   end seti
   function geti [] -> [result]
      result <- i
   end geti
end v
;
```

The residual program

```
const program == object program
  var r var rr var i
  procedure execute[d]
    var l var j2 var j1
    l5:
      if d.eq[0] then goto lbl2 else goto lbl3
    lbl2:
      l <- d
      goto Q1
```

```
    lbl3:
      result <- self
      goto Q1
    Q1:
      l.seti[1]
      rr <- i
      j1 <- rr.addj_3[d]
      l.seti[2]
      j2 <- i
      j2.add_1[]
      j2 <- j2.getj_5[]
      r <- j1.plus[j2]
      return
  end execute
  function executeresult[] -> [result]
    l46:
      result <- r
      return
  end executeresult
  procedure seti[i1]
    l32:
      i <- object toQuantifyq4
        var j
        procedure add_1[]
          l38:
            j <- j.add[9]
            return
        end add_1
        function getj_5[] -> [result]
          l42:
            result <- j
            return
        end getj_5
        function addj_3[d] -> [result]
          l23:
            if d.eq[0] then goto l24 else goto l27
          l24:
            result <- 88
            return
          l27:
            result <- 3
            return
        end addj_3
        initially
```

```
          l33:
            j <- i1
            return
        end initially
      end toQuantifyq4
      return
  end seti
  initially
    l1:
      return
  end initially
end program
;
```

B.6 Dynamic object creation

The original program

```
const a == object devious
   var i
   function New[dyn] -> [result]
      result <- object Dummy
         var newDyn
         function recur[] -> [result]
            var aVar
            var temp
            temp <- newDyn.minus[1]
            if newDyn.zero[] then
               result <- newDyn
            else
               aVar <- a.New[temp]
               result <- aVar.recur[]
               result <- result.plus[newDyn]
            end if
         end recur
         initially
            newDyn <- dyn
         end initially
      end Dummy
   end New
   procedure doit[p]
      var temp
      temp <- self.New[p]
      i <- temp.recur[]
  end doit
   function result[] -> [j]
      j <- i
   end result
end devious
;
```

The residual program

```
const a == object a
  var i
  function New_18[dyn] -> [result]
    l104:
      result <- object Dummyq5
        var newDyn
```

```
      function recur_14[] -> [result]
        var temp
        l97:
          temp <- newDyn.minus[1]
          if newDyn.zero[] then goto l99 else goto l102
        l99:
          result <- newDyn
          return
        l102:
          aVar <- self.New_18[temp]
          result <- self.recur_14[]
          result <- result.plus[newDyn]
          return
      end recur_14
      initially
        l78:
          newDyn <- dyn
          return
      end initially
    end Dummyq5
    return
end New_18
procedure doit[p]
  var temp
  l3:
    temp <- object Dummys3
      var newDyn
      function recur_2[] -> [result]
        var temp var aVar
        l70:
          temp <- newDyn.minus[1]
          if newDyn.zero[] then goto l72 else goto l75
        l72:
          result <- newDyn
          return
        l75:
          aVar <- self.New_13[temp]
          result <- aVar.recur_14[]
          result <- result.plus[newDyn]
          return
      end recur_2
      initially
        l6:
          newDyn <- p
```

```
            return
        end initially
      end Dummys3
      i <- temp.recur_2[]
      return
  end doit
  function result[] -> [j]
    l116:
      j <- i
      return
  end result
  function New_13[dyn] -> [result]
    l77:
      result <- object Dummyq5
        var newDyn
        function recur_14[] -> [result]
          var temp
          l97:
            temp <- newDyn.minus[1]
            if newDyn.zero[] then goto l99 else goto l102
          l99:
            result <- newDyn
            return
          l102:
            aVar <- self.New_18[temp]
            result <- self.recur_14[]
            result <- result.plus[newDyn]
            return
        end recur_14
        initially
          l78:
            newDyn <- dyn
            return
        end initially
      end Dummyq5
      return
  end New_13
  initially
    l1:
      return
  end initially
end a
;
```

B.7 "Self-aplication"

The original program

```
const simple == object simple
   var i var ii
   procedure seti[p]
      ii <- p
      self.recur[]
   end seti
   procedure recur[]
      if ii.eq[i] then
         skip
      else
         i <- i.plus[1]
         self.recur[]
      end if
   end recur
   function geti[] -> [result]
      result <- i
   end geti
   initially
      i <- 0
   end initially
end simple
;
```

The residual program

```
const program == object program
  var myvalue2 var thevalue4 var myvalue3 var thevalue6 var thevalue
  var returnvalue2 var thevalue7 var thevalue5 var returnvalue
  var returnvalue3 var thevalue2 var myvalue4 var thevalue3 var myvalue
  var theresult var fnctresult var fnctresult2 var programresult
  procedure fnctcall_414[thestore, callname, arglist]
    var argvalue var integervalue
    l11191:
      argvalue <- arglist.getElement_617[]
      integervalue <- argvalue.getvalue_618[]
      integervalue <- 0.plus[integervalue]
      returnvalue3 <- object anintegerobjectq160
        var myvalue
        function getvalue_618[] -> [result]
          l11201:
            result <- myvalue
```

```
          return
      end getvalue_618
      function getvalue_653[] -> [result]
        alias1:
          result <- myvalue
          return
      end getvalue_653
      function getvalue_621[] -> [result]
        alias2:
          result <- myvalue
          return
      end getvalue_621
      procedure fnctcall_414[thestore, callname, arglist]
        var argvalue var integervalue
        l11217:
          argvalue <- arglist.getElement_617[]
          integervalue <- argvalue.getvalue_621[]
          integervalue <- myvalue.plus[integervalue]
          returnvalue <- self.new_619[integervalue]
          return
      end fnctcall_414
      function fnctcallresult_623[] -> [result]
        l11228:
          result <- self
          return
      end fnctcallresult_623
      initially
        l11212:
          myvalue <- integervalue
          return
      end initially
    end anintegerobjectq160
    return
end fnctcall_414
function new_619[value] -> [result]
  l11211:
    result <- object anintegerobjectq160
      var myvalue
      function fnctcallresult_623[] -> [result]
        l11228:
          result <- self
          return
      end fnctcallresult_623
      function getvalue_653[] -> [result]
```

```
        alias1:
          result <- myvalue
          return
      end getvalue_653
      procedure fnctcall_414[thestore, callname, arglist]
        var argvalue var integervalue
        l11217:
          argvalue <- arglist.getElement_617[]
          integervalue <- argvalue.getvalue_621[]
          integervalue <- myvalue.plus[integervalue]
          returnvalue <- self.new_619[integervalue]
          return
      end fnctcall_414
      function getvalue_621[] -> [result]
        alias2:
          result <- myvalue
          return
      end getvalue_621
      function getvalue_618[] -> [result]
        l11201:
          result <- myvalue
          return
      end getvalue_618
      initially
        l11212:
          myvalue <- value
          return
      end initially
    end anintegerobjectq160
    return
end new_619
function getvalue_621[] -> [result]
  alias4:
    result <- 0
    return
end getvalue_621
function getvalue_653[] -> [result]
  alias3:
    result <- 0
    return
end getvalue_653
procedure execute[arglist]
  var thevalueQ var newargs var thevalue23 var thevalue232 var theValue
  var objectlist var theobject var thevalueQ2 var objectlist2 var theobject2
```

```
var thevalue233 var theValue2 var objectlist3 var theobject3
var objectlist22 var theobject22 var thevalue234 var thevalue235
l201:
  myvalue4 <- self
  newargs <- arglist
  thevalue23 <- newargs.getElement[1]
  myvalue2 <- thevalue23
  thevalue4 <- myvalue2
  thevalue232 <- thevalue4
  myvalue3 <- thevalue232
  thevalue6 <- myvalue3
  theobject <- thevalue6
  thevalue <- myvalue4
  theValue <- thevalue
  returnvalue2 <- object consCellq159
    var head
    function getElement_617[] -> [something]
      l11194:
        something <- head
        return
    end getElement_617
    initially
      l11069:
        head <- theValue
        return
    end initially
  end consCellq159
  objectlist <- returnvalue2
  theobject.fnctcall[self, "eq", objectlist]
  thevalue7 <- theobject.fnctcallresult[]
  thevalueQ <- thevalue7
  if thevalueQ == truthobject then goto lbl462 else goto lbl463
lbl462:
  goto lbl464
lbl463:
  thevalue5 <- myvalue4
  theobject2 <- thevalue5
  returnvalue <- object consCellq159
    var head
    function getElement_617[] -> [something]
      l11194:
        something <- head
        return
    end getElement_617
```

```
    initially
      l11069:
        head <- element
        return
    end initially
  end consCellq159
  objectlist2 <- returnvalue
  theobject2.fnctcall_414[self, "plus", objectlist2]
  thevalue2 <- theobject2.fnctcallresult_623[]
  thevalue233 <- thevalue2
  myvalue4 <- thevalue233
  localenv <- object anenvironmentemptyq161
    initially
      first1:
        return
    end initially
  end anenvironmentemptyq161
  thevalue6 <- myvalue3
  theobject3 <- thevalue6
  thevalue <- myvalue4
  theValue2 <- thevalue
  returnvalue2 <- object consCellq159
    var head
    function getElement_617[] -> [something]
      l11194:
        something <- head
        return
    end getElement_617
    initially
      l11069:
        head <- theValue2
        return
    end initially
  end consCellq159
  objectlist3 <- returnvalue2
  theobject3.fnctcall[self, "eq", objectlist3]
  thevalue7 <- theobject3.fnctcallresult[]
  thevalueQ2 <- thevalue7
  if thevalueQ2 == truthobject then goto lbl465 else goto lbl466
lbl465:
  goto lbl467
lbl466:
  thevalue5 <- myvalue4
  theobject22 <- thevalue5
```

```
    returnvalue <- object consCellq159
      var head
      function getElement_617[] -> [something]
        l11194:
          something <- head
          return
      end getElement_617
      initially
        l11069:
          head <- element
          return
      end initially
    end consCellq159
    objectlist22 <- returnvalue
    theobject22.fnctcall_414[self, "plus", objectlist22]
    thevalue2 <- theobject22.fnctcallresult_623[]
    thevalue234 <- thevalue2
    myvalue4 <- thevalue234
    self.proccall_421[]
    goto lbl467
  lbl467:
    goto lbl464
  lbl464:
    thevalue3 <- myvalue4
    thevalue235 <- thevalue3
    myvalue <- thevalue235
    theresult <- myvalue
    fnctresult <- theresult
    fnctresult2 <- fnctresult
    programresult <- fnctresult2
    programresult <- programresult.getvalue_653[]
    return
end execute
function executeresult[] -> [result]
  l11484:
    result <- programresult
    return
end executeresult
procedure proccall_421[]
  var thevalueQ var theValue var objectlist var theobject var objectlist2
  var theobject2 var thevalue23
  l11262:
    localenv <- object anenvironmentemptyq161
      initially
```

```
      first2:
        return
    end initially
  end anenvironmentemptyq161
  thevalue6 <- myvalue3
  theobject <- thevalue6
  thevalue <- myvalue4
  theValue <- thevalue
  returnvalue2 <- object consCellq159
    var head
    function getElement_617[] -> [something]
      l11194:
        something <- head
        return
    end getElement_617
    initially
      l11069:
        head <- theValue
        return
    end initially
  end consCellq159
  objectlist <- returnvalue2
  theobject.fnctcall[self, "eq", objectlist]
  thevalue7 <- theobject.fnctcallresult[]
  thevalueQ <- thevalue7
  if thevalueQ == truthobject then goto lbl335 else goto lbl336
lbl335:
  goto Q549
lbl336:
  thevalue5 <- myvalue4
  theobject2 <- thevalue5
  returnvalue <- object consCellq159
    var head
    function getElement_617[] -> [something]
      l11194:
        something <- head
        return
    end getElement_617
    initially
      l11069:
        head <- element
        return
    end initially
  end consCellq159
```

```
      objectlist2 <- returnvalue
      theobject2.fnctcall_414[self, "plus", objectlist2]
      thevalue2 <- theobject2.fnctcallresult_623[]
      thevalue23 <- thevalue2
      myvalue4 <- thevalue23
      self.proccall_421[]
      goto Q549
    Q549:
      return
  end proccall_421
  function getvalue_618[] -> [result]
    l11205:
      result <- 0
      return
  end getvalue_618
  function fnctcallresult_623[] -> [result]
    l11226:
      result <- returnvalue3
      return
  end fnctcallresult_623
  initially
    l85:
      return
  end initially
end program
const truthobject == object truthobject
  initially
    first3:
      return
  end initially
end truthobject
;
```

Appendix C

The partial evaluator

C.1 The signatures

C.1.1 ABSVAL

```
(*$ABSVAL *)

(* Abstract Values:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION
        An abstract value is a set of values. A value can be either an object,
        a basevalue or nil. Nil is considered to be either an object or a
        basevalue - it depends on the occurrence of nil!



SEE ALSO
        DEBUG SIMILI

NOTES

RCS LOG
```

```
$Log:   ABSVAL.sml,v $
Revision 2.1  92/04/29  20:52:04  marquard
Working revision.

Revision 1.1  91/12/12  16:52:48  marquard
Initial revision

*)

signature ABSVAL =
  sig
    exception Value of string

    (* TYPES *)

      type value
      (* the set of object *)

      eqtype instream
      eqtype outstream

    (* LOCAL TYPES *)
      type oc
      eqtype AOID

      datatype OID =
          Static of AOID
        | Quantified of AOID
      (* used in ProgramState, rho: OID -> code * ienv *)

      type similiBaseValue
      (* Basevalues from the Simili signature *)

      datatype BaseValue =
          Integers (* quantify *)
        | Strings  (* quantify *)
        | Chars    (* quantify *)
        | Dynamic  (* Input Dynamic - but "kind" is assumed correct *)
        (* Dynamic only used internally in PeBaseValue ! *)
        | Integer of int
        | Boolean of bool
        | String of string
        | Char of string
```

```
      | Unix
      | InStream of instream
      | OutStream of outstream

    datatype object =
        AnObject of OID
      | ABaseValue of BaseValue
      | InputDynamic
    (* local value *)

  (* CONSTANTS *)

    val GlobalOID  : OID
    val GlobalOC   : oc
    val GlobalAOID : int
    (* GlobalOID is the same as nil, actualy = AnObject(Static(1)) *)

    val initOID   : unit -> unit
    val nextOID   : unit -> int
    (* initOID and nextOID is used to generate new oids ! *)

    val empty : value
    (* empty value set *)

    val bot   : value
    (* the bottom value, i.e. empty *)

    val pnil  : OID
    (* Nil, Static(0) *)

    val mynil : object
    (* Nil, AnObject(Static(0)) *)

    val aNil  : value
    (* Nil, {AnObject(Static(0))} *)

    val True  : value
    (* truth value, {ABaseValue(Boolean(true))} *)

    val False : value
    (* falsity value, {ABaseValue(Boolean(false))} *)

    val IDVal : value
    (* Input Dyunamic value, {InputDynamic} *)
```

```
  val aBoolean : value
  (* a boolean, either true or false, {True, False} *)

(* CONVERTERS *)

  val OID2string   : OID -> string
  (* OID2string oid, returns a string representation of an oid, in a way *)
  (* similar to the datatypes of ml, i.e. oid = AnObject(4) *)

  val oid2string   : OID -> string
  val string2oid   : string -> OID
  (* oid2string and string2oid is used to converte between oids and *)
  (* strings for internal use. For every oid, *)
  (* string2oid (oid2string oid) = oid *)

  val string       : value -> string
  (* returns a string representation of the value *)

  val list         : value -> object list
  (* returns a list containing the objects the value contains *)

  val fromobject   : object -> value
  (* Convert an object to a value, i.e. fromobject a = {a} *)

  val fromBaseValue: similiBaseValue -> value
  (* Convert basevalues found i the SIMILI signature, to internal *)
  (* basevalues. *)

  val bv2similibv  : BaseValue -> similiBaseValue
  (* Convert internal basevalues to basevalues found in the *)
  (* SIMILI signature. *)

  val object2string : object -> string
  (* returns a string representation of an object. *)

  val basevalues2value : BaseValue list -> value
  (* Returns a value, containing the basevalues *)

  val object2basevalue : value list -> BaseValue list
  (* takes a value list, and returns the corresponding basevalue list *)
  (* raises Value(s) if all objects are not ABaseValue! *)

  val object2basevalues : value list -> BaseValue list list
```

```
  (* takes a value list, and returns all basevalue lists possible *)
  (* raises Value(s) if all objects are not ABaseValue! *)

  val bv2value : BaseValue -> value
  (* Convert a basevalue to a value, i.e. bv2value bv = {bv} *)

  val bv2string : BaseValue -> string
  (* returns a string representation of a basevalue *)

(* OBSERVERS *)

  val arity : value -> int
  (* Returns the number of objects in a value *)

  val includes : value * value -> bool
  (* includes v1 v2, tests whether v1 is a subset of v2. *)

  val onlyBaseValue     : value -> bool
  (* tests whether the given value is a true basevalue, i.e. is a *)
  (* ABaseValue but excludes Integers, Strings, Chars, InStream, *)
  (* OutStream and Dynamic. *)

  val includesInputDynamic : value -> bool
  (* Tests whether a value contains InputDynamic *)

  val includesBaseValue : value -> bool
  (* Tests whehter the value contains an objects of the structure *)
  (* ABaseValue(_). *)

  val includesBaseValueorInputDynamic : value -> bool
  (* Tests whehter the value contains an objects of the structure *)
  (* ABaseValue(_) or InputDynamic. *)

  val lesseq : value * value -> bool
  (* lesseq v1 v2 is true, if ... *)

  val isStatic     : value -> bool
  (* isStatic value, returns true if value does only contain one object, *)
  (* and the object is a true basevalue or AnObject(Static(_)). *)

  val isQuantified : value -> bool
  (* isQuantified value, returns true if value does only contain one *)
  (* object, and the object is a quantified basevalue or *)
  (* AnObject(Quantified(_)). *)
```

```
  val equal    : value * value -> bool
  (* equal v1 v2, tests whether two values are equal or not. *)

  val disjoint : value * value -> bool
  (* disjoint v1 v2, tests whether two values are disjoint or not. *)

  val isBot    : value -> bool
  (* Tests whether a given value is the bottom value or not, i.e. *)
  (* isBot value = equal bot value. *)

  (*val lessOID  : OID * OID -> bool*)
  (* lessOID oid1 oid2, tests whether oid1 < oid2 !!! *)

(* SELECTORS *)

  val isStatica: value -> OID Option
  (* isStatica value, returns Some(oid) if the value is static. *)
  val isStaticOrQuantified: value -> OID Option

  val object   : value -> object
  (* returns the object of the value. *)
  (* Raises Value("impossible - empty object") if the set contains *)
  (* more than one object, or none. *)

  val objects  : value -> object list
  (* objects value, returns a list of objects. *)

  val oids     : value -> OID list
  (* oids value, returns a list of oids. *)
  (* Raises Value("An object expected") if the object is not an *)
  (* AnObject(oid) !! *)

(* MANIPULATORS *)

  val lubif    : value * value -> value
  (* lubs two values, used in conditionals. *)

  val lubloop  : value * value -> value
  (* lubs two values, used in loops. *)

  val select   : value -> object * value
  (* Selects an object from the value and returns the object and the *)
  (* resulting value. *)
```

```
    (* Raises Value("select - empty value") if the value is empty. *)

    val removeNil: value -> value
    (* returns a value without nil *)

end;
```

C.1.2 ARG

```
(*$ARG *)

(* Argument:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION
        Represents argument list, i.e. list of abstract values (AbsVal)!
        The functions found in AbsVal that can be applied to argument lists
        is defined in this signature.

SEE ALSO
        AbsVal

NOTES

RCS LOG

$Log:   ARG.sml,v $
Revision 2.1  92/04/29  20:54:49  marquard
Working revision.

Revision 1.1  92/03/05  17:20:30  marquard
Initial revision

*)

signature ARG =
  sig
    exception ARG of string

    (* LOCAL TYPES *)

      type absval
```

```
  (* CONSTANTS *)

    val empty : absval list
    (* return an empty argument list, i.e. [] *)

  (* CONVERTERS *)

    val string : absval list -> string
    (* returns a string representation of an argument list *)

  (* OBSERVERS *)

    val lesseq : absval list * absval list -> bool
    (* tests whether an argument list is lesseq another argument list *)
    (* Raises ARG("argument don't have the same length") if the two *)
    (* lists does not have the same length. *)

    val eq     : absval list * absval list -> bool
    (* tests whether two argument lists are equal. *)
    (* Raises ARG("argument don't have the same length") if the two *)
    (* lists does not have the same length. *)

  (* MANIPULATORS *)

    val lubif     : absval list * absval list -> absval list
    (* lub two argument lists, in a conditional. *)
    (* Raises ARG("argument don't have the same length") if the two *)
    (* lists does not have the same length. *)

    val lubloop   : absval list * absval list -> absval list
    (* lub two argument lists, in a loop. *)
    (* Raises ARG("argument don't have the same length") if the two *)
    (* lists does not have the same length. *)
end;
```

C.1.3 DELTA

```
(*$DELTA: *)

(* Explicator mapping.

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Wed Dec 18

Maintenance:    Author

DESCRIPTION
        Implements the semantic object, Explicit_Map, as a type and
        functions operation on this type.  Information is collected
        during partial evaluation, and when fetching the program from
        the object store, explicator information is collected from this
        module and put into the abstract syntax.  This is done in the
        Print module.

SEE ALSO
        TAU RHO ABSVAL

NOTES

RCS LOG

$Log:   DELTA.sml,v $
Revision 2.1  92/04/29  20:55:02  marquard
Working revision.

Revision 1.1  91/12/18  16:53:30  marquard
Initial revision

*)

signature DELTA =
  sig
    (* TYPES *)
      type delta
```

```
  (* LOCAL TYPES *)
    type reslbl (* = SIMILI.Label *)
    type stmnt  (* = SIMILI.Statement *)
    type name   (* = Simili.Name *)
    type rho    (* = Rho.rho *)
    type rhosimple (* = RHO.rhosimple *)
    type tau    (* = TAU.Tau *)

  (* CONSTANTS *)
    val empty: delta

  (* SELECTORS *)
    val lookup : delta * reslbl -> stmnt list Option
    val string : delta -> string

  (* MANIPULATORS *)

    val lubif        : delta * delta -> delta
    (* lubs two explicator mappings. *)

    val Explicator2r : rho * rhosimple * delta -> delta
    (* Explicator2r r1 r2 delta rho, compares values in r1 and r2, *)
    (* and generate explicators for different values. *)

    val Explicator2t : tau * tau * delta * rho * (string -> name) -> delta
    (* Explicator2r t1 t2 delta rho, compares values in t1 and t2, *)
    (* and generate explicators for different values. *)

    val Explicator4  : rho * tau * rhosimple * tau * delta ->
      delta
    (* Explicator4 r1 r2 t1 t2 delta rho, compares values in r1 and r2, *)
    (* and t1 and t2, and generate explicators for different values. *)
end;
```

C.1.4 EQ_MAP

```
(*$EQ_MAP: *)

(* Equality map/set!

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Tue Mar 3

Maintenance:    Author

DESCRIPTION
        The equality map is used to mark the objects that has been
        used in an equality-test in the program. It is not possible
        to merge two objects together, that can be tested for
        equality.

SEE ALSO
        SIMILI

NOTES

RCS LOG

$Log:   EQ_MAP.sml,v $
Revision 2.1  92/04/29]  20:56:27  marquard
Working revision.

Revision 1.1  92/03/03  17:27:54  marquard
Initial revision

*)

signature EQ_MAP =
  sig
    exception EQMap of string

    (* TYPES *)
      type equality_map
```

```
    type oc (* = SIMILI.OC *)

  (* VALUES *)
    val empty  : equality_map

  (* OBSERVERS *)
    val fetch  : equality_map * oc -> bool
    val string : equality_map -> string

  (* MANIPULATORS *)
    val lift   : equality_map * oc -> equality_map
    val lubif  : equality_map * equality_map -> equality_map
    val update : equality_map * oc * oc -> equality_map
end;
```

C.1.5 INT_BIN_MAP

```
(*$INT_BIN_MAP: General *)

signature INT_BIN_MAP =

(* FINITE MAPS - BINARY TREES WITH INTEGERS AS KEY.

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Wed Mar 6

Maintenance:    Author


DESCRIPTION


RCS LOG

$Log:   INT_BIN_MAP.sml,v $
Revision 2.1  92/04/29  23:19:43  marquard
Working revision

Revision 1.1  92/03/06  13:08:10  marquard
Initial revision

*)

sig
  exception IntBinMap of string

  (* TYPES *)

  type 'b Map

  (* VALUES *)

  val empty : 'b Map
  (* Empty map. *)
```

```
(* CREATORS *)

val singleton : int * 'b -> 'b Map
(* Map containing a single element. *)


(* OBSERVERS *)

val isEmpty: 'b Map -> bool
(* Empty test on maps. *)

val lookup  : 'b Map -> int -> 'b Option
(* Look up an element in a map - may fail, returning None. *)

val dom        : 'b Map -> int list
(* Domain of a map. Might contain duplicates. *)

val range      : 'b Map -> 'b list
(* Range of a map.  Might contain elements mapped to by
   duplicate keys. *)

val string     : ('b -> string) -> 'b Map -> string
(* Returns a string representation of a mapping. *)

(* MANIPULATORS *)

val add        : (int * 'b) -> 'b Map -> 'b Map
(* Add an element to a map, rendering any existing mapping from that
   value unavailable. *)

val plus       : 'b Map -> 'b Map -> 'b Map
(* Add two maps together. Entries in the second map override entries
   on the first one (cf. the various plus operations in the SML
   semantics). *)

val composeMap : ('b -> 'c) -> 'b Map -> 'c Map
(* Is this an appropriate name? Apply a function to all elements of
   the range. *)

val composeMap' : ('b -> 'c Option) -> 'b Map -> 'c Map
(* Is this an appropriate name? Apply a function to all elements of
   the range. Returning None means that the element will be removed. *)
```

```
  val fold       : (('a * 'b) -> 'b) -> 'b -> 'a Map -> 'b
  (* Rather like the list fold operation - operates on the range of
     a map. Order of elements not guaranteed. Also, suffers from the
     duplicate problem above. *)

  val fold'      : (((int * 'b) * 'c) -> 'c)-> 'c -> 'b Map -> 'c
  (* More complex fold, with a function from (dom, range) element pairs
     to some arbitrary type. *)

  val cut        : 'a Map -> 'b Map -> 'b Map
  (* cut a b, cut the second mapping b, so it only contains elements in the
     domain of the first mapping, a. If the first mapping contains elements
     not found in the second, an exception is raised. *)

  val cut'        : 'a Map -> 'b Map -> 'b Map
  (* cut' a b, cut the second mapping b, so it only contains elements in the
     domain of the first mapping, a. If the first mapping contains elements
     not found in the second, the element is simple removed! *)


  val twofoldand:
    ('b -> 'c -> bool) -> 'b Map -> 'c Map -> bool
  (* twofoldand myfun a b, applies the function myfun to all elements in
     both mappings. If the domains of the mappings are not equal the function
     returns false. *)

  val threefoldand:
    ('b -> 'c -> 'd -> bool) -> 'b Map -> 'c Map -> 'd Map -> bool

  val twofoldand':
    ('b Option -> 'c Option -> bool) -> 'b Map -> 'c Map -> bool
  (* Like twofoldand, but used when the domains are not equal *)

  val threefoldand':
    ('b Option -> 'c Option -> 'd Option -> bool) ->
    'b Map -> 'c Map -> 'd Map -> bool

(*  val twofold:
    (('b -> 'b -> 'c) -> 'c) -> 'c -> 'b Map -> 'b Map -> 'c*)

  val twofold':
    ('b Option -> 'c Option -> 'd -> 'd) -> 'd -> 'b Map -> 'c Map -> 'd

  val mergeMap: (('b * 'b) -> 'b) -> 'b Map -> 'b Map -> 'b Map
```

```
  (* Merges two finite maps, with a composition function to apply
     to the range elements of domain elements which clash.  In such
     a case the first argument to the compose function is the range
     element of the first map argument to mergeMap.  *)

  val mergeMap': (('b Option * 'c Option) -> 'd Option) ->
    'b Map -> 'c Map -> 'd Map
  (* Merges two finite maps, with a composition function to apply
     to the range elements of domain elements which clash.  In such
     a case the first argument to the compose function is the range
     element of the first map argument to mergeMap.  *)

  val mergeMap'': (('b Option * 'c Option * 'd Option) -> 'e Option) ->
    'b Map -> 'c Map -> 'd Map -> 'e Map
  (* Merges three finite maps, with a composition function to apply
     to the range elements of domain elements which clash.  In such
     a case the first argument to the compose function is the range
     element of the first map argument to mergeMap.  *)
end;
```

C.1.6 LABELBACKWARD

```
(*$LABELBACKWARD: *)

(* Label Backward

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science,
                University of Copenhagen,
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        RHO TAU SIMILI

NOTES

RCS LOG

$Log:   LABELBACKWARD.sml,v $
Revision 2.1  92/04/29  20:56:42  marquard
Working revision.

Revision 1.1  91/12/12  13:13:07  marquard
Initial revision

*)

signature LABELBACKWARD =
  sig
    (* GENERAL *)
      exception LABELBACKWARD of string

    (* TYPES *)
      type lb

    (* LOCAL TYPES *)
      eqtype label   (* = SIMILI.Label *)
      eqtype reslbl  (* = SIMILI.Label *)
```

```
    type rho       (* = RHO.Rho *)
    type tau       (* = TAU.Tau *)

  (* CONSTANTS *)
    val empty    : lb
    val makeExit : lb

  (* CREATORS *)
    val singleton : label * reslbl * rho * tau -> lb

  (* OBSERVERS *)
    val isEmpty   : lb -> bool
    val member    : lb * label * reslbl -> bool
    val isExit    : lb -> bool
    val string    : lb -> string

  (* SELECTORS *)
    val selectRT  : lb * label * reslbl -> (rho * tau)

  (* MANIPULATORS *)
    (*val add       : lb * label * reslbl * rho * tau -> lb*)
    val union     : lb * lb -> lb
    val remove    : lb * label * reslbl -> lb
end;
```

C.1.7 LABELFORWARD

```
(*$LABELFORWARD *)

(* Label Forward:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        RHO TAU SIMILI

NOTES

RCS LOG

$Log:   LABELFORWARD.sml,v $
Revision 2.1  92/04/29  20:56:50  marquard
Working revision.

Revision 1.1  91/12/12  13:12:16  marquard
Initial revision

*)

signature LABELFORWARD =
  sig
    (* TYPES *)
      type lf

    (* LOCAL TYPES *)
      datatype LfTag = Iter | Unfold
      eqtype label   (* = SIMILI.Label *)
      eqtype reslbl  (* = SIMILI.Label *)
      type rho       (* = RHO.Rho *)
      type rhosimple (* = Rho.rhosimple *)
```

```
    type tau       (* = Tau.Tau *)

  (* CONSTANTS *)
    val empty  : lf

  (* OBSERVERS *)
    val lookup : lf * LfTag * label -> (reslbl * rhosimple * tau) Option
    val string : lf -> string

    val exists_unfold : lf * label * rho * tau -> reslbl Option

  (* MANIPULATORS *)
    val add          : lf * LfTag * label * reslbl * rho * tau -> lf
    val adddependent : lf * LfTag * label * reslbl * rho * tau -> lf
    val remove       : lf * LfTag * label * reslbl -> lf
end;
```

C.1.8 METHODBACKWARD

```
(*$METHODBACKWARD *)

(* Method Backward :

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   METHODBACKWARD.sml,v $
Revision 2.1  92/04/29  20:57:02  marquard
Working revision.

Revision 1.1  91/12/12  13:14:48  marquard
Initial revision

*)

signature METHODBACKWARD =
  sig
    exception MethodBackward of string

    (* TYPES *)
    type mb
    type mfsimple

    (* LOCAL TYPES *)
    eqtype oid
    eqtype object
    eqtype MethodName
```

```
  eqtype ResMethodName
  type rho
  type absval
  datatype MbTag = Use | Redo

  (* CONSTANTS *)
  val empty    : mb
  val makeExit : mb

  (* CREATORS *)
  val singleton :
    MbTag * oid * MethodName * ResMethodName * rho * absval list -> mb

  val string : mb -> string

  (* OBSERVERS *)
  val isEmpty : mb -> bool
  val isExit  : mb -> bool
  val isCreated : mb -> bool
  val exist   : mb * MbTag -> bool
  val member  : mb * MethodName * oid * ResMethodName -> bool

  (* SELECTORS *)
  val selectRB : mb * MbTag * MethodName * oid * ResMethodName ->
    (rho * absval list)

  (* MANIPULATORS *)
  val lubseq    : mb * mb -> mb
  val lubif     : mb * mb -> mb
  val remove    : mb * MethodName * oid * ResMethodName -> mb
  val removeExit: mb -> mb

  val raiseCreated : mb -> mb
  val get_container : mb -> mfsimple
  val set_container   : mb * mfsimple -> mb
end;
```

C.1.9 METHODFORWARD

```
(*$METHODFORWARD *)

(* Method Forward:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   METHODFORWARD.sml,v $
Revision 2.1  92/04/29  20:57:12  marquard
Working revision.

Revision 1.1  91/12/12  13:13:57  marquard
Initial revision

*)

signature METHODFORWARD =
  sig
    exception MethodForward of string

    (* TYPES *)
    type mf
    type mfsimple

    (* LOCAL TYPES *)
    type pi
    type oc
    datatype MethodTag = Iter | Unfold | Any
```

```
  eqtype oid
  eqtype MethodName
  eqtype ResMethodName
  type rho
  type rhosimple
  type absval

  (* CONSTANTS *)
  val empty  : mf
  val emptysimple  : mfsimple

  (* OBSERVERS *)
  val string : mf -> string
  val fullstring : mf -> string
  (* full string also print the rho-after of the iter-tuples *)

  val exists_unfold : mf * oid * MethodName * rho * absval list ->
    (ResMethodName * rhosimple Option * absval) Option

  val mf2simple : mf -> mfsimple

  (* SELECTORS *)
  val contains : mf * MethodTag * oid * MethodName ->
    (ResMethodName *
     rhosimple * absval list *
     rhosimple Option * absval) Option

  val lookup : mf * oid * MethodName * ResMethodName ->
    (MethodTag * rhosimple * absval)

  val buildpi : mf * oid * oc * rho -> pi

  (* MANIPULATORS *)
  val add        : mf * MethodTag * oid * MethodName * ResMethodName *
    rho * absval list * rho Option * absval -> mf
  val adddependent : mf * MethodTag * oid * MethodName * ResMethodName *
    rho * absval list * rho Option * absval -> mf
  val remove     :  mf * MethodTag * oid * MethodName * ResMethodName -> mf

  val set_container : mf * mfsimple -> mf
  val lub_exclusive : mfsimple * mfsimple -> mfsimple
end;
```

C.1.10 PE

```
(*$PE *)

(* Partial evaluator for Simili,

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        ABSVAL ARG RHO TAU LABELFORWARD LABELBACKWARD METHODFORWARD
        METHODBACKWARD UTIL DELTA QMAP EQ_MAP PEBASEVALUE PRINT

NOTES

RCS LOG

$Log:   PE.sml,v $
Revision 2.1  92/04/29  20:57:23  marquard
Working revision.

Revision 1.1  91/12/12  15:10:16  marquard
Initial revision


*)

signature PE =
  sig
    exception PE of string;

    (* TYPES *)
      datatype value =
        GOWName of string
      | InputDynamic
      | Integer of int
```

```
    | Integers
    | Boolean of bool
    | Char of string
    | Chars
    | String of string
    | Strings
    | Unix
    | NoValue

  (* LOCAL TYPES *)
    type program       (* = SIMILI.Program *)
    type procname      (* = string *)
    type fnctname      (* = string *)
    type equality_map  (* = EQ_MAP.equality_map *)

  (* MANIPULATORS *)
    val pe : program * string * procname * value list * fnctname ->
      program

    val be : program * string * procname * value list * fnctname ->
      program * equality_map
    (* be must nearly always be used. The equality_map is important *)
    (* in the postphase. The program must always be treated by the *)
    (* postphase to make sure the constant declarations is sorted! *)

end;
```

C.1.11 PEBASEVALUE

```
(*$PEBASEVALUE *)

(* Built in objects:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION
        This module implements functions on built in objects.

SEE ALSO

NOTES

RCS LOG

$Log:   PEBASEVALUE.sml,v $
Revision 2.1  92/04/29  20:57:33  marquard
Working revision.

Revision 1.1  91/12/12  17:23:29  marquard
Initial revision

*)

signature PEBASEVALUE =
  sig
    exception BaseValue of string

    (* LOCAL TYPES *)
    eqtype BaseValue
    type absval

    (* CONVERTERS *)
    val absval2basevalue : absval list -> BaseValue list
```

```
  (* MANIPULATORS *)
  val DoFnctCall : BaseValue * string * absval list -> absval
  val doFnctCall : BaseValue * string * BaseValue list -> BaseValue list
end;
```

C.1.12 PI

```
(*$PI *)

(* Pi: Object constructor creation loops!

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Sat Mar 7

Maintenance:    Author

DESCRIPTION

SEE ALSO
        ABSVAL SIMILI

NOTES

RCS LOG

$Log:   PI.sml,v $
Revision 2.1  92/04/29  20:58:08  marquard
Working revision.

Revision 1.1  92/03/07  10:47:59  marquard
Initial revision

*)

signature PI =
  sig
    exception Pi of string

    (* TYPES *)
      type pi

    (* LOCAL TYPES *)
      type oid    (* = ABSVAL.OID *)

    (* CONSTANTS *)
```

```
    val empty  : pi

  (* CONVERTERS *)
    val dom    : pi -> oid list
    (* Domain of the environment. *)

    val string : pi -> string
    (* returns a string representation of an environment. *)

  (* OBSERVERS *)

  (* CREATORS *)
    val add: pi * oid * string * string * oid -> pi

  (* SELECTORS *)

    val contains: pi * oid * string -> (string * oid) Option

  (* MANIPULATORS *)
    val lub: pi * pi -> pi
    val remove: pi * string * string * oid -> pi
end;
```

C.1.13 POSTPHASE

```
(*$POSTPHASE *)

(* Postprocess a program

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Tue Feb 25

Maintenance:    Author

DESCRIPTION

SEE ALSO
        SIMILI EQ_MAP GARBAGE_COLLECT UNFOLD

NOTES

RCS LOG

$Log:   POSTPHASE.sml,v $
Revision 2.1  92/04/29  20:58:18  marquard
Working revision.

Revision 1.1  92/03/05  16:54:11  marquard
Initial revision

*)

signature POSTPHASE =
  sig
    exception Postphase of string

    (* LOCAL TYPES *)
      type program      (* = SIMILI.Program *)
      type equality_map (* = EQ_MAP.equality_map *)

    (* CONVERTERS *)
      val doit : program * string * string * string * equality_map -> program
  end;
```

C.1.14 QMAP

```
(*$QMAP *)

(* Quantified mapping

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   QMAP.sml,v $
Revision 2.1  92/04/29  20:58:27  marquard
Working revision.

Revision 1.1  91/12/12  17:21:14  marquard
Initial revision

*)

signature QMAP =
  sig
    (* TYPES *)
    type qmap

    (* LOCAL TYPES *)
    type oc
    type OID
    type tau
    (*type ienv*)

    (* CONSTANTS *)
```

```
  val empty  : qmap

  (* OBSERVERS *)
  val string : qmap -> string

  (* SELECTORS *)
  val lookup : qmap * oc -> (OID * tau (* ienv*)) Option

  (* MANIPULATORS *)
  val update : qmap * oc * (OID * tau (* ienv*)) -> qmap

  val lub   : qmap * qmap -> qmap
  val lubif : qmap * qmap -> qmap
end;
```

C.1.15 RHO

```
(*$RHO *)

(* Rho: Object Store

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        TAU ABSVAL UTIL QMAP

NOTES

RCS LOG

$Log:   RHO.sml,v $
Revision 2.1  92/04/29  20:58:38  marquard
Working revision.

Revision 1.1  91/12/12  15:09:57  marquard
Initial revision

*)

signature RHO =
  sig
    exception Rho of string

    (* TYPES *)
      type rho
      (* the type of the abstract object store. *)

      type code
      (* is a mapping from method names into method bodies. *)
```

```
  type rescode
  (* The type of the residual code collected in the object store. *)

  type rhosimple
  (* is a simplified version of the object store. *)

(* LOCAL TYPES *)
  type 'a Set             (* = EqSet.Set *)
  eqtype name             (* = string *)
  type utilstore          (* = UTIL.store *)
  type ienv               (* = TAU.tau *)
  type tau                (* = TAU.tau *)
  type methodname         (* = string *)
  eqtype oid              (* = ABSVAL.oid *)
  eqtype aoid             (* = ABSVAL.AOID *)
  type object             (* = ABSVAL.object *)
  type oc                 (* = SIMILI.OC *)
  type absval             (* = ABSVAL.value *)
  eqtype gowname          (* = string *)
  type BasicExpression    (* = SIMILI.BasicExpression *)
  type Name               (* = SIMILI.Name *)
  type mdef               (* = SIMILI.MethodDef *)
  type body               (* = SIMILI.Body *)
  type stmnt              (* = SIMILI.Statement *)
  type reslbl             (* = SIMILI.Label *)
  type qmap               (* = QMAP.qmap *)

  datatype precond =
    Pre of rhosimple * absval list * bool * bool
  datatype procpostcond =
    PPC of rhosimple * qmap * utilstore
  datatype fnctpostcond =
    FPC of absval * rhosimple * qmap * utilstore

  datatype resproc =
    ResProc of precond * methodname * mdef * procpostcond
  | NoPReuse of methodname * mdef

  datatype resfnct =
    ResFnct of precond * methodname * mdef * fnctpostcond
  | NoFReuse of methodname * mdef

  datatype alias =
    AliasProcedure of methodname * methodname
```

```
    | AliasFunction of methodname * methodname

    datatype ProcReuse =
      PNoUse
    | PReused of string Option * rho * utilstore
    | PUseNoReuse of rhosimple * absval list * qmap

    datatype FnctReuse =
      FNoUse
    | FReused of string Option * rho * absval * utilstore
    | FUseNoReuse of rhosimple * absval list * qmap

  (* CONSTANTS *)
    val empty       : rho
    val emptysimple : rhosimple

  (* CONVERTERS *)
    val dom         : rho -> oid list
    val string      : rho -> string
    val simple2string : rhosimple -> string
    val getivars    : rho * oid -> string list
    val fullystring : rho -> string
    val OID2string  : rho * oid -> string

  (* OBSERVERS *)
    (*val samedom   : rho * rho -> bool*)
    val subdomutil  : utilstore * rho -> bool
    val isEmpty     : rho -> bool
    (*val arity     : rho -> int*)
    val equal       : rho * rho -> bool
    val equalsimple : rho * rhosimple -> bool
    val lesseq      : rho * rho -> bool
    val lesseqsimple: rho * rhosimple -> bool
    val global      : rho * absval -> bool (* static or basevalue *)

  (* SELECTORS *)
    val lookup      : rho * oid * name -> absval
    val lookuptest  : rho * oid * name -> absval Option
    val code        : rho * oid -> mdef list
    val rescode     : rho * oid ->
      string * (string * resproc list) list *
      (string * resfnct list) list * alias Set * body
    val ienv        : rho * oid -> ienv
    val ienvsafe    : rho * oid -> ienv Option
```

```
  val rho2simple  : rho -> rhosimple
  val rho2cutsimple: rho * utilstore -> rhosimple
  val difference  : rho * rhosimple -> rho

  val diff        : rhosimple * rhosimple -> rhosimple
  val patch       : rhosimple * rhosimple -> rhosimple
  (* diff and patch are used in MethodForward *)

  val rho2rho     : rho -> rho
  (* Removes rescode from rho *)

(* static or basevalue *)
  val Globalizable: rho * absval -> BasicExpression Option
  val getname     : rho * absval -> Name
  val getbasicexpr: rho * absval -> BasicExpression
  val getoc       : rho * oid -> oc

(* MANIPULATORS *)
  val CombRho     : rho * rho -> rho

  val addcode     : rho * oid * oc * mdef list -> rho
  val procedurealias :
    rho * oid * string * string -> rho
  val functionalias  :
    rho * oid * string * string -> rho

  val updateIvars : rho * oid * reslbl -> rho
  val updateRho   : rho * rho -> rho
  val updateRhosimple   : rho * rhosimple -> rho
  val updateRho0  : rho -> rho
  val update      : rho * oid * name * (absval * reslbl list) -> rho
  (*val updateall   : rho -> oid ->
    mdef list * ienv *
    (string * (string * resproc list) list *
     (string * resfnct list) list * alias Set * body) ->
    rho*)
  val updateQMAPRHO : rho * oid * oc * mdef list -> rho

  val updateResCode: rho * oid *
    (string * (string * resproc list) list *
     (string * resfnct list) list * alias Set * body) ->
    rho
  val setName     : rho * oid * string -> rho
```

```
  val AddProc     : (string * resproc list) list * string * resproc ->
    (string * resproc list) list
  val AddFnct     : (string * resfnct list) list * string * resfnct ->
    (string * resfnct list) list

  (* Routines to force reuse of a method, in the case of hardcalls *)
  val ReuseProc   :
    rho * oid * string * absval list * string * bool * bool ->
    ProcReuse
  val ReuseFnct   :
    rho * oid * string * absval list * string * bool * bool ->
    FnctReuse

  val lubif       : rho * rho -> rho
  val lubifsim    : rhosimple * rho -> rho
  val lubifsimple : rhosimple * rhosimple -> rhosimple
  val lubloop     : rho * rho -> rho
  val lubienv     : ienv * ienv -> ienv
  val lubloopsim  : rho * rhosimple -> rho
  val lubloopsimple : rhosimple * rhosimple -> rhosimple

  val SQ          : rho -> rho
  (* Fetch all static or quantified objects. Remove all global objects *)

(* ITERATORS *)
  (*val fold        : (rho -> rho -> rho) -> rho -> rho list -> rho*)
  val fold'       :
    (((aoid *
       (oc * code * ienv * (string * (string * resproc list) list *
                            (string * resfnct list) list *
                            alias Set * body))) * 'c) -> 'c) ->
    'c -> rho -> 'c

  val twofold':
    ((oc * code * ienv * rescode) Option ->
     (oc * code * ienv * rescode) Option ->
     'd -> 'd) -> 'd -> rho -> rho -> 'd

  val twofoldsimple':
    (ienv Option -> ienv Option -> 'd -> 'd) -> 'd ->
    rhosimple -> rhosimple -> 'd

  val mergeMap':
    (((oc * code * ienv * rescode) Option *
```

```
          (oc * code * ienv * rescode) Option) ->
         (oc * code * ienv * rescode) Option) -> rho -> rho -> rho
  end;
```

C.1.16 SIMILI

```
(*$SIMILI *)

(* The abstract representation of the simili language.

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   SIMILI.sml,v $
Revision 2.1  92/04/29  23:13:00  marquard
Working revision

Revision 1.1  91/12/12  16:12:37  marquard
Initial revision


*)

signature SIMILI =
  sig
    (* TYPES *)
    datatype Program =
      SPROGRAM of ConstDeclaration list
    and ConstDeclaration =
      SCONSTDECLARATION of string * Expression
    and Expression =
      SBASICEXPR of BasicExpression
      | SEQUALEXPR of BasicExpression * BasicExpression
      | SFNCTCALLEXPR of BasicExpression * string * BasicExpression list
```

```
  | SOBJECTEXPR of string * string list * MethodDef list * Body
and BasicExpression =
  SNAMEEXPR of Name
  | SBASEVALUENAMEEXPR of BaseValue
and Name =
  SSELFNAME
  | SCONSTNAME of string
  | SIVARNAME of string
  | SLVARNAME of string
  | SFORMALPAR of string
  | SRESULTNAME of string
  | UNKNOWN of string (* used for assignment statement *)
and MethodDef =
  SPROCEDURE of string * string list * string list * Body
  | SFUNCTION of string * string list * string * string list * Body
and Body =
  SBODY of BasicBlock list
and BasicBlock =
  SBB of Label * Statement list * GotoBasicBlock
and Label =
  SLABEL of string
and Statement =
  SASSIGNSTMNT of Name * Expression
  | SPROCCALLSTMNT of BasicExpression * string * BasicExpression list
  | SSKIPSTMNT
and GotoBasicBlock =
  SGOTO of Label
  | SIF of Expression * Label * Label
  | SRETURN
and BaseValue =
  SINT of int
  | SSTR of string
  | SCH of string
  | SBOOL of bool
  | SNIL
  | SUNIX

(* CONVERTERS *)
val print : string -> Program -> unit
val exp2string : Expression -> string
val oc2string : string -> string
val BaseValue2string : BaseValue -> string
val mdef2string : MethodDef -> string
val stmnt2string : Statement -> string
```

```
  val body2string : Body -> string
end;
```

C.1.17 SPECIALIZER

```
(*$SPECIALIZER: *)

(*

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Tue Mar 3

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   SPECIALIZER.sml,v $
Revision 2.1  92/04/29  20:58:48  marquard
Working revision.

Revision 1.1  92/03/05  17:22:45  marquard
Initial revision

*)

signature SPECIALIZER =
  sig
    (* Postphase flags *)

      val do_postphase        : bool ref
      (* execute the postphase?, initially true *)

      val collect_garbage     : bool ref
      (* Collect garbage?, initially true *)

      val merge_objects       : bool ref
      (* Merge objects?, initially true *)
```

```
    val unfold_methods      : bool ref
    (* Unfold methods?, initially true *)

    val fully_unfold_methods      : bool ref
    (* Unfold methods?, initially true *)

    val compress_transition : bool ref
    (* Compress transition, initially true *)

    val double_collect      : bool ref
    (* Collect garbage both before and after method unfolding and *)
    (* transition compression, initially true *)

    val string : unit -> string
    (* Returns a string of the flags *)

    val status : unit -> unit
    val print  : unit -> unit
    (* Print out the flags on std_out. *)
end;
```

C.1.18 TAU

```
(*$TAU *)

(* Tau: Local Environment

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        ABSVAL SIMILI

NOTES

RCS LOG

$Log:   TAU.sml,v $
Revision 2.1  92/04/29  20:58:56  marquard
Working revision.

Revision 1.1  91/12/12  15:10:05  marquard
Initial revision

*)

signature TAU =
  sig
    exception Tau of string

    (* TYPES *)
      type tau
      type tausimple

    (* LOCAL TYPES *)
      eqtype name (* = string *)
      type absval (* = ABSVAL.value *)
```

```
  type reslbl (* = SIMILI.Label *)

(* CONSTANTS *)
  val empty  : tau

(* CONVERTERS *)
  val dom    : tau -> name list
  (* Domain of the environment. *)

  val string : tau -> string
  (* returns a string representation of an environment. *)

(* OBSERVERS *)
  val lesseq : tau * tau -> bool
  val equal  : tau * tau -> bool

(* CREATORS *)
  val build  : name list * absval list * reslbl -> tau
  (* Build a tau from a list of names and a list of values. *)
  (* Raises Tau("Build -  name and value list must have same length") if *)
  (* the two lists does not have the same length. *)

(* SELECTORS *)
  val lookup       : tau * name -> absval
  (* Lookup a name in an environment. Returns the ABSVAL.bot if the name *)
  (* does not exists in the environment. *)

  val lookuptest   : tau * name -> absval Option
  (* Lookup a name in an environment. Returns None of the name does not *)
  (* exists in the environment. *)

  val getvalue     : tau * name -> (absval * reslbl list)
  val getvaluesafe : tau * name -> (absval * reslbl list) Option
  (* getvalue and getvaluesafe is used for lookup with explicators. *)
  (* Raises Tau("getvalue - unknown name <name>") if the name does not *)
  (* occur in the environment - only for getvalue. *)

(* MANIPULATORS *)
  val lub          : tau * tau -> tau
  (* lub two environment. *)

  val lubif        : tau * tau -> tau
  (* lub two environment. *)
```

```
    val lubloop      : tau * tau -> tau
    (* lub two environment. *)

    val update       : tau * name * (absval * reslbl list) -> tau
    val updateVars   : tau * reslbl -> tau

    val fold'       :
      (((name * (absval * reslbl list)) * 'a) -> 'a) -> 'a -> tau -> 'a

end;
```

C.1.19 UTIL

```
(*$UTIL: *)

(* Utilization mapping:

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO

NOTES

RCS LOG

$Log:   UTIL.sml,v $
Revision 2.1  92/04/29  20:59:04  marquard
Working revision.

Revision 1.1  91/12/12  17:19:13  marquard
Initial revision

*)

signature UTIL =
  sig
    exception Util of string

    (* TYPES *)
    type Store

    (* LOCAL TYPES *)
    type ivar
    type AOID
    type OID
    type Info (* datatype UtilInfo = Unused | Defined | Used *)
```

```
type Env
type 'a Map

(* CONSTANTS *)
val empty: Store

(* OBSERVERS *)
val string    : Store -> string
val isUsed    : Store * OID * ivar -> bool
val isDefined : Store * OID * ivar -> bool
val EnvUsed   : Env * ivar -> bool
val EnvDefined: Env * ivar -> bool

(* SELECTORS *)
val dom       : Store -> OID list
val ivars     : Store * OID -> ivar list
val usedivars : Store * OID -> ivar list
val fetchenv  : Store * OID -> Env Option
val EnvDom    : Env -> ivar list (* Only Used & Defined ivars *)

(* MANIPULATORS *)
val update        : Store * OID * ivar * Info -> Store
val updateif      : Store * OID * ivar * Info -> Store
val updateDefined : Store * OID * ivar -> Store
val updateUsed    : Store * OID * ivar -> Store

val lub   : Store * Store -> Store
val lubif : Store * Store -> Store

val fold'       : (((int * Env) * 'a) -> 'a)-> 'a -> Store -> 'a
(* Complex fold, with a function from (dom, range) element pairs
   to some arbitrary type. *)

val twofoldand':
  (Env Option -> 'a Option -> bool) -> Store -> 'a Map -> bool

val threefoldand':
  (Env Option -> 'a Option -> 'b Option -> bool) ->
  Store -> 'a Map -> 'b Map -> bool

val mergeMap'': ((Env Option * 'a Option * 'b Option) -> 'c Option) ->
  Store -> 'a Map -> 'b Map -> 'c Map

val cut : Store -> 'a Map -> 'a Map
```

```
end;
```

C.2 The partial evaluator

```
(*$Pe: PE ABSVAL ARG RHO TAU LABELFORWARD LABELBACKWARD METHODFORWARD
       METHODBACKWARD DEBUG UTIL DELTA QMAP EQ_MAP EQ_FIN_MAP ListPair
       PEBASEVALUE PRINT PI EQ_SET *)

(*

Created by:     Morten Marquard and Bjarne Steensgaard
                Department of Computer Science
                University of Copenhagen
                Denmark
                marquard@diku.dk, rusa@diku.dk

Date:           Thu Dec 12

Maintenance:    Author

DESCRIPTION

SEE ALSO
        ABSVAL, ARG, DEBUG, DELTA, EQ_MAP, LABELBACKWARD, LABELFORWARD,
        METHODBACKWARD, METHODFORWARD, PEBASEVALUE, PI, PRINT, QMAP,
        RHO, TAU, UTIL

NOTES

RCS LOG

$Log:   Pe.sml,v $
Revision 2.1  92/04/29  20:57:44  marquard
Working revision.

Revision 1.1  91/12/12  15:07:23  marquard
Initial revision

*)

functor Pe(structure Simili : SIMILI
           structure EqSet : EQ_SET
           structure AbsVal : ABSVAL
           sharing type AbsVal.similiBaseValue = Simili.BaseValue
               and type AbsVal.AOID = int
               and type AbsVal.oc = string
```

```
structure PeBaseValue : PEBASEVALUE
sharing type PeBaseValue.BaseValue = AbsVal.BaseValue
  and type PeBaseValue.absval = AbsVal.value
structure Arg : ARG
sharing type Arg.absval = AbsVal.value
structure Tau : TAU
sharing type Tau.name = string
        and type Tau.absval = AbsVal.value
        and type Tau.reslbl = Simili.Label
structure Pi : PI
sharing type Pi.oid = AbsVal.OID
structure Rho : RHO
sharing type Rho.oid = AbsVal.OID
    and type Rho.oc = string
    and type Rho.object = AbsVal.object
    and type Rho.name = Tau.name = string
    and type Rho.absval = AbsVal.value
    and type Rho.gowname = Simili.BasicExpression
    and type Rho.Name = Simili.Name
    and type Rho.BasicExpression = Simili.BasicExpression
    and type Rho.mdef = Simili.MethodDef
    and type Rho.body = Simili.Body
    and type Rho.stmnt = Simili.Statement
    and type Rho.reslbl = Simili.Label
    and type Rho.ienv = Tau.tau
    and type Rho.tau = Tau.tau
    and type Rho.methodname = string
    and type Rho.Set = EqSet.Set
structure LabelForward : LABELFORWARD
sharing type LabelForward.label = Simili.Label
    and type LabelForward.reslbl = Simili.Label
    and type LabelForward.rho = Rho.rho
    and type LabelForward.rhosimple = Rho.rhosimple
    and type LabelForward.tau = Tau.tau
structure LabelBackward : LABELBACKWARD
sharing type LabelBackward.label = Simili.Label
    and type LabelBackward.reslbl = Simili.Label
    and type LabelBackward.rho = Rho.rho
    and type LabelBackward.tau = Tau.tau
structure MethodForward : METHODFORWARD
sharing type MethodForward.oid = AbsVal.OID
    and type MethodForward.oc = string
    and type MethodForward.pi = Pi.pi
    and type MethodForward.MethodName = string
```

```
            and type MethodForward.ResMethodName = string
            and type MethodForward.rho = Rho.rho
            and type MethodForward.rhosimple = Rho.rhosimple
            and type MethodForward.absval = AbsVal.value
        structure MethodBackward : METHODBACKWARD
        sharing type MethodBackward.oid = AbsVal.OID
            and type MethodBackward.object = AbsVal.object
            and type MethodBackward.MethodName = string =
                MethodForward.MethodName
            and type MethodBackward.ResMethodName = string =
                MethodForward.ResMethodName
            and type MethodBackward.rho = Rho.rho
            and type MethodBackward.absval = AbsVal.value
            and type MethodBackward.mfsimple =
              MethodForward.mfsimple
        structure Util : UTIL
        sharing type Util.ivar = string
            and type Util.OID = AbsVal.OID
            and type Util.Store = Rho.utilstore
        structure Delta : DELTA
        sharing type Delta.reslbl = Simili.Label
            and type Delta.stmnt = Simili.Statement
            and type Delta.name = Simili.Name
            and type Delta.rho = Rho.rho
            and type Delta.rhosimple = Rho.rhosimple
            and type Delta.tau = Tau.tau
        structure Qmap : QMAP
        sharing type Qmap.oc = string
            and type Qmap.OID = AbsVal.OID
            and type Qmap.tau = Tau.tau
                (*and type Qmap.ienv = Tau.tau*)
            and type Qmap.qmap = Rho.qmap
        structure EQMap : EQ_MAP
        sharing type EQMap.oc = string
        structure Print : PRINT
        sharing type Print.rho = Rho.rho
            and type Print.program = Simili.Program
            and type Print.delta = Delta.delta
            and type Print.cd = Simili.ConstDeclaration
        structure Debug : DEBUG
            ) : PE =
struct
  exception PE of string;
```

```
structure Simili = Simili
structure AbsVal = AbsVal
structure Arg    = Arg
structure Rho    = Rho
structure Tau    = Tau
structure LF     = LabelForward
structure LB     = LabelBackward
structure MF     = MethodForward
structure MB     = MethodBackward
structure Util   = Util
structure Delta  = Delta
structure Qmap   = Qmap

(* Types *)
type program       = Simili.Program
type absval        = AbsVal.value
type procname      = string
type fnctname      = string
type MethodName    = MethodForward.MethodName
type ResMethodName = MethodName
type rho           = Rho.rho
type equality_map  = EQMap.equality_map

(* input values *)
datatype value =
  GOWName of string
| InputDynamic
| Integer of int
| Integers
| Boolean of bool
| Char of string
| Chars
| String of string
| Strings
| Unix
| NoValue

(* Local Types *)
datatype aCall =
  CALL of MethodName * ResMethodName * rho * absval list * string Option
                                                    (* ResultName *)

(*************************************************************************)
(* Local operators for the semantics equation system *)
```

```
val nextlabel = ref 0;
fun initlabel () =
  nextlabel := 0
fun  MakeLabel (s) =
  (nextlabel := !nextlabel + 1;
   (Simili.SLABEL(s ^ (Int.string (!nextlabel)))))

val nextlbl = ref 0;
fun initlbl () =
  nextlbl := 0
fun MakeLbl str =
  (nextlbl := !nextlbl + 1;
   Simili.SLABEL(str ^ (Int.string (!nextlbl))))

fun label2string(Simili.SLABEL(lbl)) = lbl

val nextproc = ref 0;
fun initproc () =
  nextproc := 0
fun MakeProcName procname =
  (nextproc := !nextproc + 1;
   procname ^ "_" ^ (Int.string (!nextproc)))

val nextfnct = ref 0;
fun initfnct () =
  nextfnct := 0
fun MakeFnctName fnctname =
  (nextfnct := !nextfnct + 1;
   fnctname ^ "_" ^ (Int.string (!nextfnct)))

val nextobject = ref 1;
fun initobject () =
  nextobject := 1
fun MakeObjectName (objectname) =
  (nextobject := !nextobject + 1;
   objectname ^ "_" ^ (Int.string (!nextobject)))

val TruthBaseValueName = Simili.SBASEVALUENAMEEXPR(Simili.SBOOL(true))
val FalseBaseValueName = Simili.SBASEVALUENAMEEXPR(Simili.SBOOL(false))

fun firstlabel (Simili.SBB(label, _, _) :: _) = label
fun lookupPDEF (_, [])  = None
  | lookupPDEF (procname,
    Simili.SPROCEDURE(procn, fps, lvs, body) :: rest) =
```

```
    if procname = procn then
      Some(fps, lvs, body)
    else
      lookupPDEF (procname, rest)
  | lookupPDEF (procname, Simili.SFUNCTION(_, _, _, _, _) :: rest) =
    lookupPDEF (procname, rest)

fun lookupFDEF (_, [])  = None
  | lookupFDEF (fnctname, Simili.SPROCEDURE(_, _, _, _) :: rest)=
    lookupFDEF (fnctname, rest)
  | lookupFDEF (fnctname,
    Simili.SFUNCTION(fnctn, fps, rn, lvs, body) :: rest) =
    if fnctname = fnctn then
      Some(fps, rn, lvs, body)
    else
      lookupFDEF (fnctname, rest)

fun lookupBB (_, []) = raise PE("lookupBB")
  | lookupBB (label, (bb as Simili.SBB(lbl, _, _)) :: rest) =
    if label = lbl then
      bb
    else
      lookupBB (label, rest)

(*************************************************************************)
fun lift_eqmap (eqmap, rho, beta) =
  let val objects = AbsVal.list beta
    val new_eqmap =
      List.foldR
      (fn object => fn eqmap =>
       (case Rho.Globalizable (rho, AbsVal.fromobject object) of
          Some(Simili.SNAMEEXPR(Simili.SCONSTNAME(cn))) =>
            EQMap.lift (eqmap, cn)
        | _ => eqmap))
      eqmap
      objects
  in
    new_eqmap
  end


(*************************************************************************)

(* routines *)
```

```
fun FilterOutGlobals ([], _, _) = []
  | FilterOutGlobals (_, [], _) = []
  | FilterOutGlobals (hd :: tl, car :: cdr, rho) =
    if Rho.global (rho, hd) then
      FilterOutGlobals (tl, cdr, rho)
    else
      car :: FilterOutGlobals (tl, cdr, rho)

(*************************************************************************)
fun PeBNameExpr (Simili.SSELFNAME, rho, tau, oid, util) =
  (AbsVal.fromobject (AbsVal.AnObject(oid)),
   Simili.SNAMEEXPR(Simili.SSELFNAME), util)
  | PeBNameExpr (Simili.SCONSTNAME(cn), rho, tau, oid, util) =
    let val beta = Rho.lookup (rho, AbsVal.GlobalOID, cn) in
      (beta, (Simili.SNAMEEXPR(Simili.SCONSTNAME(cn))), util)
    end
  | PeBNameExpr (Simili.SIVARNAME(iv), rho, tau, oid, util) =
    let val beta = Rho.lookup (rho, oid, iv)
        val newutil =
          Util.updateUsed (util, oid, iv)
    in
      (beta, Simili.SNAMEEXPR(Simili.SIVARNAME(iv)), newutil)
    end
  | PeBNameExpr (Simili.SLVARNAME(lv), rho, tau, oid, util) =
    let val beta = Tau.lookup (tau, lv) in
      (beta, Simili.SNAMEEXPR(Simili.SLVARNAME(lv)), util)
    end
  | PeBNameExpr (Simili.SFORMALPAR(fp), rho, tau, oid, util) =
    let val beta = Tau.lookup (tau, fp) in
      (beta, Simili.SNAMEEXPR(Simili.SFORMALPAR(fp)), util)
    end
  | PeBNameExpr (Simili.SRESULTNAME(rn), rho, tau, oid, util) =
    let val beta = Tau.lookup (tau, rn) in
      (beta, Simili.SNAMEEXPR(Simili.SRESULTNAME(rn)), util)
    end
  | PeBNameExpr (Simili.UNKNOWN(_), _, _, _, _) =
    raise PE("Fatal error: impossible - PeBNameExpr")

fun PeNewBExpr (Simili.SNAMEEXPR(name), rho, tau, oid, util) =
  let val (beta, newname, newutil) =
      PeBNameExpr (name, rho, tau, oid, util)
  in
    (beta, newname, newutil)
  end
```

```
  | PeNewBExpr (Simili.SBASEVALUENAMEEXPR(bv), _, _, _, util) =
    (AbsVal.fromBaseValue bv, Simili.SBASEVALUENAMEEXPR(bv), util)

fun PeBExpr (be, rho, tau, oid, util) =
  let val (beta, newbe, util1) = PeNewBExpr (be, rho, tau, oid, util) in
    if AbsVal.equal (beta, AbsVal.fromobject (AbsVal.AnObject oid)) then
      (beta, Simili.SNAMEEXPR(Simili.SSELFNAME), util1)
    else
      case Rho.Globalizable (rho, beta) of
        Some(globalbe) =>
          (beta, globalbe, util1)
      | None =>
          (beta, newbe, util1)
  end

fun PeBExprs (bexprs, rho, tau, oid, util) =
  List.foldR
  (fn bexpr => fn (betas, exprs, util) =>
   let val (beta, expr, newutil) =
     PeBExpr (bexpr, rho, tau, oid, util)
   in
     (beta :: betas, expr :: exprs, newutil)
   end)
  ([], [], util)
  bexprs

(*************************************************************************)
(* Semantic equation system *)
fun PeProgram ([], rho, eqmap, delta, cds) = (rho, eqmap, delta, cds)
  | PeProgram (Simili.SCONSTDECLARATION(name, expr)
               :: rest, rho, eqmap, delta, cds) =
    let val dyn = false
      val mf = MF.empty
      val oid = AbsVal.GlobalOID
      val tau = Tau.empty
      val qmap = Qmap.empty
      val call = CALL("","", rho, [], None)
      val (value, newrho, neweqmap, newdelta, newexp) =
        (case expr of
           Simili.SOBJECTEXPR(oc, ivars, mdefs, ibody) =>
             let val newaoid = AbsVal.nextOID()
               val newoid = AbsVal.Static(newaoid)
               val rho1temp = Rho.addcode (rho, newoid, oc, mdefs)
               val rho1 = Rho.setName (rho1temp, newoid, name)
```

```
            val pi1 = Pi.empty
            val (_, mb, _, newibbs, rho2, _, _, _, delta1, eqmap1, pi2) =
              PeBody (ibody, dyn, call, mf, newoid,
                      rho1, tau, qmap, delta, eqmap, pi1)
            val newibody = Simili.SBODY(newibbs)
            val stringoid = AbsVal.oid2string newoid
            val (_, a1, a2, a3, _) = Rho.rescode (rho2, newoid)
            val rescode = (name, a1, a2, a3, newibody)
            val rho3 = Rho.updateResCode (rho2, newoid, rescode)
          in
            (AbsVal.fromobject (AbsVal.AnObject(newoid)),
             rho3, eqmap1, delta1, None)
          end
      | _ =>
          let val (value, NewExp, _, newrho, _, _, delta1, eqmap1, _) =
            PeExpr (expr, dyn, call, mf, oid,
                    rho, tau, Util.empty, qmap, delta, eqmap, Pi.empty)
          in
            case Rho.Globalizable (newrho, value) of
              Some(be) =>
                (value, newrho,
                 eqmap1, delta1, Some(Simili.SBASICEXPR(be)))
            | None =>
                (case AbsVal.isStatica value of
                   Some(oid) =>
                     let val rho2 = Rho.setName (newrho, oid, name)
                     in
                       (value, rho2, eqmap1, delta1, Some(NewExp))
                     end
                 | None =>
                     (if AbsVal.arity value = 1 then
                          (value, newrho,
                           eqmap1, delta1, Some(NewExp))
                      else
                          raise PE("PeProgram - impossible 44 " ^
                              (AbsVal.string value))))
          end)
    val _ = Debug.print "Pe" 6
      (fn () => ("ConstName = " ^ name ^ " processed!"))
    val newcds =
      (case newexp of
         None => cds
       | Some(expr) =>
          Simili.SCONSTDECLARATION(name, expr) ::
```

```
              cds)
    in
      PeProgram (rest,
                 Rho.update (newrho, AbsVal.GlobalOID, name, (value, [])),
                 neweqmap,
                 newdelta,
                 newcds)
    end

(* PeBody is now tail-recursive => changes where it is used !!! *)
(* PeBody does not conform to the standard definition in the formulas. *)
(* the first and third parameter should not be used, and a list of *)
(* basic blocks is returned instead of a body. *)
and PeBody (Simili.SBODY([]), dyn, call, mf, oid,
            rho, tau, qmap, delta, eqmap, pi) =
  (LB.empty, MB.makeExit(*empty*), Simili.SLABEL("dummy"), [], rho, tau,
   Util.empty, qmap, delta, eqmap, pi)
  | PeBody (Simili.SBODY(bb :: rest), dyn, call, mf, oid,
            rho, tau, qmap, delta, eqmap, pi) =
    let val _ = Debug.print "Pe" 6 (fn () => "PeBody")
    in
      PeBB (bb, bb :: rest, dyn, call, mf, LF.empty, oid,
            rho, tau, Util.empty, qmap, delta, eqmap, pi)
    end

and PeBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, mf, lf,
          oid, rho, tau, util, qmap, delta, eqmap, pi) =
  (case LF.exists_unfold (lf, label, rho, tau) of
     Some(reslbl) =>
       (LB.empty, MB.empty, reslbl, [], Rho.empty, Tau.empty,
        util, qmap, delta, eqmap, pi)
  | None =>
      (case LF.lookup (lf, LF.Iter, label) of
         Some(reslbl, rho1, tau1) =>
           if (Rho.lesseqsimple (rho, rho1)) andalso
             (Tau.lesseq (tau, tau1)) then
             let val delta1 =
               Delta.Explicator4(rho, tau, rho1, tau1, delta)
               val rho2 = Rho.updateRho0 rho
               val _ = Debug.print "LF" 5
                 (fn () =>
                  ("Loop Iteration finished. Label: " ^
                   (label2string label) ^ " " ^ (Qmap.string qmap)))
               val _ = Debug.print "QMAP" 5
```

```
                (fn () => "Result: " ^ (Rho.fullystring rho2) ^
                 " >> " ^ (Rho.string rho) ^
                 "\n   >> " ^ (Rho.simple2string rho1) ^
                 "\n >> " ^ (Tau.string tau) ^
                 "\n   >> " ^ (Tau.string tau1) ^
                 "\n")
            in
              (LB.makeExit(*empty*), MB.makeExit(*empty*),
               reslbl, [], rho2, Tau.empty,
               util, qmap, delta1, eqmap, pi)
            end
          else
            let val _ = Debug.print "LF" 5
              (fn () =>
               ("Another Loop Iteration Needed. Label: " ^
                (label2string label) ^ " " ^ (Qmap.string qmap)))
                val _ = Debug.print "LF" 9
                  (fn () => "Rho:" ^
                   "\n >> " ^ (Rho.string rho) ^
                   "\n   >> " ^ (Rho.simple2string rho1) ^
                   "\n >> " ^ (Tau.string tau) ^
                   "\n   >> " ^ (Tau.string tau1) ^
                   "\n")
                val rhosave = Rho.difference (rho, rho1)
            in
              (LB.singleton (label, reslbl, rhosave, tau), MB.empty,
               reslbl, [], Rho.empty, Tau.empty, util, qmap, delta,
               eqmap, pi)
            end
        | None =>
            PeDoBB (Simili.SBB(label, stmnts, jump),
                    bblist, dyn, call, mf, lf, oid, rho, tau,
                    util, qmap, delta, eqmap, pi)))

and PeDoBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, mf, lf,
            oid, rho, tau, util, qmap, delta, eqmap, pi) =
  let val _ = Debug.print "Pe" 7
    (fn () => "PeDoBB: " ^ (label2string label) ^ " " ^ (Qmap.string qmap))
      val (lb, mb, lbl, list, rho1, tau1,
           util1, qmap1, delta1, eqmap1, pi1) =
        PeBBBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, lf, mf,
                oid, rho, tau, util, qmap, delta, eqmap, pi)
      val lb1 = LB.remove (lb, label, lbl)
  in
```

```
    if (not (LB.isEmpty lb1)) orelse
      (not (LB.member (lb, label, lbl))) then
      let val _ = Debug.print "Pe" 7
        (fn () => "PeDoBB finish!")
      in
        (lb1, mb, lbl, list, rho1, tau1, util1, qmap1, delta1, eqmap1, pi1)
      end
    else
      PeDoBBTail(lf, rho, tau, label, stmnts, jump, bblist, call,
                 mf, oid, util, qmap, delta, eqmap, dyn, pi)
  end

and PeDoBBTail(lf, rho, tau, label, stmnts, jump, bblist, call,
               mf, oid, util, qmap, delta, eqmap, dyn, pi) =
  let val lbl2 = MakeLabel("l")
    val lf1 = LF.add (lf, LF.Iter, label, lbl2, rho, tau)
    val dyn1 = true
    val (lb1, mb1, list2, rho2, tau2, rho3, tau3,
         util2, qmap2, delta2, eqmap2) =
      PeIterBB (Simili.SBB(label, stmnts, jump), bblist, dyn1, call,
                lf1, mf, oid, lbl2, rho, tau, util, qmap, delta, eqmap, pi)
    val delta3 =
      Delta.Explicator4(rho, tau, Rho.rho2simple rho2, tau2, delta2)
  in
    (lb1, mb1, lbl2, list2,
     rho3, tau3, util2, (if dyn then qmap2 else qmap), delta3, eqmap2, pi)
  end

and PeIterBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, lf, mf,
              oid, lbl, rho, tau, util, qmap, delta, eqmap, pi) =
  let val _ = Debug.print "LF" 5
    (fn () => "IterBB called (" ^ (label2string label) ^ ", " ^
     (label2string lbl) ^ ") " ^ (Qmap.string qmap))
    val (lb, mb, lbl1, list1, rho1, tau1, util1, qmap1, delta1, eqmap1, _) =
    PeBBBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, lf, mf,
            oid, rho, tau, util, qmap, delta, eqmap, pi)
    val lb1 = LB.remove (lb, label, lbl)
  in
    if (not (LB.isExit lb)) andalso
      (LB.isEmpty lb1) andalso
      (MB.isEmpty mb) then
      let val _ = Debug.print "LF" 6
        (fn () => "Unfold tuple added: " ^ (label2string label))
        val lf1 =
```

```
            LF.add (LF.remove (lf, LF.Iter, label, lbl),
                    LF.Unfold, label, lbl, rho, tau)
        in
          PeIterBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, lf1,
                    mf, oid, lbl, rho, tau, util, qmap, delta, eqmap, pi)
        end
      else
        let val _ = Debug.print "LB" 9
            (fn () => ("Label, lbl: (" ^ (label2string label) ^ "," ^
                       (label2string lbl) ^ ")" ^
                       "LabelBackward: " ^ (LB.string lb1) ^ " " ^
                       (LB.string lb)))
        in
          if (not (LB.isEmpty lb1)) orelse
            (not (LB.member (lb, label, lbl))) then
            let val list = Simili.SBB(lbl, [], Simili.SGOTO(lbl1)) :: list1
            in
              (lb1, mb, list, rho, tau, rho1, tau1,
               util1, qmap1, delta1, eqmap1)
            end
          else
            let val (R, T) = LB.selectRT (lb, label, lbl)
              val rho2 = Rho.lubloop (rho, R)
              val tau2 = Tau.lubloop (tau, T)
              val lf1 = LF.add (LF.remove (lf, LF.Iter, label, lbl),
                                LF.Iter, label, lbl, rho2, tau2)
            in
              PeIterBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call,
                        lf1, mf, oid, lbl, rho2, tau2,
                        util, qmap1, delta1, eqmap1, pi)
            end
        end
    end

  (* Make better Garbage Collection *)
  and PeSGOTO(label1, bblist, dyn, call, mf, lf, oid, rho1, tau1, util1,
              qmap1, delta1, eqmap1, mb, lbl2, NewStmnt, pi1) =
    let val (lb, mb1, lbl, list, rho2, tau2,
             util2, qmap2, delta2, eqmap2, pi2) =
      PeBB (lookupBB (label1, bblist), bblist, dyn, call, mf, lf,
            oid, rho1, tau1, util1, qmap1, delta1, eqmap1, pi1)
        val mb2 = MB.lubseq (mb, mb1)
        val list2 =
          Simili.SBB(lbl2, NewStmnt,
```

```
                    Simili.SGOTO(lbl)) :: list
        val _ = Debug.print "MB" 9
          (fn () => ("After Goto: " ^ (MB.string mb2)))
    in
      (lb, mb2, lbl2, list2, rho2, tau2, util2, qmap2, delta2, eqmap2, pi2)
    end

  and PeBBBB (Simili.SBB(label, stmnts, jump), bblist, dyn, call, lf, mf,
              oid, rho, tau, util, qmap, delta, eqmap, pi) =
    let val _ = Debug.print "Pe" 7 (fn () => "PeBBBB called")
      val lbl2 = MakeLabel("l")
      val (mb, NewStmnt, rho1, tau1, util1, qmap1, delta1, eqmap1, pi1) =
        PeStmnts (stmnts, lbl2, dyn, call, mf, oid,
                  rho, tau, util, qmap, delta, eqmap, pi)
      val _ = Debug.print "Pe" 7 (fn () => "PeStmnts returned to PeBBBB")
    in
      case jump of
        Simili.SRETURN =>
          let val list = [Simili.SBB(lbl2, NewStmnt, Simili.SRETURN)]
            val lb = LB.makeExit
            val mb1 = MB.lubseq (mb, (MB.makeExit))
            val _ = Debug.print "MB" 7 (fn () => ("After return: " ^
                                                  (MB.string mb1)))
          in
            (lb, mb1, lbl2, list, rho1, tau1,
             util1, qmap1, delta1, eqmap1, pi1)
          end
      | Simili.SGOTO(label1) =>
          PeSGOTO(label1, bblist, dyn, call, mf, lf, oid, rho1, tau1, util1,
                  qmap1, delta1, eqmap1, mb, lbl2, NewStmnt, pi1)
      | Simili.SIF(expr, label1, label2) =>
          let val (beta, NewExp, mb1, rho2,
                   util2, qmap2, delta2, eqmap2, _ (*pi*)) =
            PeExpr (expr, dyn, call, mf, oid,
                    rho1, tau1, util1, qmap1, delta1, eqmap1, pi1)
            val mbnew = MB.lubseq (mb, mb1)
          in
            if (not (AbsVal.includesInputDynamic beta)) andalso
              (not (AbsVal.includes (AbsVal.False, beta))) andalso
              (AbsVal.includes (AbsVal.True, beta)) then (* TRUE *)
              PeSIFStaticTest(label1, bblist, dyn, call, mf, lf, oid, rho2,
                              tau1, util2, qmap2, delta2, eqmap2, mbnew,
                              NewStmnt, lbl2, pi1)
            else if (not (AbsVal.includesInputDynamic beta)) andalso
```

```
            (not (AbsVal.includes (AbsVal.True, beta))) andalso
            (AbsVal.includes (AbsVal.False, beta)) then (* FALSE *)
            PeSIFStaticTest(label2, bblist, dyn, call, mf, lf, oid, rho2,
                            tau1, util2, qmap2, delta2, eqmap2, mbnew,
                            NewStmnt, lbl2, pi1)
          else if (not (AbsVal.includesInputDynamic beta)) andalso
            (not (AbsVal.includes (AbsVal.True, beta))) andalso
            (not (AbsVal.includes (AbsVal.False, beta))) then (* ERROR *)
            let val _ = Debug.print "Pe" 6
              (fn () => "TEST - neither true nor false in:" ^
               (AbsVal.OID2string oid))
            in
              (LB.empty, MB.empty, lbl2,
               [Simili.SBB(lbl2, NewStmnt,
                           Simili.SIF(NewExp, lbl2, lbl2))],
               Rho.empty, Tau.empty, Util.empty, Qmap.empty,
               Delta.empty, EQMap.empty, Pi.empty)
            end
          else (* DYNAMIC TEST *)
            let val CALL(MethodName, ResMethodName,
                         rhocall, betascall, _) = call
                val mf1 =  MF.adddependent (mf, MF.Iter, oid, MethodName,
                                            ResMethodName, rhocall,
                                            betascall, None, AbsVal.empty)
                val lf1 =
                  LF.adddependent (lf, LF.Iter, label, lbl2, rho, tau)
                val _ = Debug.print "Dyn" 3
                  (fn () => "DYNAMIC TEST in: " ^ MethodName ^ " > " ^
                   (AbsVal.OID2string oid) ^ " >> " ^
                   (AbsVal.string beta))
                val _ = Debug.print "LF" 9
                  (fn () => "LabelForward: " ^ (LF.string lf) ^
                   "LabelForward1: " ^ (LF.string lf1))
            in
              PeSIFDynamicTest(label1, label2, bblist, dyn, call, mf1, lf1,
                               oid, lbl2, rho2, tau1, util2, qmap2, delta2,
                               eqmap2, mbnew, NewStmnt, NewExp, pi1)
            end
        end
    end

and PeSIFDynamicTest(label1, label2, bblist, dyn, call, mf1, lf1, oid,
                     lbl2, rho2, tau1, util2, qmap2, delta2, eqmap2,
                     mbnew, NewStmnt, NewExp, pi1) =
```

```
  let val (lbu1, mbu1, lblu1, listu1, rhou1, tauu1,
           utilu1, qmapu1, deltau1, eqmap3, piu1) =
    PeBB (lookupBB (label1, bblist), bblist, dyn, call, mf1,
          lf1, oid, rho2, tau1, util2, qmap2, delta2, eqmap2, pi1)
      val _ = Debug.print "Dyn" 5
        (fn () => "DYNAMIC TEST - between branches: (" ^
         (label2string label1) ^ ", " ^ (label2string label2) ^ ")")
      (* reuse code generated in the first branch *)
      val newrho = Rho.CombRho(rho2, rhou1)
      val (mbnew2, mbu1new) =
        if MB.isExit mbu1 then
          (MB.lubseq (mbnew, mbu1), MB.makeExit)
        else
          (MB.lubif (mbnew, mbu1), MB.empty)
  in
    PeSIFDynamicTestTail(label2, bblist, dyn, call, mf1, lf1, oid,
                         lbl2, newrho, tau1, util2, qmapu1, deltau1,
                         eqmap3, pi1, mbnew2, utilu1, NewStmnt, NewExp,
                         lbu1, mbu1new, lblu1, listu1, rhou1, tauu1, piu1)
  end

and PeSIFDynamicTestTail(label2, bblist, dyn, call, mf1, lf1, oid,
                         lbl2, newrho, tau1, util2, qmapu1, delta2,
                         eqmap3, pi1, mbnew, utilu1, NewStmnt, NewExp,
                         lbu1, mbu1new,
                         lblu1, listu1, rhou1, tauu1, piu1) =
  let val (lbu2, mbu2, lblu2, listu2, rhou2, tauu2,
           utilu2, qmap3, delta3, eqmap4, piu2) =
        PeBB (lookupBB (label2, bblist), bblist, dyn, call, mf1,
              lf1, oid, newrho, tau1, util2, qmapu1, delta2, eqmap3, pi1)
      val mb2 = MB.lubseq (mbnew, MB.lubif (mbu1new, mbu2))
      val lb1 = LB.union (lbu1, lbu2)
      val rho3 = Rho.lubif (rhou1, rhou2)
      val _ = Debug.print "Dyn" 9
        (fn () => Rho.fullystring rho3)
      val tau2 = Tau.lubif (tauu1, tauu2)
      val util3 = Util.lubif (utilu1, utilu2)
      val pi2 = Pi.lub (piu1, piu2)
      val CALL(_, _, _, _, rn) = call
      val (tau11, tau22) =
        case rn of
          None => (Tau.empty, Tau.empty)
        | Some(rn) =>
            let val t1 =
```

```
              (case Tau.getvaluesafe (tauu1, rn) of
                 Some(value) => Tau.update (Tau.empty, rn, value)
               | None => Tau.empty)
                val t2 =
                  (case Tau.getvaluesafe (tauu2, rn) of
                     Some(value) => Tau.update (Tau.empty, rn, value)
                   | None => Tau.empty)
            in
              (t1, t2)
            end
      val delta4 = Delta.Explicator4(rhou1, tau11,
                                     Rho.rho2simple rhou2, tau22, delta3)
      val list = Simili.SBB(lbl2, NewStmnt,
                            Simili.SIF(NewExp, lblu1, lblu2)) ::
        (listu1 @ listu2)
  in
    (lb1, mb2, lbl2, list, rho3, tau2, util3, qmap3, delta4, eqmap4, pi2)
  end

and PeSIFStaticTest(label1, bblist, dyn, call, mf, lf, oid, rho2, tau1,
                    util2, qmap2, delta2, eqmap2, mbnew, NewStmnt,
                    lbl2, pi1) =
  let val (lb, mb2, lbl, list, rho3, tau2,
           util3, qmap3, delta3, eqmap3, pi2) =
    PeBB (lookupBB (label1, bblist), bblist, dyn, call, mf, lf,
          oid, rho2, tau1, util2, qmap2, delta2, eqmap2, pi1)
      val mb3 = MB.lubseq (mbnew, mb2)
      val list2 = Simili.SBB(lbl2, NewStmnt, Simili.SGOTO(lbl))
        :: list
  in
    (lb, mb3, lbl2, list2, rho3, tau2, util3, qmap3, delta3, eqmap3, pi2)
  end

and myPeStmnts (stmnts, lbl, dyn, call, mf, oid,
                rho, tau, util, qmap, delta, eqmap, pi) =
  List.foldR
  (fn stmnt =>
   fn (mb, resstmnts, rho, tau, util, qmap, delta, eqmap, pi) =>
   let val (mb1, stmnts1, rho1, tau1,
            util1, qmap1, delta1, eqmap1, pi1) =
     PeStmnt (stmnt, lbl, dyn, call, mf, oid,
              rho, tau, util, qmap, delta, eqmap, pi)
   in
     (MB.lubseq (mb, mb1), resstmnts @ stmnts1, rho1, tau1,
```

```
      util1, qmap1, delta1, eqmap1, pi1)
   end)
  (MB.empty, [], rho, tau, util, qmap, delta, eqmap, pi)
  stmnts

and PeStmnts (stmnts, lbl, dyn, call, mf, oid,
              rho, tau, util, qmap, delta, eqmap, pi) =
  (Debug.print "Pe" 7 (fn () => "PeStmnts called");
   myPeStmnts ((rev stmnts), lbl, dyn, call, mf, oid,
               rho, tau, util, qmap, delta, eqmap, pi))

and PeStmnt (Simili.SASSIGNSTMNT(name, expr), lbl, dyn, call, mf, oid,
             rho, tau, util, qmap, delta, eqmap, pi) =
  (let val (beta, NewExp, mb, rho1, util1, qmap1, delta1, eqmap1, pi1) =
     PeExpr (expr, dyn, call, mf, oid, rho, tau,
             util, qmap, delta, eqmap, pi)
   in
     if Rho.global (rho1, beta) then
       case name of
         Simili.SIVARNAME(n) =>
           (mb, [], Rho.update (rho1, oid, n, (beta, [lbl])), tau,
            Util.updateDefined (util1, oid, n),
            qmap1, delta1, eqmap1, pi1)
       | Simili.SLVARNAME(n) =>
           (mb, [], rho1, Tau.update (tau, n, (beta, [lbl])),
            util1, qmap1, delta1, eqmap1, pi1)
       | Simili.SRESULTNAME(n) =>
           (mb, [],
            rho1, Tau.update (tau, n, (beta, [lbl])),
            util1, qmap1, delta1, eqmap1, pi1)
       | _ =>
           raise PE("Fatal error: impossible - PeStmnt")
     else
       case name of
         Simili.SIVARNAME(n) =>
           (mb, Simili.SASSIGNSTMNT(name, NewExp) :: [],
            Rho.update (rho1, oid, n, (beta, [])), tau,
            Util.updateDefined (util1, oid, n),
            qmap1, delta1, eqmap1, pi1)
       | Simili.SLVARNAME(n) =>
           (mb, Simili.SASSIGNSTMNT(name, NewExp) :: [], rho1,
            Tau.update (tau, n, (beta, [])),
            util1, qmap1, delta1, eqmap1, pi1)
       | Simili.SRESULTNAME(n) =>
```

```
            (mb, Simili.SASSIGNSTMNT(name, NewExp) :: [], rho1,
             Tau.update (tau, n, (beta, [])),
             util1, qmap1, delta1, eqmap1, pi1)
        | _ =>
            raise PE("Fatal error: impossible - PeStmnt")
    end
  handle Rho.Rho(s) => raise PE("PeStmnt (rho: " ^ s ^ ")"))

  | PeStmnt (Simili.SSKIPSTMNT, lbl, dyn, call, mf, oid,
             rho, tau, util, qmap, delta, eqmap, pi) =
    (MB.empty, [], rho, tau, util, qmap, delta, eqmap, pi)

  | PeStmnt (Simili.SPROCCALLSTMNT(be, ProcName, bes), lbl, dyn, call, mf,
             oid, rho, tau, util, qmap, delta, eqmap, pi) =
    let val (beta, NewBExpr, util1) = PeBExpr (be, rho, tau, oid, util)
      val (betas, NewBExprs, util2) = PeBExprs (bes, rho, tau, oid, util1)
      val beta1 = AbsVal.removeNil beta
    in
      if AbsVal.arity(beta1) = 0 then
        let val NewProcName = MakeProcName ProcName
        in
          (MB.empty,
           Simili.SPROCCALLSTMNT(NewBExpr, NewProcName, NewBExprs) :: [],
           Rho.empty, Tau.empty, Util.empty, qmap, delta, eqmap, pi)
        end
      else
        (case AbsVal.isStaticOrQuantified(*isStatica*) beta1 of
          Some(oid1:AbsVal.OID) =>
            let val NewProcName = MakeProcName ProcName
              val man = true
              val (ResProcName, newarg, mb, rho1,
                   util3, qmap1, delta1, eqmap1, pi1) =
                PeProcCall (oid1, ProcName, betas, NewProcName,
                            man, dyn, mf, rho, qmap, delta, eqmap, pi)
              val util4 = Util.lub (util2, util3)
              val _ = Debug.print "Pe" 6
                (fn () =>
                 ("PeProcCall finished*** -->" ^ (Arg.string newarg)))
              val NewArgs = FilterOutGlobals (newarg, NewBExprs, rho1)
               val NPN =
                 case ResProcName of
                    None => NewProcName
                  | Some(rpn) => rpn
              val stmnt =
```

```
            (Simili.SPROCCALLSTMNT(NewBExpr, NPN, NewArgs) :: [])
          val mb1 = MB.removeExit mb
        in
          (mb1, stmnt, rho1, tau, util4, qmap1, delta1, eqmap1, pi1)
        end
    | None => (* NON MANIFEST CALL *)
        (if (not (AbsVal.includesInputDynamic(beta1))) andalso
           (not (AbsVal.includesBaseValue(beta1))) then
           let val CALL(MethodName, ResMethodName,
                        rhocall, betascall, _) = call
               val mf1 = MF.adddependent (mf, MF.Iter, oid, MethodName,
                                          ResMethodName, rhocall,
                                          betascall, None, AbsVal.empty)
               val NewProcName = MakeProcName ProcName
               val oids = AbsVal.oids beta1
               val man = false
               val stmnt = (Simili.SPROCCALLSTMNT(NewBExpr,
                                                  NewProcName,
                                                  NewBExprs) :: [])
               val _ = Debug.print "Man" 3
                 (fn () => "Non manifest call: " ^
                  (AbsVal.string beta) ^ " " ^ ProcName ^ " " ^
                  (MF.string mf1))
               val (_, mb1, rho1,
                    util3, qmap1, deltaxs, eqmap1, pi1) =
                 List.foldR
                 (fn oid => fn (rho, mbxs, rhoxs, utilxs,
                                qmapxs, deltaxs, eqmap, pixs) =>
                  let val (_, _, mbx, rhox, utilx,
                           qmapx, deltax, eqmap1, pi1) =
                        PeProcCall (oid, ProcName, betas, NewProcName,
                                    man, dyn, mf1, rho,
                                    qmapxs, deltaxs, eqmap, pi)
                  in
                    (Rho.CombRho(rho, rhox),
                     MB.lubif (MB.removeExit mbx,  mbxs),
                     Rho.lubif (rhox, rhoxs),
                     Util.lubif (utilx, utilxs),
                     (*Qmap.lubif (qmapx, qmapxs),*)
                     qmapx,
                     Delta.Explicator2r(rhoxs, Rho.rho2simple rhox,
                                        deltaxs),
                     eqmap1,
                     Pi.lub (pi1, pixs))
```

```
                end)
               (rho, MB.empty, Rho.empty,
                util2, qmap, delta, eqmap, Pi.empty)
               oids
           val _ = Debug.print "MB" 9 (fn () => MB.string mb1)
           (*val deltaxs1 = List.map
             (fn (deltax, rhox) =>
              Delta.Explicator2r(rho1, rhox, deltax))
             deltaxs
           val delta1 = List.foldR Delta.lubif Delta.empty deltaxs1
             *)
       in
         (mb1, stmnt, rho1, tau,
          util3, qmap1, deltaxs(*1*), eqmap1, pi1)
       end
     else (* HARD PROCCALL - includes InputDynamic or BV *)
       let val objects = AbsVal.objects beta1
           val CALL(MethodName, ResMethodName,
                    rhocall, betascall, _) = call
           val mf1 = MF.adddependent (mf, MF.Iter, oid, MethodName,
                                      ResMethodName, rhocall,
                                      betascall, None, AbsVal.empty)
           val _ = Debug.print "Man" 3
             (fn () => "Non manifest hard call: " ^
              (AbsVal.string beta) ^ " " ^ ProcName ^ " " ^
              (MF.string mf1))
           val stmnt =
             Simili.SPROCCALLSTMNT(NewBExpr, ProcName, NewBExprs)
             :: []
           val man = false
           val (_, mb1, rho1,
                util3, qmap1, deltaxs, eqmap1, pi1) =
             List.foldR
             (fn object => fn (rho, mbxs, rhoxs, utilxs,
                               qmapxs, deltaxs, eqmap, pixs) =>
              let val (mbx, rhox,
                       utilx, qmapx, deltax, eqmap1, pi1) =
                PeHardProcCall (object, ProcName, betas,
                                man, dyn, mf1, rho,
                                qmapxs, deltaxs, eqmap, pi)
              in
                (Rho.CombRho(rho, rhox),
                 MB.lubif (MB.removeExit mbx, mbxs),
                 Rho.lubif (rhox, rhoxs),
```

```
                        Util.lubif (utilx, utilxs),
                        (*Qmap.lubif (qmapx, qmapxs),*)
                        qmapx,
                        Delta.Explicator2r(rhoxs, Rho.rho2simple rhox,
                                           deltaxs),
                        eqmap1,
                        Pi.lub (pi1, pixs))
                     end)
                    (rho, MB.empty, Rho.empty,
                     util2, qmap, delta, eqmap, Pi.empty)
                    objects
                  val _ = Debug.print "MB" 9 (fn () => MB.string mb1)
                  (*val deltaxs1 = List.map
                    (fn (deltax, rhox) =>
                     Delta.Explicator2r(rho1, rhox, deltax))
                    deltaxs
                  val delta1 = List.foldR Delta.lubif Delta.empty deltaxs1
                    *)
              in
                (mb1, stmnt, rho1, tau,
                 util3, qmap1, deltaxs, eqmap1, pi1)
              end))
    end

and PeHardProcCall (AbsVal.ABaseValue(bv), ProcName, args,
                    man, dyn, mf, rho, qmap, delta, eqmap, pi) =
  (* Manifest flag, man, always false when called *)
    (MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
  | PeHardProcCall (AbsVal.InputDynamic, ProcName, args,
                    man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    (MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
  | PeHardProcCall (AbsVal.AnObject(oid), ProcName, args,
                    man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    if oid = AbsVal.GlobalOID then
      (MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
    else
      case Rho.ReuseProc(rho, oid, ProcName, args, ProcName, dyn, man) of
        Rho.PNoUse =>
          let val dyn1 = true
            val (_, _, mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
              PeAProcCall (oid, ProcName, args, ProcName,
                           man, dyn1, mf, rho, qmap, delta, eqmap, pi)
          in
            (mb, rho1, util, qmap1, delta1, eqmap1, pi1)
```

```
            end
        | Rho.PReused(rpn, newrho, util) =>
            (MB.empty, newrho, util, qmap, delta, eqmap, pi)
        | Rho.PUseNoReuse(rhob4, argsb4, qmapb4) =>
            let val rhonew = Rho.lubifsim (rhob4, rho)
              val deltanew = Delta.Explicator2r (rhonew, rhob4, delta)
              val argnew = Arg.lubif (argsb4, args)
              val qmapnew = Qmap.lubif (qmapb4, qmap)
              val dyn1 = true
              val (_, _, mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
                PeAProcCall (oid, ProcName, argnew, ProcName, man, dyn1,
                             mf, rhonew, qmapnew, deltanew, eqmap, pi)
            in
              (mb, rho1, util, qmap1, delta1, eqmap1, pi1)
            end

    and PeExpr (Simili.SBASICEXPR(be), dyn, call, mf, oid,
                rho, tau, util, qmap, delta, eqmap, pi) =
      let val (beta, newbe, util1) = PeBExpr (be, rho, tau, oid, util) in
        (beta, Simili.SBASICEXPR(newbe), MB.empty, rho,
         util1, qmap, delta, eqmap, pi)
      end
      | PeExpr (Simili.SEQUALEXPR(be1, be2), dyn, call, mf, oid,
                rho, tau, util, qmap, delta, eqmap, pi) =
        let val (beta1, be1, util1) = PeBExpr (be1, rho, tau, oid, util)
          val (beta2, be2, util2) = PeBExpr (be2, rho, tau, oid, util1)
          val _ = Debug.print "Pe" 7
            (fn () => "Equality expression: " ^ (AbsVal.string beta1) ^ " == "
             ^ (AbsVal.string beta2))
        in
          if (AbsVal.isStatic beta1) andalso (AbsVal.equal (beta1, beta2)) then
            (AbsVal.True, Simili.SBASICEXPR(TruthBaseValueName), MB.empty,
             rho, util2, qmap, delta, eqmap, pi)
            else if AbsVal.disjoint (beta1, beta2) then
              (AbsVal.False, Simili.SBASICEXPR(FalseBaseValueName), MB.empty,
               rho, util2, qmap, delta, eqmap, pi)
              else
                let val new_eqmap =
                  lift_eqmap (eqmap, rho, AbsVal.lubif (beta1, beta2))
                in
                  (AbsVal.aBoolean, Simili.SEQUALEXPR(be1, be2), MB.empty,
                   rho, util2, qmap, delta, new_eqmap, pi)
                end
        end
```

```
| PeExpr (Simili.SOBJECTEXPR(oc, ivars, mdefs, ibody), dyn, call, mf,
         oid, rho, tau, util, qmap, delta, eqmap, pi) =
  if (not dyn) then
    let val newaoid = AbsVal.nextOID()
      val newoid = AbsVal.Static(newaoid)
      val rho1 = Rho.addcode (rho, newoid, oc, mdefs)
      val pi1 =
        if dyn then
          pi
        else
          Pi.lub (pi, MF.buildpi (mf, newoid, oc, rho1))
      val (_, mb, _, newibbs, rho2, tau2,
           util1, qmap1, delta1, eqmap1, pi2) =
        PeBody (ibody, dyn, call, mf, newoid,
                rho1, tau, qmap, delta, eqmap, pi1)
      val newibody = Simili.SBODY(newibbs)
      val newutil = Util.lub (util, util1)
      val beta = AbsVal.fromobject (AbsVal.AnObject(newoid))
    in
      PeExprStaticObject(rho2, newoid, oc, ivars, newibody, beta, mb,
                         newutil, qmap1, delta1, eqmap1, pi2)
    end
  else (* Quantified objects - dyn true *)
    (case Qmap.lookup (qmap, oc) of
       None =>
         let val newaoid = AbsVal.nextOID()
           val newoid = AbsVal.Quantified(newaoid)
           val rho1 = Rho.addcode (rho, newoid, oc, mdefs)
           val ienv1 = Rho.ienv (rho, oid)
           val pi1 = pi
             (*Pi.lub (pi, MF.buildpi (mf, newoid, oc, rho1))*)
           val (_, mb, _, newibbs, rho2, tau2,
                util1, qmap1, delta1, eqmap1, pi2) =
             PeBody (ibody, dyn, call, mf, newoid,
                     rho1, tau, qmap, delta, eqmap, pi1)
           val newibody = Simili.SBODY(newibbs)
           val beta = AbsVal.fromobject (AbsVal.AnObject(newoid))
           val (_, a1, a2, a3, _) = Rho.rescode (rho2, newoid)
           val rescode = ("", a1, a2, a3, newibody)
           val rho3 = Rho.updateResCode (rho2, newoid, rescode)
           val stringoid = AbsVal.oid2string newoid
           val qmap2 = Qmap.update (qmap1, oc, (newoid, tau(*, ienv1*)))
           val mb1 = MB.raiseCreated mb
```

```
            val newutil = Util.lub (util, util1)
          in
            (beta,
             Simili.SOBJECTEXPR(stringoid,
                                [], [], Simili.SBODY([])),
             mb1, rho3, newutil, qmap2, delta1, eqmap1, pi2)
          end
      | Some(newoid, taulast(*, ienvlast*)) =>
          let val stringoid = AbsVal.oid2string newoid
            val _ = Debug.print "Pe" 5
              (fn () => "reuse quantified object " ^
               (Rho.OID2string (rho, newoid)))
            val rho1 = Rho.updateQMAPRHO (rho, newoid, oc, mdefs)
            val delta1 =
              Delta.Explicator2t(taulast, tau, delta,
                                 rho1, Simili.SLVARNAME)
            val tau2 = Tau.lubif (tau, taulast)
            val pi1 = pi
              (*Pi.lub (pi, MF.buildpi (mf, newoid, oc, rho1))*)
            val (_, mb, _, newibbs, rho2, tau4,
                 util1, qmap1, delta2, eqmap1, pi2) =
              PeBody (ibody, dyn, call, mf, newoid,
                      rho1, tau2, qmap, delta1, eqmap, pi1)
            val newibody = Simili.SBODY(newibbs)
            val beta = AbsVal.fromobject (AbsVal.AnObject(newoid))
            val (_, a1, a2, a3, _) = Rho.rescode (rho2, newoid)
            val rescode = ("", a1, a2, a3, newibody)
            val rho3 = Rho.updateResCode (rho2, newoid, rescode)
            val stringoid = AbsVal.oid2string newoid
            val qmap2 =
              Qmap.update (qmap1, oc, (newoid, tau2(*, ienvlast*)))
            val mb1 = MB.raiseCreated mb
            val newutil = Util.lub (util, util1)
            val _ = Debug.print "QMAP" 5
              (fn () => "Reuse code: " ^ (Rho.fullystring rho3))
          in
            (beta,
             Simili.SOBJECTEXPR(stringoid,
                                [], [], Simili.SBODY([])),
             mb1, rho3, newutil, qmap2, delta2, eqmap1, pi2)
          end)

    | PeExpr (Simili.SFNCTCALLEXPR(be, FnctName, bes), dyn, call, mf, oid,
              rho, tau, util, qmap, delta, eqmap, pi) =
```

```
    let val (beta, NewExp, mb, rho1, newutil, qmap1, delta1, eqmap1, pi1) =
      PeEvalFnct (be, FnctName, bes, dyn, call, mf, oid,
                  rho, tau, util, qmap, delta, eqmap, pi)
      val util1 = Util.lub (util, newutil)
    in
      case Rho.Globalizable (rho1, beta) of
        Some(abexpr) =>
          (beta, Simili.SBASICEXPR(abexpr), mb, rho1,
           util1, qmap1, delta1, eqmap1, pi1)
      | None =>
          (beta, NewExp, mb, rho1, util1, qmap1, delta1, eqmap1, pi1)
    end

and PeExprStaticObject(rho2, newoid, oc, ivars, newibody, beta, mb, util,
                       qmap1, delta1, eqmap1, pi2) =
  let val myself = AbsVal.fromobject (AbsVal.AnObject(newoid)) in
    if List.forAll
      (fn ivarname =>
       (case Rho.lookuptest (rho2, newoid, ivarname) of
          None => true
        | Some(value) =>
            (AbsVal.equal (value, myself)) orelse
            (Rho.global (rho2, value))))
      (Rho.getivars (rho2, newoid))
      then (* GLOBALIZABLE *)
        let val name = MakeObjectName oc
          val initlabel = MakeLbl "init"
          val (_, a1, a2, a3, _) = Rho.rescode (rho2, newoid)
          val newibody =
            Simili.SBODY(Simili.SBB(initlabel,
                                    [],
                                    Simili.SRETURN) :: [])
          val rescode = (name, a1, a2, a3, newibody)
          val rho3 =
            Rho.updateIvars (rho2, newoid, (initlabel))
          val rho4 = Rho.updateResCode (rho3, newoid, rescode)
          val mb1 = MB.raiseCreated mb
        in
          (beta,
           Simili.SBASICEXPR(Simili.SNAMEEXPR(Simili.SCONSTNAME(name))),
           mb1, rho4, util, qmap1, delta1, eqmap1, pi2)
        end
    else
      let val (_, a1, a2, a3, _) = Rho.rescode (rho2, newoid)
```

```
          val rescode = ("", a1, a2, a3, newibody)
          val rho3 = Rho.updateResCode (rho2, newoid, rescode)
          val stringoid = AbsVal.oid2string newoid
          val mb1 = MB.raiseCreated mb
        in
          (beta,
           Simili.SOBJECTEXPR(stringoid,
                              [], [], Simili.SBODY([])),
           mb1, rho3, util, qmap1, delta1, eqmap1, pi2)
        end
    end

  and PeEvalFnct (be, FnctName, bes, dyn, call, mf, oid,
                  rho, tau, util, qmap, delta, eqmap, pi) =
      let val (beta, newbexpr, util1) = PeBExpr (be, rho, tau, oid, util)
        val (betas, newbexprs, util2) = PeBExprs (bes, rho, tau, oid, util1)
        val _ = Debug.print "Pe" 8
          (fn () =>
           ("EvalFnct: " ^ FnctName ^ ", called in object: " ^
            (AbsVal.string beta) ^ ", with arguments " ^
            (Arg.string betas)))
        val beta1 = AbsVal.removeNil beta
      in
        if AbsVal.arity(beta1) = 0 then
          let val NewFnctName = MakeFnctName FnctName
          in
            (beta1,
             Simili.SFNCTCALLEXPR(newbexpr, NewFnctName, newbexprs),
             MB.empty, Rho.empty, Util.empty, qmap, delta, eqmap, pi)
          end
        else
          (case AbsVal.isStaticOrQuantified(*isStatica*) beta1 of
             Some(oid) =>
               let val NewFnctName = MakeFnctName FnctName
                 val man = true
                 val (ResFnctName, beta2, betas1, mb, rho1,
                      util3, qmap1, delta1, eqmap1, pi1) =
                   PeFnctCall (oid, FnctName, betas, NewFnctName,
                               man, dyn, mf, rho, qmap, delta, eqmap, pi)
                 val NewArg = FilterOutGlobals (betas1, newbexprs, rho1)
                 val _ = Debug.print "Pe" 6
                   (fn () => ("result (" ^ FnctName ^ ") = " ^
                              (AbsVal.string beta2)))
                 val newutil = Util.lub (util2, util3)
```

```
          val mb1 = MB.removeExit mb
          val NFN =
            case ResFnctName of
               None => NewFnctName
             | Some(rfn) => rfn
        in
          (beta2,
           Simili.SFNCTCALLEXPR(newbexpr, NFN, NewArg),
           mb1, rho1, newutil, qmap1, delta1, eqmap1, pi1)
        end
    | None => (* NON MANIFEST CALL *)
        (if (not (AbsVal.includesInputDynamic beta1)) andalso
           (not (AbsVal.includesBaseValue beta1)) then
           let val CALL(MethodName, ResMethodName,
                        rhocall, betascall,_) = call
               val mf1 = MF.adddependent (mf, MF.Iter, oid,
                                          MethodName, ResMethodName,
                                          rhocall, betascall, None,
                                          AbsVal.empty)
               val NewFnctName = MakeFnctName FnctName
               val oids = AbsVal.oids beta1
               val _ = Debug.print "Man" 3
                 (fn () => "Non manifest call: " ^
                  (AbsVal.string beta) ^ " " ^ FnctName ^ " " ^
                  (MF.string mf1))
               val man = false
               val (_, beta2, mb1, rho1, util3,
                    qmap1, deltaxs, eqmap1, pi1) =
                 (List.foldR
                  (fn oid =>
                   fn (rho, betass, mbs, rhos, utils,
                       qmaps, deltas, eqmap, pixs) =>
                   let val (_, beta, betaxs, mb, rhox,
                            util, qmap, delta, eqmap1, pi1) =
                     PeFnctCall (oid, FnctName, betas, NewFnctName,
                                 man, dyn, mf1, rho,
                                 qmaps, deltas, eqmap, pi)
                   in
                     (Rho.CombRho(rho, rhox),
                      AbsVal.lubif (beta, betass),
                      MB.lubif (MB.removeExit mb, mbs),
                      Rho.lubif (rhox, rhos),
                      Util.lubif (util, utils),
                      (*Qmap.lubif (qmap, qmaps),*)
```

```
                    qmap,
                    Delta.Explicator2r(rhos, Rho.rho2simple rhox,
                                       deltas),
                    eqmap1,
                    Pi.lub (pi1, pixs))
                end)
               (rho, AbsVal.empty, MB.empty, Rho.empty, util2,
                qmap, delta, eqmap, Pi.empty)
               oids)
          val _ = Debug.print "Pe" 6
            (fn () => ("result(" ^ FnctName ^ ") = " ^
                       (AbsVal.string beta2)))
          val _ = Debug.print "MB" 9 (fn () => MB.string mb1)
          (*val deltaxs1 = List.map
            (fn (deltax, rhox) =>
             Delta.Explicator2r(rho1, rhox, deltax))
            deltaxs
          val delta1 =
            List.foldR Delta.lubif Delta.empty deltaxs1*)
      in
        (beta2,
         Simili.SFNCTCALLEXPR(newbexpr, NewFnctName, newbexprs),
         mb1, rho1, util3, qmap1, deltaxs, eqmap1, pi1)
      end
    else
      let val CALL(MethodName, ResMethodName,
                   rhocall, betascall, _) = call
          val mf1 = MF.adddependent (mf, MF.Iter, oid,
                                     MethodName, ResMethodName,
                                     rhocall, betascall, None,
                                     AbsVal.empty)
          val objects = AbsVal.objects beta1
          val _ = Debug.print "Man" 3
            (fn () => "Non manifest call: " ^
             (AbsVal.string beta) ^ " " ^ FnctName ^ " " ^
             (MF.string mf1))
          val man = false
          val (_, beta2, mb1, rho1, util3,
               qmap1, deltas, eqmap1, pi1) =
            List.foldR
            (fn object => fn (rho, betaxs, mbs, rhoxs, utils,
                              qmaps, deltas, eqmap, pixs) =>
             let val (beta, mb, rho1, util,
                      qmap1, delta1, eqmap1, pi1) =
```

```
                        PeHardFnctCall (object, FnctName, betas,
                                        man, dyn, mf1, rho,
                                        qmaps, deltas, eqmap, pi)
                    in
                      (Rho.CombRho(rho, rho1),
                       AbsVal.lubif (beta, betaxs),
                       MB.lubif (MB.removeExit mb, mbs),
                       Rho.lubif (rho1, rhoxs),
                       Util.lubif (util, utils),
                       (*Qmap.lubif (qmap1, qmaps),*)
                       qmap1,
                       Delta.Explicator2r(rhoxs, Rho.rho2simple rho1,
                                          deltas),
                       eqmap1,
                       Pi.lub (pi1, pixs))
                     end)
                    (rho, AbsVal.empty, MB.empty, Rho.empty, util2,
                     qmap, delta, eqmap,
                     Pi.empty)
                    objects
                  val _ = Debug.print "MB" 9 (fn () => MB.string mb1)
                  (*val deltas1 = List.map
                    (fn (deltax, rhox) =>
                     Delta.Explicator2r(rho1, rhox, deltax))
                    deltas
                  val delta1 = List.foldR Delta.lubif Delta.empty deltas1
                    *)
                  val _ = (Debug.print "Pe" 6
                           (fn () => "result(" ^ FnctName ^ ") = " ^
                            (AbsVal.string beta2)))
              in
                (beta2,
                 Simili.SFNCTCALLEXPR(newbexpr, FnctName, newbexprs),
                 mb1, rho1, util3, qmap1, deltas, eqmap1, pi1)
              end))
    end

and PeProcCall (oid1, ProcName, betas, NewProcName,
                man, dyn, mf, rho, qmap, delta, eqmap, pi) =
  case Rho.ReuseProc(rho, oid1, ProcName, betas, NewProcName, dyn, man) of
    Rho.PNoUse =>
      PeAProcCall(oid1, ProcName, betas, NewProcName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi)
  | Rho.PReused(rpn, newrho, newutil) =>
```

```
        (rpn, betas, MB.empty, newrho, newutil, qmap, delta, eqmap, pi)
    | Rho.PUseNoReuse(_, _, _) =>
        raise PE("Impossible - internal error. Method: " ^ ProcName ^
                 "invoked.  Rho: " ^ (Rho.fullystring rho))

  and PeAProcCall(oid1, ProcName, betas, NewProcName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    let val _ = Debug.print "Pe" 5
      (fn () =>
       ("PeProcCall: " ^ ProcName ^ ", called in object: " ^
        (AbsVal.OID2string oid1) ^ " arguments " ^
        (Arg.string betas) ^ " " ^ (MF.string mf)))
    in
      (case MF.exists_unfold (mf, oid1, ProcName, rho, betas) of
         Some(ResProcName, rho2, _) =>
           let val mb =
             MB.singleton (MB.Use, oid1, ProcName, ResProcName, rho, betas)
             val rho3 = (case rho2 of
                           None => rho
                         | Some(rho2) =>
                             Rho.updateRhosimple (rho, rho2))
             val (rpn, rho4) =
               if man then
                 (Some(ResProcName), rho3)
               else
                 (None,
                  Rho.procedurealias (rho3, oid1, ResProcName, NewProcName))
           in
             (rpn, betas, mb, rho4, Util.empty, qmap, delta, eqmap, pi)
           end
       | None =>
           (case MF.contains (mf, MF.Iter, oid1, ProcName) of
              Some(ResProcName, rho1, betas1, rho2, _) =>
                (case rho2 of
                   None =>
                     let val mb = MB.singleton (MB.Redo, oid1, ProcName,
                                                ResProcName, rho, betas)
                     in
                       (None, betas, mb, rho,
                        Util.empty, qmap, delta, eqmap, pi)
                     end
                 | Some(rho2) =>
                     if (Rho.lesseqsimple (rho, rho1)) andalso
                       (Arg.lesseq (betas, betas1)) then
```

```
                    let val delta1 =
                      Delta.Explicator2r(rho, rho1, delta)
                        val mb = MB.singleton (MB.Use, oid1, ProcName,
                                               ResProcName, rho, betas)
                        val rho3 = Rho.updateRhosimple (rho, rho2)
                        val (rpn, rho4) =
                          if man then
                            (Some(ResProcName), rho3)
                          else
                            (None, Rho.procedurealias (rho3, oid1,
                                                       ResProcName,
                                                       NewProcName))
                        val _ = Debug.print "MB" 6
                          (fn () => ("PeProcCall: result reached " ^
                                     (AbsVal.OID2string oid1) ^
                                     ", alias " ^ NewProcName ^
                                     " with " ^ ResProcName ^
                                     (MB.string mb)))
                    in
                      (rpn, betas1, mb, rho4,
                       Util.empty, qmap, delta1, eqmap, pi)
                    end
                  else
                    let val rhosave = Rho.difference (rho, rho1)
                      val mb = MB.singleton (MB.Redo, oid1, ProcName,
                                             ResProcName, rhosave, betas)
                      val _ = Debug.print "MB" 6
                        (fn () => "PeProcCall: try again " ^
                         (AbsVal.OID2string oid1) ^ ", alias " ^
                         NewProcName ^ " with " ^ ResProcName ^
                         (MB.string mb))
                      val _ = Debug.print "MB" 9
                        (fn () => "Rho: " ^ (Rho.string rho) ^
                         "\nRho1: " ^ (Rho.simple2string rho1) ^
                        "\nArg less-equal? " ^
                         (Bool.string (Arg.lesseq (betas, betas1))) ^
                         "\nArg: " ^ (Arg.string betas) ^
                         "\nArg1: " ^ (Arg.string betas1))
                      val rho3 = Rho.updateRhosimple (rho, rho2)
                    in
                      (None, betas, mb, rho3,
                       Util.empty, qmap, delta, eqmap, pi)
                    end)
              | None =>
```

```
                let val _ = Debug.print "MF" 9
                  (fn () => "PeProcCall - not member of MF")
                in
                  (case Pi.contains (pi, oid1, ProcName) of
                     Some(ResProcName, oid2) =>
                       let val _ = Debug.print "Pi" 3
                         (fn () => "Pi: " ^ (Pi.string pi))
                           val mb1 = MB.singleton(MB.Redo, oid2, ProcName,
                                                  ResProcName, Rho.empty,
                                                  Arg.empty)
                       in
                         (None, betas, mb1, Rho.empty, Util.empty,
                          qmap, delta, eqmap, pi)
                       end
                   | None =>
                       let val mf1 =
                         MF.remove (mf, MF.Any, oid1, ProcName, NewProcName)
                       in
                         PeDoProcCall (oid1, ProcName, betas, NewProcName,
                                       man, dyn, mf1, rho, qmap, delta,
                                       eqmap, pi)
                       end)
                end))
    end

  and PeDoProcCall (oid1, ProcName, betas, NewProcName,
                    man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    let val (mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
      PeProcBody (oid1, ProcName, betas, NewProcName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi)
      val mb1 = MB.remove (mb, ProcName, oid1, NewProcName)
    in
      if MB.exist (mb1, MB.Redo) then
        let val pi2 = Pi.remove (pi1, ProcName, NewProcName, oid1)
          val (newrho, newargs) =
            MB.selectRB (mb, MB.Redo, ProcName, oid1, NewProcName)
          val resrho = Rho.lubloop (rho, newrho)
          val resargs = Arg.lubloop (betas, newargs)
          val mf2 =
            MF.add (mf, MF.Iter, oid1, ProcName, NewProcName,
                    resrho, resargs, Some(rho1), AbsVal.empty)
          val mfsimple = MF.mf2simple mf2
          val mb2 = MB.set_container (mb1, mfsimple)
        in
```

```
        (None, betas, mb2, rho1, util, qmap1, delta1, eqmap1, pi2)
      end
    else if not (MB.member (mb, ProcName, oid1, NewProcName)) then
      let val pi2 = Pi.remove (pi1, ProcName, NewProcName, oid1)
      in
        (None, betas, mb1, rho1, util, qmap1, delta1, eqmap1, pi2)
      end
    else (* Give is used to reuse the first call of PeProcBody !!! *)
        if dyn orelse (not (MB.isCreated mb)) then
          let val mf1 = MF.add (mf, MF.Iter, oid1, ProcName, NewProcName,
                                rho, betas, None, AbsVal.empty)
            val dyn1 = true
            val rhosimple = Rho.rho2simple rho
            val cont = fn (rho, delta) =>
              (Delta.Explicator2r(rho, rhosimple, delta))
          in
            PeProcIterFirst (oid1, ProcName, betas, NewProcName, man, dyn1,
                             mf1, rho, qmap, delta, eqmap, pi,
                             (mb, rho1, util, qmap1, delta1, eqmap1, pi1),
                             cont)
          end
        else
          let val mf1 = MF.add (mf, MF.Iter, oid1, ProcName, NewProcName,
                                rho, betas, None, AbsVal.empty)
            val dyn1 = true
            val rhosimple = Rho.rho2simple rho
            val cont = fn (rho, delta) =>
              (Delta.Explicator2r(rho, rhosimple, delta))
          in
            PeProcIter (oid1, ProcName, betas, NewProcName,
                        man, dyn1, mf1, rho, qmap, delta, eqmap, pi, cont)
          end
  end

and PeProcIterFirst (oid1, ProcName, betas, NewProcName,
                man, dyn, mf, rho, qmap, delta, eqmap, pi, give, cont) =
  let val _ = Debug.print "Pe" 3
    (fn () => "**PeProcIterFirst called: " ^ (Rho.OID2string (rho, oid1)) ^
     " " ^ ProcName ^ " " ^ NewProcName ^ (Qmap.string qmap) ^
     (MF.string mf))
    val (mb, rho1, util, qmap1, delta1, eqmap1, _(*pi*)) = give
    val _ = Debug.print "RhoIter" 5
      (fn () => "PeProcIterFirst, after procbody\n" ^
       (Rho.fullystring rho1))
```

```
    val mb1 = MB.remove (mb, ProcName, oid1, NewProcName)
    val _ = Debug.print "MB" 7 (fn () => "PeProcIter " ^ MB.string mb)
  in
    if (not (MB.isExit mb)) andalso (MB.isEmpty mb1) then
      let val mf2 =
        MF.add (MF.remove (mf, MF.Iter, oid1, ProcName, NewProcName),
                MF.Unfold, oid1, ProcName, NewProcName, rho, betas,
                None, AbsVal.empty)
          val mfsimple = MB.get_container mb
          val mf3 = MF.set_container (mf2, mfsimple)
      in
        PeProcIter (oid1, ProcName, betas, NewProcName,
                    man, dyn, mf3, rho, qmap1, delta, eqmap, pi, cont)
      end
    else
      if not (MB.exist (mb, MB.Redo)) then
        let val (flag, rhoold, _) =
          MF.lookup (mf, oid1, ProcName, NewProcName)
          val result = Rho.lesseqsimple (rho1, rhoold)
          val _ = Debug.print "MF" 8
            (fn () => "lesseq RHO " ^ (Bool.string result) ^
             (if result then
                ""
              else
                (Rho.fullystring rho1) ^ " " ^
                (Rho.simple2string rhoold)))
        in
          if result then
            let val mfsimple = MF.mf2simple mf
              val mb2 = MB.set_container (mb1, mfsimple)
              val rho2 = Rho.updateRhosimple (rho1, rhoold)
              val newdelta = cont (rho, delta1)
            in
              (None, betas, mb2, rho2, util, qmap1, newdelta, eqmap1, pi)
            end
          else
            let val mf2 =
              MF.add (MF.remove (mf, flag, oid1, ProcName, NewProcName),
                      flag, oid1, ProcName, NewProcName, rho, betas,
                      Some(Rho.lubloopsim (rho1, rhoold)), AbsVal.empty)
              val mfsimple = MB.get_container mb
              val mf3 = MF.set_container (mf2, mfsimple)
            in
              PeProcIter (oid1, ProcName, betas, NewProcName, man, dyn,
```

```
                              mf3, rho, qmap1, delta1, eqmap1, pi, cont)
            end
        end
      else
        if (MB.exist (mb1, MB.Redo)) orelse
          (not (MB.member (mb, ProcName, oid1, NewProcName))) then
          let val mfsimple = MF.mf2simple mf
            val mb2 = MB.set_container (mb1, mfsimple)
            val newdelta = cont (rho, delta1)
          in
            (None, betas, mb2, rho1, util, qmap1, newdelta, eqmap1, pi)
          end
        else
          let val (R, B) =
            MB.selectRB (mb, MB.Redo, ProcName, oid1, NewProcName)
              val rho2 = Rho.lubloop (rho, R)
              val betas1 = Arg.lubloop (betas, B)
              val (flag, rhoold, _) =
                MF.lookup (mf, oid1, ProcName, NewProcName)
              val mf2 =
                MF.add (MF.remove (mf, flag, oid1, ProcName, NewProcName),
                        flag, oid1, ProcName, NewProcName, rho2, betas1,
                        Some(Rho.lubloopsim (rho1, rhoold)), AbsVal.empty)
              val mfsimple = MB.get_container mb
              val mf3 = MF.set_container (mf2, mfsimple)
          in
            PeProcIter (oid1, ProcName, betas1, NewProcName, man, dyn,
                        mf3, rho2, qmap1, delta1, eqmap1, pi, cont)
          end
  end

and PeProcIter (oid1, ProcName, betas, NewProcName,
                man, dyn, mf, rho, qmap, delta, eqmap, pi, cont) =
  let val _ = Debug.print "Pe" 3
    (fn () => "**PeProcIter called: " ^ (Rho.OID2string (rho, oid1)) ^
     " " ^ ProcName ^ " " ^ NewProcName ^ (Qmap.string qmap) ^
     (MF.string mf))
    val (mb, rho1, util, qmap1, delta1, eqmap1, _(*pi*)) =
      PeProcBody (oid1, ProcName, betas, NewProcName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi)
    val _ = Debug.print "RhoIter" 5
      (fn () => "PeProcIter, after procbody\n" ^ (Rho.fullystring rho1))
    val mb1 = MB.remove (mb, ProcName, oid1, NewProcName)
    val _ = Debug.print "MB" 7 (fn () => "PeProcIter " ^ MB.string mb)
```

```
in
  if (not (MB.isExit mb)) andalso (MB.isEmpty mb1) then
    let val mf2 =
      MF.add (MF.remove (mf, MF.Iter, oid1, ProcName, NewProcName),
              MF.Unfold, oid1, ProcName, NewProcName, rho, betas,
              None, AbsVal.empty)
        val mfsimple = MB.get_container mb
        val mf3 = MF.set_container (mf2, mfsimple)
    in
      PeProcIter (oid1, ProcName, betas, NewProcName,
                  man, dyn, mf3, rho, qmap1, delta, eqmap, pi, cont)
    end
  else
    if not (MB.exist (mb, MB.Redo)) then
      let val (flag, rhoold, _) =
        MF.lookup (mf, oid1, ProcName, NewProcName)
        val result = Rho.lesseqsimple (rho1, rhoold)
        val _ = Debug.print "MF" 8
          (fn () => "lesseq RHO " ^ (Bool.string result) ^
           (if result then
              ""
            else
              (Rho.fullystring rho1) ^ " " ^
              (Rho.simple2string rhoold)))
      in
        if result then
          let val mfsimple = MF.mf2simple mf
            val mb2 = MB.set_container (mb1, mfsimple)
            val rho2 = Rho.updateRhosimple (rho1, rhoold)
            val newdelta = cont (rho, delta1)
          in
            (None, betas, mb2, rho2, util, qmap1, newdelta, eqmap1, pi)
          end
        else
          let val mf2 =
            MF.add (MF.remove (mf, flag, oid1, ProcName, NewProcName),
                    flag, oid1, ProcName, NewProcName, rho, betas,
                    Some(Rho.lubloopsim (rho1, rhoold)), AbsVal.empty)
            val mfsimple = MB.get_container mb
            val mf3 = MF.set_container (mf2, mfsimple)
          in
            PeProcIter (oid1, ProcName, betas, NewProcName, man, dyn,
                        mf3, rho, qmap1, delta1, eqmap1, pi, cont)
          end
```

```
          end
        else
          if (MB.exist (mb1, MB.Redo)) orelse
            (not (MB.member (mb, ProcName, oid1, NewProcName))) then
            let val mfsimple = MF.mf2simple mf
              val mb2 = MB.set_container (mb1, mfsimple)
              val newdelta = cont (rho, delta1)
            in
              (None, betas, mb2, rho1, util, qmap1, newdelta, eqmap1, pi)
            end
          else
            let val (R, B) =
              MB.selectRB (mb, MB.Redo, ProcName, oid1, NewProcName)
                val rho2 = Rho.lubloop (rho, R)
                val betas1 = Arg.lubloop (betas, B)
                val (flag, rhoold, _) =
                  MF.lookup (mf, oid1, ProcName, NewProcName)
                val mf2 =
                  MF.add (MF.remove (mf, flag, oid1, ProcName, NewProcName),
                          flag, oid1, ProcName, NewProcName, rho2, betas1,
                          Some(Rho.lubloopsim (rho1, rhoold)), AbsVal.empty)
                val mfsimple = MB.get_container mb
                val mf3 = MF.set_container (mf2, mfsimple)
            in
              PeProcIter (oid1, ProcName, betas1, NewProcName, man, dyn,
                          mf3, rho2, qmap1, delta1, eqmap1, pi, cont)
            end
    end

  and PeProcBody (oid as (AbsVal.Static(1)), (* nil *)
    ProcName, betas, NewProcName, man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    (Debug.print "Pe" 1 (fn () => ("Nil invoked : " ^ ProcName));
     (MB.empty, Rho.empty, Util.empty, qmap, delta, eqmap, pi))
    | PeProcBody (oid, ProcName, betas,
                  NewProcName, man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    if Rho.isEmpty rho then
      (Debug.print "Pe" 6 (fn () => "Rho is empty");
       (MB.empty, Rho.empty, Util.empty, qmap, delta, eqmap, pi))
    else
      let val mdefs = Rho.code (rho, oid)
      in
        case lookupPDEF (ProcName, mdefs) of
          None =>
            (Debug.print "Pe" 2
```

```
          (fn () => ("Procedure not found: " ^ ProcName) ^ " " ^
           (Rho.OID2string (rho, oid) ^ (Rho.string rho)));
          (MB.empty, Rho.empty, Util.empty, qmap, delta, eqmap, pi))
    | Some(fps, lvs, pbody) =>
        let val _ = Debug.print "Pe" 4
          (fn () => "ProcBody " ^ ProcName ^ (Arg.string betas) ^
           " : " ^ (Rho.OID2string (rho, oid)))
          val initlabel = MakeLabel "m"
          val tau = Tau.build (fps, betas, initlabel)
          val call = CALL(ProcName, NewProcName, rho, betas, None)
          val (_, mb, _, newpbbs, rho1, tau1,
               util, qmap1, delta1, eqmap1, pi1) =
            PeBody (pbody, dyn, call, mf, oid, rho, tau,
                    qmap, delta, eqmap, pi)
          val newpbody =
            Simili.SBODY(Simili.SBB(initlabel, [],
                                    Simili.SGOTO(firstlabel newpbbs)) ::
                         newpbbs)
          val newfps =
            (if man then FilterOutGlobals (betas, fps, rho1) else fps)
          val residproc =
            Simili.SPROCEDURE(NewProcName, newfps, lvs, newpbody)
          val newproc =
            if ((not (MB.isCreated mb)) orelse dyn) andalso
              Rho.subdomutil(util, rho) andalso
              Rho.subdomutil(util, rho1) then
              Rho.ResProc(Rho.Pre(Rho.rho2cutsimple(rho, util),
                                  betas, man, MB.isCreated mb),
                          NewProcName, residproc,
                          Rho.PPC(Rho.rho2cutsimple(rho1, util),
                                  qmap1, util))
            else
              Rho.NoPReuse(NewProcName, residproc)
          val (name, procset, fnctset, alias, ribody) =
            Rho.rescode (rho1, oid)
          val newprocset = Rho.AddProc (procset, ProcName, newproc)
          val newrescode =
            (name, newprocset, fnctset, alias, ribody)
          val rho2 = Rho.updateResCode (rho1, oid, newrescode)
        in
          (mb, rho2, util, qmap1, delta1, eqmap1, pi1)
        end
end
```

```
and PeFnctCall(oid1, FnctName, betas, NewFnctName,
               man, dyn, mf, rho, qmap, delta, eqmap, pi) =
  case Rho.ReuseFnct(rho, oid1, FnctName, betas, NewFnctName, dyn, man) of
    Rho.FNoUse =>
      PeAFnctCall(oid1, FnctName, betas, NewFnctName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi)
  | Rho.FReused(rfn, newrho, resultbeta, newutil) =>
      (rfn, resultbeta, betas, MB.empty, newrho, newutil, qmap,
       delta, eqmap, pi)
  | Rho.FUseNoReuse(_, _, _) =>
      raise PE("Impossible - internal error. Method: " ^ FnctName ^
               "invoked.  Rho: " ^ (Rho.fullystring rho))

and PeAFnctCall (oid1, FnctName, betas, NewFnctName,
                 man, dyn, mf, rho, qmap, delta, eqmap, pi) =
  let val _ = Debug.print "Pe" 5
    (fn () =>
     ("PeFnctCall: " ^ FnctName ^ ", called in object: " ^
      (AbsVal.OID2string oid1) ^ " arguments " ^
      (Arg.string betas) ^ " " ^ (MF.string mf)))
  in
    (case MF.exists_unfold (mf, oid1, FnctName, rho, betas) of
       Some(ResFnctName, rho2, beta) =>
         let val mb = MB.singleton (MB.Use, oid1, FnctName,
                                    ResFnctName, rho, betas)
           val rho3 = (case rho2 of
                         None => rho
                       | Some(rho2) =>
                           Rho.updateRhosimple (rho, rho2))
           val (rfn, rho4) =
             if man then
               (Some(ResFnctName), rho3)
             else
               (None,
                Rho.functionalias (rho3, oid1, ResFnctName, NewFnctName))
         in
           (rfn, beta, betas, mb, rho4,
            Util.empty, qmap, delta, eqmap, pi)
         end
     | None =>
         (case MF.contains (mf, MF.Iter, oid1, FnctName) of
            Some(ResFnctName, rho1, betas1, rho2, beta) =>
              (case rho2 of
                 None =>
```

```
        let val mb = MB.singleton (MB.Redo, oid1, FnctName,
                                   ResFnctName, rho, betas)
        in
          (None, beta, betas, mb, rho,
           Util.empty, qmap, delta, eqmap, pi)
        end
    | Some(rho2) =>
        if (Rho.equalsimple (rho, rho1)) andalso
          (Arg.lesseq (betas, betas1)) then
          let val mb = MB.singleton (MB.Use, oid1, FnctName,
                                     ResFnctName, rho, betas)
            val _ = Debug.print "MB" 6
              (fn () =>
               ("PeFnctCall: result reached, alias " ^
                NewFnctName ^ " with " ^ ResFnctName))
            val _ = Debug.print "Pe" 9
              (fn () => "Rho b4 alias: " ^ (Rho.fullystring rho))
            val rho3 = Rho.updateRhosimple (rho, rho2)
            val (rfn, rho4) =
              if man then
                (Some(ResFnctName), rho3)
              else
                (None, Rho.functionalias (rho3, oid1,
                                          ResFnctName,
                                          NewFnctName))
            val _ = Debug.print "Pe" 9
              (fn () => "Rho after alias: " ^
               (Rho.fullystring rho4))
          in
            (rfn, beta, betas1, mb, rho4,
             Util.empty, qmap, delta, eqmap, pi)
          end
        else
          let val rhosave = Rho.difference (rho, rho1)
            val mb = MB.singleton (MB.Redo, oid1, FnctName,
                                   ResFnctName, rhosave, betas)
            val _ = Debug.print "MB" 6
              (fn () => "PeFnctCall: try again\n:Rho equal? " ^
               (Bool.string (Rho.equalsimple (rho, rho1))))
            val _ = Debug.print "MB" 9
              (fn () => "Rho: " ^ (Rho.string rho) ^
               "\nRho1: " ^ (Rho.simple2string rho1) ^
              "\nArg less-equal? " ^
               (Bool.string (Arg.lesseq (betas, betas1))) ^
```

```
                        "\nArg: " ^ (Arg.string betas) ^
                        "\nArg1: " ^ (Arg.string betas1))
                    val rho3 = Rho.updateRhosimple (rho, rho2)
                in
                  (None, beta, betas, mb, rho3,
                   Util.empty, qmap, delta, eqmap, pi)
                  end)
          | None =>
              (case Pi.contains (pi, oid1, FnctName) of
                 Some(ResFnctName, oid2) =>
                   let val _ = Debug.print "Pi" 3
                     (fn () => "Pi: " ^ (Pi.string pi))
                       val mb1 = MB.singleton(MB.Redo, oid2, FnctName,
                                              ResFnctName, Rho.empty,
                                              Arg.empty)
                   in
                     (None, AbsVal.empty, betas, mb1, Rho.empty,
                      Util.empty, qmap, delta, eqmap, pi)
                   end
               | None =>
                   let val mf1 =
                     MF.remove (mf, MF.Any, oid1, FnctName, NewFnctName)
                   in
                     PeDoFnctCall (oid1, FnctName, betas, NewFnctName,
                                   man, dyn, mf, rho, qmap, delta,
                                     eqmap, pi)
                   end)))
  end

and PeDoFnctCall (oid1, FnctName, betas, NewFnctName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi) =
  let val _ = Debug.print "Pe" 6 (fn () => "PeDoFnctCall - entry point")
    val (beta, mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
    PeFnctBody (oid1, FnctName, betas, NewFnctName,
                man, dyn, mf, rho, qmap, delta, eqmap, pi)
    val mb1 = MB.remove (mb, FnctName, oid1, NewFnctName)
  in
    if (MB.exist (mb1, MB.Redo)) orelse
      (not (MB.member (mb, FnctName, oid1, NewFnctName))) then
      let val pi2 = Pi.remove (pi1, FnctName, NewFnctName, oid1)
      in
        (None, beta, betas, mb1, rho1, util, qmap1, delta1, eqmap1, pi2)
      end
    else
```

```
      let val give =
        if dyn orelse (not (MB.isCreated mb)) then
          Some(beta, mb, rho1, util, qmap1, delta1, eqmap1, pi1)
        else
          None
      in
        PeDoFnctCallTail(mf, oid1, FnctName, NewFnctName, rho, betas, man,
                         qmap, delta, eqmap, dyn, pi, give)
      end
end

and PeDoFnctCallTail(mf, oid1, FnctName, NewFnctName, rho, betas, man,
                     qmap, delta, eqmap, dyn, pi, give) =
  let val mf1 = MF.add (mf, MF.Iter, oid1, FnctName, NewFnctName,
                        rho, betas, None, AbsVal.empty)
      val dyn1 = true
      val (beta1, betas1, mb2, rho2, rho3,
           util, qmap2, delta2, eqmap2) =
        PeFnctIter (oid1, FnctName, betas, NewFnctName,
                    man, dyn1, mf1, rho, qmap, delta, eqmap, pi, give)
      val delta3 = Delta.Explicator2r(rho, Rho.rho2simple rho2, delta2)
      val _ = Debug.print "Pe" 9
        (fn () => ("PeDoFnctCall - exit point (after PeFnctIter)" ^
                   (MB.string mb2) ^ "\n" ^ (Rho.fullystring rho3)))
  in
    (None, beta1, betas1, mb2, rho3, util,
     (if dyn then qmap2 else qmap), delta3, eqmap2, pi)
  end

and PeFnctIter (oid1, FnctName, betas, NewFnctName,
                man, dyn, mf, rho, qmap, delta, eqmap, pi, give) =
  let val _ = Debug.print "Pe" 3
    (fn () => "**PeFnctIter called: " ^ (Rho.OID2string (rho, oid1)) ^
     " " ^ FnctName ^ " " ^ NewFnctName)
      val (beta, mb, rho1, util, qmap1, delta1, eqmap1, _(*pi*)) =
        (case give of
           Some(value) => value
         | None =>
             PeFnctBody (oid1, FnctName, betas, NewFnctName,
                         man, dyn, mf, rho, qmap, delta, eqmap, pi))
      val _ = Debug.print "RhoIter" 5
        (fn () => "PeFnctIter, after fnctbody\n" ^ (Rho.fullystring rho1))
      val mb1 = MB.remove (mb, FnctName, oid1, NewFnctName)
      val _ = Debug.print "MB" 5
```

```
      (fn () => ("PeFnctIter >> " ^ (MB.string mb)))
in
  if (not (MB.isExit mb)) andalso (MB.isEmpty mb1) then
    let val mf2 =
      MF.add (MF.remove (mf, MF.Iter, oid1, FnctName, NewFnctName),
              MF.Unfold, oid1, FnctName, NewFnctName,
              rho, betas, None, AbsVal.empty)
        val mfsimple = MB.get_container mb
        val mf3 = MF.set_container (mf2, mfsimple)
    in
      PeFnctIter (oid1, FnctName, betas, NewFnctName,
                  man, dyn, mf3, rho, qmap, delta, eqmap, pi, None)
    end
  else
    if not (MB.exist (mb, MB.Redo)) then
      let val (flag, rhoold, betaold) =
        MF.lookup (mf, oid1, FnctName, NewFnctName)
      in
        if (AbsVal.lesseq (beta, betaold)) andalso
          (Rho.lesseqsimple (rho1, rhoold)) then
          let val rho2 = Rho.updateRhosimple (rho1, rhoold)
          in
            (betaold, betas, mb1, rho, rho2, util, qmap1, delta1, eqmap1)
          end
        else
          let val mf2 =
            MF.add (MF.remove (mf, flag, oid1, FnctName, NewFnctName),
                    flag, oid1, FnctName, NewFnctName, rho, betas,
                    Some(Rho.lubloopsim (rho1, rhoold)),
                    AbsVal.lubloop (beta, betaold))
            val mfsimple = MB.get_container mb
            val mf3 = MF.set_container (mf2, mfsimple)
          in
            PeFnctIter (oid1, FnctName, betas, NewFnctName, man, dyn,
                        mf3, rho, qmap1, delta1, eqmap1, pi, None)
          end
      end
    else
      if (MB.exist (mb1, MB.Redo)) orelse
        (not (MB.member (mb, FnctName, oid1, NewFnctName))) then
        let val mfsimple = MF.mf2simple mf
          val mb2 = MB.set_container (mb1, mfsimple)
        in
          (beta, betas, mb2, rho, rho1, util, qmap1, delta1, eqmap1)
```

```
            end
          else
            let val (R, B) =
              MB.selectRB (mb, MB.Redo, FnctName, oid1, NewFnctName)
                val rho2 = Rho.lubloop (rho, R)
                val betas1 = Arg.lubloop (betas, B)
                val (flag, rhoold, betaold) =
                  MF.lookup (mf, oid1, FnctName, NewFnctName)
                val mf2 =
                  MF.add (MF.remove (mf, flag, oid1, FnctName, NewFnctName),
                          flag, oid1, FnctName, NewFnctName, rho2, betas1,
                          Some(Rho.lubloopsim (rho1, rhoold)),
                          AbsVal.lubloop (beta, betaold))
                val mfsimple = MB.get_container mb
                val mf3 = MF.set_container (mf2, mfsimple)
            in
              PeFnctIter (oid1, FnctName, betas1, NewFnctName, man, dyn,
                          mf3, rho2, qmap1, delta1, eqmap1, pi, None)
            end
    end

  and PeFnctBody (oid as (AbsVal.Static(1)), FnctName, betas, NewFnctName,
                  man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    (Debug.print "Pe" 1 (fn () => "Nil invoked : " ^ FnctName);
     (AbsVal.empty, MB.empty, Rho.empty,
      Util.empty, qmap, delta, eqmap, pi))
    | PeFnctBody (oid, FnctName, betas,
                  NewFnctName, man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    if Rho.isEmpty rho then
      (AbsVal.empty, MB.empty, Rho.empty,
       Util.empty, qmap, delta, eqmap, pi)
    else
      let val mdefs = Rho.code (rho, oid) in
        case lookupFDEF (FnctName, mdefs) of
          None => (Debug.print "Pe" 2
                   (fn () => ("Function not found: " ^ FnctName ^
                              " " ^ (Rho.OID2string (rho, oid)) ^
                                    (Rho.string rho)));
                   (AbsVal.empty, MB.empty, Rho.empty,
                    Util.empty, qmap, delta, eqmap, pi))
        | Some(fps, rn, lvs, fbody) =>
            let val initlabel = MakeLabel "m"
              val tau = Tau.build (fps, betas, initlabel)
              val call = CALL(FnctName, NewFnctName, rho, betas,
```

```
                Some(rn))
val _ = Debug.print "Pe" 4
  (fn () => "FnctBody " ^ FnctName ^ " " ^
   (Rho.OID2string (rho, oid) ^ " " ^
    (Arg.string betas)))
val _ = Debug.print "Pe" 9
  (fn () => "PeFnctBody -> RHO: " ^ (Rho.fullystring rho))
val (_, mb, _, newfbbs, rho1, tau1,
     util, qmap1, delta1, eqmap1, pi1) =
   PeBody (fbody, dyn, call, mf, oid, rho, tau,
           qmap, delta, eqmap, pi)
val newfbody =
  Simili.SBODY(Simili.SBB(initlabel, [],
                          Simili.SGOTO(firstlabel newfbbs)) ::
               newfbbs)
val betaresult = Tau.lookup (tau1, rn)
val _ = Debug.print "Pe" 5
  (fn () => ("After FnctBody " ^ FnctName ^ " : " ^
             (Rho.OID2string (rho, oid)) ^ " " ^
             " -> [" ^ (AbsVal.string betaresult) ^ "]"))
val newfps =
  (if man then FilterOutGlobals (betas, fps, rho1) else fps)
val (name, procset, fnctset, alias, ribody) =
  (Rho.rescode (rho1, oid)
   handle Rho.Rho(s) => raise PE("FnctBody, " ^ s))
val (global, resfbody, reslvs) =
  (case Rho.Globalizable (rho1, betaresult) of
     None => (false, newfbody, lvs)
   | Some(be) =>
       let val lbl = MakeLabel("n") in
         (true,
          Simili.SBODY(Simili.SBB(lbl,
                                  Simili.SASSIGNSTMNT(Simili.SRESULTNAM
                                  Simili.SBASICEXPR(be)) :: [],
                                  Simili.SRETURN) :: []), [])
       end)
val residfnct =
  Simili.SFUNCTION(NewFnctName, newfps, rn, reslvs, resfbody)
val newfnct =
  if ((not (MB.isCreated mb)) orelse dyn) andalso
    Rho.subdomutil(util, rho) andalso
    Rho.subdomutil(util, rho1) then
    Rho.ResFnct(Rho.Pre(Rho.rho2cutsimple(rho, util),
                        betas, man, MB.isCreated mb),
```

```
                              NewFnctName, residfnct,
                              Rho.FPC(betaresult,
                                      Rho.rho2cutsimple(rho1, util),
                                      qmap1, util))
                  else
                    Rho.NoFReuse(NewFnctName, residfnct)
              val newfnctset = Rho.AddFnct (fnctset, FnctName, newfnct)
              val newrescode =
                (name, procset, newfnctset, alias, ribody)
              val rho2 = Rho.updateResCode (rho1, oid, newrescode)
            in
              if global andalso man then
                let val _ = Debug.print "Pe" 5
                  (fn () => "Global and manifest function: " ^
                   (Rho.OID2string (rho, oid)) ^ "." ^ FnctName ^ " removed")
                in
                  (betaresult, mb, rho1, util, qmap1, delta1, eqmap1, pi1)
                end
              else
                (betaresult, mb, rho2, util, qmap1, delta1, eqmap1, pi1)
            end
      end

  and PeHardFnctCall (AbsVal.ABaseValue(bv), FnctName, args,
                      man, dyn, mf, rho, qmap, delta, eqmap, pi) =
    (* Manifest flag, man, always false when called *)
    let val beta = PeBaseValue.DoFnctCall (bv, FnctName, args)
    in
      (beta, MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
    end
    | PeHardFnctCall (AbsVal.InputDynamic, FnctName, args,
                      man, dyn, mf, rho, qmap, delta, eqmap, pi) =
      (AbsVal.fromobject(AbsVal.InputDynamic),
       MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
    | PeHardFnctCall (AbsVal.AnObject(oid), FnctName, args,
                      man, dyn, mf, rho, qmap, delta, eqmap, pi) =
      if oid = AbsVal.GlobalOID then
        (AbsVal.empty, MB.empty, rho, Util.empty, qmap, delta, eqmap, pi)
      else
        case Rho.ReuseFnct(rho, oid, FnctName, args, FnctName, dyn, man) of
          Rho.FNoUse =>
            let val dyn1 = true
              val (_, beta, _, mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
                PeAFnctCall (oid, FnctName, args, FnctName,
```

```
                              man, dyn1, mf, rho, qmap, delta, eqmap, pi)
            in
              (beta, mb, rho1, util, qmap1, delta1, eqmap1, pi1)
            end
        | Rho.FReused(rfn, newrho, beta, util) =>
            (beta, MB.empty, newrho, util, qmap, delta, eqmap, pi)
        | Rho.FUseNoReuse(rhob4, argsb4, qmapb4) =>
            let val rhonew = Rho.lubifsim (rhob4, rho)
              val deltanew = Delta.Explicator2r (rhonew, rhob4, delta)
              val argnew = Arg.lubif (argsb4, args)
              val qmapnew = Qmap.lubif (qmapb4, qmap)
              val dyn1 = true
              val (_, beta, _, mb, rho1, util, qmap1, delta1, eqmap1, pi1) =
                PeAFnctCall (oid, FnctName, argnew, FnctName,
                             man, dyn1, mf, rhonew, qmapnew,
                             deltanew, eqmap, pi)
            in
              (beta, mb, rho1, util, qmap1, delta1, eqmap1, pi1)
            end


(* Main routine *)
local
  fun name2absval (name, rho) =
    let val value =
      Rho.lookup (rho, AbsVal.GlobalOID, name)
    in
      value
    end

  fun value2absval (GOWName(name), rho) = name2absval (name, rho)
    | value2absval (InputDynamic, _) = AbsVal.IDVal
    | value2absval (Integer(i), _) = AbsVal.bv2value(AbsVal.Integer(i))
    | value2absval (Integers, _) = AbsVal.bv2value(AbsVal.Integers)
    | value2absval (Boolean(b), _) = AbsVal.bv2value(AbsVal.Boolean(b))
    | value2absval (Char(c), _) = AbsVal.bv2value(AbsVal.Char(c))
    | value2absval (Chars, _) = AbsVal.bv2value(AbsVal.Chars)
    | value2absval (String(s), _) = AbsVal.bv2value(AbsVal.String(s))
    | value2absval (Strings, _) = AbsVal.bv2value(AbsVal.Strings)
    | value2absval (Unix, _) = AbsVal.bv2value(AbsVal.Unix)
    | value2absval (NoValue, _) = AbsVal.empty
in
  fun mype (Simili.SPROGRAM(program),
            constname, procname, arglist, fnctname) =
```

```
let val _ = (initlabel();
             initlbl();
             initproc();
             initfnct();
             initobject();
             AbsVal.initOID())
  val _ = Debug.print "Pe" 3 (fn () => "Start to load program.")
  val _ = Debug.change ~4
  val rho37 =
    Rho.addcode (Rho.empty, AbsVal.GlobalOID, AbsVal.GlobalOC, [])
  val rho = Rho.updateResCode (rho37, AbsVal.GlobalOID,
                               ("nil", [], [],
                                EqSet.empty, Simili.SBODY([])))
  val (rho1, eqmap1, delta, newcds) =
    PeProgram (program, rho, EQMap.empty, Delta.empty, [])
  val absvallist =
      List.map (fn name => value2absval (name, rho1)) arglist
  val absval = name2absval (constname, rho1)
  val _ = Debug.change 4
  val _ = Debug.print "Delta" 7 (fn () => (Delta.string delta))
in
  case AbsVal.isStatica absval of
    Some(oid) =>
      let val _ = Debug.print "Pe" 3
        (fn () =>
         ("Program loaded\n" ^
          "run : " ^ procname ^ " in object " ^ constname ^
          " with arguments [" ^
          (String.implode
           (List.map AbsVal.string absvallist)) ^ "]"))
          val _ = Debug.print "Pe" 7
            (fn () => "RHO after load:\n" ^ (Rho.fullystring rho1))
          val (ResProcName, _, _, newrho,
               util, qmap1, delta1, eqmap2, pi1) =
            PeProcCall (oid, procname, absvallist, procname, true,
                        false, MF.empty, rho1, Qmap.empty,
                        delta, eqmap1, Pi.empty)
          val _ = Debug.print "Pe" 3
            (fn () => "PeProcCall OK --> PeFnctCall")
          val _ = Debug.print "QMAP" 5
            (fn () => Rho.fullystring newrho)
          val (_, _, _, _, resrho, util1, qmap2, delta2, eqmap3, _) =
            PeFnctCall (oid, fnctname, [], fnctname, false, false,
                        MF.empty, newrho, qmap1, delta1, eqmap2, pi1)
```

```
            val _ = Debug.print "Pe" 3 (fn () => "Program transformed!")
            val (newcn, _, _, _, _) = Rho.rescode (resrho, oid)
        in
          (resrho, delta2, newcn, newcds, eqmap3)
        end
    | None =>
        raise PE("pe - constname not an object! " ^ constname ^ " " ^
                 (AbsVal.string absval))
  end

fun localpe (program, constname, procname, arglist, fnctname) =
  let val (rho, delta, _, newcds, eq_map) =
    mype (program, constname, procname, arglist, fnctname)
  in
    (Print.program (rho, delta, newcds), eq_map)
  end handle
  PE(s) =>
    (Outstream.write std_out
     ("Error in Pe module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| Rho.Rho(s) =>
    (Outstream.write std_out
     ("Error in Rho module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| Tau.Tau(s) =>
    (Outstream.write std_out
     ("Error in Tau module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| PeBaseValue.BaseValue(s) =>
    (Outstream.write std_out
     ("Error in BaseValue module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| LB.LABELBACKWARD(s) =>
    (Outstream.write std_out
     ("Error in LabelBackward module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| MB.MethodBackward(s) =>
    (Outstream.write std_out
     ("Error in MethodBackward module: " ^ s ^ "\n");
     raise PE("Abort pe"))
| AbsVal.Value(s) =>
    (Outstream.write std_out
     ("Error in AbsVal module: " ^ s ^ "\n");
     raise PE("Abort pe"))
```

```
    fun pe (program, constname, procname, arglist, fnctname) =
      let val (newprogram, _) =
        localpe (program, constname, procname, arglist, fnctname)
      in
        newprogram
      end
    fun be (program, constname, procname, arglist, fnctname) =
      let val value =
        localpe (program, constname, procname, arglist, fnctname)
      in
        value
      end
  end
end;
```

Appendix D

The selfinterpreter

D.1 A list

```
const stdin == Unix.stdin[]
const stdout == Unix.stdout[]

(* Index: [1..n] and NOT [0..(n-1)] - very important. *)
const list == object list
   function of [listtype] -> [result]
      result <- object thelist
 var mytype
 var parsevalue

 function cons[element, alist] -> [result]
    result <- object consCell
       var head
       var tail
       var returnvalue

       function head [] -> [something]
  something <- head
       end head

       function tail [] -> [something]
  something <- tail
       end tail

       function getElement[int] -> [something]
  var nextint

  if int.gt[1] then
     nextint <- int.minus[1]
```

```
   something <- tail.getElement[nextint]
elseif int.eq[1] then
   something <- head
else
   something <- nil.error["Negative argument to getElement"]
end if
     end getElement

     procedure setElement[int, something]
var nextint

if int.gt[1] then
   nextint <- int.minus[1]
   tail.setElement[nextint, something]
elseif int.eq[1] then
   head <- something
else
   nextint <- nil.error["Negative  argument to setElement"]
end if
     end setElement

     procedure reverse [newtail]
if newtail == nil then
   skip
elseif newtail == emptylist then
   skip
else
   newtail.reverse[self]
end if
tail <- newtail
     end reverse

     function reverse[newtail] -> [result]
result <- genericlist.cons[head, newtail]
if tail == nil then
   skip
elseif tail == emptylist then
   skip
else
   result <- tail.reverse[result]
end if
     end reverse

     function length [] -> [result]
```

```
var temp

temp <- tail.length[]
result <- temp.plus[1]
     end length

     function asString [] -> [result]
var str

result <- head.asString []
str <- tail.asString []
str <- " ".catenate[str]
result <- result.catenate[str]
     end asString

     function asStringFull [astr] -> [result]
var str

str <- head.asString []
result <- astr.catenate[str]
str <- tail.asStringFull [astr]
str <- " ".catenate[str]
result <- result.catenate[str]
     end asStringFull

     procedure PrettyPrint [stream]
var str

str <- self.asString []
stream.putString[str]
     end PrettyPrint

     procedure load [theStore]
head.load[theStore]
tail.load[theStore]
     end load

     procedure introduce [Env]
Env.introduce[head]
tail.introduce[Env]
     end introduce

     procedure initialize [Env, alist]
var temp
```

```
Env.introduce[head]
temp <- alist.head[]

Env.setvalue[head, temp]
temp <- alist.tail[]
tail.initialize[Env, temp]
     end initialize

     procedure evaluate [theStore, IEnv, LEnv, CurrentSelf]
var theValue
var theRest

head.evaluate[theStore, IEnv, LEnv, CurrentSelf]
theValue <- head.evaluateresult[]
tail.evaluate[theStore, IEnv, LEnv, CurrentSelf]
theRest <- tail.evaluateresult[]
returnvalue <- genericlist.cons[theValue, theRest]
     end evaluate

     function evaluateresult [] -> [result]
result <- returnvalue
     end EvaluateResult

     procedure execute [theStore, IEnv, LEnv, CurrentSelf]
head.execute[theStore, IEnv, LEnv, CurrentSelf]
tail.execute[theStore, IEnv, LEnv, CurrentSelf]
     end evaluate

     procedure bbexec [thestore, IEnv, LEnv, CurrentSelf]
head.bbexec[thestore, IEnv, LEnv, CurrentSelf, self]
     end bbexec

     function lookup[key] -> [result]
if head.same_key[key] then
   result <- head
else
   result <- tail.lookup[key]
end if
     end lookup

     initially
head <- element
tail <- alist
```

```
      end initially
   end consCell
end cons

procedure parse []
   var theToken
   var templist
   var tempelement

   theToken <- token.gettoken[]
   templist <- emptylist
   if theToken.equal["list"] then
      token.readnexttoken[]
      loop
 theToken <- token.gettoken[]
 exit when theToken.equal["end"]
 mytype.parse[]
 tempelement <- mytype.parseresult[]
 templist <- self.cons[tempelement, templist]
      end loop
      token.readnexttoken[]
      parsevalue <- templist.reverse [emptylist]
   else
      parsevalue <- nil.error["Error in program (no LIST keyword)\n"]
   end if
end parse

function parseresult [] -> [result]
   result <- parsevalue
end parseresult

initially
   mytype <- listtype
end initially
     end thelist
  end of
end list

const token == object token
   var stringpointer
   var programstring
   var stringlength
   var linenumber
   var thistoken
```

```
   procedure setstring[astring]
      programstring <- astring.catenate[" End of string "]
      stringlength <- programstring.length[]
      stringpointer <- 0
      linenumber <- 1
   end setstring

   function linenumber [] -> [line]
      line <- linenumber
   end linenumber

   function gettoken [] -> [result]
      result <- thistoken
   end gettoken

   procedure readnexttoken []
      var exitcond
      var exitcondtwo
      var startpointer
      var temppointer
      var tempchar

      if stringlength.lt[stringpointer] then
 tempchar <- nil.error["error, end of string reached\n"]
 thistoken <- ""
      else
 loop (* skip whitespace *)
    tempchar <- programstring.getElement[stringpointer]
    stringpointer <- stringpointer.plus[1]
    if tempchar.eq['\n'] then
       linenumber <- linenumber.plus[1]
    end if
    exitcond <- tempchar.neq[' ']
(*     exitcondtwo <- tempchar.neq['\t']
    exitcond <- exitcond.and[exitcondtwo]*)
    exitcondtwo <- tempchar.neq['\n']
    exitcond <- exitcond.and[exitcondtwo]
    exit when exitcond
 end loop
 startpointer <- stringpointer.minus[1]
 if tempchar.eq['"'] then (* it is a string *)
    loop
       tempchar <- programstring.getElement[stringpointer]
```

```
       exit when tempchar.eq['"']
       stringpointer <- stringpointer.plus[1]
    end loop
    thistoken <- programstring.getSlice[startpointer, stringpointer]
 else
    loop
       exitcond <- tempchar.eq[' ']
       exitcondtwo <- tempchar.eq['\n']
       exitcond <- exitcond.or[exitcondtwo]
       exit when exitcond
       tempchar <- programstring.getElement[stringpointer]
       stringpointer <- stringpointer.plus[1]
    end loop
    temppointer <- stringpointer.minus[2]
    thistoken <- programstring.getSlice[startpointer, temppointer]
 end if
      end if
   end readnexttoken
end token

const emptylist == object emptylist
   function getElement[int] -> [something]
      something <- nil.error["List is not long enough in getElement (emptylist)"]
   end getElement

   procedure setElement[int, something]
      var dummy

      dummy <- nil.error["List is not long enough in setElement (emptylist)"]
   end setElement

   procedure reverse [newtail]
      var dummy

      dummy <- nil.error["List is not long enough in reverse (emptylist)"]
   end reverse

   function reverse[newtail] -> [result]
      result <- self
   end reverse

   function length [] -> [result]
      result <- 0
   end length
```

```
function asString [] -> [result]
   result <- ""
end asString

function asStringFull [a] -> [result]
   result <- ""
end asStringFull

procedure PrettyPrint [stream]
   skip
end PrettyPrint

procedure evaluate [theStore, IEnv, LEnv, CurrentSelf]
   skip
end evaluate

function evaluateresult [] -> [result]
   result <- self
end evaluateresult

procedure load [theStore]
   skip
end load

procedure introduce [Env]
   skip
end introduce

procedure initialize [Env, alist]
   skip
end initialize

procedure execute [theStore, IEnv, LEnv, CurrentSelf]
   skip
end execute

procedure bbexec  [theStore, IEnv, LEnv, CurrentSelf]
   skip
end bbexec

function lookup [key] -> [result]
   result <- nil.error["Key not found in lookup"]
end lookup
```

```
end emptylist

(*const genericlist == list.of[nil] -- Old version*)
const genericlist == list.of[emptylist]
;
```

D.2 An environment

```
const emptyenvironment == nil

const environment == object environment

   function empty[] -> [result]
      result <- (*self.envcell["not-defined", "error", emptyenvironment]*)
         object anenvironmentempty
    var tail

    function lookup [thisname] -> [result]
       if tail == emptyenvironment then
  result <- nil.error["error in environment (lookup)"]
       else
  result <- tail.lookup [thisname]
       end if
    end lookup

    procedure setvalue [thisname, newvalue]
       var myvalue

       if tail == emptyenvironment then
  myvalue <- nil.error["error in environment (update)"]
       else
  tail.setvalue [thisname, newvalue]
       end if
    end setvalue

    procedure introduce [thisname]
       if tail == emptyenvironment then
  tail <- environment.envcell[thisname, nilobject, tail]
       else
  tail.introduce[thisname]
       end if
    end introduce

    (*procedure settail [atail]
       tail <- atail
    end settail*)

    function asString [] -> [result]
       if tail == emptyenvironment then
  skip
```

```
      else
 result <- tail.asString []
      end if
   end asString

   procedure PrettyPrint [astream]
      var str

      str <- self.asString []
      astream.putString[str]
   end PrettyPrint

   initially
      tail <- (*atail*) emptyenvironment
   end initially
end anenvironmentempty
  end empty

  function envcell [aname, avalue, atail] -> [result]
     result <- object anenvironment
var myname
var myvalue
var tail

function lookup [thisname] -> [result]
   if thisname.equal[myname] then
      result <- myvalue
   elseif tail == emptyenvironment then
      result <- nil.error["error in environment (lookup)"]
   else
      result <- tail.lookup [thisname]
   end if
end lookup

procedure setvalue [thisname, newvalue]
   if thisname.equal[myname] then
      myvalue <- newvalue
   elseif tail == emptyenvironment then
      myvalue <- nil.error["error in environment (update)"]
   else
      tail.setvalue [thisname, newvalue]
   end if
end setvalue
```

```
procedure introduce [thisname]
   if tail == emptyenvironment then
      tail <- environment.envcell[thisname, nilobject, tail]
   else
      tail.introduce[thisname]
   end if
end introduce

(*procedure settail [atail]
   tail <- atail
end settail*)

function asString [] -> [result]
   var str

   result <- myname.asString []
   result <- "(".catenate[result]
   str <- myvalue.asString[]
   str <- " -> ".catenate[str]
   str <- str.catenate[")"]
   result <- result.catenate[str]
   if tail == nil then
      skip
   else
      str <- tail.asString []
      result <- result.catenate[" "]
      result <- result.catenate[str]
   end if
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   myname <- aname
   myvalue <- avalue
   tail <- atail
end initially
     end anenvironment
  end cell
```

```
end environment
;
```

D.3 Builtin objects

```
(*****************************************************************************)

const objecttags == object Tag
   function booleantag[] -> [result]
      result <- 1
   end booleantag
   function integertag[] -> [result]
      result <- 2
   end integertag
   function charactertag[] -> [result]
      result <- 3
   end chartag
   function stringtag[] -> [result]
      result <- 4
   end stringtag
   function niltag[] -> [result]
      result <- 5
   end niltag
   function unixtag[] -> [result]
      result <- 6
   end unixtag
   function usertag[] -> [result]
      result <- 7
   end usertag
   function instreamtag [] -> [result]
      result <- 8
   end instreamtag
   function outstreamtag [] -> [result]
      result <- 9
   end outstreamtag
end Tag

const truthobject == object truthobject
   var thevalue

   function objecttag[] -> [result]
      result <- objecttags.booleantag[]
   end objecttag

   procedure fnctcall [thestore, callname, arglist]
      var argvalue
```

```
      if callname.equal["and"] then
 argvalue <- arglist.getElement[1]
 if argvalue == truthobject then
    thevalue <- truthobject
 elseif argvalue == falseobject then
    thevalue <- falseobject
 else
    thevalue <- nil.error[
       "error in truthobject (argument to and must be a boolean\n"]
 end if
      elseif callname.equal["or"] then
 thevalue <- truthobject
      elseif callname.equal["not"] then
 thevalue <- falseobject
      elseif callname.equal["asString"] then
 thevalue <- stringobject.new["true"]
      else
 thevalue <- nil.error["error in truthobject (no such function)"]
      end if
   end fnctcall

   function fnctcallresult [] -> [result]
      result <- thevalue
   end fnctcallresult

   procedure proccall [thestore, callname, arglist]
      var temp

      temp <- nil.error["error in truthobject (no such procedure)\n"]
   end proccall

   function asString [] -> [result]
      result <- "true"
   end asString
end truthobject

const falseobject == object falseobject
   var thevalue

   function objecttag[] -> [result]
      result <- objecttags.booleantag[]
   end objecttag

   procedure fnctcall [thestore, callname, arglist]
```

```
      var argvalue

      if callname.equal["and"] then
 thevalue <- falseobject
      elseif callname.equal["or"] then
 argvalue <- arglist.getElement[1]
 if argvalue == truthobject then
    thevalue <- truthobject
 elseif argvalue == falseobject then
    thevalue <- falseobject
 else
    nil.error["error in falseobject (argument to or must be boolean)"]
 end if
      elseif callname.equal["not"] then
 thevalue <- truthobject
      elseif callname.equal["asString"] then
 thevalue <- stringobject.new["false"]
      else
 thevalue <- nil.error["error in falseobject (no such function)\n"]
      end if
   end fnctcall

   function fnctcallresult [] -> [result]
      result <- thevalue
   end fnctcallresult

   procedure proccall [thestore, callname, arglist]
      var temp

      temp <- nil.error["error in falseobject (no such procedure)\n"]
   end proccall

   function asString [] -> [result]
      result <- "false"
   end asString
end falseobject

const integerobject == object integerobject

   function new [value] -> [result]
      result <- object anintegerobject
 var returnvalue
 var myvalue
 var stringvalue
```

```
function objecttag [] -> [result]
   result <- objecttags.integertag[]
end objecttag

function getvalue [] -> [result]
   result <- myvalue
end getvalue

procedure proccall [thestore, callname, arglist]
   var temp

   temp <- nil.error["error in integerobject (nu such procedure)\n"]
end proccall

procedure fnctcall [thestore, callname, arglist]
   var argvalue
   var integervalue

   if callname.equal["plus"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvalue[]
      integervalue <- myvalue.plus[integervalue]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["minus"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvalue[]
      integervalue <- myvalue.minus[integervalue]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["times"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvalue[]
      integervalue <- myvalue.times[integervalue]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["div"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvalue[]
      integervalue <- myvalue.div[integervalue]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["sqr"] then
      integervalue <- myvalue.sqr[]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["even"] then
      if myvalue.even[] then
```

```
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
  elseif callname.equal["lt"] then
     argvalue <- arglist.getElement[1]
     integervalue <- argvalue.getvalue[]
     if myvalue.lt[integervalue] then
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
  elseif callname.equal["leq"] then
     argvalue <- arglist.getElement[1]
     integervalue <- argvalue.getvalue[]
     if myvalue.leq[integervalue] then
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
  elseif callname.equal["gt"] then
     argvalue <- arglist.getElement[1]
     integervalue <- argvalue.getvalue[]
     if myvalue.gt[integervalue] then
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
  elseif callname.equal["gte"] then
     argvalue <- arglist.getElement[1]
     integervalue <- argvalue.getvalue[]
     if myvalue.gte[integervalue] then
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
  elseif callname.equal["eq"] then
     argvalue <- arglist.getElement[1]
     integervalue <- argvalue.getvalue[]
     if myvalue.eq[integervalue] then
returnvalue <- truthobject
     else
returnvalue <- falseobject
     end if
```

```
    elseif callname.equal["neq"] then
       argvalue <- arglist.getElement[1]
       integervalue <- argvalue.getvalue[]
       if myvalue.neq[integervalue] then
  returnvalue <- truthobject
       else
  returnvalue <- falseobject
       end if
    elseif callname.equal["asString"] then
       stringvalue <- myvalue.asString[integervalue]
       returnvalue <- stringobject.new[stringvalue]
    else
       returnvalue <- nil.error["error in integer (nu such function)\n"]
    end if
 end fnctcall

 function fnctcallresult [] -> [result]
    result <- returnvalue
 end fnctcallresult

 function asString [] -> [result]
    result <- myvalue.asString[]
 end asString

 initially
    myvalue <- value
 end initially
      end anintegerobject
   end new
end integerobject

const stringobject == object stringobject
   function new [value] -> [result]
      result <- object astring
 var myvalue
 var returnvalue

 function objecttag [] -> [result]
    result <- objecttags.stringtag[]
 end objecttag

 function getvalue [] -> [result]
    result <- myvalue
 end getvalue
```

```
procedure fnctcall [thestore, callname, arglist]
   var argvalue
   var integervalue
   var charactervalue
   var stringvalue
   var intvalue

   if callname.equal["length"] then
      integervalue <- myvalue.length[]
      returnvalue <- integerobject.new[integervalue]
   elseif callname.equal["getElement"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvaluel[]
      charactervalue <- myvalue.getElement[integervalue]
      returnvalue <- characterobject.new[charactervalue]
   elseif callname.equal["getSlice"] then
      argvalue <- arglist.getElement[1]
      integervalue <- argvalue.getvalue[]
      argvalue <- arglist.getElement[2]
      intvalue <- argvalue.getvalue[]
      stringvalue <- myvalue.getSlice[integervalue,intvalue]
      returnvalue <- stringobject.new[stringvalue]
   elseif callname.equal["equal"] then
      argvalue <- arglist.getElement[1]
      stringvalue <- argvalue.getvalue[]
      if myvalue.equal[stringvalue] then
 returnvalue <- truthobject
      else
 returnvalue <- falseobject
      end if
   elseif callname.equal["catenate"] then
      argvalue <- arglist.getElement[1]
      stringvalue <- argvalue.getvalue[]
      stringvalue <- myvalue.catenate[stringvalue]
      returnvalue <- stringobject.new[stringvalue]
   elseif callname.equal["asinteger"] then
      integervalue <- myvalue.asInteger[]
      returnvalue <- integerobject.new[integervalue]
   else
      returnvalue <-
 nil.error["error in stringobject (no such function)\n"]
   end if
end fnctcall
```

```
 function fnctcallresult [] -> [result]
    result <- returnvalue
 end fnctcallresult

 procedure proccall [thestore, callname, arglist]
    var temp

    temp <- nil.error["error in stringobject (no such procedure)\n"]
 end proccall

 function asString [] -> [result]
    result <- myvalue.asString[]
 end asString

 initially
    myvalue <- value
 end initially
      end astring
   end new
end stringobject

const characterobject == object characterobject
   var charlist

   function new [value] -> [result]
      result <- charlist.getElement[value]
   end new

   function create[count] -> [result]
      var counter
      var thischaracter

      counter <- 2 (* 255 *)
      charlist <- emptylist
      loop
 exit when counter.lt[1]
 thischaracter <- object acharacterobject
    var myvalue
    var returnvalue

    function objecttag [] -> [result]
       result <- objecttags.charactertag[]
    end objecttag
```

```
  function getvalue [] -> [result]
     result <- myvalue
  end getvalue

  procedure proccall [thestore, callname, arglist]
     var temp

     temp <-
nil.error["error in characterobject (no such procedure)\n"]
  end proccall

  procedure fnctcall [thestore, callname, arglist]
     var integervalue
     var argvalue
     var charactervalue
     var stringvalue

     if callname.equal["ord"] then
integervalue <- myvalue.ord[]
returnvalue <- integerobject.new[integervalue]
     elseif callname.equal["lt"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if myvalue.lt[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
end if
     elseif callname.equal["leq"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if  myvalue.leq[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
end if
     elseif callname.equal["gt"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if  myvalue.gt[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
```

```
end if
     elseif callname.equal["gte"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if  myvalue.gte[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
end if
     elseif callname.equal["eq"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if  myvalue.eq[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
end if
     elseif callname.equal["neq"] then
argvalue <- arglist.getElement[1]
charactervalue <- argvalue.getvalue[]
if  myvalue.neq[charactervalue] then
   returnvalue <- truthobject
else
   returnvalue <- falseobject
end if
     elseif callname.equal["asString"] then
stringvalue <- myvalue.asString[]
returnvalue <- stringobject.new[stringvalue]
     else
returnvalue <-
   nil.error["error in characterobj. (no such function)\n"]
     end if
  end fnctcall

  function fnctcallresult [] -> [result]
     result <- returnvalue
  end fnctcallresult

  function asString [] -> [result]
     result <- myvalue.asString[]
  end asString

  initially
     myvalue <- counter.chr[]
```

```
    end initially
 end acharacterobject
 charlist <- genericlist.cons[thischaracter, charlist]
 counter <- counter.minus[1]
      end loop
      charlist <- charlist.reverse[emptylist]
   end create

   initially
charlist <- self.create[2] (* 256 / 128 *)
   end initially
end characterobject

const nilobject == object nilobject

   function objecttag [] -> [result]
      result <- objecttags.niltag[]
   end objecttag

   procedure proccall [callname, arglist]
      var temp

      temp <- nil.error["error in nilobjec no such procedure\n"]
   end proccall

   procedure fnctcall [thestore, callname, arglist]
      var temp

      temp <- nil.error["error in nilobject (no such function)\n"]
   end fnctcall

   function fnctcallresult[] -> [result]
      result <- self
   end fnctcallresult

   function asString [] -> [result]
      result <- "nil"
   end asString
end nilobject

const unixobject == object unixobject
   var thevalue

   function objecttag [] -> [result]
```

```
      result <- objecttags.unixtag[]
   end objecttag

   procedure proccall [callname, arglist]
      var temp

      temp <- nil.error["error in unixobject (no such procedure)\n"]
   end proccall

   procedure fnctcall [thestore, callname, arglist]
      var astream
      var argvalue
      var filename

      if callname.equal["stdin"] then
 thevalue <- similiinstream.new[stdin, "StdIn"]
      elseif callname.equal["stdout"] then
 thevalue <- similioutstream.new[stdout, "StdOut"]
      elseif callname.equal["openin"] then
 argvalue <- arglist.getElement[1]
 filename <- argvalue.getvalue []
 astream <- unix.openin[filename]
 thevalue <- similiinstream.new[astream, filename]
      elseif callname.equal["openout"] then
 argvalue <- arglist.getElement[1]
 filename <- argvalue.getvalue []
 astream <- unix.openout[filename]
 thevalue <- similioutstream.new[astream, filename]
      else
 thevalue <- nil.error["error in unixobject (no such function)\n"]
      end if
   end fnctcall

   function fnctcallresult [] -> [result]
      result <- thevalue
   end fnctcallresult

   function asString [] -> [result]
      result <- "unix"
   end asString
end unixobject

const similiinstream == object similiinstream
   function new [instream, filename] -> [result]
```

```
     result <- object aninstream
var myinstream
var returnvalue
var myfilename

function objecttag[] -> [result]
   result <- objecttags.instreamtag[]
end objecttag

procedure proccall [thestore, callname, arglist]
   var avalue
   var acharacter
   var temp

   if callname.equal["backwardchar"] then
      avalue <- arglist.getElement[1]
      acharacter <- avalue.getvalue[]
      myinstream.backwardchar[acharacter]
   elseif callname.equal["forwardchar"] then
      avalue <- arglist.getElement[1]
      acharacter <- avalue.getvalue[]
      myinstream.forwardchar[acharacter]
   elseif callname.equal["forwardstring"] then
      avalue <- arglist.getElement[1]
      acharacter <- avalue.getvalue[]
      myinstream.forwardstring[acharacter]
   elseif callname.equal["close"] then
      myinstream.close[]
   else
      temp <- nil.error["error in instream (unknown procedure)\n"]
   end if
end proccall

procedure fnctcall [thestore, callname, arglist]
   var acharacter
   var astring

   if callname.equal["getchar"] then
      acharacter <- myinstream.getchar[]
      returnvalue <- characterobject.new[acharacter]
   elseif callname.equal["getstring"] then
      astring <- myinstream.getstring[]
      returnvalue <- stringobject.new[astring]
   elseif callname.eq["eos"] then
```

```
      if myinstream.eos[] then
  returnvalue <- truthobject
      else
  returnvalue <- falseobject
      end if
    else
      returnvalue <- nil.error["error in instream (no such function)\n"]
    end if
 end fnctcall

 function fnctcallresult [] -> [result]
    result <- returnvalue
 end fnctcallresult

 function asString [] -> [result]
    result <- "instream(".catenate[myfilename]
    result <- result.catenate[")"]
 end asString

 initially
    myinstream <- instream
    myfilename <- filename
 end initially
     end aninstream
   end new
end similiinstream

const similioutstream == object similioutstream
   function new [outstream, filename] -> [result]
      result <- object anoutstream
 var myoutstream
 var myfilename

 function objecttag [] -> [result]
    result <- objecttags.outstreamtag[]
 end objecttag

 procedure proccall [thestore, callname, arglist]
    var thevalue
    var acharacter
    var astring

    if callname.equal["putChar"] then
       thevalue <- arglist.getElement[1]
```

```
      acharacter <- thevalue.getvalue[]
      myoutstream.putchar[acharacter]
   elseif callname.equal["putString"] then
      thevalue <- arglist.getElement[1]
      astring <- thevalue.getvalue[]
      myoutstream.putstring[astring]
   elseif callname.equal["close"] then
      myoutstream.close[]
   else
      thevalue <- nil.error["error in stdout (no such procedure)\n"]
   end if
 end proccall

 procedure fnctcall [thestore, callname, arglist]
   var temp

   temp <- nil.error["error in stdout (no such function name)\n"]
 end fnctcall

 function asString [] -> [result]
   result <- "outstream(".catenate[myfilename]
   result <- result.catenate[")"]
 end asString

 initially
   myoutstream <- outstream
   myfilename <- filename
 end initially
     end anoutstream
  end new
end similioutstream

;
```

D.4 The interpreter itself

```
(*****************************************************************************)

const program == object program
   var parsevalue
   var programresult

   function new [alist] -> [result]
      result <- object aprogram
 var decllist
 var mystore
 var fnctresult

 function asString [] -> [result]
    result <- decllist.asString []
    result <- result.catenate["\n;\n"]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString[]
    astream.putString[str]
 end PrettyPrint

 procedure load[]
    mystore <- environment.empty[]
    decllist.load[mystore]
 end load

 function loadresult [] -> [result]
    result <- mystore
 end loadresult

 procedure proccall[objectname, methodname, arglist]
    var theobject

    theobject <- mystore.lookup[objectname]
    theobject.proccall[mystore, methodname, arglist]
 end proccall

 procedure fnctcall[objectname, methodname, arglist]
    var theobject
```

```
   theobject <- mystore.lookup[objectname]
   theobject.fnctcall[mystore, methodname, arglist]
   fnctresult <- theobject.fnctcallresult[]
end fnctcall

function fnctcallresult [] -> [result]
   result <- fnctresult
end fnctcallresult

initially
   decllist <- alist
end initially
     end aprogram
  end new

  procedure interpret[program constname procname arglist fnctname]
     var aprogram
     var args
     var newargs
     var mystore
     var tagvalue
     var temp

     (* Parse *)
     token.setstring[program]
     self.parse[]
     aprogram <- parsevalue

     (* Load *)
     aprogram.load[]
     mystore <- aprogram.loadresult[]

     (* Parse & Evaluate arguments *)
     token.setstring[arglist]
     token.readnexttoken[]
     argumentlist.parse[]
     args <- argumentlist.parseresult[]
     args.evaluate[mystore, nil, nil, nilobject]
     newargs <- args.evaluateresult[]

     (* Execute procedure *)
     aprogram.proccall[constname, procname, newargs]
     args <- emptylist
```

```
      (* Evaluate function *)
      aprogram.fnctcall[constname, fnctname, args]
      programresult <- aprogram.fnctcallresult[]

      (* Fetch return object *)
      tagvalue <- programresult.objecttag[]
      temp <- objecttags.niltag[]
      if tagvalue.leq[temp] then
 programresult <- programresult.asString[]
      end if
   end interpret

   function interpretresult [] -> [result]
      result <- programresult
   end interpretresult

   procedure parse []
      var adecllist
      var thisdecl
      var thetoken

      adecllist <- emptylist
      token.readnexttoken[]
      constdecllist.parse[]
      adecllist <- constdecllist.parseresult[]
      parsevalue <- self.new[adecllist]
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end program

const constdecl == object constdecl
   var parsevalue

   function new[name, exp] -> [result]
      result <- object aconstdecl
 var myname
 var myexpression

 function asString [] -> [result]
    var str
```

```
   result <- "const ".catenate[myname]
   result <- result.catenate[" == "]
   str <- myexpression.asString []
   result <- result.catenate[str]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

procedure load[astore]
   var thevalue
   var newienv
   var newlenv

   astore.introduce[myname]
   newienv <- environment.empty[]
   newlenv <- environment.empty[]
   myexpression.evaluate[astore, newienv, newlenv, nilobject]
   thevalue <- myexpression.evaluateresult[]
   astore.setvalue[myname, thevalue]
end load

initially
   myname <- name
   myexpression <- exp
end initially
     end aconstdecl
  end new

  procedure parse []
     var thetoken
     var thename
     var theexp

     thetoken <- token.gettoken[]
     if thetoken.equal["const"] then
token.readnexttoken[]
thename <- token.gettoken[]
```

```
 token.readnexttoken[]
 expression.parse[]
 theexp <- expression.parseresult[]
 parsevalue <- self.new[thename, theexp]
      else
 parsevalue <- nil.error["error in program (no const keyword)\n"]
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end constdecl

const basicexpression == object basicexpression
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      token.readnexttoken[]
      if thetoken.equal["constname"] then
 thetoken <- token.gettoken[]
 parsevalue <- constnameexpression.new[thetoken]
 token.readnexttoken[]
      elseif thetoken.equal["self"] then
 parsevalue <- selfexpression
      elseif thetoken.equal["ivarname"] then
 thetoken <- token.gettoken[]
 parsevalue <- ivarnameexpression.new[thetoken]
 token.readnexttoken[]
      elseif thetoken.equal["lvarname"] then
 thetoken <- token.gettoken[]
 parsevalue <- lvarnameexpression.new[thetoken]
 token.readnexttoken[]
      elseif thetoken.equal["formal"] then
 thetoken <- token.gettoken[]
 parsevalue <- formalparexpression.new[thetoken]
 token.readnexttoken[]
      elseif thetoken.equal["result"] then
 thetoken <- token.gettoken[]
 parsevalue <- resultnameexpression.new[thetoken]
 token.readnexttoken[]
```

```
      elseif thetoken.equal["basevalue"] then
 basevalue.parse[]
 parsevalue <- basevalue.parseresult[]
      else
 parsevalue <- nil.error["error in program (no such basic expression)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end basicexpression

const basevalue == object basevalue
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["integer"] then
 integerexpression.parse[]
 parsevalue <- integerexpression.parseresult[]
      elseif thetoken.equal["char"] then
 characterexpression.parse[]
 parsevalue <- characterexpression.parseresult[]
      elseif thetoken.equal["string"] then
 stringexpression.parse[]
 parsevalue <- stringexpression.parseresult[]
      elseif thetoken.equal["false"] then
 parsevalue <- falseexpression
 token.readnexttoken[]
      elseif thetoken.equal["true"] then
 parsevalue <- truthexpression
 token.readnexttoken[]
      elseif thetoken.equal["nil"] then
 parsevalue <- nilexpression
 token.readnexttoken[]
      else
 parsevalue <- nil.error["error in program (no such basevalue)\n"]
      end if
   end parse

   function parseresult [] -> [result]
```

```
      result <- parsevalue
   end parseresult
end basevalue

const expression == object expression
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["basic"] then
 token.readnexttoken[]
 basicexpression.parse[]
 parsevalue <- basicexpression.parseresult[]
      elseif thetoken.equal["equal"] then
 equalityexpression.parse[]
 parsevalue <- equalityexpression.parseresult[]
      elseif thetoken.equal["invoke"] then
 functioncallexpression.parse[]
 parsevalue <- functioncallexpression.parseresult[]
      elseif thetoken.equal["object"] then
 objectconstr.parse[]
 parsevalue <- objectconstr.parseresult[]
      else
 parsevalue <- nil.error["error in program (no such expression)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end expression

const truthexpression == object truthexpression

  procedure evaluate[thestore, ienv, lenv, currentself]
  end evaluate

  function evaluateresult [] -> [result]
     result <- truthobject
  end evaluateresult

  function asString [] -> [result]
```

```
     result <- "true"
  end asString

  procedure PrettyPrint [astream]
     astream.putString["true"]
  end PrettyPrint
end truthexpression

const falseexpression == object falseexpression

   procedure evaluate[thestore, ienv, lenv, currentself]
   end evaluate

   function evaluateresult [] -> [result]
      result <- falseobject
   end evaluateresult

   function asString [] -> [result]
      result <- "false"
   end asString

   procedure PrettyPrint [astream]
      astream.putString["false"]
   end PrettyPrint
end falseexpression

const selfexpression == object selfexpression
   var theresult

   procedure evaluate[thestore, ienv, lenv, currentself]
      theresult <- currentself
   end evaluate

   function evaluateresult [] -> [result]
      result <- theresult
   end evaluateresult

   function asString [] -> [result]
      result <- "self"
   end asString

   procedure PrettyPrint [astream]
      astream.putString["self"]
   end PrettyPrint
```

```
end selfexpression

const unixexpression == object unixexpression

   procedure evaluate[thestore, ienv, lenv, currentself]
   end evaluate

   function evaluateresult[] -> [result]
      result <- unixobject
   end evaluateresult

   function asString [] -> [result]
      result <- "unix"
   end asString

   procedure PrettyPrint [astream]
      astream.putString["unix"]
   end PrettyPrint
end unixexpression

const nilexpression == object nilexpression

   procedure evaluate[thestore, ienv, lenv, currentself]
   end evaluate

   function evaluateresult [] -> [result]
      result <- nilobject
   end evaluateresult

   function asString [] -> [result]
      result <- "nil"
   end asString

   procedure PrettyPrint [astream]
      astream.putString["nil"]
   end PrettyPrint
end nilexpression

const integerexpression == object integerexpression
   var parsevalue

   function new [int] -> [result]
      result <- object anintegerexpression
 var theinteger
```

```
procedure evaluate [thestore, ienv, lenv, currentself]
end evaluate

function evaluateresult [] -> [result]
   result <- theinteger
end evaluateresult

function asString [] -> [result]
   var thenumber

   thenumber <- theinteger.getvalue[]
   result <- thenumber.asString[]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   theinteger <- integerobject.new[int]
end initially
     end anintegerexpression
  end new

  procedure parse[]
     var thetoken
     var myvalue

     thetoken <- token.gettoken[]
     if thetoken.equal["integer"] then
token.readnexttoken[]
thetoken <- token.gettoken[]
myvalue <- thetoken.asInteger[]
parsevalue <- self.new[myvalue]
token.readnexttoken[]
     else
     end if
  end parse

  function parseresult[] -> [result]
```

```
      result <- parsevalue
   end parseresult

end integerexpression

const characterexpression == object characterexpression
   var parsevalue

   function new [char] -> [result]
      result <- object acharacterexpression
 var thecharacter

 procedure evaluate [thestore, ienv, lenv, currentself]
 end evaluate

 function evaluateresult [] -> [result]
    result <- thecharacter
 end evaluateresult

 function asString [] -> [result]
    result <- thecharacter.getvalue[]
    result <- "'".catenate[result]
    result <- result.catenate["'"]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString []
    astream.putString[str]
 end PrettyPrint

 initially
    thecharacter <- characterobject.new[char]
 end initially
      end acharacterexpression
   end new

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["char"] then
 token.readnexttoken[]
```

```
 thetoken <- token.gettoken[]
 parsevalue <- self.new[thetoken]
 token.readnexttoken[]
      else
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult

end characterexpression

const stringexpression == object stringexpression
   var parsevalue

   function new [astring] -> [result]
      result <- object astringexpression
 var thestring

 procedure evaluate [thestore, ienv, lenv, currentself]
 end evaluate

 function evaluateresult [] -> [result]
    result <- thestring
 end evaluateresult

 function asString [] -> [result]
    var myquote
    var thevalue

    myquote <- '"'
    thevalue <- thestring.getvalue[]
    myquote <- myquote.asString []
    result <- myquote.catenate[thevalue]
    result <- result.catenate[myquote]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString []
    astream.putString[str]
 end PrettyPrint
```

```
 initially
    thestring <- stringobject.new[astring]
 end initially
      end astringexpression
   end new
   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["string"] then
 token.readnexttoken[]
 thetoken <- token.gettoken[]
 parsevalue <- self.new[thetoken]
 token.readnexttoken[]
      else
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult

end stringexpression

const constnameexpression == object constnameexpression

   function new [aname] -> [result]
      result <- object aconstnameexpression
 var thename
 var thevalue

 procedure evaluate [thestore, ienv, lenv, currentself]
    thevalue <- thestore.lookup[thename]
 end evaluate

 function evaluateresult [] -> [result]
    result <- thevalue
 end evaluateresult

 function asString [] -> [result]
    result <- thename
 end asString
```

```
 procedure PrettyPrint [astream]
    astream.putString[thename]
 end PrettyPrint

 initially
    thename <- aname
 end initially
      end aconstnameexpression
   end new
end constnameexpression

const ivarnameexpression == object ivarnameexpression

   function new [aname] -> [result]
      result <- object aivarnameexpression
 var thename
 var thevalue

 procedure evaluate [thestore, ienv, lenv, currentself]
    thevalue <- ienv.lookup[thename]
 end evaluate

 function evaluateresult [] -> [result]
    result <- thevalue
 end evaluateresult

 function asString [] -> [result]
    result <- thename
 end asString

 procedure PrettyPrint [astream]
    astream.putString[thename]
 end PrettyPrint

 initially
    thename <- aname
 end initially
      end aivarnameexpression
   end new
end ivarnameexpression

const lvarnameexpression == object lvarnameexpression

   function new [aname] -> [result]
```

```
      result <- object alvarnameexpression
 var thename
 var thevalue

 procedure evaluate [thestore, ienv, lenv, currentself]
    thevalue <- lenv.lookup[thename]
 end evaluate

 function evaluateresult [] -> [result]
    result <- thevalue
 end evaluateresult

 function asString [] -> [result]
    result <- thename
 end asString

 procedure PrettyPrint [astream]
    astream.putString[thename]
 end PrettyPrint

 initially
    thename <- aname
 end initially
      end alvarnameexpression
   end new
end lvarnameexpression

const formalparexpression == lvarnameexpression

const resultnameexpression == lvarnameexpression

const equalityexpression == object equalityexpression
   var parsevalue

   function new [first, second] -> [result]
      result <- object aequalityexpression
 var firstexp
 var secondexp
 var thevalue

 procedure evaluate [thestore ienv, lenv, currentself]
    var thefirstvalue
    var thesecondvalue
```

```
   firstexp.evaluate[thestore, ienv, lenv, currentself]
   thefirstvalue <- firstexp.evaluateresult []
   secondexp.evaluate[thestore, ienv, lenv, currentself]
   thesecondvalue <- secondexp.evaluateresult []
   if thefirstvalue == thesecondvalue then
      thevalue <- truthobject
   else
      thevalue <- falseobject
   end if
end evaluate

function evaluateresult [] -> [result]
   result <- thevalue
end evaluateresult

function asString [] -> [result]
   var str

   result <- firstexp.asString []
   result <- result.catenate[" == "]
   str <- secondexp.asString []
   result <- result.catenate[str]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   firstexp <- first
   secondexp <- second
end initially
     end aequalityexpression
  end new

  procedure parse[]
     var first
     var second
     var thetoken

     thetoken <- token.gettoken []
```

```
      if thetoken.equal["equal"] then
 token.readnexttoken[]
 basicexpression.parse[]
 first <- basicexpression.parseresult []
 basicexpression.parse[]
 second <- basicexpression.parseresult []
 parsevalue <- self.new[first, second]
      else
 parsevalue <- nil.error["error in program (no equal keyword)\n"]
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end equalityexpression

const functioncallexpression == object fnctcallexpression
   var parsevalue

   function new[aexp, aname, alist] -> [result]
      result <- object afnctcallexpression
 var theexp
 var thefnctname
 var thearglist
 var thevalue

 procedure evaluate [thestore, ienv, lenv, currentself]
    var theobject
    var objectlist

    theexp.evaluate[thestore, ienv, lenv, currentself]
    theobject <- theexp.evaluateresult []

    thearglist.evaluate[thestore, ienv, lenv, currentself]
    objectlist <- thearglist.evaluateresult []

    theobject.fnctcall[thestore, thefnctname, objectlist]
    thevalue <- theobject.fnctcallresult []
 end evaluate

 function evaluateresult [] -> [result]
    result <- thevalue
 end evaluateresult
```

```
function asString [] -> [result]
   var str

   str <- theexp.asString []
   result <- str.catenate["."]
   result <- result.catenate[thefnctname]
   result <- result.catenate["["]
   str <- thearglist.asString []
   result <- result.catenate[str]
   result <- result.catenate["]"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   theexp <- aexp
   thefnctname <- aname
   thearglist <- alist
end initially
     end afnctcallexpression
  end new

  procedure parse []
     var thetoken
     var thebexp
     var thename
     var thelist

     thetoken <- token.gettoken[]
     if thetoken.equal["invoke"] then
token.readnexttoken[]
basicexpression.parse[]
thebexp <- basicexpression.parseresult[]
thename <- token.gettoken[]
token.readnexttoken[]
basicexpressionlist.parse[]
thelist <- basicexpressionlist.parseresult[]
parsevalue <- self.new[thebexp, thename, thelist]
```

```
      else
 parsevalue <- nil.error["error in program (no invoke keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end fnctcallexpression

const objectconstr == object objectdecl
   var parsevalue

   function new[aname, ivarlist, proclist, fnctlist, bblist] -> [result]
      result <- object anobjectdecl
 var thename
 var theivarlist
 var thefnctlist
 var theproclist
 var theinitially

 var thevalue

 procedure evaluate [thestore, ienv, lenv, currentself]
    var templocalenv
    var tempstate
    var tempfnctlist
    var tempproclist
    var tempinitially

    (* Build a new IEnv *)
    tempstate <- environment.empty[]
    theivarlist.introduce[tempstate]

    templocalenv <- lenv

    tempfnctlist <- thefnctlist
    tempproclist <- theproclist
    tempinitially <- theinitially

    thevalue <- object newobject
       var thestate
       var theprocedures
       var thefunctions
```

```
      var fnctresult
      var initiallybody

      function objecttag [] -> [result]
 result <- objecttags.usertag []
      end objecttag

      procedure fnctcall [astore, aname, arglist]
 var thefunction

 thefunction <- thefunctions.lookup[aname]
 thefunction.invoke[astore, thestate, arglist, self]
 fnctresult <- thefunction.getresult[]
      end fnctcall

      function fnctcallresult [] -> [result]
 result <- fnctresult
      end fnctcallresult

      procedure proccall [astore, aname, arglist]
 var theprocedure

 theprocedure <- theprocedures.lookup[aname]
 theprocedure.invoke[astore, thestate, arglist, self]
      end proccall

      procedure doinitialize [thestore, templocalenv]
 initiallybody.bbexec[thestore, thestate, templocalenv, self]
      end doinitialize

      initially
 thestate <- tempstate
 thefunctions <- tempfnctlist
 theprocedures <- tempproclist
 initiallybody <- tempinitially
      end initially
   end newobject

   thevalue.doinitialize[thestore, templocalenv]
end evaluate

function evaluateresult [] -> [result]
   result <- thevalue
end evaluateresult
```

```
function asString [] -> [result]
   var str

   result <- "object ".catenate[thename]
   result <- result.catenate["\n"]
   str <- theivarlist.asStringFull ["var "]
   result <- result.catenate[str]
   result <- result.catenate["\n"]
   str <- theproclist.asString []
   result <- result.catenate[str]
   result <- result.catenate["\n"]
   str <- thefnctlist.asString []
   result <- result.catenate[str]
   result <- result.catenate["\ninitially\n"]
   str <- theinitially.asString []
   result <- result.catenate[str]
   result <- result.catenate["end initially\n"]
   str <- "\nend ".catenate[thename]
   result <- result.catenate[str]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   thename <- aname
   theivarlist <- ivarlist
   theproclist <- proclist
   thefnctlist <- fnctlist
   theinitially <- bblist
end initially

     end anobjectdecl
  end new

  procedure parse[]
     var thename
     var theivarlist
```

```
      var theproclist
      var thefnctlist
      var thebblist
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["object"] then
 token.readnexttoken[]
 thename <- token.gettoken[]
 token.readnexttoken[]
 ivardecllist.parse[]
 theivarlist <- ivardecllist.parseresult[]
 proclist.parse[]
 theproclist <- proclist.parseresult[]
 fnctlist.parse[]
 thefnctlist <- fnctlist.parseresult[]
 bblist.parse[]
 thebblist <- bblist.parseresult[]
 parsevalue <-
    self.new[thename, theivarlist, theproclist, thefnctlist, thebblist]
      else
 parsevalue <- nil.error["error in program (no object keyword)\n"]
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end objectdecl

const proceduredef == object proceduredef
   var parsevalue

   function new[aname, afparlist, alvarlist, abblist] -> [result]
      result <- object aproceduredef
 var procedurename
 var formallist
 var localvarlist
 var mybblist

 function same_key [key] -> [result]
    result <- procedurename.equal[key]
 end same_key
```

```
procedure invoke[astore, ienv, arglist, currentself]
   var localenv

   localenv <- environment.empty[]
   formallist.initialize[localenv, arglist]
   localvarlist.introduce[localenv]

   mybblist.bbexec[astore, ienv, localenv, currentself]
end invoke

function asString [] -> [result]
   var str

   result <- "procedure ".catenate[procedurename]
   result <- result.catenate["["]
   str <- formallist.asString []
   result <- result.catenate[str]
   result <- result.catenate["]\n"]
   result <- result.catenate[str]
   str <- localvarlist.asStringFull ["var "]
   result <- result.catenate[str]
   result <- result.catenate["\n"]
   str <- mybblist.asString []
   result <- result.catenate[str]
   result <- result.catenate["\n"]
   str <- "end ".catenate[procedurename]
   result <- result.catenate[str]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   procedurename <- aname
   formallist <- afparlist
   localvarlist <- alvarlist
   mybblist <- abblist
end initially
     end aproceduredef
```

```
   end new

   procedure parse[]
      var thetoken
      var thename
      var thefparlist
      var thelvarlist
      var thebblist

      thetoken <- token.gettoken[]
      if thetoken.equal["procedure"] then
 token.readnexttoken[]
 thename <- token.gettoken[]
 token.readnexttoken[]
 fparlist.parse[]
 thefparlist <- fparlist.parseresult[]
 lvardecllist.parse[]
 thelvarlist <- lvardecllist.parseresult[]
 bblist.parse[]
 thebblist <- bblist.parseresult[]
 parsevalue <-
    self.new[thename, thefparlist, thelvarlist, thebblist]
      else
 parsevalue <- nil.error["error in program (no procedure keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end proceduredef

const functiondef == object functiondef
   var parsevalue

   function new[aname, afparlist, resname, alvarlist, abblist] ->
 [result]
      result <- object afunctiondef
 var functionname
 var formallist
 var myresultname
 var localvarlist
 var mybblist
 var theresult
```

```
function same_key [key] -> [result]
   result <- functionname.equal[key]
end same_key

procedure invoke[astore, ienv, arglist, currentself]
   var localenv

   localenv <- environment.empty[]
   formallist.initialize[localenv, arglist]
   localvarlist.introduce[localenv]

   mybblist.bbexec[astore, ienv, localenv, currentself]
   theresult <- localenv.lookup[myresultname]
end invoke

function getresult [] -> [result]
   result <- theresult
end getresult

function asString [] -> [result]
   var str

   result <- "function ".catenate[functionname]
   result <- result.catenate["["]
   str <- formallist.asString []
   result <- result.catenate[str]
   result <- result.catenate["] -> ["]
   result <- result.catenate[myresultname]
   result <- result.catenate["]\n"]
   str <- localvarlist.asStringFull ["var "]
   result <- result.catenate[str]
   result <- result.catenate["\n"]
   str <- mybblist.asString[]
   result <- result.catenate[str]
   result <- result.catenate["\nend "]
   result <- result.catenate[functionname]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
```

```
   astream.putString [str]
end PrettyPrint

initially
   functionname <- aname
   formallist <- afparlist
   myresultname <- resname
   localvarlist <- alvarlist
   mybblist <- abblist
end initially
     end afunctiondef
  end new

  procedure parse[]
     var thetoken
     var thename
     var thefparlist
     var theresultname
     var thelvarlist
     var thebblist

     thetoken <- token.gettoken[]
     if thetoken.equal["function"] then
token.readnexttoken[]
thename <- token.gettoken[]
token.readnexttoken[]
fparlist.parse[]
thefparlist <- fparlist.parseresult[]
theresultname <- token.gettoken[]
if theresultname.equal["result"] then
   token.readnexttoken[]
   theresultname <- token.gettoken[]
   token.readnexttoken[]
   lvardecllist.parse[]
   thelvarlist <- lvardecllist.parseresult[]
   bblist.parse[]
   thebblist <- bblist.parseresult[]
   parsevalue <- self.new[thename, thefparlist, theresultname,
      thelvarlist, thebblist]
else
   parsevalue <- nil.error["error in program (no result keyword)\n"]
end if
     else
parsevalue <- nil.error["error in program (no function keyword)\n"]
```

```
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end functiondef

const ivardecl == object ivardecl
   var parsevalue

   procedure parse[]
      parsevalue <- token.gettoken[]
      token.readnexttoken[]
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end ivardecl

const lvardecl == ivardecl

const fpar == lvardecl

const statement == object statement
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["ivarassign"] then
 ivarassignstatement.parse[]
 parsevalue <- ivarassignstatement.parseresult[]
      elseif thetoken.equal["lvarassign"] then
 lvarassignstatement.parse[]
 parsevalue <- lvarassignstatement.parseresult[]
      elseif thetoken.equal["resultassign"] then
 resultassignstatement.parse[]
 parsevalue <- resultassignstatement.parseresult[]
      elseif thetoken.equal["skip"] then
 parsevalue <- skipstatement
 token.readnexttoken[]
```

```
      elseif thetoken.equal["invoke"] then
 procedurecallstatement.parse[]
 parsevalue <- procedurecallstatement.parseresult[]
      else
 parsevalue <- nil.error["error in program (no such statement)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end statement

const ivarassignstatement == object ivarassignstatement
   var parsevalue

   function new[aname, aexp] -> [result]
      result <- object anivarassignstatement
 var myvarname
 var myexpression

 procedure execute [thestore, ienv, lenv, currentself]
    var thevalue

    myexpression.evaluate[thestore, ienv, lenv, currentself]
    thevalue <- myexpression.evaluateresult[]
    ienv.setvalue[myvarname, thevalue]
 end execute

 function asString [] -> [result]
    var str

    result <- myvarname.catenate[" <- "]
    str <- myexpression.asString []
    result <- result.catenate[str]
    result <- result.catenate["\n"]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString []
    astream.putString [str]
 end PrettyPrint
```

```
 initially
    myvarname <- aname
    myexpression <- aexp
 end initially
      end anivarassignstatement
   end new

   procedure parse[]
      var thetoken
      var thename
      var theexp

      thetoken <- token.gettoken[]
      if thetoken.equal["ivarassign"] then
 token.readnexttoken[]
 thename <- token.gettoken[]
 token.readnexttoken[]
 expression.parse[]
 theexp <- expression.parseresult[]
 parsevalue <- self.new[thename, theexp]
      else
 parsevalue <- nil.error["error in program (no ivarassign keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end ivarassignstatement

const lvarassignstatement == object lvarassignstatement
   var parsevalue

   function new[aname, aexp] -> [result]
      result <- object anlvarassignstatement
 var myvarname
 var myexpression

 procedure execute [thestore, ienv, lenv, currentself]
    var thevalue

    myexpression.evaluate[thestore, ienv, lenv, currentself]
    thevalue <- myexpression.evaluateresult[]
```

```
   lenv.setvalue[myvarname, thevalue]
end execute

function asString [] -> [result]
   var str

   result <- myvarname.catenate[" <- "]
   str <- myexpression.asString []
   result <- result.catenate[str]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString [str]
end PrettyPrint

initially
   myvarname <- aname
   myexpression <- aexp
end initially
     end anlvarassignstatement
  end new

  procedure parse[]
     var thetoken
     var thename
     var theexp

     thetoken <- token.gettoken[]
     if thetoken.equal["lvarassign"] then
token.readnexttoken[]
thename <- token.gettoken[]
token.readnexttoken[]
expression.parse[]
theexp <- expression.parseresult[]
parsevalue <-self.new[thename, theexp]
     else
parsevalue <- nil.error["error in program (no lvarassign keyword)\n"]
     end if
  end parse
```

```
   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end lvarassignstatement

const resultassignstatement == object resultassignstatement
   var parsevalue

   function new[aname, aexp] -> [result]
      result <- object anresultassignstatement
 var myvarname
 var myexpression

 procedure execute [thestore, ienv, lenv, currentself]
    var thevalue

    myexpression.evaluate[thestore, ienv, lenv, currentself]
    thevalue <- myexpression.evaluateresult[]
    lenv.setvalue[myvarname, thevalue]
 end execute

 function asString [] -> [result]
    var str

    result <- myvarname.catenate[" <- "]
    str <- myexpression.asString []
    result <- result.catenate[str]
    result <- result.catenate["\n"]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString []
    astream.putString [str]
 end PrettyPrint

 initially
    myvarname <- aname
    myexpression <- aexp
 end initially
      end anresultassignstatement
   end new
```

```
   procedure parse[]
      var thetoken
      var thename
      var theexp

      thetoken <- token.gettoken[]
      if thetoken.equal["resultassign"] then
 token.readnexttoken[]
 thename <- token.gettoken[]
 token.readnexttoken[]
 expression.parse[]
 theexp <- expression.parseresult[]
 parsevalue <-self.new[thename, theexp]
      else
 parsevalue <- nil.error["error in program (no resultassign keyword)\n"]
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end resultassignstatement

const skipstatement == object skipstatement
   procedure execute [thestore, ienv, lenv, currentself]
   end execute

   function asString [] -> [result]
      result <- "skip\n"
   end asString

   procedure PrettyPrint [astream]
      astream.putString["skip\n"]
   end PrettyPrint
end skipstatement

const procedurecallstatement == object proccallstatement
   var parsevalue

   function new[abexp, aname, aarglist] -> [result]
      result <- object aproccallstatement
 var theexp
 var theprocname
 var thearglist
```

```
procedure execute [thestore, ienv, lenv, currentself]
   var theobject
   var objectlist

   theexp.evaluate[thestore, ienv, lenv, currentself]
   theobject <- theexp.evaluateresult[]

   thearglist.evaluate[thestore, ienv, lenv, currentself]
   objectlist <- thearglist.evaluateresult[]

   theobject.proccall[thestore, theprocname, objectlist]
end execute

function asString [] -> [result]
   var str

   str <- theexp.asString []
   str <- str.catenate["."]
   str <- str.catenate[theprocname]
   result <- str.catenate["["]
   str <- thearglist.asString []
   result <- result.catenate[str]
   result <- str.catenate["]\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString [str]
end PrettyPrint

initially
   theexp <- abexp
   theprocname <- aname
   thearglist <- aarglist
end initially
     end aproccallstatement
  end new

  procedure parse[]
     var thetoken
     var thebexp
```

```
      var thename
      var thelist

      thetoken <- token.gettoken[]
      if thetoken.equal["invoke"] then
 token.readnexttoken[]
 basicexpression.parse[]
 thebexp <-basicexpression.parseresult[]
 thename <- token.gettoken[]
 token.readnexttoken[]
 basicexpressionlist.parse[]
 thelist <- basicexpressionlist.parseresult[]
 parsevalue <- self.new[thebexp, thename, thelist]
      else
 parsevalue <- nil.error["error in program (no invoke keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end proccallstatement

const ifgoto == object ifgoto
   var parsevalue

   function new[aexp, anameone, anametwo] -> [result]
      result <- object anifgoto
 var testexpression
 var thenname
 var elsename

 procedure jump [thestore, ienv, lenv, currentself, bblist]
    var thevalue
    var thebb

    testexpression.evaluate[thestore, ienv, lenv, currentself]
    thevalue <- testexpression.evaluateresult[]
    if thevalue == truthobject then
       thebb <- bblist.lookup[thenname]
    else
       thebb <- bblist.lookup[elsename]
    end if
    thebb.bbexec[thestore, ienv, lenv, currentself, bblist]
```

```
end jump

function asString [] -> [result]
   var str

   str <- testexpression.asString []
   result <- "if ".catenate[str]
   result <- result.catenate[" then goto "]
   result <- result.catenate[thenname]
   result <- result.catenate[" else goto "]
   result <- result.catenate[elsename]
   result <- result.catenate["\n"]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString [str]
end PrettyPrint

initially
   testexpression <- aexp
   thenname <- anameone
   elsename <- anametwo
end initially
     end anifgoto
  end new

  procedure parse[]
     var theexp
     var nameone
     var nametwo
     var thetoken

     thetoken <- token.gettoken[]
     if thetoken.equal["if"] then
token.readnexttoken[]
expression.parse[]
theexp <- expression.parseresult[]
nameone <- token.gettoken[]
token.readnexttoken[]
nametwo <- token.gettoken[]
token.readnexttoken[]
```

```
 parsevalue <- self.new[theexp, nameone, nametwo]
      else
 parsevalue <- nil.error["error in program (no if keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end ifgoto

const justgoto == object justgoto
   var parsevalue

   function new[aname] -> [result]
      result <- object ajustgoto
 var thename

 procedure jump [thestore, ienv, lenv, currentself, bblist]
    var thebb

    thebb <- bblist.lookup[thename]
    thebb.bbexec[thestore, ienv, lenv, currentself, bblist]
 end jump

 function asString [] -> [result]
    result <- "goto ".catenate[thename]
    result <- result.catenate["\n"]
 end asString

 procedure PrettyPrint [astream]
    var str

    str <- self.asString []
    astream.putString[str]
 end PrettyPrint

 initially
    thename <- aname
 end initially
      end ajustgoto
   end new

   procedure parse []
```

```
      var thetoken
      var thelabel

      thetoken <- token.gettoken[]
      if thetoken.equal["goto"] then
 token.readnexttoken[]
 thelabel <- token.gettoken[]
 parsevalue <- self.new[thelabel]
 token.readnexttoken[]
      else
 parsevalue <- nil.error["error in program (no goto keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end justgoto

const justreturn == object justreturn
   var parsevalue

   function new[] -> [result]
      result <- object ajustreturn
 procedure jump [thestore, ienv, lenv, currentself, bblist]
 end jump

 function asString [] -> [result]
    result <- "return\n"
 end asString

 procedure PrettyPrint [astream]
    astream.putString["return\n"]
 end PrettyPrint
      end ajustreturn
   end new

   procedure parse []

      parsevalue <- token.gettoken[]
      if parsevalue.equal["return"] then
 parsevalue <- self.new[]
 token.readnexttoken[]
      else
```

```
 parsevalue <- nil.error["error in program (no return keyword)\n"]
      end if
   end parse

   function parseresult[] -> [result]
      result <- parsevalue
   end parseresult
end justreturn

const jump == object jump
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["goto"] then
 justgoto.parse[]
 parsevalue <- justgoto.parseresult[]
      elseif thetoken.equal["if"] then
 ifgoto.parse[]
 parsevalue <- ifgoto.parseresult[]
      elseif thetoken.equal["return"] then
 justreturn.parse[]
 parsevalue <- justreturn.parseresult[]
      else
 parsevalue <- nil.error["error in program (no such jump)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end jump

const bb == object bb
   var parsevalue

   function new[alabel, astatement, ajump] -> [result]
      result <- object abb
 var thelabel
 var thestatements
 var thejump
```

```
function same_key [key] -> [result]
   result <- thelabel.equal[key]
end same_key

procedure bbexec[thestore, ienv, lenv, currentself, bblist]
   var nextbb

   thestatements.execute[thestore, ienv, lenv, currentself]
   thejump.jump[thestore, ienv, lenv, currentself, bblist]
end bbexec

function asString [] -> [result]
   var str

   result <- thelabel.catenate[":\n"]
   str <- thestatements.asString []
   result <- result.catenate[str]
   str <- thejump.asString []
   result <- result.catenate[str]
end asString

procedure PrettyPrint [astream]
   var str

   str <- self.asString []
   astream.putString[str]
end PrettyPrint

initially
   thelabel <- alabel
   thestatements <- astatement
   thejump <- ajump
end initially

     end abb
  end new

  procedure parse []
     var thetoken
     var thelabel
     var thestatements
     var thejump

     thetoken <- token.gettoken[]
```

```
      if thetoken.equal["bb"] then
 token.readnexttoken[]
 thelabel <- token.gettoken[]
 token.readnexttoken[]
 statementlist.parse[]
 thestatements <- statementlist.parseresult[]
 jump.parse[]
 thejump <- jump.parseresult[]
 parsevalue <- self.new[thelabel, thestatements, thejump]
      else
 parsevalue <- nil.error["error in program (no bb keyword)\n"]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end bb

const argument == object argument
   var parsevalue

   procedure parse[]
      var thetoken

      thetoken <- token.gettoken[]
      if thetoken.equal["constname"] then
 token.readnexttoken[]
 thetoken <- token.gettoken[]
 parsevalue <- constnameexpression.new[thetoken]
 token.readnexttoken[]
      else
 basevalue.parse[]
 parsevalue <- basevalue.parseresult[]
      end if
   end parse

   function parseresult [] -> [result]
      result <- parsevalue
   end parseresult
end argument

(*************************************************************************)
const constdecllist == list.of[constdecl]
```

```
const basicexpressionlist == list.of[basicexpression]

const ivardecllist == list.of[ivardecl]

const lvardecllist == list.of[lvardecl]

const fparlist == list.of[fpar]

const proclist == list.of[proceduredef]

const fnctlist == list.of[functiondef]

const argumentlist == list.of[argument]

const bblist == list.of[bb]

const statementlist == list.of[statement]

;
```