University of Southampton

Static Analysis for Distributed Prograph

by

Benoît Lanaspre

A thesis submitted for the degree of

Doctor of Philosophy

in the

Faculty of Engineering and Applied Science

Department of Electronics and Computer Science

October 1997

Abstract

Prograph [Cox and Pietrzykowski 1988] provides a sophisticated application builder, together with a visual programming language, supported by a powerful program development environment. The programming language uses an object-oriented model for data abstraction and the logic is based on a dataflow model of computation, specified graphically.

Graphical dataflow gives programmers a clear view of the potential for exploitation of concurrency and so the Prograph language appears to give some leverage for the programming of parallel or distributed systems. However parallel scheduling of operations based solely on the dataflow dependencies might result in the incorrect execution of programs in a distributed environment.

This thesis investigates the issues to be addressed to develop a distributed version of Prograph. A first issue is that of a programming model for Distributed Prograph and a second issue is that of the design of mechanisms to implement the model. The need for a static analysis to support the implementation of the model is justified.

The proposed analysis is divided into three logical parts: a type inference, an effect inference and an effect synthesis. Suitable representations for the type and effect information are presented. The inference algorithms are described and the implementation of the analysis tool and test results are discussed.

Acknowledgements

I would like to thank my supervisor, Hugh Glaser, for proposing and supervising this research. Hugh Glaser never had too little time to answer my questions or to discuss theoretical issues and implementation problems. His constant support helped me retain my motivation throughout the duration of this work.

Pictorius Inc. generously provided both software and technical support for this research.

I am also indebted to the University Research Council which paid the tuition fees for my studies in the department of Electronics and Computer Science.

I wish to express my gratitude to Stuart Maclean who reviewed some of the chapters of this thesis.

Special thanks are due to Luc Moreau and Danius Michaelides for the fruitful technical discussions that we had.

My parents deserve a special mention for their continuous and generous support throughout the years. Without their help, this PhD degree would not have been possible.

Contents

1 Introduction

This chapter presents the motivations to undertake this research in its first section. The second section sets the goals for this research. Major previous works related to this research are presented in the third section. The reader can find an overview of this thesis in the fourth section. The last section reviews the expected contributions of this work.

1.1. Motivations

1.1.1. Distributed Systems

The evolution of computing since its inception in the 1940's has not been limited to the progress, however impressive that progress may be, of the hardware. The evolution has also been that of the applications and of the tools to develop these applications. Over the years the certitude has also grown that improvements in the hardware technology are not the only way towards greater performance and functionality. This goal can also be achieved by exploiting several computers to perform a common task, where the meaning of common task is left open at this stage.

In [Bal, Steiner and Tanenbaum 1989], the following definition is given:

"A distributed computing system consists of multiple autonomous processors that do not share primary memory but cooperate by sending messages over a communication network."

This definition encompasses a broad spectrum of systems ranging from *tightly coupled* systems such as distributed memory parallel machines to loosely coupled systems such as remote computers connected by a Wide-Area Network.

The variety of hardware configurations reflects the variety of classes of distributed applications.

1.1.2 Distributed Programming

Distribution results in an added complexity for program development, as distributed applications have to deal with communication between activities, concurrency and synchronisation. If the benefits of distribution are to be realised, it is necessary to be able to develop applications with a reasonable level of productivity. The design and implementation of distributed applications can be facilitated by three different factors: a programming model, tools to implement the application and other tools to test it.

Considerable work has been undertaken to develop models for distributed systems. The role of the model is to present the programmer with useful abstractions to deal with the different dimensions of distributed programming and to make the reasoning about applications tractable. A good model strikes a good compromise between the ease with which the programmer understands it and the efficiency with which it can be implemented.

Tools are also available to assist the user to write and to test programs. An example of a programming tool for distributed applications is the interface compiler. Such a compiler generates the templates for the application code and some low-level code for the networking operations from an interface specified using an interface definition language. The task of the programmer may also be alleviated by reusing software contained in libraries of procedures or classes.

The behaviour of a distributed application is potentially more difficult to understand than that of a sequential application. Tools have been developed to ease the task of distributed application testing and debugging and to monitor the execution of distributed programs. Some of these tools rely on a visual representation of the computation to help the user understand its behaviour.

1.1.3 Prograph

Prograph is a graphical programming environment and language which grew out of the work of Pietrzykowski and Cox on graphical languages for functional programming, but has evolved considerably since then, and is currently marketed as a general-purpose programming language and programming environment.

The programming language uses an object-oriented model for data abstraction and the logic is based on a dataflow model of computation, specified graphically.

The application builder consists of an extensive set of classes (*Application Building Classes* or *ABC*'s) which provide a framework with which to build applications. ABC editors let the user create and modify ABC objects without leaving the familiar *WIMP* (*Windows, Icons, Menus, Pointer*) paradigm.

The environment also provides an interpreter with substantial interactive debugging and editing facilities. Data values can be checked, edited and copied at run-time. Several evaluation modes are available: animate, single step, trace. It is possible to roll execution forward and backward, set breakpoints and monitor the computation stack.

The combination of the application builder and the interpreter makes possible an incremental style of programming.

1.1.4 Static analysis

Static analysis aims at obtaining information about the behaviour of a program without actually executing the program. Instead, the analysis makes use of the language semantics to derive information from the program source.

The information serves two different purposes: verification and optimisation.

The analysis may verify program properties such as type correctness, termination or, in the context of concurrent programming, deadlock freedom.

Optimisation refers to the improvement of the execution of a program. Improvements are not only concerned with execution speed as memory usage is also relevant. As the degree of abstraction offered by programming languages increases steadily, the requirement for optimisation becomes more stringent. In [Field and Harrison 1988] p.445, it is argued that optimisation is "an essential component of any viable implementation of a functional language". Information about different properties allows different types of optimisations:

- In the context of functional languages, *strictness* analysis checks whether the values of the arguments passed to a function can be computed now or if their evaluation must be delayed.
- Information about the lifetime of data objects is useful to improve the management of memory.

1.2 Objectives

It is believed that a distributed version of the Prograph language, *Distributed Prograph,* would give some leverage for the programming of distributed systems. Three features of the language support this view:

- Visual Dataflow allows the programmer to represent parallelism in a more natural way than text-based languages, because textual representations are, by their very nature, sequential.
- Object-orientation provides the user with powerful encapsulation and abstraction mechanisms that are essential as the size and the complexity of applications grow. Such facilities may provide a good mechanism for encapsulating parallelism constructs in the same way that they already provide for the design of user-interfaces.
- The Prograph development environment greatly contributes to the ease of use that users report.

The target architecture for Distributed Prograph is that of several workstations connected by a local area network.

Distributed Prograph is aimed at using distributed systems for parallel programming where the main motivation is to speed-up application through the use of several computing resources.

The original sequential model should be extended to support distribution while retaining its simplicity. Some of the features of the language might not lend themselves easily to the development of a distributed version and so distribution mechanisms which would maintain the current semantics of the language might be difficult to implement and/or highly inefficient.

One of the main goals of this work is to develop a static analysis to support distribution in Prograph to alleviate these difficulties. The analysis is focused on the effect properties of operations; that is, how the arguments of an operation or some global variables are accessed or modified during the execution of this operation. The information gathered by the analysis is to be used to elaborate a distribution strategy which both respects the semantics of the model and, at the same time, ensures a reasonable level of efficiency.

The static analysis comprises two components: a type inference mechanism and an effect inference and synthesis mechanism. Although type information is not directly relevant to distribution, knowledge of the values' types is relevant to the analysis as it will be shown that in Prograph types and side-effects are not completely orthogonal.

1.3 Related Work

This work is concerned with three different research areas: distributed programming languages, type inference and effect analysis. The following subsections give a summary of the work undertaken in these areas. The discussion of the related work is expanded in the appropriate chapters of this thesis.

1.3.1 Distributed programming languages

Languages for distributed programming have attracted considerable research interest. Given Prograph's ancestry and its object-oriented nature, two classes of distributed languages should be investigated in particular.

- Prograph was at its beginnings thought of as a visual functional language, and distributed programming languages or environments based on functional languages should provide useful information on how to proceed with the design of Distributed Prograph.
- Prograph also presents some similarities with Smalltalk and the work undertaken to develop distributed versions of Smalltalk and other object-oriented languages is of interest to this work.

1.3.2 Type Inference

Type inference is a form of static analysis whose purpose is to compute the type of all expressions occurring in a program in the absence of type declarations from the programmer. Type inference has been investigated for languages belonging to various paradigms: procedural, functional and object-oriented and the type information inferred can be used for various purposes including establishing type correctness and allowing some code optimisations.

Kaplan and Ullman [Kaplan and Ullman 1980] devised an algorithm for the inference of the types of the program variables for an abstract imperative language.

Milner's work [Milner 1978] on the ML type system led to a successful type inference algorithm which is efficient and supports a flexible type system. ML combines both interactivity and strong typing. Milner's type checking system has been incorporated in other functional languages.

The dynamic nature of Smalltalk's type system greatly contributes to the flexibility of the language. However, being able to obtain type information statically offers several benefits, an important one is the ability to anticipate run-time binding errors. Suzuki [Suzuki 1981] proposed an inference algorithm for Smalltalk drawing heavily on Milner's experience. More recently, Palsberg and Schwartzbach [Palsberg and Schwartzbach 1991] developed an original type theory for object-oriented languages and presented an inference algorithm for a Smalltalk-like language.

1.3.3 Effect analysis

Effect analysis is another important form of static analysis which has been applied to procedural and functional languages in a variety of contexts.

Two statements in a program can be connected by a *control dependence*, which means that one statement must be executed before the other one or by a *data dependence* which means that one statement reads some data and the other modifies it. Control dependencies can be detected by control flow analysis and the detection of data dependencies relies on the results of an effect analysis. Automatic parallelisation and a range of compile-time optimisations rely on dependency analysis.

The task of a parallelising compiler is to partition a computation into smaller subcomputations that can be executed in parallel. The parallel schedule must respect the data dependencies existing between the subcomputations. The *Miprac* parallelising compiler converts C, FORTRAN or Scheme programs into an intermediate language which can be analysed [Chow and Harrison 1992]. The analysis gathers information about program properties such as side-effects but also object lifetime, data dependence and unordered accesses.

Functional languages offer a higher degree of abstraction than their imperative counterparts. However, functional languages are usually less efficient than imperative ones. For the sake of efficiency, some functional languages have been augmented with a limited number of imperative constructs. Several effect analyses have been proposed [Wright 1993] to extend Milner's type discipline to be able to type the newly added imperative constructs.

In the context of software engineering, effect analysis has been used to select test data [Rapps and Weyuker 1982] or for *program slicing* [Horwitz, Reps and Binkley 1988]. Program slicing means extracting from a program source the statements that are necessary to understand a certain property of the program, for example, the computation of the return value of a procedure.

1.4 Overview of the thesis

This thesis is divided into eight chapters. The content of each chapter is now presented:

- The Prograph language and development environment bring together several advanced features such as visual programming, dataflow and object-orientation. Each of these features has considerable implications for the programmer and the implementer; the aim of Chapter 2 is to give an appreciation of the language in a manner which is relevant to this research. Details of the language implementation that are relevant for the static analysis are explained.

- Numerous programming models have been proposed and several of these models and their language implementations are described and discussed in the third chapter. The benefits of this review are twofold; firstly, the requirements of distributed programming are presented in a pragmatic manner through several case studies, and secondly the Distributed Prograph model is explained in the context of other distributed programming models. Chapter 3 also discusses the expected benefits of a distributed version of Prograph.

- Chapter 4 looks at the design issues that have to be tackled in the implementation of the Distributed Prograph model. Distributed Object Based Systems have attracted considerable research interest over the last two decades and the state-of-the-art has progressed quite steadily.

Chapter 4 presents the aspects of this accumulated experience that are relevant to the implementation of Distributed Prograph. State and behaviour consistency is of particular interest to this research and are discussed in depth. As stated in the thesis title, this research work has been mainly concerned with static analysis; the question of the purpose and motivation for the proposed analysis is addressed at the end of chapter 4.

• Chapter 5 begins with a comparison of the different approaches to typing in programming languages in general and object-oriented languages in particular. Various attempts at type inference are compared, the points of comparison being that of the language to which the inference is applied, the purpose of the inference and the representation chosen for the type information. The type inference for Prograph is presented along similar lines.

• Chapter 6 is devoted to effect inference and synthesis. Previous work undertaken in the field of effect analysis is reviewed. Effect inference is concerned with the *effect signature* of operations and the effect inference algorithm that derives the effect signature is described. The effect synthesis combines the effect signature of an operation with its context, to produce an approximation of the effects induced by the execution of an operation.

• Chapter 7 highlights significant aspects of the implementation of the analysis. The analysis is illustrated by several commented examples. The applicability of the analysis is also discussed. The last part of chapter 7 makes suggestion on how the results of the analysis can be used.

• Chapter 8 concludes the thesis with a discussion of its contributions and the future work that could be undertaken.

1.6 Contributions

The contributions of this work are:

• A type inference mechanism for Prograph is proposed.

• A prototype of the type inference system is implemented in Prograph.

- Building on the experience of type inference, an effect inference mechanism is proposed.
- A prototype effect inference is implemented.
- A synthesis algorithm is proposed and implemented.

2 Prograph features

Prograph is a comparatively new language and embodies some of the latest trends in programming language and environment design.

The history of Prograph is presented in the first section of this chapter. In the second section, the features of the language and of its interpreter are reviewed. It is also necessary to give some explanations about the techniques used to implement the language in order to understand better the analysis described in the following chapters. The third section explains that both applications and application development tools are built from a set of *Application Building Classes* (*ABC*'s), which allows for the easy customisation of applications and development tools. The last section of this chapter is devoted to the facilities provided by the current version of Prograph for distributed programming.

2.1 Prograph history

Prograph originated at Acadia University around 1983. The first implementation of the language, in Pascal, was due to Pietrzykowski [Matwin and Pietrzykowski 1985]. The impetus for the development of Prograph was to better understand applications written in the functional language FP [Backus 1978]. The name "Prograph" was obtained by analogy to the word "program", where the suffix "-gram" meaning "writing" was changed to "-graph" meaning "drawing" [Cox 1996]. A compiler was also developed [Cox and Mulligan 1985].

About 1985, a second experimental implementation of the language was built in Prolog by Pietrzykowski and Cox at the Technical University of Nova Scotia. An editor, interpreter and debugger were also developed [Cox and Pietrzykowski 1985].

Experiments with the first version of the language highlighted the need for redefined language constructs and for a data abstraction mechanism. The `if - then - else -` constructs of the original language were replaced by a Prolog-like case structure with success/failure as the trigger, the iterative construct `while - do` also disappeared and the list and loop annotations were introduced to express iteration. Object-orientation was introduced in

1987 as the mechanism for modularisation and data abstraction. The language which resulted from these modifications, described in [Cox and Pietrzykowski 1988], has known no major alteration since then.

Commercial development of the language and of the environment associated with it began in 1986 at *The Gunkara Sun Systems Limited* company in 1986 and commercial exploitation has been pursued by the successive instances of the company: *TGS, Prograph International* and now *Pictorius Inc.*

The first commercial version of Prograph for the Macintosh was released in 1989 and comprised an editor/interpreter (written in C) and a development environment with a small library of *System Classes*. The System Classes were special classes whose behaviour was implemented in C and which could be manipulated using special editors also written in C. 1990 saw the introduction of a compiler for generating standalone applications.

The release of the *Cross Platform Environment* (CPX) version of the language in 1993 marked an important evolution of the development environment. Prograph CPX was designed to exploit the code reuse and component-based approach associated to object-oriented programming to a larger extent than the previous versions of Prograph. Prograph CPX (whose current version is 1.4) ships with a large library of *Application Building Classes* and a sizeable part of the application editor is now implemented in Prograph in the form of *Application Building Editors*.

The modular architecture of the current version of Prograph has allowed the development of components by third-party developers. Application development tools have been built on top of Prograph CPX. For example, Entrada!, released in 1995, is targeted at the construction of client/server applications, including Internet based client/server applications.

In order to broaden its appeal and realise the benefits of cross platform portability, Prograph is being reimplemented for the Windows95 platform and a preliminary version has already been made available to developers at the time of this writing.

2.2 Language features

2.2.1 Introductory example

The introductory example presented in this subsection is taken from [Cox and Smedley 1996]. Note that Prograph is an object-oriented language, hence the term *method* is used to refer to entities known as *procedures* in standard programming languages.

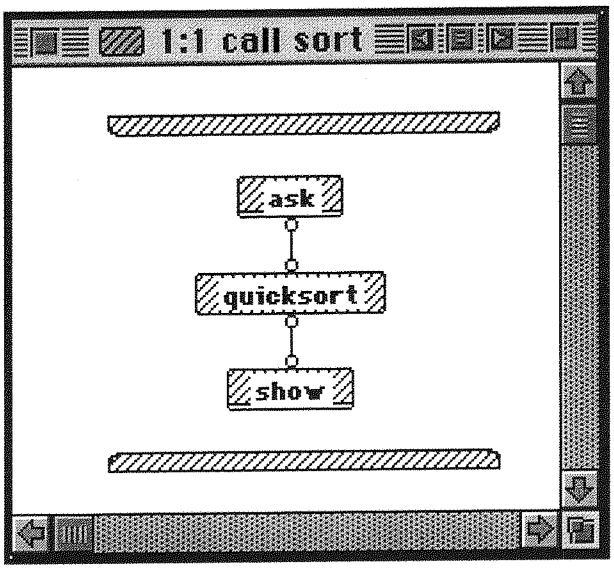


Fig. 2.1: `call sort` method

Fig. 2.1 shows the details of the method `call sort`, a dataflow diagram in which three *operations* are connected sequentially by lines called *datalinks*. A datalink transmits data from an output of an operation, represented by a small icon called a *root* on the bottom of the operation, to the inputs of other operations, represented by *terminal* icons on the tops of operations. The first operation in this diagram, `ask`, is a *primitive* that calls system-supplied code to produce a dialogue requesting input from the user. Note that the icon for a primitive is distinguished from other operation icons by the white line along its bottom edge. When `ask` has been executed, the data input by the user flows down the datalink to the operation `quicksort`, invoking the method `quicksort`. This method expects to receive a list, which it sorts as explained below, outputting the sorted list, which flows down the datalink to the show primitive. The show produces a dialogue displaying the sorted list. Fig. 2.2 shows a Prograph implementation of the well known algorithm `quicksort` for sorting a list into ascending order.

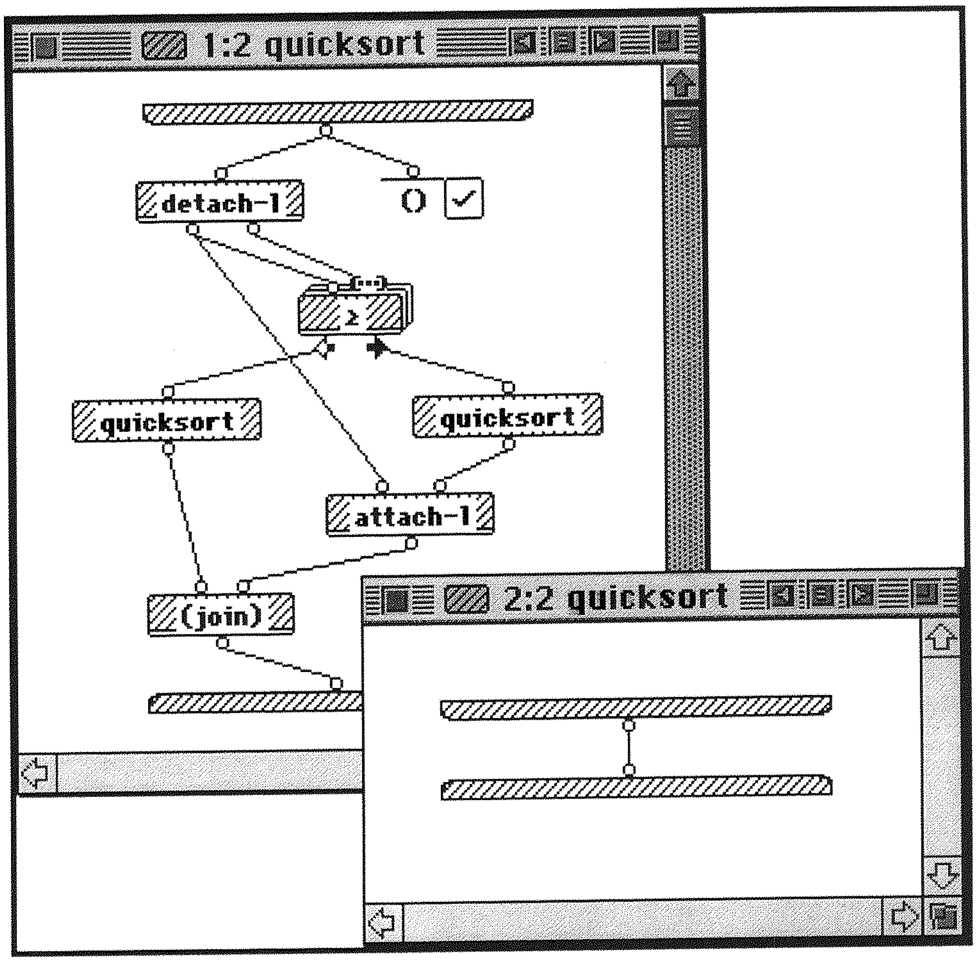


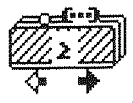
Fig. 2.2: A `quicksort` method

The method `quicksort` consists of two *cases*, represented by the dataflow diagrams as shown in fig. 2.2. The first case, shown in the window entitled `1:2 quicksort`, implements the recursive case of the algorithm, while the second implements the base case. In general, a method consists of a sequence of cases. The bars at the top and bottom of cases are special operations called the *input bar* and *output bar* respectively. The input bar is always the first operation executed, and copies the values of parameters from the terminals of the calling operation to the input bar roots. Similarly, if the case executes to conclusion, the output bar is the last operation executed, copying the values on its terminals to the roots of the calling operation.

In the first case of `quicksort`, the first operation to be executed is a `match`, () ✓, a special operation which tests to see if the incoming data is the empty list. The icon attached to the right end of the match is a `Next`

`Case on success` *control,* which is triggered by success of the match, immediately terminating the execution of the first case and initiating execution of the second. If this happens, the empty list is simply passed through as the output of the second case, and execution of `quicksort` finishes, producing the empty list.

In the first case of `quicksort`, if the input list is not empty, the control on the `match` operation in the first case is not triggered, and the first case is executed to completion. The operation to be executed immediately after the match is the primitive `detach-l` which outputs the first element of the list and the remainder of the list on its left and right roots respectively.

The next operation to be executed,  , is an example of a *multiplex,* of which there are several kinds in Prograph, determined by visual annotations. First, the three-dimensional nature of the icon, common to all multiplexes, indicates that the operation ≥ will be applied repeatedly. Second, the list annotation (··) on the right-hand terminal indicates that a list is expected as data, one element of which will be consumed by each execution of the operation. Execution of this multiplex, therefore, uses ≥ to compare the first element of the original list with each of the other elements. Finally the special roots ◇ and ◆ indicate that this particular multiplex is a *partition,* which divides the list arriving on the list annotated terminal into two lists; items for which the embedded operation succeeds and those for which it fails. These two lists appear on the ◇ and ◆ roots respectively.

The lists produced by the partition multiplex are sorted by recursive calls to the `quicksort` method. The final sorted list is then assembled using the two primitive operations `attach-l`, which constructs a new list by attaching an element to the left end of a list, and `(join)`, which concatenates two lists.

The execution mechanism of Prograph is data-driven dataflow. That is, an operation executes when all its input data is available. In practice, a linear execution order for the operations in a case is predetermined by topologically sorting the directed acyclic graph of operations and datalinks, subject to certain constraints. For example, an operation with a control should be executed as early as possible.

The pure dataflow model prohibits alteration of data objects; if a data object is to be modified, a copy of it is made instead. However, application of the strict dataflow principles would result in performance costs when complex data structures are involved. For the sake of efficiency, Prograph implements a modified version of dataflow principles. *Primitive type* data objects are copied but instances of classes are modified "in place". Memory management is automatic in Prograph and data which is no longer used is automatically garbage-collected.

In the example shown in fig. 2.2, the method `quicksort` has only one input and one output, and therefore does not show how the terminals of an operation are matched with the roots on the input bar of a case of the method it invokes. These terminals and roots must be of equal number, and are matched from left to right. A similar relationship exists between the roots of an operation and the terminals of the output bar in a case of a method invoked by the operation.

One important kind of operation not illustrated in the above example is the *local* operation. A local operation is one that does not call a separately defined method such as `quicksort`. Instead, it contains its own sequence of cases, called a local method. It is therefore analogous to a parametrised `begin-end` block in a standard procedural language.

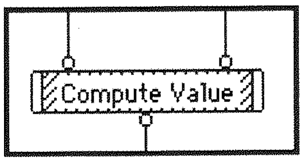


Fig. 2.3 : A call to a local method

As mentioned earlier, Prograph is an object-oriented language and therefore provides facilities for defining new datatypes as classes. The methods in the above example, called *universal methods*, deal with simple data rather than instances of classes, and therefore do not belong to any class. It is important to note, however, that classes also contain methods, that several classes may have methods of the same name, and that an operation may invoke different methods at different times during execution.

External methods are calls to the operating system. Fig. 2.4 shows the pictorial representation for an operation calling an external method:

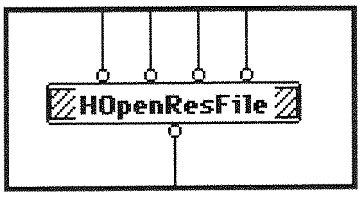


Fig. 2.4: A call to an `external` method

However, application code seldom uses external methods directly. The functionality of the external methods is provided either by the primitive methods or by the Application Building Classes.

2.2.2 Control of execution

The example presented in section 2.2.1 shows that controls and annotations may affect the flow of control and data.

Conceptually the computation is driven by the availability of data. When all the input values of an operation are available, the operation can be executed. The execution produces a `succeed` *execution message* and the results are output on the roots of the operation, alternatively it produces a `fail` execution message.

A control is the combination of an execution message and of an action. The range of possible actions is explained:

Control name	Description	Symbol (`fail` execution message).
`Next Case`	When a `Next Case` control is activated, the flow of control is transferred to the next case of the method being called.	☒
`Fail`	When a `Fail` control is actived in the case of a method, the operation that called that method produces a `fail` execution message.	⊗
`Continue`	The combination of a `succeed` execution message with a `Continue` action is semantically equivalent to no control, `Continue` with a `fail` execution message ignores the failure of the operation to which the control is attached	☒
`Terminate`	The `Terminate` control is used to control the iterations of the calling operation. When a `Terminate` control is activated in the case of a method, the remainder of the case is skipped and the iterations of the operation that called the method interrupted.	☒
`Finish`	The `Finish` control is also used to control the iterations of the calling operation. When a `Finish` control is activated in the case of a method, the remainder of the case is executed but the iterations of the operations that called the method are interrupted.	☒

A `list` annotated terminal indicates that the argument on this terminal will be a list and that the operation should be applied to every element of the list until the end of the list is reached or a control is activated.

A `loop` annotation creates a cycle whereby the value coming out of a root is fed back into a terminal until the iterations are completed or interrupted.

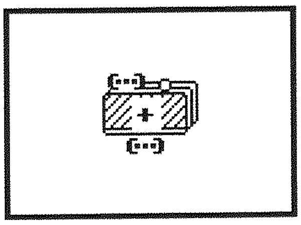


Fig. 2.5.a: `list` terminals

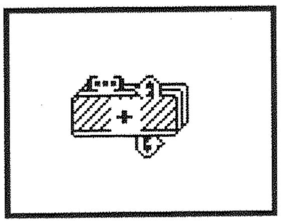


Fig. 2.5.b: `loop` terminals

The behaviour of an operation can also be modified by a *multiplex* annotation:

- A `repeat` annotated operation executes repeatedly until a `Terminate` or `Finish` control is activated in one of the cases of the method called by this operation (see fig. 2.6.a).
- A `partition` operation converts a predicate operation into a filter operation which splits an input list into two result lists (see fig. 2.6.b).

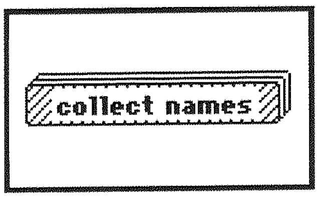


Fig. 2.6.a: A `repeat` operation

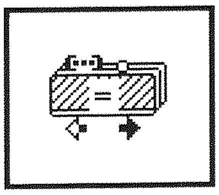


Fig. 2.6.b: A `partition` operation

The execution of a computation can be controlled by using an *inject* terminal and the `call` primitive. An inject terminal allows the naming of an operation at run time:

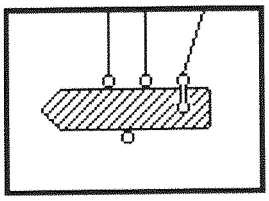


Fig. 2.7: A `Set` operation with an inject terminal

In fig.2.7 the rightmost terminal of the operation is an inject terminal. The name of the `Set` operation is given at run time.

A distinguished primitive, `call`, calls the method whose name is passed as argument.

2.2.3 Object-orientation

2.2.3.1 Terms and definitions

Wegner [Wegner 1987] distinguishes three generic features for object-oriented systems: *object, class* and *inheritance.*

- An object encapsulates both state and the operations to manipulate that state. Objects are instances of classes.
- A class is an abstract template describing the internal state and the behaviour of its instances.
- Inheritance is a mechanism for code reuse; a subclass inherits the behaviour and state of its parent class. The subclass *extends* the behaviour of its superclasses by *overriding* the inherited methods or by *defining* new methods.

Data abstraction requires that the state of objects is accessed only through the operations of the objects. Although this is not systematic, object-oriented languages often enforce data abstraction.

2.2.3.2 Prograph class system

The Prograph class system supports single inheritance, where each class inherits from at most one class. "At most" is significant in the sense that the Prograph class hierarchy is "a forest of trees" and not a "tree" as in Smalltalk [LaLonde and Pugh 1990] in which all classes inherit from the `Object` class.

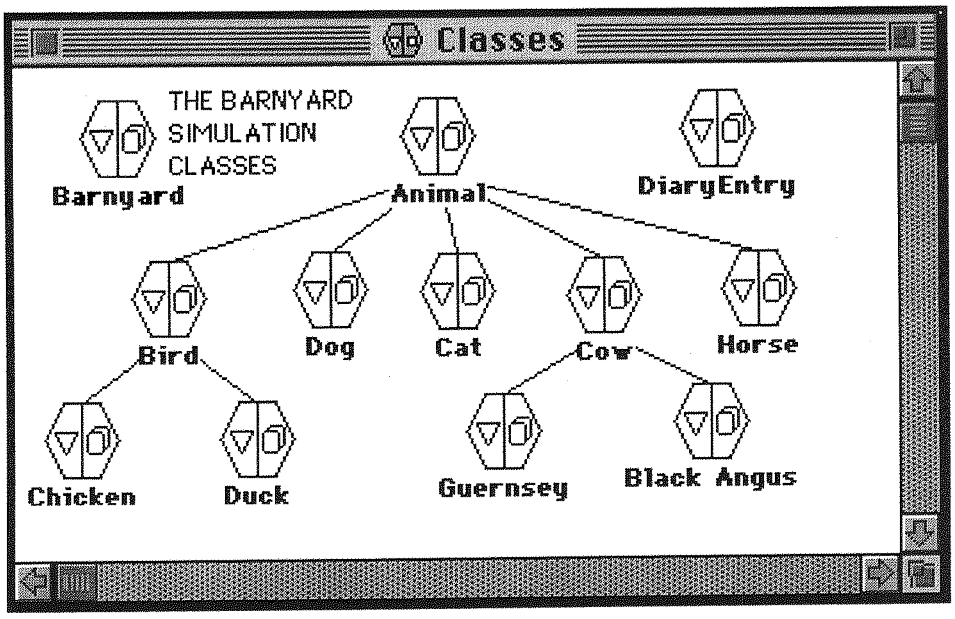


Fig. 2.8: Class icons and inheritance trees.

The collection of classes shown in fig. 2.8 contains two single classes `Barnyard` and `DiaryEntry` as well as an inheritance tree with the class `Animal` at its root.

2.2.3.3 Object state and behaviour

Unlike Smalltalk, Prograph has not adopted "the everything is an object" philosophy. In Prograph, the data flowing along the arcs of the dataflow graph can be a value of a Prograph primitive type, or, it can be an object whose type is the class of the object.

Prograph provides the following primitive datatypes: `boolean`, `external`, `integer`, `list`, `none`, `null`, `real`, `string` or `undefined`. Most type names are self-explanatory. `external` is the type of operating system data structures. `none` is the type of a distinguished value `NONE` which is passed to an operation when the matching terminal is not connected. `NONE` is suppressed from the list constructed on the list root of a multiplex operation. `undefined` is the type of another distinguished value, `UNDEFINED`, which is used when a computation is rolled forward during debugging.

It is important to note that Prograph does not view classes as objects and that classes are not first-class values (classes are first class values in Smalltalk). Each class supports two sets of attributes, the *class attributes* and the *instance attributes*. Each attribute can be inherited from a superclass or *defined* in the class. The value of a class attribute is shared by all the instances of the class while the value of an instance attribute is private to each instance.

Contrary to the principles of data abstraction, the values of all the attributes can be accessed or modified using default `Get` or `Set` operations respectively (see fig. 2.9).

The name of a `Get` operation is that of the attribute whose value must be returned. The arity of the `Get` operation is fixed: one terminal and two roots; the leftmost root returns the value flowing into the terminal and the second root returns the value of the attribute.

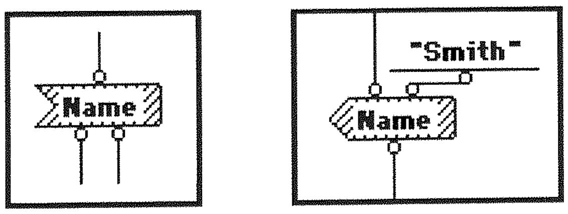


Fig. 2.9: `Get` and `Set` operations

Likewise, the name of a `Set` operation is that of the attribute whose value is to be modified; its arity is two terminals and one root, the new value of the attribute flows into the second terminal of the operation, the value on the root of the `Set` operation is the value flowing into the leftmost terminal.

The state of a class consists of the values of the class attributes and of the default values of the instance attributes defined by the class. `Get` and `Set` operations may be used to access the state of classes, however a class cannot be passed as argument to a `Get` or `Set` operation. Instead a string whose value is the name of the class is passed to the operation. Thus the semantics of a `Get` or a `Set` operation depends on the type of the value flowing on the terminal of the `Get` operation or the leftmost terminal of the `Set` operation.

A default *initialisation* or `Init` operation is provided for each class. The default `Init` operation has the same name as the class from which a new object is instantiated. A new instance comes with the default attribute values defined for its class. The `Init` operation may take as optional argument a list of (attribute name, attribute value) pairs to overwrite the default values or set new class attribute values at instantiation time. Fig. 2.10 shows the creation of a new instance of the class `Person` with the value `"01 05 96"` for the attribute `DOB`.

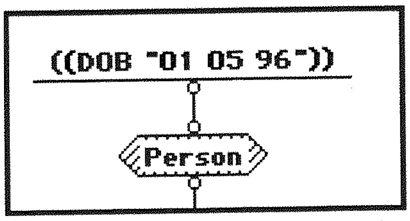


Fig. 2.10: Instance creation

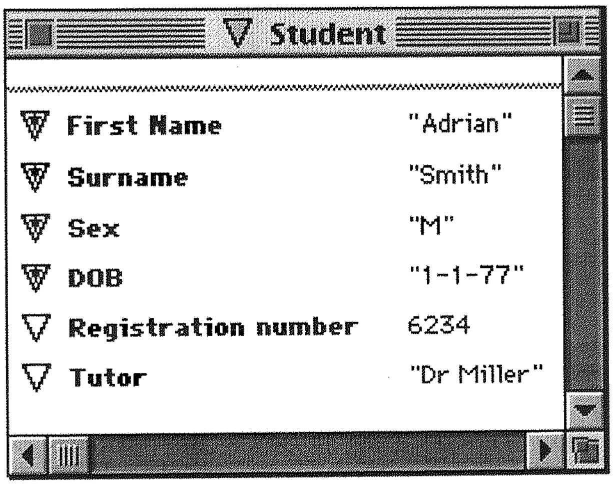
It is possible to override the behaviour of the default initialisation operation by defining a custom initialisation method which is added to the class.

The behaviour of the instances of a class is implemented by a set of class methods. *Simple* class methods are methods of arbitrary arity. It is sometimes necessary to extend the behaviour of the default `Get` or `Set` operations by defining custom `Get` or `Set` methods. The methods must have the same arity as the corresponding operations. A `Get` or a `Set` method may have a name which does not correspond to any attribute in the class. A *virtual attribute* is thus defined.

The definition of a class is presented using a visual form. Different symbols are used to distinguish instance attributes from class attributes and inherited attributes from the attributes defined by the class.

Attribute	Symbol
Class attribute, inherited	⏣
Class attribute, defined by the class	⬡
Instance attribute, inherited	⛛
Instance attribute, defined by the class	▽

Fig. 2.11 shows the attribute window of the class `Student`, all the attributes of the class are listed in this window.


Student

First Name	"Adrian"
Surname	"Smith"
Sex	"M"
DOB	"1-1-77"
Registration number	6234
Tutor	"Dr Miller"

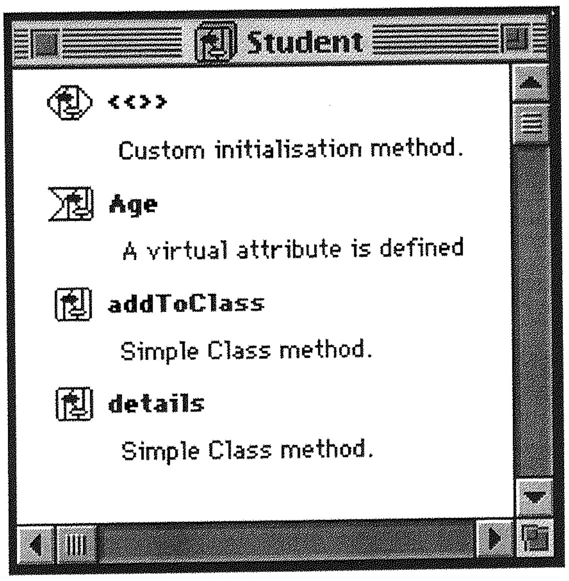
Fig. 2.11: The attribute window of the class Student

In fig. 2.11, all the attributes of the Student class are instance attributes (if Student had class attributes, these attributes would be displayed above the horizontal line in the window). The attributes First Name, Surname, Sex and DOB of the Student class are inherited from a superclass, whereas the attributes Registration number and Tutor are defined by the Student class.

The method window of a class definition displays the methods defined by this class (inherited methods are not displayed). Different symbols are used to indicate whether a method is an Init, a Get, a Set or a simple method.

Method type	Symbol
Init method	
Get method	
Set method	
Simple method	

The window of fig. 2.12 lists the methods of the Student class.



<<>>
Custom initialisation method.
Age
A virtual attribute is defined
addToClass
Simple Class method.
details
Simple Class method.

Fig. 2.12: The method window of the class `Student`.

There exists a set of simple methods not attached to any class, called *universal* methods.

Prograph is a dynamically typed language. In dynamically typed languages, the information is not associated with variables but with values. This means that the same variable can store successive values of different types.

2.2.3.4 Polymorphism

In their survey devoted to data abstraction and polymorphism, Cardelli and Wegner [Cardelli and Wegner 1985] distinguish four kinds of polymorphism:

- *Overloading* is a facility which allows the reuse of the same name for different behaviours. For example in C, the + primitive (addition) is overloaded as there exist two implementations of the addition, one to add integer values and a second one to add real values. Likewise, some of the primitive methods are overloaded in Prograph (e.g. the ≤ relational primitive method). Overloading is not supported for universal methods in Prograph as the universal methods share a single name space with the primitive operations and thus a universal method and a primitive may not have the same name. But different classes can define methods with the same name.

• *Coercion* is an implicit type conversion. An example would be the computation of 5.3 + 4. The integer value 4 is coerced into the real value 4.0 to be added to the real value 5.3. Prograph coerces the arguments of the arithmetic and relational operators.

• *Inclusion polymorphism* is a consequence of inheritance. Each method defined for a class is also *applicable* to its subclasses (unless the method is redefined by a subclass).

• *Parametric polymorphism* can be defined as the property of a function to accept arguments of a potentially infinite number of types. Functions exhibiting parametric polymorphism are termed *generic* functions. One example of a generic function is the computation of the length of a list, where the element type of the list is arbitrary.

A fifth type of polymorphism is often mentioned and should be added to the list:

• *Data polymorphism* refers to the ability of the same variable to hold successive values of different types.

In Prograph, at run-time, an operation calls a method and failure to bind the operation to a method results in a run-time error. The binding depends on the type of reference used by the calling operation, the type of its arguments and the method case in which the calling operation appears.

Prograph offers four types of references:

• The first type of reference is a *universal* reference (shown in fig. 2.13.a). The operation is associated with a universal, a primitive or an external method.

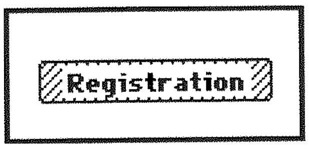


Fig. 2.13.a: A universal reference

• The second type is a data-determined reference, which has the semantics of a message send in Smalltalk. In the object model, objects communicate by sending messages to each other. Upon receipt of a message by an object, the method binding mechanism looks up all the methods with the same name as the message selector and despatches the method applicable to the class of the receiver object, that is, the method that the class of the receiver defines or inherits. In the Prograph dataflow object-oriented model, the receiver is the object flowing in the leftmost terminal of the operation. If there exists no method applicable to the class of the receiver, a universal method or a primitive method may be called. Fig. 2.13.b shows that a data-determined reference consists of the / character followed by the name of the method to be called

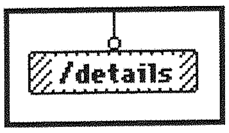


Fig. 2.13.b: A data-determined reference

• The third type is an *explicit* reference. The operation name consists of both a class name and a method name separated by the / character (see fig. 2.13.b). Explicit reference leaves no ambiguity about which method will be called at run-time.

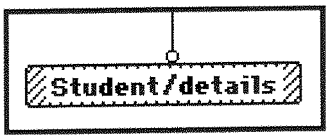


Fig. 2.13.c: An explicit reference

• The fourth type is a *context-determined* reference. The method called is the method applicable to the class of the method containing the operation. It is thus impossible to name operations with a context-based reference in one of the cases of a universal method. A context-determined reference can be *super* annotated. The method called is that applicable to the superclass of the class of the method containing the operation. A data-determined reference consists of two / characters followed by the name of the method to be called (see fig. 2.13.d). Fig. 2.13.e shows the representation of the super annotation.

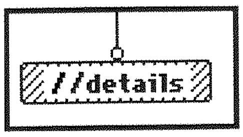


Fig. 2.13.d: A context-determined reference

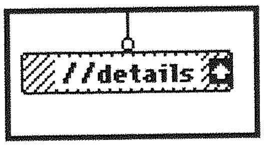


Fig. 2.13.e: A context-determined reference with a super annotation

2.2.4 Persistents

Prograph provides *persistents*. Persistents are named elements which can hold any value. Two operations are available on persistents, a `Get` operation reads the value of the persistent and a `Set` operation modifies the value of the persistent (see fig. 2.14).

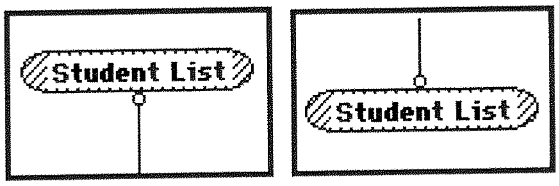


Fig. 2.14: Persistent `Get` and `Set`.

Persistence allows data values to have extent beyond a single execution of a program.

2.2.5 The language editor & interpreter

A visual editor allows the programmer to manipulate the visual components making up the Prograph language. Navigation through nested dataflow diagrams is possible through selecting and clicking on the elements to be inspected. Documenting applications is facilitated by a hypertext facility which combines explanation about primitives, user-written comments on classes, methods and variables. A search facility allows the user to find all the occurrences of a given operation or attribute in the code.

The interpreter fully exploits the visual paradigm to make the behaviour of programs easy to understand. A wide range of features is supported to assist the programmer during the debugging phase: breakpoints, roll backward and roll forward and four execution modes. The programmer can visually monitor how the execution of the code progresses by setting the interpreter in the *trace* or *animate* execution mode, in which operations on the graph are highlighted in a different colour after they have been executed. The interpreter offers a high degree of *reactiveness*, that is, the ability to inspect and edit the data objects taking part in a computation. The stack of the computation currently executed is also available for inspection in a visual form.

Debugging is facilitated by the exhaustive error reporting provided by the interpreter. Run-time error messages include invalid argument type, out-of-range values, stack overflow, activation of a `Next Case` control in the last case of a method, no control attached to a failed operation, no method can be despatched or the named attribute does not exist in the argument class and wrong operation arity.

Execution and editing are tightly integrated. When an error occurs at run-time, the execution is suspended at the faulty program point, the case containing the offending operation is opened for inspection and the interpreter suggests a possible solution to the programmer (e.g. add an attribute or a method to the receiver's class). Execution can be resumed after the faulty value or code has been edited.

Prograph encourages a development methodology whereby the programmer can develop and test an application using the support tools available in the editor/interpreter environment and then use the compiler to generate a stand alone version of the application.

2.2.6 Implementation overview

Although references are seldom manipulated directly by programs, they underpin the implementation of the language.

All method arguments and return values are passed by reference. The use of references has several benefits:

- It allows data polymorphism, since structures do not store values but references to values.
- Values can be shared. The existence of several references to the same value is called value *aliasing*.

The implementation of data values can be described using C structures. All data object structures have three fields in common:

- The `type` field contains an integer value which identifies the type of the value (dynamic typing).

- The `save` field is used when cycles in data linkage need to be detected.
- The `use` field keeps track of the number of times a value is referenced. This reference count is used by the garbage collector.

This basic structure is extended with the necessary fields to represent different data objects. Boolean values are implemented by adding an extra field, called `value`, which stores 0 when the boolean value is `False` and 1 when it is `True`. The declaration in C of the boolean structure would be:

```
typedef struct
{
        Int2 type;
        Nat2 save;
        Nat2 use;
        Bool value;
} CS_boolean, *C_boolean;
```

The representation of instances requires the addition of two fields to the basic structure. A `size` field records the number of slots in the instance. The `C_object` field points to an array of references, the size of which is kept in the `size` field. The field `class` is a reference to the class from which the object has been instantiated. The other references are to the values of the instance (but not class) attributes. The definition of an instance structure would be:

```
typedef struct
{
        Int2 type;
        Nat4 save;
        Nat4 use;
        Nat2 size;
        Handle class;
        C_object* attrs[];
} *C_instance
```

In Prograph, classes are not first class values and the structure underlying their implementation differs slightly from those used to construct data objects; it does not make sense for example to record a use count as class structures are not garbage collected. A class structure is too complex to be usefully explained in detail but an outline is given now. A class structure points to:

- the name of the class.
- an array of references to attribute descriptor structures. An attribute descriptor points to the name of the attribute; a `flags` field carries other information about the attribute: class or instance attribute, inherited or defined for the class.
- an array of references to the values of the class attributes.
- an array of references to the default values of the instance attributes.
- an array of references to the methods supported by the class.
- the structure of its parent, sibling and children classes.

2.3 Application development

2.3.1 Structure of applications

The current version of Prograph is available only for the Apple Macintosh. The traditional area of strength of the Macintosh applications lies in user-centred, *event-driven* applications, where the construction of the user-interface usually makes up a sizeable part of the application.

The requirements of this class of applications are reflected in the structure of Prograph programs. These are built as an event-processing loop: events are inserted in an event stream and during each iteration of the loop the event-handler gets the next event from the event stream and despatches it to the application component that can respond to it. The way an application component should respond to an event is described by a *behaviour*. A behaviour associates the name of a method or primitive with input *specifiers*. When an event is despatched, the behaviour associated with the event is executed by the *commander*.

The code for an application, or *project* in Prograph terminology, is divided into *sections*. A section consists of sets of classes, universal methods and persistents. Sections have no significance for the semantics of the language. They merely facilitate the modular development of applications and the sharing of code between projects.

The *Application Building Classes* (*ABCs*) form a collection of reusable classes from which the programmer creates and customises the components of an application. These components are concerned with:

- User interface building with classes such as `Window`, `Menu` and `Button`.
- Document management with classes for file handling. Datafile primitives offer low-level support for operations on indexed files.
- Utilities, with classes and methods to manage printing, clipboards and system resources.
- Application behaviour, with a collection of related classes such as `Event Handler`, `Commander`, `Task` and `Behaviour` and their associated methods.

The application is structured as an object containment hierarchy (see fig. 2.15). An instance of the `Application` class is at the top of the containment hierarchy. The attributes of the `Application` object store the components making an application (file, menu and windows). Each component may in turn contain several further components. For example in fig. 2.15, the `Application` instance contains an instance of the `Desktop` class, which in turn contains an instance of the `Menubar` class.

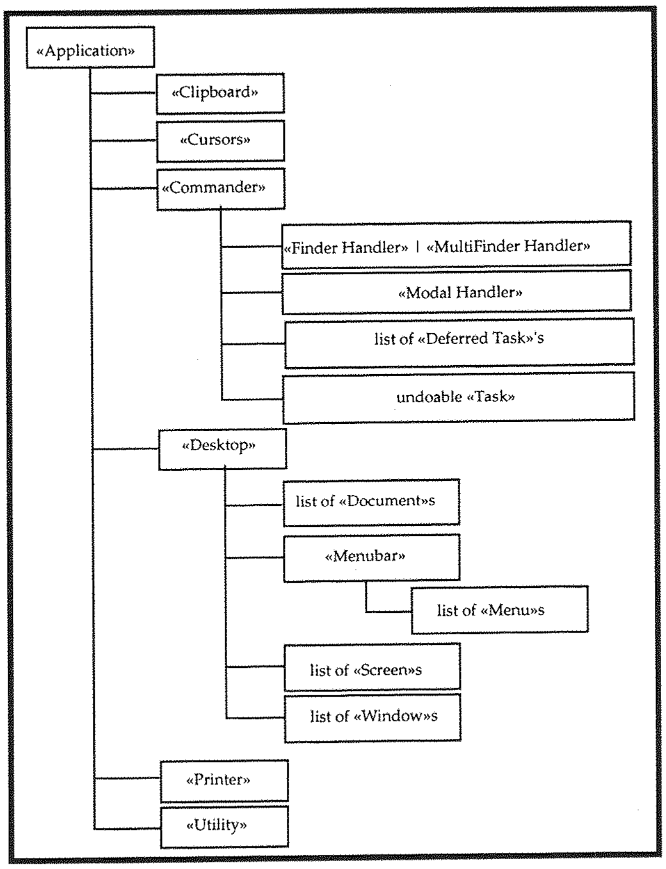


Fig. 2.15: The application containment hierarchy

2.3.2 Application building tools

ABC objects can be created and modified in a direct manipulation fashion using the *Application Building Editors* (*ABE*'s). Visual components of applications such as windows and buttons are drawn in a *WYSIWYG* (*What you see is what you get*) fashion. However, editors are not restricted to the manipulation of visual objects. Behaviours, for example, may also be specified by typing a method name and choosing the input specifiers from a pull down menu.

When creating an application component from an ABC, the associated ABE actually creates a subclass of the ABC. The values required for the instantiation of the component are recorded as the default values of the attributes of the ABC subclass.

2.4 Support for distribution in Prograph

The current version of Prograph provides simple mechanisms for distributed programming.

Packing is the transformation of an object into a stream of bytes for storage on a permanent media or transmission over a network. *Unpacking* is the reconstruction of an object from a stream of bytes.

When packing an object, it is meaningless to save references as these references will have no meaning in another context, so instead the values pointed at by the references should be packed as well. The `to-bytes` primitive in Prograph packs data objects of arbitrary complexity and produces a map which can be used when unpacking the object. The values of instance variables are packed with the instance that refers to them. The primitive `from-bytes` reconstruct objects from their byte stream representation and the associated map.

Different communication protocols can be used to transmit packed objects. Prograph provides a set of primitive methods to wrap up calls to the *Apple Transaction Protocol* (*ATP*), which is one of the protocols supported by the *Appletalk* protocol suite [Sidu, Andrews and Oppenheimer 1990]. Other communications protocols, notably TCP-IP, are supported by add-on products.

2.5 Summary

- Prograph was first conceived as a functional language using a visual dataflow representation.

- The current version of the language features an object system to provide data abstraction.

- For efficiency reasons, Prograph does not adhere to the pure dataflow model. Instead, complex data structures can be modified in place. Values, in Prograph, are dynamically typed and memory management is automatic.

- A powerful editing/debugging environment supports the task of the programmer. Applications can be created from a large collection of reusable classes.

- Support for distribution is limited in the current version of Prograph.

3 Distributed Programming models

The term *distributed systems* encompasses a broad range of hardware configurations and applications.

A classification for such systems is proposed in the first section of this chapter. The second section discusses the role of a distributed programming model. Sections three, four, five and six are devoted to four of these models. The last section presents a model for Distributed Prograph.

3.1 Classes of Distributed Applications

In [Bal, Steiner and Tanenbaum 1989], the reasons for using distributed systems are put in four different categories:

- Execution speed-up for a single computation can be achieved through parallelism. Numerical applications are characterised by the regularity of both their data structures (vectors or matrices) and control structures (loops). Symbolic programming handles symbolic data such as deduction rules in expert systems. This data is represented by complex and irregular data structures. The requirements of symbolic programming have motivated the development of declarative languages. The higher level of abstraction provided by these languages, automatic memory management for example, makes symbolic programming more tractable than with imperative languages. Intensive numerical or symbolic computations can be split into smaller granularity computations. Flow modelling and the implementation of a true-or parallel facility are examples of numerical and symbolic parallel applications respectively. Parallel applications are often characterised by a high communication to computation ratio. Workstations connected by a local area network are establishing themselves as an alternative to parallel machines and vector processors to execute such parallel applications.

- The benefits of distribution with respect to fault-tolerance have been recognised for some time. Fault-tolerance can be provided by purpose-built distributed hardware. Alternatively, duplication of data and functions on autonomous machines increases the reliability and availability of the system. Mechanisms must be devised to ensure the consistency of the replicated data and a proper synchronisation of the replicated activities. The ISIS toolkit [Birman 1993] provides support to build distributed fault-tolerant applications using normal hardware.

• Resource sharing is possible through the functional specialisation of parts of the system, known as resource managers or *servers*. This class of applications is known as *client/server* applications. Client/server applications are usually deployed in multi-user environments. Heterogeneity is an important issue in the design of client/server applications. This heterogeneity ranges over hardware, operating systems, communication protocols and application programming languages. Client/server applications are progressively replacing mainframe centred applications within commercial organisations as is reported in the 1996 Datapro client/server survey [BYTE 1996].

• Some applications such as Automatic Teller Machines (ATMs) are intrinsically distributed. This class of distributed applications emerged as early as the 1950s with the early developments exemplified by the SABRE airline reservation system. These systems can be described as loosely coupled as they involve wide-area network communication. The relevance of this class of applications is illustrated by the growing popularity of the Internet and the other distributed applications it has spawned such as electronic mail and the World-Wide-Web.

3.2 Distributed Programming models

"A distributed programming model is one which enables us to set up and coordinate activities residing at multiple autonomous machines connected by a network" [Coulouris, Dollimore and Kindberg 1992]. Programming *in the small* refers to the task of describing the individual activities, whereas the coordination of these activities is referred to as programming *in the large.*

A distributed programming model should present the programmer with the necessary abstractions to deal with parallelism, communication and synchronisation between activities. The level of abstraction supported may vary considerably from one model to another.

A model may shelter the programmer from the issues arising from distribution. The opposite view states that a programming model consists of a *computation model* and a *coordination model* and these two models can be developed separately [Gelernter and Carriero 1992].

A model is often biased toward a class of applications. The communication pattern between the activities involved in a parallel computation differs from that in a client/server application. The requirements of the class of applications targeted are reflected in the abstractions provided by the model.

There are two approaches to parallelism. The first approach advocates *implicit* parallelism where the programmer need not indicate which portions of the program should be executed in parallel. With *explicit* parallelism, language constructs are available to express parallelism within a program. *Mapping* is concerned with the assignment of activities to the processing resources available within the distributed system.

The activities making up a distributed computation need to exchange data by means of communication over the network. A programming model may abstract communication to various extents. Some models offer some high-level views of network operations. At the other end of the spectrum, communication may be completely hidden from the programmer and dealt with by the implementation.

The model also deals with synchronisation requirements. Activities are said to be *synchronised* if the progress of one is conditional upon an event caused by the other.

3.3 Process Model

3.3.1 The process abstraction

The process model is often implemented by procedural languages. A procedural language describes a computation as a sequence of instructions which access and modify data stored in memory. Procedures are groups of instructions that can be referred to by a name and can be called. Procedural languages view computations as a set of procedure definitions and a sequence of procedure calls.

The process is the abstraction of the hardware used to execute a computation. A process encapsulates the program data, its code as well as one or more threads of execution. A thread is the abstraction of an activity.

3.3.2 Distributed processes

In the context of distributed programming, processes are seen as units of concurrency and distribution. Processes do not share state and communicate by the sole means of message passing. The purpose of a message might be to transmit data or invoke some behaviour of another process. *Point-to-point communication* is the name of the former type of message-passing and *Remote Procedure Call* that of the latter.

3.3.2.1 Point-to-point communication

With point-to-point communication, processes exchange unidirectional messages using communication primitives. Variations are possible depending on whether the operations are blocking or non-blocking. A `send` operation is blocking if it does not return until a corresponding `receive` operation is issued. A blocking `receive` operation does not complete until a message arrives.

In [Bal, Steiner and Tanenbaum 1989], processes and point-to-point communication are described as the "basic model". This suggests that the model closely reflects the distributed architecture; it can be implemented efficiently and is widely used for parallel numeric applications.

3.3.2.2 Remote Procedure Call

Remote Procedure Call [Birrel and Nelson 1984] extends the functionality provided by one-to-one messages as they transmit not only data but also a reference to a procedure defined in the interface of the callee. RPC builds on the well-understood notion of a local procedure call. The semantics of a procedure call implies that the caller remains blocked while the callee is executing the procedure. A remote procedure call provides a means of exchanging data and synchronisation between activities and hides the details of the network operation from the application programmer.

RPC is often chosen to implement client/server applications. In that particular context, the caller is referred to as the *client* and the callee as the *server*. A server manages some resources on behalf of its clients. The service is described by an interface.

The RPC model of process interaction restricts the potential for parallelism between clients and servers. More parallelism can be introduced by allowing several threads of execution within the client, that is, procedure invocations can proceed in parallel. At the server level, multiple threads might allow the server to service requests in parallel. Asynchronous RPC has also been investigated [Liskov and Shrira 1988]. Both multithreaded code and asynchronous RPC complicate the programmer's task as they introduce further synchronisation requirements.

3.3.3 Implementations of the process model

The Occam language implements the *Communicating Sequential Processes* (CSP) model [Hoare 1978]. Occam processes are single threaded computations. They communicate by sending messages through *channels*. A channel is the abstraction of an unbuffered, unidirectional data path between two processes. An input operation reads the value available through the channel and an output operation writes a value on the channel. Both operations are blocking, as an input operation must be matched by an output operation. Processes can be composed using the `SEQ`, `PAR` and `ALT` statements. `SEQ` requires the processes to be executed sequentially, `PAR` in parallel and `ALT` provides non-determinism.

```
CHAN OF INT chan3, chan4:
PAR
        INT fred:
        SEQ
                chan3? fred
                fred := fred+1
        INT jim:
        SEQ
                chan4 ? jim
                jim := jim+1
```

In the above example taken from [Pountain and May 1987], two channels transmitting integer values are declared, `chan3` and `chan4`, two identical processes proceed in parallel to read the values from `chan3` and `chan4` and increment them by one.

The conceptual simplicity of the CSP model has also made it attractive as a coordination model to be used with a variety of sequential languages (a list can be found in [Bal, Steiner and Tanenbaum 1989]).

The conjunction of a conventional programming language such as C or FORTRAN with a message passing library such as the Parallel Virtual Machine (PVM) [Sunderam 1990] or the Message Passing Interface (MPI) [MPI forum 1993] provides an evolutionary approach. These libraries abstract away the complexities of the networking operations and present to the user a set of high level communication functions to send and receive data in a machine independent format.

```
/* Sending Process */
initsend();
putstring("The square root of");
putint(2);
putstring("is");
putfloat(1.414)
send("receiver", 4, 99)
```

The example above shows the use of functions of the PVM libraries from within a program written in C. A message is constructed and then sent. The `send` function takes as arguments the (process name, instance) pair (there might be several instances of the same process) and the message type. A message type permits the selective reception of messages.

The Argus language [Liskov 1988] supports RPC as a language construct. RPC, however, is more often implemented as a run-time support infrastructure for existing languages. An RPC mechanism is one of the components of the Open Software Foundation (OSF) Distributed Computing Environment (DCE) [OSF 1992] (see fig. 3.1).

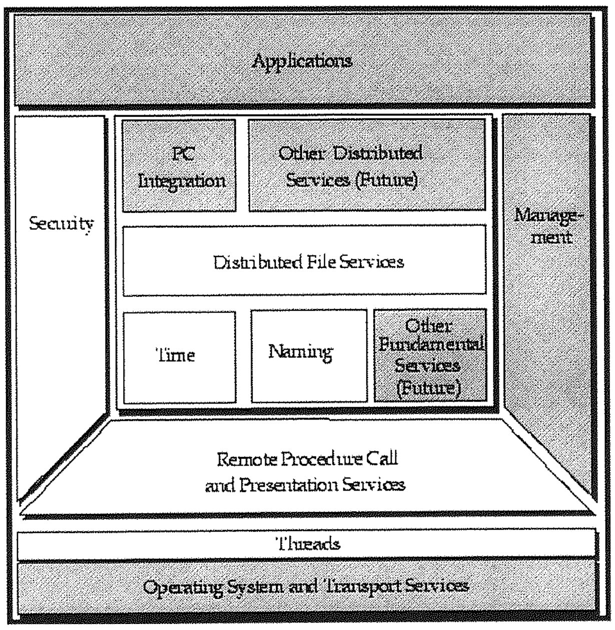

Applications
PC Integration
Other Distributed Services (Future)
Security
Manage-ment
Distributed File Services
Time
Naming
Other Fundamental Services (Future)
Remote Procedure Call and Presentation Services
Threads
Operating System and Transport Services

Fig. 3.1: The architecture of the Distributed Computing Environment

The interface between a callee and the callers is described using a special purpose *Interface Definition Language*. The following example is taken from [Shirley, Hu and Magid 1994]:

```
[
uuid(40554daa-6b3b-11cf-8a42-08002be7a203),
version(1.0)
] interface arithmetic

{
      const unsigned short ARRAY_SIZE = 10;
      typedef long long_array[ARRAY_SIZE];
      void sum_arrays(
            [in] long_array    a,
            [in] long_array    b,
            [out] long_array   c);
      int sum_ints([in] int a, [in] int b));
}
```

The syntax of the DCE IDL is very close to that of ANSI C. The interface is identified by a *Unique Universal Identifier* (*UUID*), its version number and its name. This interface defines a constant, an array datatype: `long_array` and two operations, `sum_arrays` and `sum_ints`, which compute the sum of two arrays or of two integers. The interface is compiled by the interface compiler which generates a skeleton for the implementation of the server and the low-level distribution code which is transparent for the user. A binding mechanism, *Cell Directory Service* (*CDS*) in DCE terminology, allows servers to export their reference and client applications to acquire the reference of the servers they need to invoke.

3.4 Distributed Objects

3.4.1 The basic object model

An object encapsulates both state (the attribute variables of the object) and behaviour (the methods). Objects are instances of classes. Classes serve as templates that define the implementation of their instances. Inheritance is a mechanism whereby a subclass inherits the behaviour and the structure of its superclasses. The primary goal of inheritance is code reuse and sharing. Objects communicate exclusively by message passing. An object invokes another object by sending it a message. When the object receives a message, it determines whether it has a method to respond to the message. The matching at run-time of an operation name and of the corresponding method is called dynamic binding. The semantics of object invocation is that the calling object remains blocked until the invoked object returns.

3.4.2 Objects and distribution

The message passing mode of interaction between objects extends naturally to distributed programming. Data abstraction is of obvious benefit to distributed programming because it reduces coupling between the different parts of a distributed application. Other features of the original object model require special consideration for distribution.

Inheritance can be difficult to implement in a distributed environment. Bennett [Bennett 1987] writes: "A major disadvantage of inheritance is the potentially awkward separation of object behaviour and state". The cost of method despatching appears too high. Some models do not support it. Prototype-based languages [Lieberman 1986] constitute another variation from the original object model. These languages do not have classes from which objects can be instantiated, rather objects are created by *cloning* a *prototype* object. Code sharing and reuse is achieved through *delegation*. Objects can delegate to one or more ancestors the responsibility for performing an operation or keeping part of its state.

Object models have different approaches to typing. Dynamically typed languages give greater flexibility to the programmer. However type errors can be discovered only at run-time and this might prove particularly unacceptable in a distributed system. Static typing stresses tighter typing discipline since type errors are detected at compile time.

Object models also vary in the way objects and activities are related, two approaches can be distinguished

- In the *active object model,* activity is associated with the object. Parallelism in the active object model results from the instantiation of several active objects. Intra-object parallelism is possible by allowing several activities to execute in an object.
- In the *passive object model,* objects and (potentially multithreaded) processes are distinct entities, the process being responsible for executing methods of passive objects. Processes are units of parallel execution in the passive object model.

In all models, message passing is the means for objects to communicate whether they be in the same address space or separate address spaces. Synchronous message passing also provides inter-object synchronisation, although some models allow objects to send messages asynchronously in order to increase concurrency.

Several mechanisms are available for the internal synchronisation of objects. For example synchronisation variables such as mutexes can be part of the internal state of the object or monitor constructs may be available for object operations.

The requirements of parallel computing led to the development of another variation of the basic object model, the Actor model [Agha 1990]. Actors encapsulate state and behaviour as well as activity. Actors communicate by asynchronously sending messages to other Actor's mailboxes. A mailbox name uniquely identifies the actor to which the mailbox belongs and it can also be transmitted in a message. The mailbox queues the messages for its actor and the messages invoke the actor's behaviour. A behaviour may spawn new actors with new mailboxes or a successor actor with the same mailbox.

Client/server applications are often deployed in heterogeneous environments and the good abstraction capabilities of the object model have led to its widespread acceptance for this class of applications. However, the use of object-oriented languages for parallel programming has also been the subject of some interest.

3.4.3 Distributed Object Systems

The systems presented in this subsection illustrate the various design approaches mentioned in the previous subsection.

Smalltalk is seen as the archetypal object-oriented language. [Bennett 1987] describes a possible way of distributing the language. The aim of the project was to retain as much as possible of the original object model, hence the semantics of Distributed Smalltalk is the same as that of the sequential language. The implementation provides a message forwarding and reply service to remote objects. The Smalltalk language and programming environment offer a high degree of reflection and interactivity. Reflection means that the representation and execution characteristics of the language are exposed using the language constructs; these characteristics can be easily customised. Distributed Smalltalk preserves this design philosophy; Distributed Smalltalk is largely implemented in Smalltalk and the error reporting and analysis facilities remain available with distributed computation.

The Obliq language [Cardelli 1995] is an interpreted prototype-based language designed for distributed programming. Obliq's object model does not use delegation on the grounds that the sharing resulting from delegation causes implementation difficulties in a distributed context. Obliq objects are self-contained. The attributes and methods of an object are embedded into it. The fragment of code below shows the definition of an object:

```
let o =
        { x => 3,
        inc => meth(s, y) s.x := s.x+y; s end,
        next => meth(s) s.inc(1).x end};
```

An object `o` is defined with an attribute `x` and two methods `inc` and `next`. Code reuse is achieved by the cloning operation. The expression:

```
clone(a1, ..., an)
```

creates a new object with all the attributes and methods of objects a_1 to a_n. Distribution is not transparent in Obliq. A *site* is the abstraction of an execution context and it is designated by a name. The execution of a procedure can be explicitly mapped to a site by the programmer. Obliq supports lexical binding with identifiers retaining the value to which they were bound at their first occurrence. In Cardelli's view, lexical scoping prevents the unpredictable results caused by dynamic scoping in a distributed environment.

The following example (adapted from Cardelli's paper) explains how distributed lexical scoping works. The server site registers itself with the name server (called *Namer*) under the name of `ComputeServer` and exports an `exec` procedure which, when invoked, executes the closure it receives as argument:

```
net_export("ComputeServer", Namer, {exec => meth(s, p) p() end})
```

At the client site, the `ComputeServer` object is obtained from the name server *Namer*, bound to an identifier `computeServer` and its `exec` method is invoked:

```
let computeServer =
      net_import("ComputeServer", Namer);
var x = 0;
computeServer.exec(proc() x:=x+1 end);
```

As the result of the invocation, the value of x at the client site is 1, even if a variable x with another value was defined at the server site. Type checking in Obliq is dynamic.

The designers of the Emerald language [Black et al. 1986] have chosen an object model which does not support inheritance. They have introduced a strong typing discipline based on the notion of subtyping. Emerald objects can have a process attached to them. A monitor construct is available for the object methods in order to synchronise object invocations. Objects and their activity can be migrated in Emerald but migration is not completely transparent to the user. A small set of language primitives allows the programmer to control mobility:

- `locate` returns the node where an object resides.
- `move` relocates an object to another site.
- `fix` and `unfix` respectively disable and re-enable object mobility
- `refix` performs the sequence `unfix`, `move` and `fix` in an atomic fashion.

An `attached` annotation for the object attribute declaration allows the programmer to describe how objects should be migrated. The values of all the attached attributes of an object are automatically migrated with the object.

UC++ [Winder, Wei and Roberts 1992] is a parallel object-oriented programming language based on C++. UC++ relies on active objects to express the potential for parallel execution. An `active` keyword allows the programmer to make the instance of any class an active object. An active object is executed by a virtual processor. Each virtual processor supports only one object. Virtual processors are designated by integer values and the programmer can optionally specify to which processor a new active object should be allocated, using the keyword `on`. The fragment of code taken from [Winder, Wei and Roberts 1992] shows the creation of an active instance of the class `PrimeFilter` on the virtual processor `0`:

```
active PrimeFilter two (2, outputObject) on 0;
```

The mapping of virtual processors onto physical processors can be done automatically by the run-time system or it can be described in a *loading file*. The `split` keyword allows the invocation of the methods of active objects asynchronously or to spawn a thread within a method of an active object.

Guide [Balter, Lacourte and Riveill 1994] supports inheritance and makes a clear distinction between subtyping and subclassing. Types are concerned with the interface and classes with the implementation. The Guide type system will be described in greater detail in chapter 5. Guide is based on the passive execution model where objects and processes are orthogonal. Synchronisation in Guide takes the form of activation conditions attached to methods. For each method the following set of counters is defined:

- `invoked (m)` : number of invocations of method m
- `started (m)` : number of accepted invocations for m
- `completed (m)`: number of completed executions of m
- `current(m) = started(m) - completed(m)`
- `pending (m) = invoked(m) - started(m)`.

The activation conditions of a method are expressed as conditions which the method counters must satisfy in order for the method invocation to proceed. The following conditions are defined for the activation of the `Put` and `Get` methods of a `FixedSizeBuffer` object.

```
Put: (completed(Put)-completed(Get) <size) AND current(Put)=0;
Get: (completed(Put)>completed(Get)) AND current(Get)= 0;
```

The conditions state that:

- `Put` may proceed if the number of items in the buffer is smaller than the maximum size of the buffer and no invocation of `Put` is currently proceeding.
- `Get` may proceed if there is at least one item in the buffer and no other invocation of `Get` is currently proceeding.

As with other models, considerable effort has been put into offering an evolutionary path to distributed objects. An Object Request Broker (ORB) forwards object invocation across separate object contexts. The Object Management Group (OMG) has worked on the standardisation of a Common Object Request Broker Architecture (CORBA) [OMG 1996] (see fig. 3.2). CORBA defines not only the architecture of a request broker but also a series of associated services such as naming, persistence and transactions.

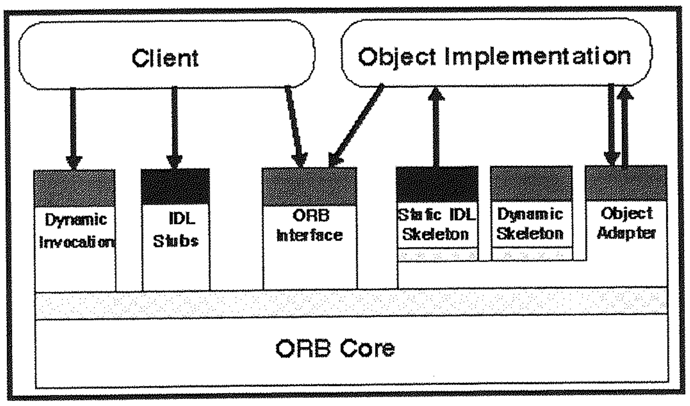


Fig. 3.2 : Architecture of an Object Request Broker.

The interfaces of these services and that of the application objects are described using the CORBA Interface Definition Language. The operations defined in the interface of an object can be invoked by clients or alternatively the invocation can be constructed through a mechanism called dynamic invocation. The extra complexity incurred by dynamic invocation is justified in cases where the invocation parameters (message selector or the arguments of the invocation) cannot be known at compile time. Application objects can be implemented using a variety of languages, including non object-oriented ones.

3.5 Functional parallelism

3.5.1 Functional languages

Functional languages along with logic languages are said to be *declarative,* in contrast to imperative languages. Declarative languages let the programmer concentrate on the description of a solution to a problem and do not require the programmer to describe how the computation should be sequenced or how memory should be managed.

With the functional model, the function is the abstraction of a computation and programs are built from function definitions and function applications. Functions, in that context, are pure mathematical functions. Program state is passed as an argument to the function and returned by it. State is not modified as a side-effect of the computation.

Hudak [Hudak 1989] highlights some of the features common to most functional languages:

- Good support for data abstraction.
- Functions are first class values.
- Use of recursion for looping.
- Equational feel and pattern matching.

The use of pattern matching and of recursion is illustrated by the definition of factorial in SML:

```
fun factorial 0 = 1
| factorial x = x * factorial (x-1);
```

However, some features may vary considerably from one functional language to another. For example, efficiency concerns have motivated the addition of a limited number of imperative features to some functional languages, called *impure* functional languages. Functional languages may also differ in the way functions are evaluated. Two strategies can be distinguished: *eager* and *lazy* evaluation. With eager evaluation, all the arguments of a function must be evaluated before the function can be applied to them. With lazy evaluation, an argument is evaluated only if its value is needed to compute the value of the function.

Functional languages offer a high level of abstraction from implementation details and they appear both more concise and more expressive than their imperative counterparts. Also, their clear semantics make them amenable to formal analysis. Functional languages also appear to be well suited for parallelism because the freedom from side-effects ensures, that, in the absence of data dependencies, functions can be evaluated in parallel.

3.5.2 Parallel functional models

Steele [Steele 1995] lists several ideas for the development of a parallel version of LISP and, by extension, of a parallel functional language. The last idea is mentioned in [Hammond 1994]:

- Completely independent processes: A computation is described as a set of processes implemented using a sequential functional language and the processes communicate using some communication facilities such as channels.

• Processes in a shared address space: A primitive initiates the evaluation of a piece of code by a new process. Processes execute concurrently, access and modify the data available in a (virtually) shared address space. Communication and synchronisation are implicit.

• Futures: The *future* construct [Kranz, Halstead and Mohr 1989] spawns the evaluation of a LISP expression in parallel. The computation which spawned the future need not wait for the return of the value and receives a future data object which acts as a place holder for the value to return and is in an *unresolved* state until the value becomes available. If the value of the expression is needed and the future is still unresolved, the evaluation can be forced by *touching* the future, thus forcing the two tasks to synchronise.

• Parallel evaluation of arguments. The *pcall* (parallel call) form [Halstead 1984] spawns different tasks to evaluate in parallel the different arguments of a function before applying the function to them.

• The data-parallel model: The application of the same functions to all the elements of a large regular data structure can be performed in parallel [Steele and Hillis 1986]. This leads to fine-grain parallelism which is best supported by tightly coupled parallel architectures, particularly SIMD (single instruction multiple data) architectures.

• Purely functional model: In the idealised functional model [Goldberg and Hudak 1986], no language constructs are necessary to indicate parallelism. Communication and synchronisation remain also implicit for the programmer. This idealised model might lead to a fine grain parallelism which cannot be exploited efficiently. Annotations have been introduced to control parallelism, evaluation order and mapping of expression evaluations to processors [Hudak 1986].

• Algorithmic skeletons [Cole 1989] offer a high level view of the program structures of a parallel computation. Skeletons capture patterns of parallelism common to classes of applications; one example of such a pattern is pipelining.

3.5.3 Case studies

The languages presented below exemplify some of the approaches discussed in the previous subsection.

Facile [Giacalone, Mishra and Prasad 1989] combines the SML functional language with a coordination model based on processes and channels. Processes in Facile are SML computations which communicate and synchronise themselves using a channel facility. The example below shows the creation of channels in Facile. Firstly, the user calls the function `channel` and gives the type of the channel.

```
- val ch1 = channel () : int channel;
val ch1 = channel : int channel
```

Then processes communicate using the `send` and `receive` functions:

```
val send: 'a channel * 'a -> unit
val receive: 'a channel -> 'a
- send(ch1,7);
val it = () : unit
- receive ch2;
val it = 8 : int
```

ICSLA [Queinnec and DeRoure 1992], a LISP family language, structures parallel computations as a collection of processes in a virtually shared address space. ICSLA expresses concurrency with a *breed* function.

(breed [*thunk* ...])

`breed` takes an arbitrary number of thunks (a thunk is a function with no arguments) as its arguments and replaces the current task with the necessary number of tasks to evaluate the thunks. The `remote` and `placed-remote` functions distribute data and tasks and `placed-remote` specifies the processor where the task or data should be sent.

The Connection Machine LISP [Steele and Hillis 1986] extends LISP with some language constructs tailored to exploit data parallelism inherent in the Connection Machine SIMD architecture. A *xapping* data structure is a combination of an array and a hash table. Operations on the entries of the xapping can be carried out in parallel. The following xapping maps symbols to other symbols:

```
{blue → sky red → apple green → grass}
```

The α notation indicates that a function must be performed on all the elements of a xapping in parallel. For example:

```
αcons '{ blue → sky red → apple green → grass} '{ blue → sea red →
wine green → emerald}
```

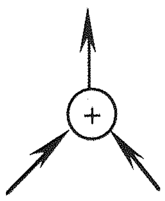
returns:

```
{ blue → (sky, sea) red → (apple, wine) green → (grass, emerald)}
```

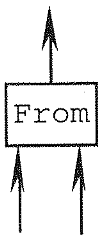
Alfalfa [Goldberg and Hudak 1986] is an implementation for distributed memory multiprocessor of the Alfl functional language [Hudak 1984]. Alfl contains no explicit construct for expressing parallelism and thus Alfalfa can be seen as an example of a purely functional parallel system. The implementation of Alfalfa is based on the graph reduction model. The graph reduction model represents a computation as a directed graph of nodes.

[Fasel and Keller 1986] define a notation in their introduction to graph reduction. Other notations have been proposed (one example can be found in [Field and Harrison 1988]) but the intuitive nature of Fasel and Keller's notation makes it well suited for the purpose of the explanation that follows. In this notation, a node can be:

- A value which is a leaf node of the graph.
- A primitive operation, represented by an oval node e.g.



- A function application, represented by a rectangle node. The example shown below is a function of two arguments.



The computation is driven by the *reduction* of the graph, that is by replacing the evaluable nodes by their value.

The reduction of a primitive node replaces the node by its value. The reduction of a function application requires the expansion of the function application node into the graph defining the behaviour of the function. The graph can then be reduced.

The computation is demand-driven, that is a node is not evaluated until its value is needed. This property is illustrated in [Fasel and Keller 1986] with the `from` function which generates an infinite list of integer values starting with the value of its argument `n` (see fig. 3.3):

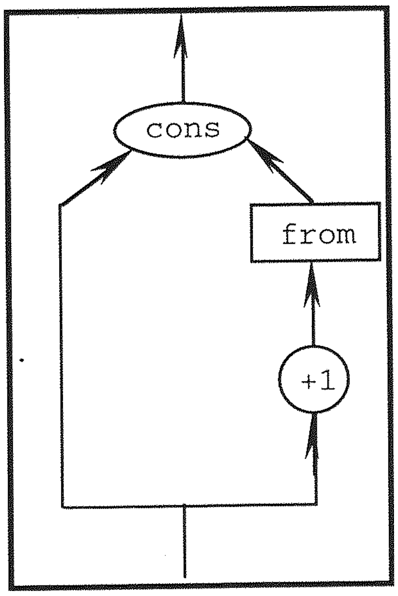


Fig. 3.3: The `from` function

The `cons` operator does not require the evaluation of its second argument and `from` will be evaluated again when the value `n+1` (that is the second element of the list) is accessed

The interest of the graph reduction model is that nodes may be reduced independently and potentially in parallel. The example shown in fig. 3.4 taken from the introduction of [Fasel and Keller 1986] highlights the potential for parallelism in the evaluation of the expression:

```
(a+b)/(c×d) - (c×d)/(e+f)
```

In fig. 3.4, the three nodes in grey could be evaluated in parallel, the two divide nodes could also be evaluated in parallel at a later stage during the computation.

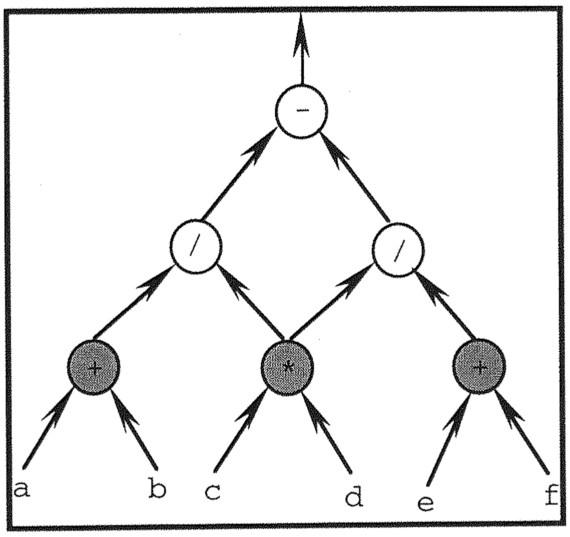


Fig. 3.4: Graph representation of a functional expression

ParAlfl [Hudak 1986] is another development based on the Alfl language. ParAlfl provides a notation so that the programmer can control the mapping of a program onto a target machine architecture. The processors of the target machine are designed by their PID, which is an integer value. The evaluation of an expression can be mapped to a processor explicitly. The code:

```
(f(x) $on 0)+(g(y) $on 1)
```

maps the evaluation of `f(x)` on processor 0 and `g(y)` on processor 1. `$self` is bound to the PID of the currently evaluating processor. It is possible to designate processors relatively to the current processor. For example:

```
(f(x) $on left($self))+(g(y) $on right($self))
```

maps the evaluation of `f(x)` to the processor on the left of the current processor and the evaluation of `g(y)` to the processor on the right of the current processor (the meaning of left and right depends on the topology of the target machine). Alfl has a lazy evaluation strategy, however, a special notation, #, allows the programmer to force the evaluation of an expression. For example:

```
f(x,#y,z)
```

forces the evaluation of `y` in parallel with the evaluation of `f`.

Algorithmic skeletons outline a pattern of parallelism without knowledge of the individual tasks to be carried out in parallel. The Structured Coordination Language (SCL) [Darlington et al. 1995] is a functional language for composing procedures written in a sequential language (e.g. FORTRAN). SCL provides constructs to specify partitioning, data movement and control flow. Data parallelism constitutes the underlying model for SCL. The built-in α distributed array type allows operations on its elements to be carried out in parallel. The following example is taken from [Darlington et al. 1995]:

```
rotate :: Int → ParArray Int α → ParArray Int α
```

The function `rotate` takes as arguments an integer value (called the distance of rotation), a one dimensional parallel array whose indices are of integer type and elements of type α and returns a similar array. The code for `rotate`:

```
rotate k A=<<i:=A((i+k) mod SIZE(A)) | i ← [1 .. SIZE(A)]>>
```

In [Rabhi 1993], Haskell serves as both the base and the coordination language. Higher-order functions control the application of supplied functions following a pre-established pattern. The higher-order function `rp$` captures the pattern of recursively partitioned algorithms:

```
rp$ ind solve divide combine prob
        | ind prob = solve prob
        | otherwise = combine prob
                              (map (rp$ ind solve divide combine)
                                    (divide prob))
```

with:

- `ind prob` is a predicate which returns `TRUE` if the problem `prob` is indivisible.
- `solve prob` is the function which solves an indivisible instance of the problem `prob`.
- `divide prob` partitions the problem `prob` into subproblems
- `combine prob sols` combines the solutions `sols`.

3.6 Dataflow model

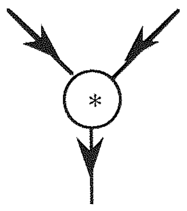
3.6.1 Dataflow computations

The dataflow model promotes a view of computation even further removed from control flow languages than graph reduction. The dataflow model has a simplified and quite straightforward evaluation regime, and execution is driven solely by the availability of data.

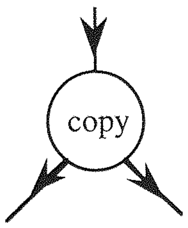
A dataflow computation can be represented by a dataflow graph. The graph comprises nodes called operations and arcs representing the flow of data between operations. The notion of variable does not exist in the dataflow model. The values are anonymous and side-effects do not exist in the dataflow model.

The dataflow model presented in [Glaser, Hankin and Till 1984] distinguishes six types of nodes and a notation for function definition.

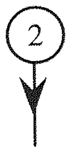
- A `primitive` node applies the operation to its argument.



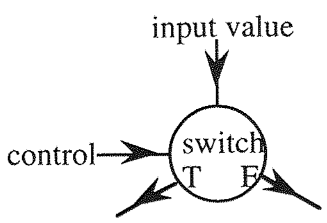
- The `copy` node duplicates its incoming argument onto two or more outgoing arcs.



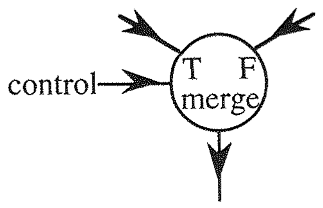
- The `value` node has no input value and outputs a constant value, a primitive or a user-defined function represented as a closure.



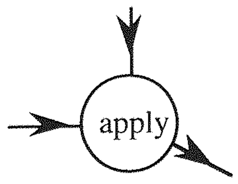
- The `switch` node controls the flow of data according to the boolean value flowing into its control input.



• The `merge` node selects which input value is returned as output value.



• The `apply` node applies the function which is passed to it on its leftmost input to the argument on the second input (functions of more than one argument are partially applied).



The example shown in fig. 3.5 computes the roots of a quadratic equation of the form ax^2+bx+c.

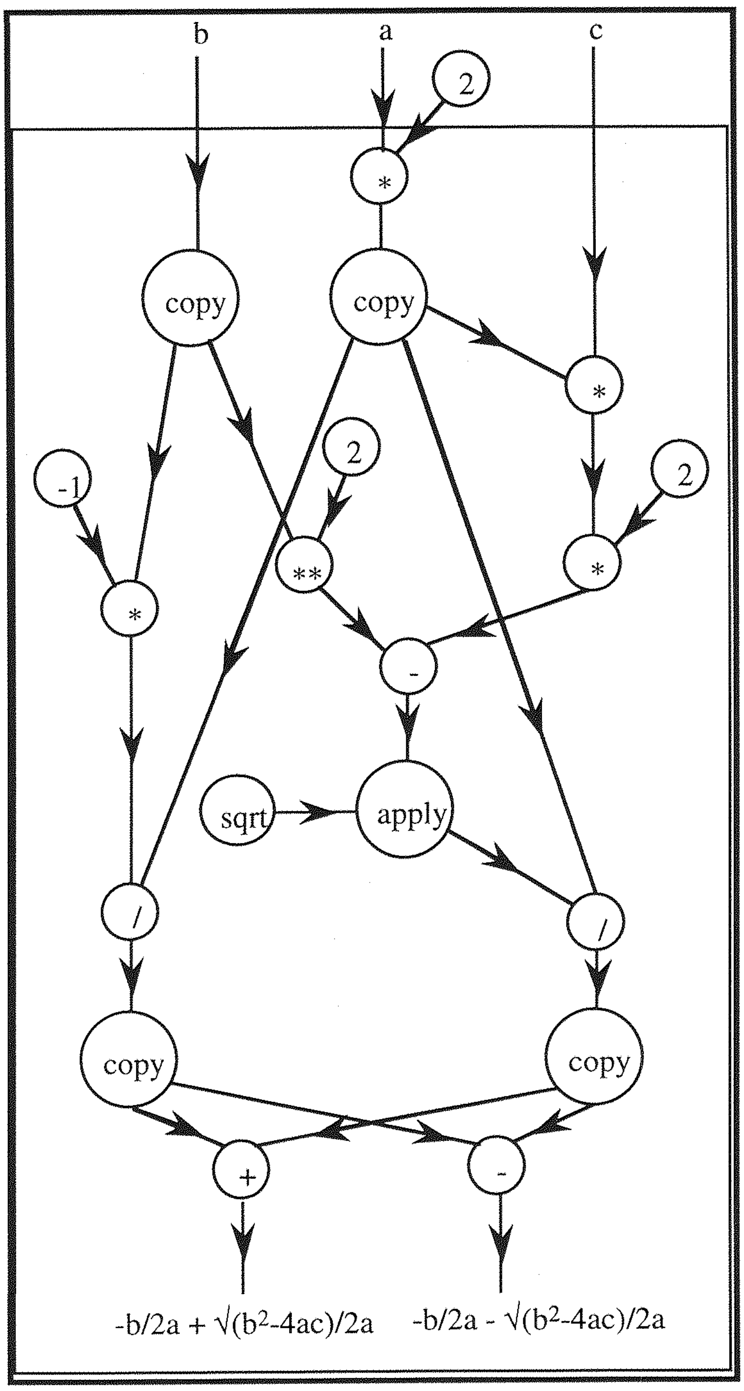


Fig. 3.5: Data flow graph for the computation of the quadratic roots.

The interest of the dataflow model is that it is inherently parallel. Operations can be seen as fine grain units of parallelism; the execution sequence is constrained only by the partial order defined by the data dependencies. Communication and synchronisation are implicit.

3.6.2 Dataflow languages

Several dataflow languages have been developed, Id Nouveau [Nikhil, Pengali and Arvind 1986] is a recent example. In order to increase their expressive power, these languages extend the pure dataflow model with procedural and data abstractions, conditional choice and iterations. The fine grain parallelism inherent in the dataflow model requires tightly coupled parallel machines or specialised hardware to be exploited successfully. Several dataflow machine architectures were investigated in the 1970s and a survey of dataflow machine architectures can be found in [Treleaven, Brownbridge and Hopkins 1982]. However, results have proved disappointing and recent research work focuses on hybrid dataflow/Von Neuman architectures [Lee and Hurson 1994].

SISAL (Streams and Iterations in a Single Assignment Language) [Feo, Cann and Oldehoeft 1990] is considered a dataflow language. Although it supports variables in a limited form, variables can be assigned only once. SISAL is mainly targeted at numerical applications.

SISAL functions may take several arguments and return several values. The control structures of the language `if…then…else` and the `for` loop are higher order functions. SISAL provides a built-in array type and supports user-defined types, including records. The statement below defines an array of integer type:

```
type One_Dim_I = array [integer];
```

The record type `element_record` is used for the periodic classification of chemical elements:

```
type element_record = record [name : array [character];
        number : integer;
        weight : real]
```

The following code computes the roots of a quadratic but it is not equivalent to the program described by the graph in fig. 3.5 as the function shown below tests the sign of the value of the discriminant.

```
function quad.roots( a, b, c : real returns real, real )
        let
                denom = 2.0 * a;
```

```
            discrim := b * b - 4.0 * a * c;
        in
            if discrim >= 0.0
                then
                    -b + sqrt( discrim ) / denom,
                    -b - sqrt( discrim ) / denom
                else
                    -9.99e99, -9.99e99
            end if
        end let
    end function
```

SISAL has been implemented on a variety of shared-memory multiprocessors with some success [Cann 1992]. A SISAL compiler for distributed memory parallel machines and a network of workstations is described in [Freeh and Andrews 1995].

Dataflow is seen as a possible glue to describe the communication and synchronisation patterns between coarse grain activities. Such an approach is called Large Grain Dataflow (LGDF) [Babb 1984]. LGDF is a programming methodology for the development of parallel programs. The methodology goes over successive steps to convert a dataflow graph into a program written in an imperative language.

Several parallel programming environments use visual dataflow as their coordination language with nodes corresponding to sequential computations and the links of the graph representing dependencies between the activities. [Browne et al. 1994] review the benefits of representing parallel programs visually:

- The visual representation exposes large scale program structures and allows a natural representation of parallelism in programs.
- This representation enforces good programming practices as programming in the large and programming in the small become distinct concerns.
- Debugging can be carried out in the same framework as programming.

Parallex [Alvisi et al. 1992] is a programming environment for parallel scientific computing in a distributed system. A Parallex computation is described by a dataflow graph whose nodes are coarse grain C or FORTRAN computations. There exists no shared data between the nodes.

3.7 A model for Distributed Prograph

This last section discusses the choice of a model for Distributed Prograph. The model should take into account both the class of applications intended for Distributed Prograph and the features of the current version of the language. It should also provide the right abstractions to handle parallelism, communication and synchronisation.

Distributed Prograph targets parallel programming in a distributed system. The overall goal of parallel programming is to speed-up the execution of an application through the use of multiple computing resources.

Program sequencing is based on the dataflow model. In the sequential version of the language, operations are triggered sequentially. The schedule follows the partial order defined by the data dependencies. Synchronisation links and the controls attached to the operations also affect the flow of control (see 2.2.1).

3.7.1 Parallelism

3.7.1.1 Potential for parallelism

Without adding further abstractions to the existing model, parallelism can be achieved in two different ways: data parallelism and operation level parallelism:

- Processing of lists (which are a Prograph built-in type) offers some potential for data parallelism. This form of parallelism can be conveniently called *multiplex parallelism.*
- Following the dataflow model, operations with no data dependencies are obvious candidates for parallelism and could be fired in parallel. This latter form can be called *operation parallelism.*

It appears that other forms of parallelism would be less easy to introduce. The conjunction of Prograph with a CSP like model would require the addition of a channel abstraction. Parallelism such as promoted by the RPC or Distributed Objects model would impose a more extensive use of names than in the current version of the language and the introduction of new abstractions such as remote server or remote object.

3.7.1.2 Expressing the parallelism

Another question which arises is how to express parallelism. Fig. 3.6 lists the possible alternatives:

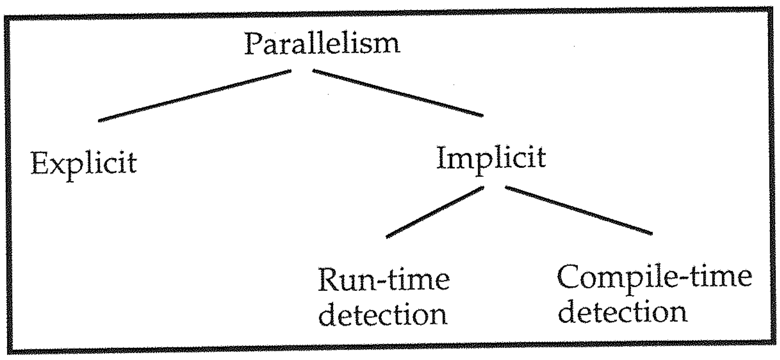


Fig. 3.6: Expression of parallelism

With implicit parallelism:

- parallelism can be detected at run-time.
- parallelism can be detected at compile-time by some automatic parallelisation tools. The object code is then split into a set of subcomputations that can be executed in parallel.

The first approach is that taken in the design of hardware dataflow machines and can be emulated by software. For example the run-time support of the Strand_88 language [Foster and Taylor 1990], the Strand Abstract Machine, (SAM) mimics the behaviour of a dataflow machine and schedules functions for evaluation based on the data dependencies. However, this approach leads to fine grain parallelism; the overhead induced by run-time scheduling and the cost of communication leads to a highly inefficient implementation on a loosely coupled architecture such as that intended for Distributed Prograph.

Automatic parallelisation has been the subject of much research work and has been applied to both imperative languages such as FORTRAN and functional languages. This option also does not appear viable for Distributed Prograph as both the features of the language and the structure of applications would make the automatic parallelisation of Prograph programs difficult. Prograph provides dynamic binding and Prograph applications are event-driven. Consequently, the execution path of a Prograph application is more complex to build than that of control-driven scientific applications to which automatic parallelisation techniques are often applied.

Explicit parallelism requires the programmer to explicitly indicate which portions of the code should be executed in parallel. This is the option retained for Distributed Prograph. The user has to annotate an operation to indicate that it is a candidate for distribution at run-time.

3.7.1.3 Benefits and drawbacks

This approach offers several benefits:

- The idea of an operation annotation is already familiar to the Prograph programmer (e.g. multiplex annotation) and should limit the amount of recoding necessary to distribute existing Prograph code.
- It builds on the recognised strength of visual languages to grasp and express the potential for parallelism.
- The user's appreciation of issues such as granularity, cost of communication and side-effect is required to distribute Prograph operations. Static analysis tools should assist the programmer in those decisions.

The idea of introducing a notation to indicate parallelism may bear some resemblance to the concept of future. The benefit of using futures is that parallelism is not restricted by prematurely blocking a computation by waiting for the value of the future. The drawback is that it introduces a new abstraction the programmer has to deal with and that, for efficiency reasons, futures should probably be implemented as a built-in datatype. Another difficulty is the use of futures in presence of controls, if the computation proceeds without waiting for the results of an operation with a control, the activation of the control may result in wasted computation.

The designers of the Prograph language have consciously hidden pointers and operations on pointers from the programmer's view to mimic the data-driven nature of a pure dataflow language. Introducing futures would twist the design philosophy of Prograph and it appears preferable to retain the original data-driven semantics of the language at the expense of a gain in parallelism.

The drawbacks of this approach are:

- It leads to a specialised model which can only support parallel applications. Linda according to [Gelernter and Carriero 1992] is an example of a more universal model, Linda can support message passing between sibling computations, task pool model of parallelism and RPC-like interaction between computations.
- Programmers have no control over the allocation of operations annotated for distribution to remote processors. Precious programmer insight might be lost and this would result in a less efficient execution of applications.

3.7.2 Communication and synchronisation

Communication related activities should be carried on completely transparently to the user.

The current version of Prograph provides a synchronisation facility in the form of synchros; otherwise synchronisation should only be constrained by data dependencies.

3.7.3 A metaphor

The Distributed Prograph model can be better understood with the metaphor of a dataflow machine. In such a machine, parallelism is achieved by having several functional units executing operations simultaneously. Information items appear as operation packets and data tokens. Under the control of a sequencing unit, operation packets are sent to the functional units. An operation packet consists of an operation code and the operands; the operations results are returned as data tokens to the sequencing unit. The Prograph run-time acts as a virtual dataflow machine with a single sequencing unit and a single control unit. A network of workstations can be seen as a machine with multiple functional units. Operation packets and data tokens flow to and from the functional units under the control of the sequencing unit.

3.8 Summary

- Distributed systems encompass a broad range of applications and hardware platforms.
- Various distributed programming models provide the necessary abstractions to deal with the requirements of different classes of distributed applications (i.e. parallel, fault-tolerant, client/server and wide-area distributed applications).
- With Distributed Prograph, the interest lies in obtaining speed-ups through parallel execution.
- After reviewing several models, dataflow appears to be the most straightforward way to extend the Prograph sequential model.
- The Distributed Prograph model requires the programmer to annotate the operations for distribution. Communication is transparent and synchronisation constructs are already available in the sequential version of the language.

4 Prograph and Distribution

The previous chapter concluded that the dataflow model should be used to extend the Prograph sequential model into a distributed one. In its first section, this chapter discusses a range of issues to be considered in the design of Distributed Prograph. The second section is concerned with the implementation of the language. The third section of the chapter justifies the need for a static analysis for Distributed Prograph and the last section sets the objectives of this analysis.

4.1 Design issues

Design issues look at a high-level, functional view of the policies and mechanisms for distribution. The object of this section is not to commit Distributed Prograph to a set of policies or mechanisms but rather to review the range of policies available for its design. The options available will be better understood if they are presented in the context of the Distributed Prograph model presented in 3.7.3.

Distributed Prograph aims to execute a single application on a set of machines. The application is initiated on the user machine. The execution context on the user machine is called the *originator context*. Operations annotated for distribution can then be sent to a remote processor for execution. The execution context at the remote processor is called the *recipient context*. The user machine acts as the sequencing and update unit. It is responsible for distributing operations initially and all results are ultimately returned to it.

The various activities involved in the execution of a remote operation fall into four broad categories:

- Preparation of an operation packet in the originator context.
- Transmission of the packet from the originator context into a recipient context after selecting a remote processor.
- Execution of an operation in the recipient context.
- Return of the results to the originator context.

Various mechanisms must be designed to support the execution of these activities. The following subsections discuss possible solutions in close relation with the features these mechanisms must provide.

4.1.1 Operation packet

The operation packet is the information sent for execution in a recipient context. This information includes at least the name of the operation along with its arguments.

Two approaches are possible for the transmission of argument values: *proxies* [Decouchant 1986] (see fig. 4.1) and replication.

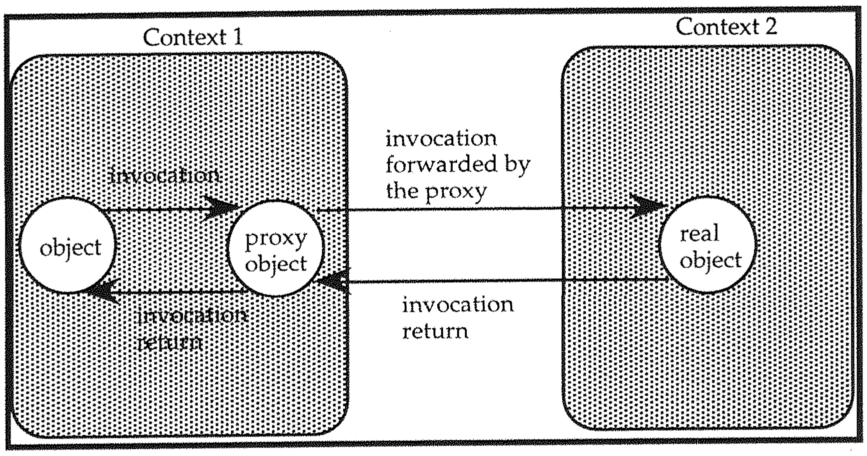


Fig. 4.1: Proxy objects

A *proxy* object acts as a *surrogate* for an object residing in another execution context and invocations on the proxy are trapped and are forwarded across contexts to the remote object. Distribution requires that objects are not only identified in a local context but also across several contexts. A new naming scheme must be introduced in order to generate global identifiers. With modern object-oriented languages, parameters and return values are often passed by reference. Proxies denote remote references and thus logically extend the pass-by-reference mechanism to distributed environments. The possibility to alias objects is preserved. However, forwarding invocations can prove expensive. To remedy this, several optimisations have been proposed. Passing arguments by value may provide better performance. Immutable values, such as integers or booleans, can be passed by value. In [Dollimore, Miranda and Xu 1991] it is also suggested that objects which are not aliased can also be passed by value.

An alternative solution is replication. All objects are replicated, packed and transmitted with the operation packet. Replication reduces the load on the remote objects and increases concurrency as distributed operations proceed with their own copies of the objects taking part in the computation. However, the speed-up gained from replication may be partially or totally offset by the cost of maintaining consistency between replicas of the same object. Schemes for the management of replicated data can provide various levels of consistency between replicas.

Pessimistic replication schemes supporting full consistency can be achieved by serialising update operations on all the replicas. The GARF system [Garbinato, Guerraoui and Mazouni 1994] uses a multicast protocol to update replicas. Full consistency results in significant communication costs. The purpose of replication in such a context is to provide fault-tolerance rather than to increase performance. Weaker models of consistency have been studied. DMEROON [Queinnec 1995] supports causal consistency. In DMEROON, values can be cached for read operations. Coherency is monitored by a clock-based algorithm. Write operations are always performed on the original object and a clock records the number of modifications that occurred to the object monitored. When a cached object is accessed, the value of its clock is compared with that of the clock of the original object; different clock values mean that the cached value is invalid. *Optimistic* replication management schemes allow both read and write operations on replicas. The values of the replicas can then be *reconciled* at a later stage or inconsistencies are not important; such a decision depends on the semantics of the application.

POOM [Kristensen and Low 1995] allows the programmer to specify the consistency required and the mechanism to manage replicated data.

In Prograph, persistents and class variables can be seen as global variables. Their value need not be passed as arguments to a method to be accessible in the cases of the method called by an operation.

Obliq [Cardelli 1995] supports *lexical scoping* such that a global variable retains the value to which it was bound in its original context. The implementation of lexical scoping relies on global identifiers. The free variables occurring in an object method are described by global identifiers and their values can be obtained from the context in which they were created.

The distributed version of Smalltalk developed by [Schelvis and Bledoeg 1988] distinguishes between two types of global variables. *Home* variables are those whose values are only relevant in their local context (variables describing the user interface are an example of home variables). The other variables have values which must be consistent across several execution contexts. Thus both lexical and dynamic scoping mechanisms can coexist. This does however require the programmer to declare which variables should be dynamically scoped and which ones are lexically scoped.

The SOS distributed operating system [Shapiro, Gautron and Mosseri 1989] provides a migration mechanism for C++ objects. Migratable objects must be instances of subclasses of the `sosObject` class. The `sosObject` class defines the migration behaviour of all migratable objects. Further, the state of sosObjects contains a set of *prerequisite* objects. The prerequisite objects contain the required information to reconstruct a migrated object in a new context. One example of prerequisite is the code of the class to which the migrated object belongs.

4.1.2 Operation scheduling

In Distributed Prograph, once the operation packet has been readied, it is pooled to be exported for remote execution. In the absence of any mapping annotation from the programmer, the allocation of a resource to execute an operation is left to the language run-time support.

Two strategies are possible to export pooled operations. With *work sharing*, operation packets are distributed eagerly to remote processors for execution. In *work stealing*, idle remote processors steal operation packets from the pool of other processors.

The two strategies are compared in [Hammond 1994]. Work sharing runs the risk of distributing operations where there exists no idle processing capacity. To be efficient, a work sharing algorithm requires accurate load information in the distributed system. The drawback of work stealing is that it may increase latency before operations are executed, because they are exported on demand instead of being exported eagerly.

Another issue is how operation packets are scheduled for execution. Operation packets can be distributed on a last-in first out (LIFO) basis or on a first-in-first-out (FIFO) basis. [Hammond 1994] contrasts the effects of the two scheduling policies. The effect of scheduling operation packets on a LIFO basis is more like sequential execution whereas FIFO scheduling stimulates the execution of a greater number of operations in parallel. It must be noted that these comments are based on the observation of the evaluation of a recursive function, called `nfib`, which computes the values of the fibonacci series and may as such, only reflect the effect of these two scheduling strategies for the evaluation of the `nfib` function. Other strategies such as combining both LIFO and FIFO to control parallelism dynamically are also possible.

4.1.3 Remote execution of an operation

Classes are the templates from which instances are constructed and describe the behaviour of these instances. Class information is therefore required to unpack the instances transmitted with the operation packet and to invoke the behaviour of its instances.

Bennett [Bennett 1987] reviews a range of possibilities for the design of Distributed Smalltalk:

- Instances point back to their classes in the context in which they were instantiated. Method despatching and read and write operations on class variables require access to information stored in the class description. It becomes clear that this option would greatly increase the number of network operations and result in poor performance.

- Classes become immutable and can be freely replicated. Reactiveness is the degree to which objects are easily presented for inspection and modification. Disallowing class modification would severely restrict the reactiveness of the system. In the context of Prograph, this would mean that it would no longer be possible to execute and edit the code simultaneously, modifications to classes having being disabled.

- Classes are replicated and can be altered. Ensuring compatibility across contexts can be left to the programmer or it can be supported by a caching scheme, a distributed database of classes or versioning. Letting the programmer handle class compatibility problems is not consistent with the goal of distribution transparency. The distributed database or caching schemes might be difficult to implement efficiently.

Smalltalk views classes as objects and so class migration can be implemented using an object migration mechanism [Dollimore, Nascimento and Xu 1992].

In the commercial implementation of Distributed Smalltalk described in [LaLonde and Pugh 1996], the migration of classes is left to the user and is decided statically. A distributed application may encompass several object contexts (or *images* in Smalltalk terminology) residing over separate machines. Application classes are grouped into packages, and the user controls the distribution of packages. Packages can be loaded entirely or as a *shadow*, only proxy objects can be created from shadow classes.

In Prograph, upon successful completion, an operation may return some values or a `fail` execution message. Return values and execution message must be returned to the originator context. The execution of the operation may trigger a run-time error. Run-time errors include invalid type, out-of-range value and no method can be despatched. Although errors are well documented, Prograph provides no facility for exception handling and it is the programmer's responsibility to edit the faulty code fragment and proceed with or abort the execution of the program. A run-time error during the execution of a compiled application results in an abnormal termination of the execution. In Distributed Prograph, if the execution of a remote operation produces a `fail` execution message or triggers a run-time error, the remote processor should be able to forward the execution message or error type to the originator of the operation packet and to resume its normal activity.

The execution of the operation might update the values of some global variables or of its arguments. The modified arguments might not be returned explicitly by the exported operation to the originator context. The propagation of updates to operation arguments will depend on the approach chosen for the passing of arguments. If the arguments are passed as proxy objects, the update is immediately forwarded to the original object. Alternatively, an optimistic replication scheme would allow *reconciliation* in the originator context to be deferred.

4.1.4 Reception of the results

To retain the data driven semantics of the Prograph language, the execution of a remote operation will be considered finished when either the results or a `fail` execution message or an error message are returned to the originator context.

To maximise concurrency, the retrieval of the results should be postponed as long as possible. If when trying to retrieve the results of an operation, these results are not yet available, two different options can be considered:

- Another remote operation not yet exported may be scheduled for local execution. This offers the benefit of increased parallelism as the local processor evaluates an operation while waiting for results to be returned.
- If no other operation is available for local execution, the operations which have already been exported but whose results are still being awaited may be executed locally.

The reception of the results may be the right time to reconcile the replicas if an optimistic replication management scheme has been chosen.

4.1.5 Help to the programmer

The ultimate goal of Distributed Prograph is to allow programmers to build stand-alone, compiled applications. However, reactiveness is one of the recognised strengths of the Prograph programming environment as it lets programmers gain an in-depth understanding of the application they are developing. A distributed interpreter would give programmers a useful insight on the behaviour of their application, speed-up provided by distribution and ratio of local computation to distributed computation.

Prograph provides live editing, where values can be inspected or changed directly by the user. Various facilities for debugging have been proposed in different Distributed Smalltalk implementations: remote inspection enabled but remote editing disabled; full featured remote debugger.

The stack of a Prograph computation can also be inspected and computations can be rolled forward or backward. It would be challenging to support this facility in a distributed environment.

4.2 Implementation

[Briot and Guerraroui 1996] distinguish three approaches for object-based parallel and distributed programming:

- The first approach, called the *applicative* approach, uses the language to structure distributed systems.
- The *integrative* approach extends an existing programming language or creates a new one with language constructs to deal with concurrency and distribution.
- The last approach, termed the *reflexive* approach, makes use of the reflection, that is the property of a language of being self-descriptive and modifiable.

These approaches can be looked at from two different perspectives. The first perspective is that of language design and the second one, which is of interest to this section, that of implementation techniques. It is worth noting that the classification is also relevant for the design of distributed languages based on other paradigms: procedural and functional.

4.2.1 The applicative approach

The applicative approach aims to use the abstraction capabilities of the language to hide the complexity of the distribution mechanisms.

The applicative approach for procedural languages consists of writing libraries of procedures to hide the low level details of communication and providing the necessary abstractions such as threads and synchronisation variables. Application programs use the data structures and call the procedures provided by the libraries. The PVM message passing library [Sunderam 1990] and the DCE environment [OSF 1992] are examples of library-based systems.

In [Rabhi 1993], skeletons for parallel computations are implemented as higher order functions written in Haskell.

Object-oriented languages integrate the mechanism for distribution within a class hierarchy; one example is HP Distributed Smalltalk [Keremitsis and Fuller 1995] which implements a CORBA-compliant Object Request Broker within its class hierarchy.

The benefit of the applicative approach is that it exploits the features of an existing programming language to provide new abstractions for distributed programming. However, this approach requires the application programmer to deal with a potentially larger number of concepts than for the development of sequential applications.

As explained in Chapter 2, Prograph already makes an extended use of this approach. For example, Prograph applications are described as a containment hierarchy of Prograph objects.

4.2.2 The integrative approach

The abstractions for distributed programming are integrated within the language. This solution often requires significant modifications to an existing language or the design of a new language. The language reflects closely the distributed programming model and thus has greater expressive power than a language initially designed for sequential programming.

The designers of the Guide language [Balter, Lacourte and Riveill 1994] have taken the view that distributed programming justified the design of a new language.

However, such a choice requires the mobilisation of significant resources to carry the implementation of an interpreter and/or compiler and associated run-time support for the language. Acceptance of a new language frequently presents a significant problem.

The integrative approach seems incompatible with one of the Distributed Prograph goals, which is to remain as close as possible to the sequential version of Prograph.

4.2.3 The reflexive approach

A *metacircular* evaluator is an evaluator where the defining language is the same as the defined one. Lisp interpreters exploit metacircularity. Metacircularity is a powerful feature to control and extend the language. Quasi-Parallel Lisp (QPL) [DeRoure 1990] extends the Lisp language to integrate abstractions for communication, the *stream*, and activity, the *process*.

Reflection in object-oriented systems results from the possibility of defining the semantics of objects in an object-oriented model through a set of objects called *meta-objects*. This organisation is named a *Meta-object-protocol*. Meta-objects can control some features of the object model such as message sending, method look-up, execution and state accessing. Meta-objects may also extend and modify resources management such as scheduling and naming.

The GARF [Garbinato, Guerraoui and Mazouni 1994] class hierarchy separates two programming levels. At the functional level, a collection of *data classes* describes the logic of the application as if the application was developed in a centralised, sequential environment. The behavioural level controls behavioural features related to concurrency, persistence distribution and fault-tolerance.

The reflexive approach offers the benefit of a great flexibility. Complexity and potential inefficiency can be seen as shortcomings of reflexive architectures.

This option does not appear viable for the implementation of Distributed Prograph as the language presents little reflection.

Briot and Guerraroui see in the development of generic run-time systems for distributed languages a dual approach to the use of reflection. The distribution mechanisms are integrated within the run-time support for the language. This choice is motivated by the search for greater efficiency with some of the flexibility of the reflexive approach. The requirement for efficiency becomes more urgent as the functionality and the complexity of the run-time system increase. The GUM run-time uses PVM to implement a task pool for the scheduling of distributed tasks. The Chorus Object-Oriented Layer (COOL) [Lea, Jacquemot and Pillevesse 1993] is a run-time system upon which distributed object-oriented languages (C++ and Eiffel) can be built.

The integration of the Prograph interpreter with an existing distribution infrastructure is an option worth investigating.

4.3 Need for analysis

The section on design issues surveyed a range of policies and mechanisms to implement these policies and to extend the features of Prograph to provide distributed programming. This section discusses why a static analysis may be useful to support these mechanisms.

4.3.1 Interferences

In the current version of Prograph, the operations on the dataflow graph are executed according to a serial schedule of execution and each operation constitutes an atomic unit of execution. In some cases, the results of a computation do not depend on the order of execution of the operations; however the programmer has sometimes to impose a special ordering for the execution of the operations. This is typically the case when one of the operations is a `Match` or calls a method providing I/O functionality. The concurrent execution of operations might lead to interferences. Interference occurs when two or more parallel operations read the same data and at least one of them updates the data. Thus, distributed execution may introduce further synchronisation requirements.

4.3.2 Global variables

The pure dataflow model is side-effect free and does not provide global variables. However, Prograph has both side-effects and global variables, in the form of persistents and class variables.

When a computation is distributed over several execution contexts, it is necessary to ensure that the global state is kept consistent. The value of a global variable is the same as if the operation had been executed in its originator context following a serial schedule of execution. According to the policy adopted for the management of the global variables, different concerns must be addressed:

- If consistency is ensured by maintaining a single copy of each global variable, the cost of the access to this single copy is the main concern.
- If global variables are replicated, consistency of the replicas becomes the issue and the question of whether the replicated global values in the recipient context can be trusted.

4.3.3 Updated values and aliases

Operations may induce side-effects on their arguments and/or on some global state. Preserving the semantics of Prograph requires that the side-effects are implemented, including the aliases created by write operations. The side-effects should be visible when the execution of the remote operation is considered completed. Once again the choice is between forwarding operations across execution contexts on a single copy or managing replicas in different contexts.

The update problem is compounded by the possibility of creating aliases, as the semantics of Prograph implies that aliases be preserved even across separate contexts.

4.3.4 Behaviour maintenance

Behaviour maintenance is primarily concerned with the consistency of the class definitions, that is, of the attributes (not the values of the attributes) and of the methods, as well as that of the universal methods. This issue should not be overlooked especially if Distributed Prograph is to be used as a distributed interpreter. This work does not however tackle the issue of behaviour maintenance across several contexts.

Behaviour maintenance in Prograph is a problem only in interpreted mode because class definitions and methods can be manipulated only using the editor. Therefore it might be more advisable to build support within the interpreter instead of a general distribution mechanism to be included with all applications. In addition, updates to arguments and global variables are more common in programs and therefore a more urgent problem to solve.

4.4 Aims of the analysis

The correct execution of operations in parallel is conditioned by the absence of interference between the operations, the availability of the current values of the global variables and the implementation of the side-effects induced by the operations. The development of a static analysis will help to check that some of the conditions for the correct execution of parallel operations are met.

The analysis will not tackle the problem of interferences. As a consequence of aliasing, interference may occur in a *stealthy* manner, that is the same data can be accessed via different paths by different operations. Program results are saved between sessions of the Prograph interpreter. Consequently, aliases may result from previous executions of the program. Thus it is believed that there would extremely difficult to detect aliases by statically analysing program code and, by extension, to solve the problem of interferences between operations executed concurrently.

The analysis to be developed will address the issues of access to global variables and updates to arguments. It aims at characterising the side-effects induced by a subcomputation. Characterising an effect means identifying unambiguously its nature (read or write), and the data upon which it operates (class, persistent value or operation input). The effected data should be referred to by their symbolic names.

There are several purposes to this effect analysis:

- It will provide the user with some useful information about the behaviour of operations annotated for distribution. Clearly operations with a purely functional behaviour are more suited for remote execution than those which heavily affect their arguments or global variables. Such feedback might help the user to annotate the program.

- The results produced by the analysis can be exploited to optimise the distribution mechanisms. Being able to statically anticipate the accesses and updates to operation arguments and global variables reduces the cost of keeping the value of global variables consistent and implementing side-effects. Alternatively, it might be decided that the overhead incurred by these mechanisms is too high and that operations inducing certain effects should not be distributed.

It is important to keep a clear separation between the analysis and the distribution policies so that the analysis does not become biased towards supporting a particular policy.

4.5 Summary

This chapter has discussed the following points:

- A range of issues has to be addressed for the design of Distributed Prograph. The most important issues are transmission of the operation arguments and results, the scheduling of operations in parallel and the maintenance of the global state and behaviour across several separate contexts.

• Three different approaches can be distinguished for the implementation of a distributed programming language. However, only the applicative approach seems to be exploitable for the implementation of Distributed Prograph.

• Static analysis of programs can help in solving the problem of the consistency of values across several contexts.

5 Type Inference

A type inference algorithm constitutes the first component of the analysis described in this thesis. This chapter details the design of the inference mechanism after reviewing several type-related issues.

The first section discusses the purpose of types in programming languages and the different approaches to typing, namely static versus dynamic typing. The second section surveys issues related to the design of a type inference algorithm. Previous work on type inference is the topic of the third section. The fourth section exposes the need for type inference as part of the analysis developed in this work. The inference algorithm is outlined in the fifth section. The sixth and the seventh sections present a suitable type representation and the rules to type the different expressions of the Prograph language. The details of the type inference algorithm are presented in the eighth section. Type inference is illustrated by two examples in the ninth section. The tenth section discusses shortcomings of the algorithm.

5.1 Types in programming languages

Wegner [Wegner 1986] defines the properties that types should have to constitute a type system for object-oriented programming languages. By removing the references to features which are specific to object-oriented languages (e.g. inheritance), the definition can be extended to cover the purpose and properties of types in programming languages:

- Application programmer's view: Types partition values into equivalence classes with common attributes and operations.

- System evolution view: Types are behaviour specifications that may be composed and incrementally modified to form new behaviour specifications.

- Type checking view: Types impose syntactic constraints on expressions so that operators and operands of composite expressions are compatible.

- Verification view: Types determine behavioural invariants that instances of the type are required to satisfy.

- System programming and security view: Types are a suit of clothes (armour) that protects raw information (bit strings) from unintended interpretations.

- Implementer's view: Types specify a storage mapping for values.

Typing means associating a type with every expression of a program. Two broad approaches to typing can be distinguished, the static and the dynamic one. With the static approach, the type is statically associated with all the variables and expressions of a computation. Dynamic typing distinguishes itself from static typing in the sense that the type information is available only at run-time and that types are not bound to variables but to the values instead.

5.1.1 Static typing

Static typing in procedural languages requires the programmer to declare the types of the variables and procedures. The correctness of the type declarations is established by a *type checker.*

Functional languages such as ML and Miranda are equipped with an *implicit* static type system. Implicit typing requires a minimum of type declarations. Type information is inferred from the local context and type correctness can be established.

Static knowledge of the types of the values involved in a computation provides a safety guarantee and enables optimisation. Declaring type information is also seen as conducive to good software engineering practices: type declarations serve as partial specifications.

Most procedural languages are *monomorphic*, and the types of variables, functions and procedures are invariant, remaining the same throughout the execution of the program. Such languages may not describe generic procedures where algorithms are applicable to values of different types. A procedure to compute the length of a list is an example of generic procedure.

ML-like type systems introduce the concept of type variable. These variables can be instantiated to different types thus allowing functions to accept arguments of different types.

Type systems for object-oriented languages often use interfaces as types. All objects of a given class have their interface described by an *abstract type*.

An abstract type definition contains:

- The signatures of the methods supported by the class. A method signature specifies the types of the input and output arguments of a method.

- The type of the attributes of the instances of the class.

The following example (taken from [Balter, Lacoutre and Riveill 1994]) defines a document description for a computerised library catalogue:

```
TYPE Document_descr IS
    key: Integer;
    title, author: String;
    date_borrowed, date_returned: REF Date;
    METHOD Init;        //set initial values
    METHOD Consult;     //display information about the
                        //document
    METHOD Get_text: REF Document
                        // gives access to the text of the
                        //document
END Document_descr.
```

A `Document-descr` has a `key` attribute, `title` and `author` attributes and `date_borrowed` and `date_returned` attributes. The methods defined for the `Document_descr` abstract type are `Init`, `Consult` and `Get_text`. The three methods take an object of type `Document_descr` as (implicit) argument. The method `Get_text` returns a reference to a document (`REF Document`).

Types can be partially ordered; `B` is a *subtype* or *is included* in `A` when all the values of type `B` are also values of type `A`. $B \leq A$ denotes the inclusion of type `B` in type `A`. The inclusion rules for method signatures ($A \rightarrow B$ denotes a method signature with an argument type `A` and a return type `B`) is:

$$A' \rightarrow B' \leq A \rightarrow B \text{ iff } A \leq A' \text{ and } B' \leq B$$

This rule, known as *contravariance*, states that the type of method m_b is the subtype of method m_a if the argument type of m_b (A') is more general than the argument type of m_a (A) and the return type of m_b (B') is more specialised than the return type of m_a (B).

Abstract type `A` is the supertype of abstract type `B` if:

- Each method m_a defined in the interface of `A` is matched by a method m_b of the same name in the interface of `B` and that the type of m_b is the subtype of the type of method m_a.

- Each attribute of A is defined in B and the type of the attribute of A is the supertype of that of the attribute of B.

To provide inclusion polymorphism and static typing, the following rules must be respected:

- The type of the value assigned to a variable is a subtype of the type declared for that variable.

- A message send is type correct if a method of corresponding type is defined in the signature of the receiver and the types of the message arguments conform to the types of the formal method arguments.

There exists no consensus on how class and types should be composed:

- With the concept of *Class Type*, a class defines a type and by extension inheritance and subtyping are considered equivalent. The type of a subclass has to conform to that of its superclass.

- It has been argued that subclassing and subtyping are different notions: subtyping is the sharing of abstract behaviour whereas subclassing is a mechanism which provides code reuse [LaLonde and Pugh 1991]. Guide [Balter, Lacoutre and Riveill 1994] separates types which describe the interface and the classes that implement the type. This approach allows greater flexibility as two unrelated classes may conform to the same type and conversely, a subclass does not have to conform to the type of its superclass.

Parametrisable classes are a feature of several statically typed object-oriented languages and allow code reuse through parametric polymorphism.

Type systems for statically typed object-oriented languages still attract considerable research interest. *Covariance*, unlike contravariance, allows the programmer to specialise argument types of methods in subclasses; covariance remains the subject of investigations [Shang 1994].

5.1.2 Dynamic typing

Dynamic type systems bind types not to variables but to values, thus variables can be polymorphic.

Dynamic typing gives languages a greater flexibility than their statically typed counterparts, most notably for the implementation of collections of elements of heteregenous types and the manipulation of these collections.

Dynamically typed languages are often associated with an interactive and incremental style of programming appropriate for experimentation and prototyping. From a language designer point of view, Cardelli [Cardelli 1995] considers that untyped languages are easier to prototype.

The drawbacks of dynamic typing are, firstly, the possibility of run-time type errors as type checking can only be performed at run-time and secondly that it incurs several overheads over static typing:

- Memory overhead: values must carry a type tag attached to them and their implementation takes more memory space. Also the absence of type information at compile time may rule out optimisation such as dead code elimination and the use of machine data types instead of program data types.
- Performance overhead: the absence of type information prevents performance oriented optimisations including inlining and use of machine data types. Run-time type checking induces an extra performance cost. In [Steenkiste and Hennessy 1987] the cost of type computations for LISP applications is estimated to increase the execution time by 25%, on average.

5.1.3 Bridging the gap

It may be considered that the two typing approaches are not antagonistic but complementary as suggested in [Palsberg and Schwartzbach 1993] p.72.

Providing type information during the prototyping phase places an unnecessary burden on the programmer and type correctness is not very important at that early stage. Wegner [Wegner 1987] suggests that dynamic typing is more appropriate during the development phase.

Type inference aims at computing type information from an untyped language. The transition is illustrated by the following diagram taken from [Palsberg and Schwartzbach 1993]:

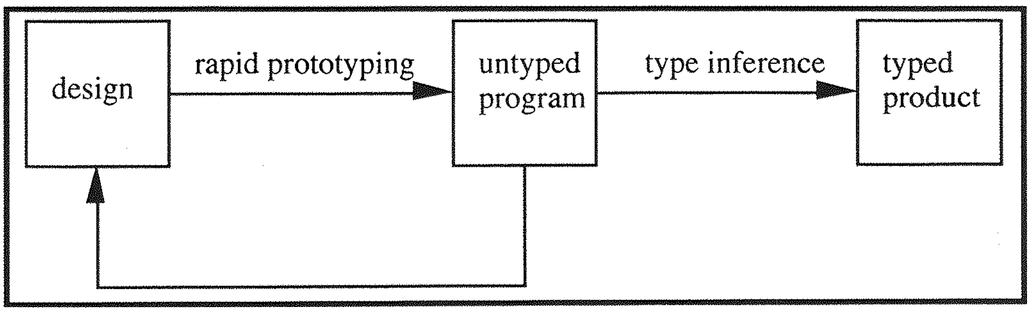


Fig. 5.1: From untyped to typed program.

5.2 Issues for type inference

Type inference has been explored for various language paradigms: procedural, object-oriented and functional. The approaches investigated may also vary in:

- purpose;
- world assumption;
- type system;

The remainder of this section discusses these three issues in turn.

5.2.1 Purpose of inferring types

The type information produced may serve several purposes: safety, optimisation and help to the programmer.

In the context of object-oriented languages, the overriding safety concern is that method despatching might fail. Type inference will check that the message send cannot fail at run-time.

Efficiency concerns encompass performance and compactness of the compiled code. Dynamic binding incurs a run-time overhead. Static knowledge of the concrete type of the receiver of a message allows the message to be bound to a method statically, thus eliminating the cost of a method look-up. The code can be further optimised by *inlining* methods. Dynamic binding also has an impact on the size of the compiled executable as the code for classes that may never be instantiated or methods that may not be called is included. Type inference may help to detect that a method cannot be called during the execution of the program so that the *dead code* can be removed.

The information inferred can be fed to analysis and debugging tools. In connection with the safety concern expressed above, the programmer can be warned about the likely failure of a message send. Type information feedback helps the programmer to check that the type of the method is compatible with the intended use for that method.

5.2.2 World assumptions

A *closed-world* assumption is that all parts making up an application are available for analysis and compilation. In object-oriented languages where code and data are encapsulated in classes, a closed-world assumption thus requires the set of classes to be known statically.

Under an *open-world* assumption, a program is divided into separate modules or libraries that can be analysed and compiled separately. The *open-world* assumption, although a more desirable approach to software engineering, may not be compatible with all the possible purposes for doing type inference. For example an open-world assumption does not help to address efficiency concerns because it does not allow the identification of which code is executed at run-time.

A *modular* type inference may analyse methods one method at a time. A *non-modular* inference is performed on a whole program at a time and requires a closed world view.

5.2.3 Type systems

A type system for a programming language defines:

- a set of type expressions;
- operations to manipulate type expressions (e.g. *Cartesian product* to describe record type and *disjoint sum*);
- a set of rules for associating a type with all the expressions of the programming language.

In his discussion of *Type Systems and Polymorphism*, Agesen [Agesen 1996] distinguishes two dimensions to classify type systems: the concrete/abstract dimension and the general/specific dimension (see fig. 5.2). Discussion of a type system cannot be divorced from the assumption under which the type inference operates and the purpose of the inference itself.

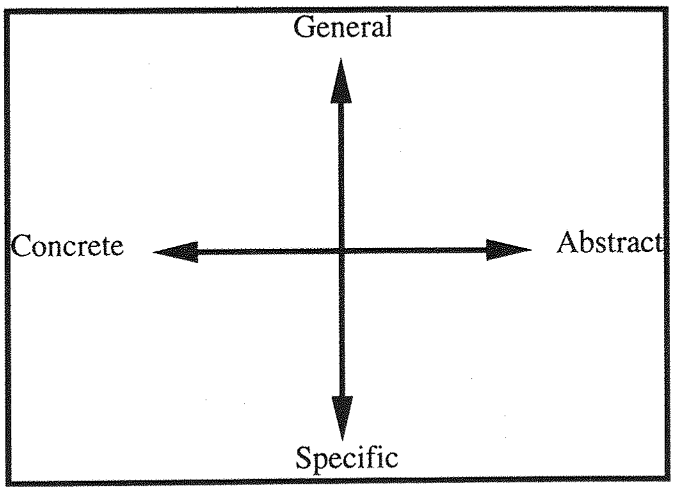


Fig. 5.2: Dimensions of type system

The *concrete type* of an object is the set of classes of which this object can be an instance. Concrete type inference requires a closed-world view and it is mainly aimed at optimisation.

Abstract types have already been introduced in 5.1.1, they describe the external behaviour of an object. Abstract types are useful to prove type safety and are compatible with an open world view.

In Agesen's view, class types represent a half-way house between concrete and abstract types. The distinction between concrete and class type can be illustrated by an example taken from the class hierarchy depicted in fig. 5.3. The class type `Bird` encompasses instances of `Bird`, `Chicken` and `Duck`. The corresponding concrete type would be the set `{Bird, Chicken, Duck}`. Under a closed world view, class types can be converted into concrete types and vice-versa.

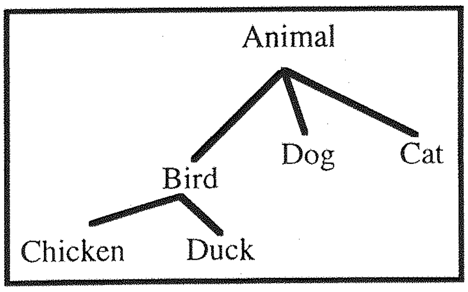


Fig. 5.3: A class hierarchy.

General types describe the way a method or an object may be used, that is, all the possible legal types for the object or method. *Specific types* describe the use of an object or of a method in the context of a particular application and assume a closed-world view.

5.3 Previous work

5.3.1 Kaplan/Ullman

[Kaplan and Ullman 1980] presented an inference method for an untyped imperative language for the purpose of compile-time optimisation.

The program is modelled as a *flowgraph*. A flowgraph is a directed graph with nodes consisting of one or more assignment statements of the form:

$$y \leftarrow f(x_1 \ldots x_n)$$

and the edges following the control flow of the program. The graph starts from and finishes at a *Start-Finish* (*SF)* node. The set of all the program variables is constructed and a mapping from the set of the program variables to the set of types is defined. The mapping is refined by successive iterations of the *forward analysis* and of the *backward analysis* over the nodes.

Forward analysis infers the type of the program variables after the execution of a statement from the types of the program variables before execution of the statement and the signature of the statement.

Backward analysis infers the type of program variables from the type information available after the execution of a statement and the signature of the statement.

- Types are elements of a type lattice (see fig. 5.4) with a top and a bottom element. A *least-upper bound* and *greatest-lower bound* operation are defined on the lattice.

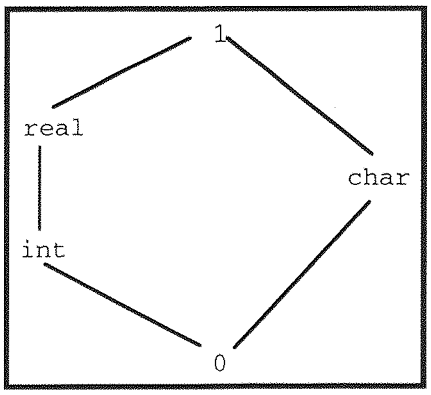


Fig. 5.4: Type Lattice

- Statements' signatures are constructed using *T-functions*. T^0 is a function which takes the types of the arguments $x_1 \dots x_n$ of the statement as its arguments and returns the best approximation of the type of the return value of the statement after its execution. T^0 is used during the iterations of the forward analysis. Similarly, for each argument x_j of the statement, a function T^j is defined. During the backward passes of the analysis T^j is applied to the types of the arguments $x_1 \dots x_n$ and the type of the return value y of the statement. Thus T^j provides the best approximation of the type of x_j before assignment from the information available after assignment.

As an example, consider `f1(x)`, a simple function which returns the greatest integer smaller than or equal to x if x is a number, otherwise, if x is a char, x is translated into lower case.

T^0 takes the type of x as argument and returns the type of the return value of `f1(x)`. The definition of T^0 (see fig. 5.5.a) shows if the type of x is `real`, the type of the return value will be `int` (second row of the matrix)

x	T^0
1	1
real	int
int	int
char	char
0	0

Fig. 5.5.a: Definition of T^0

T^1 is used during the backward analysis, the leftmost column contains the current approximation of the type of x, the argument of `f1(x)` and the top row the type of y, the return value of the `f1(x)`. The intersection of a type of x and y returns a new approximation of the type of x.

y / x	1	real	int	char	0
1	1	real	real	char	0
real	real	real	real	0	0
int	int	int	int	0	0
char	char	0	0	char	0
0	0	0	0	0	0

Fig. 5.5.b: Definition of T^1

5.3.2 Hindley/Milner

Milner extended Hindley's type inference algorithm and used it for an ML type system [Milner 1978]. ML is a functional language with lexical binding and higher-order functions. Hindley/Milner inference is more concerned with type safety than with optimisation. It computes the *principal* types (most general types) of expressions to rule out run-time type errors. All of the types of a polymorphic expression are an instance of the principal type of the expression.

The inference algorithm handles functions one by one. The function is converted into an expression tree where the leaves of the tree are either constants or variable accesses.

The function:

```
let rec length = fun(l) if null(l)
            then
                  0
            else
                  succ(length(tl(l)))
in ...
```

is converted into the tree shown in fig. 5.6.

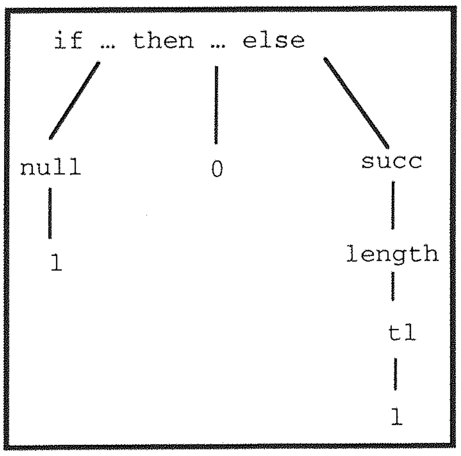


Fig. 5.6: Expression tree for `length`

Type information can be represented as:

- Basic types such as `Int` or `Boolean`.

- Type variables, denoted by a Greek letter: α, β, γ range over the complete set of types. A distinction is made between *generic* and *non-generic* type variables. All occurrences of a non generic type variable must be instantiated to the same type values. Generic type variables may be instantiated to different type values. The difference between non-generic and generic type variables is illustrated by the following example taken from [Cardelli 1987].

`fun (f) pair(f(3)) (f(true))` cannot be typed because the type of `f` which should be of the form $\alpha \rightarrow \beta$ is instantiated to `Int` $\rightarrow \beta$ for the first application of f and to `Boolean` $\rightarrow \beta$ for the second application. The type variable α appears in the type of the `fun`-bound identifier `f` and is therefore non generic and cannot be instantiated to different type values.

The expression `let f = fun(a) a in pair (f(3)) (f(true))can` be typed. In the type of `f` $\alpha \rightarrow \alpha$, the α's are generic and `f` can take the types `Int` $\rightarrow$ `Int` and `Boolean` $\rightarrow$ `Boolean`.

- Function types $\rho \rightarrow \sigma$. A function type maps the type of the function argument to the type of the function return value.

All the leaves have a type associated to them. A constant leaf is assigned the type corresponding to the value of the constant. The type assigned to a variable access leaf is the type variable of the variable accessed.

The inference mechanism maintains a map called *assumptions* from the set of variables to the set of types. Every time an occurrence of a variable is found, the set of assumptions is accessed to yield the type of that occurrence.

Inference rules describe how the type of an expression can be deduced from the types of its subexpressions. The rule for the typing of an `if e then e' else e"` expression states that, if the type of `e` is boolean, the type of `e'` is τ and the type of `e"` is τ, then the type of the expression is τ. The rule is formalised by an expression of the form:

$$\frac{A \vdash e\text{: bool} \qquad A \vdash e' : \tau \qquad A \vdash e'' : \tau}{A \vdash (\texttt{if } e \texttt{ then } e' \texttt{ else } e'') : \tau}$$

where the horizontal bar means *imply*, the line above the horizontal bar is the premise and the line below the bar the conclusion. $A \vdash e: \tau$ means that from the set of assumptions `A`, it can be deduced that the type of `e` is τ.

Inference proceeds in a bottom-up fashion from the leaves to the root of the expression tree. The application of an inference rule yields a set of equalities on the types of the subexpressions. The equalities are solved by *unification*

[Robinson 1965]. Failure to solve the equations means that the function cannot be typed.

5.3.3 Suzuki

Suzuki [Suzuki 1981] proposed a type inference mechanism for Smalltalk. The aim of the system was, firstly, to substitute static binding for dynamic binding and thus improve efficiency and secondly to provide type information to the programmer.

Suzuki's work draws on the functional approach. Unlike ML, Smalltalk has:

- Dynamic binding
- Data polymorphism

These features make some modifications to the Milner's original algorithm necessary. The modifications concern the representation of types, inference rules and the constraints on types.

Types can be:

- Basic types: they are sets of classes
- Type variables, denoted by Greek letters.
- Function types are used to describe the signatures of methods. Function types are of the form: $\alpha \times \beta \rightarrow \gamma$, the types on the left hand side of the arrow are that of the receiver and the arguments of the method, the type on the right hand side of the arrow is that of the method return value.

Milner's algorithm solves a set of equalities over type values but with Suzuki's approach types are sets of types and constraints are expressed as set inclusions.

The inference rules must also be altered for conditional expressions and more importantly for function applications. To be able to type message sends (the linguistic equivalent of function application in ML), Suzuki's algorithm assumed that all the methods with the matching name can be despatched. The type of the message send is the union of the types of all the methods that may be invoked.

Smalltalk allows variables to hold values of different types (data polymorphism). The implemented algorithm did not attempt to infer the type of the instance and global variables. Instead, the algorithm took the view that the type of such variables is the set of all the classes in the system.

5.3.4 The EULisp type inference system

EULisp [Padget, Nuyens and Bretthauer 1993] belongs to the family of LISP dialects and features object-orientation. The language unifies the functional and object-oriented paradigms, providing classes and functions. A *generic function* is a function for which several implementations are available. Each implementation or *method* is defined with a distinct domain which specifies the applicability of the method to supplied arguments. Unlike a message send which despatches the method applicable to the class of the receiver and ignores the types of the method arguments, the application of a generic function will despatch a method only if the class of each argument is a subclass of the corresponding domain class formally declared in the method definition. The following example, provided by [Kind 1996] declares a generic function `element` which returns the i-th element of an ordered collection.

```
(defgeneric element (x y))
```

The methods for the generic function are defined for the different domains. For a string argument, `element` is implemented as:

```
(defmethod element ((x <string>) (i <integer>))
 (string-ref x i))
```

For an argument of the vector class as:

```
(defmethod element ((x <vector>) (i <integer>))
 (vector-ref x i))
```

For an argument of the list class as:

```
(defmethod element ((x <list>) (i <integer>))
 (if (= i 0)
  (car x)
 (element (cdr x) (- i 1))))
```

`Element` can be used as any other function.

```
(defun foo (x)
...
 (element x 1)
...)
```

[Kind and Friedrich 1993] have proposed a type inference mechanism for EULisp. The type information inferred is used mainly for optimisation during the compilation of applications.

The representation for types draws heavily on Milner's work with some improvements to handle inclusion polymorphism and provide more precise type information about functions

- Lattice types: A lattice L is constructed over the set of the *base* types B. The set of the base types can in turn be defined as the set of the concrete types augmented with a set of *strategic* types. The concrete types correspond to EULisp classes, for example `<list>`, `<vector>` or `<string>`. Strategic types have been introduced to describe distinguished values and aim to yield a more precise typing of predicates and conditional expressions, examples of such types include: `singleton`, `zero`, `one`. The lattice reflects the subtype relation over the sets of types. For example, the lattice type `<number>` is the sublattice of L whose vertex is the base type `<number>`.

- Type variables: They are used to express type dependencies between argument types or between argument types and the result type of a function. The range of a type variable may be restricted, for example, $\alpha^{\texttt{<number>}}$ denotes a type variable whose upper bound is `<number>`.

- Generic type schemes embody constraints on the argument and result types of functions; they contain several lines of the form:

$$\tau_1 \times \tau_2 \times \tau_3 \rightarrow \tau_4$$

where τ_1, τ_2 and τ_3 are the types of the function arguments and τ_4 the type of the function's return value. Using a number of lines, type dependencies can be expressed more precisely.

Inference is performed in three steps:

- A local inference computes the most-general type of functions. The local inference starts with a set of initial type constraints on the function's parameters, literals, constants and variables. A type scheme must be available for each function call within the function's body. If a type scheme is missing, the current analysis is suspended until the missing scheme has been inferred. The type scheme for a generic function application contains the lines of the schemes of all the methods that may be despatched. Solving the constraints by unification yields a scheme for the function.

• A global inference aims at deriving concrete type information by examining all the calls to a given function.

• The local inference is iterated again with feedback from the global inference.

5.3.5 Palsberg and Schwartzbach

The language of interest for the type inference described in [Palsberg and Schwartzbach 1991] is a subset of Smalltalk called BOPL (Basic Object Programming Language). The analysis is applied to programs globally, under a closed-world assumption. Palsberg and Schwartzbach envision several uses for the type information inferred: safety, dead code elimination, static method binding and information to the user.

Types are defined to be sets of classes. Inheritance, parametric polymorphism and data polymorphism are tackled by some program transformations:

• Inheritance is expanded away. Each inherited method or variable is duplicated in the class which inherits it and all the occurrences of the pseudovariable `super` are replaced by `self`. If a method is redefined by a subclass, the name of the superclass is appended to the name of the inherited method. Some aspects of the transformation are illustrated by the example below:

```
class Rectangle
    var l, w
    method Base(l0,w0)
            l:=l0;w:=w0;self
    method Area()
            l*w
    method Scale(s)
            l:=l*s; w:=w*s; self
end Rectangle
class Box inherits Rectangle
    var h
    method Height(h0)
            h:=h0
    method Volume()
            self.Area()*h
    end
    method Scale(s)
           super.Scale(s); h:=h*s; self
end Box
```

The definition of the class `Rectangle` is left unchanged by the transformation but the class `Box` becomes:

```
class Box
    var l, w, h
    method Base(l0,w0)
           l:=l0;w:=w0;self
    method Area()
           l*w
    method Scale$Rectangle(s)
           l:=l*s; w:=w*s; self
    method Height(h0)
           h:=h0
    method Volume()
           self.Area()*h
    method Scale(s)
           self.Scale$Rectangle(s); h:=h*s; self
end Box
```

• To handle parametric polymorphism, methods are duplicated for each message send in which their name occcurs. This transformation is illustrated by the following example:

```
class C
    method id(x)
           x
end C
((C new).id(7))+10;
((C new).id(true)) or false
```

A different version of `id` is created for the message sends in which `id` occurs.

```
class C
    method id@1(x)
           x
    method id@2(x)
```

```
        x
end C
((C new).id@1(7))+10;
((C new).id@2(true)) or false
```

• A copy of the class is created for each instantiation. This transformation allows a precise treatment of data polymorphism.

```
class Container
    var x
    method put(val)
            x:= val;
            self
    method get()
            x
end Container
(Container new).put(7).get.()+10;
(Container new).put(false).get() or true
```

The `Container` class is entirely duplicated and becomes `Container@1` and `Container@2`. The modified program is:

```
(Container@1 new).put(7).get()+10;
(Container@2 new).put(false).get() or true
```

The expanded program is converted into a *trace graph*. The nodes of the graph represent methods and the edges of the graph message sends. A condition is attached to each edge: the edge will be traversed only if the type of the receiver contains the class which implements the method represented by the node to which the edge is leading.

The construction of the graph is illustrated for the following simple program:

```
class A                         class B
        method m: e                     var temp
                e n                     method m: e
                                                (temp := e) n
                                        method n
                                                temp p
                                        method p
                                                self p
end A                           end B
(A new) m: (B new)
```

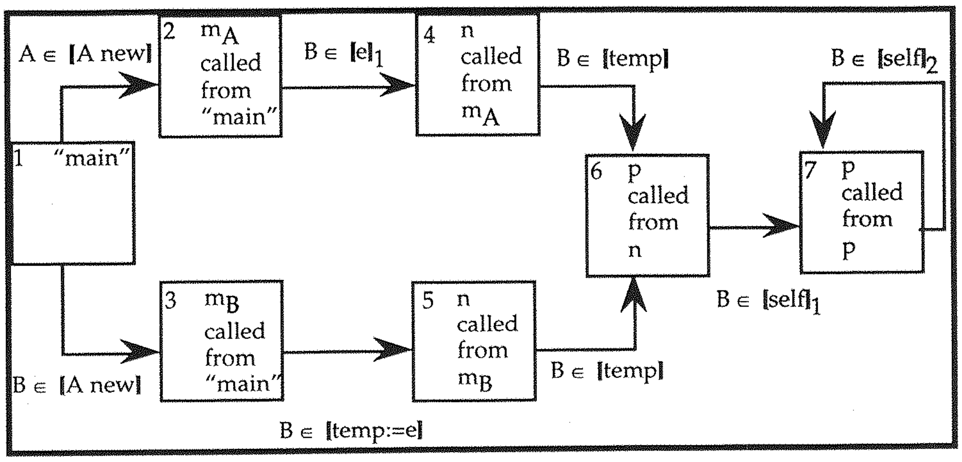


Fig. 5.7: A Trace graph

In the trace graph of fig. 5.7, the first node corresponds to the execution of the program. Node 2 represents the invocation of the method m defined in class A and node 3 represents the invocation of the method m defined in class B. The condition A ∈ [A new] attached to the edge between node 1 and node 2 expresses the requirement that the class A must be an element of the type of the expression A new if the method m defined in class A is to be called. Similarly, the condition B ∈ [A new] attached to the edge between node 1 and node 3 means that the class B must be an element of the type of the expression A new if the method m defined in class B is to be called.

A set of constraints is derived from inference rules. Constraints can be of three different types. Constraints can be:

- Local constraints reflect the semantics of the method body. For the node 3 on fig. 5.7. The constraints are:

  - $[\![\text{temp}]\!] \supseteq [\![\text{e}]\!]_2$

  - $[\![\texttt{temp := e}]\!] \supseteq [\![\text{e}]\!]_2$. This constraint and the one above result from the assignement of the value of the variable e to the variable temp.

  - $[\![\texttt{temp := e}]\!] \subseteq \{\texttt{B}\}$. This constraint reflects the invocation of the method n in the body of m defined by class B. It states that the type of the expression temp:= e must be included in the set of classes defining a method n (here the singleton {B}).

- Connecting constraints reflects the semantics of message sends. They embody the matching of the type of the actual and formal arguments of the method as well as the matching of the return value with the result of the invoked method's body. For the transition from node 1 to node 3, the following constraints can be derived:

- $[\![$`B new`$]\!] \subseteq [\![e]\!]_2$ (Constraint on the type of the argument).

- $[\![$`(A new)m: (B new)`$]\!] \supseteq [\![$`(temp:=e) n`$]\!]$ (Constraint on the type of the return value)

• Global constraints state that a path is executable if all the edge conditions encountered hold. Global constraints are constructed by traversing all the paths from the main node in the graph. Walking along a path yields an expression of the form:

$$K_1, K_2, \ldots, K_n \Rightarrow L \cup C$$

with K a condition on an edge, L the local constraints of the final node of the path and C the connecting constraints to reach the final node.

The set of constraints is solved to yield a type for all expressions in the program or it fails if the program is untypable. The description of an efficient implementation of the constraint solver can be found in [Oxhøj, Palsberg and Schwartzbach 1992].

5.4 Motivations for inferring types in Prograph

The ultimate objective of the analysis described in this thesis is to obtain an approximation of the side-effects that the execution of an operation annotated for distribution may induce. Type information will help to reach this goal because:

• Types and effects are not orthogonal, as instances of primitive datatypes are immutable (except for list) and knowledge of types is useful to infer side-effects. Class variables can be reached via the instances of the class and thus knowing the class to which the instance belongs gives extra information about class variable accesses or updates.

• The type inference will help to reduce the uncertainty due to the dynamic binding of operations to methods. A better approximation of the effects will be possible if the side-effects of the methods that cannot be despatched are ignored.

5.5 Outline of the type inference system

This section outlines the type inference developed for Prograph.

5.5.1 Method-wide analysis

The inference algorithm proceeds in a modular fashion: it is applied to individual methods but it operates under a closed-world assumption (it is in that respect similar to the type inference algorithm for Smalltalk presented in [Suzuki 1981]).

The method-wide inference reflects the structure of the method to which it is applied and relies in turn on a sequence of case-wide inferences, one for each case in the method. The analysis of a case depends on the results obtained for the previous one. Once the case-wide inferences have finished, the type information for the whole method can be synthesised.

5.5.2 Case-wide analysis

A case describes some computation using a visual dataflow graph. Such a representation lends itself easily to analysis and the information necessary for the inference is attached to the elements of the graph. It is important to distinguish between the type information attached to the datalinks and that attached to the operations (or nodes) of the graph.

- Type of values: different datalinks propagate the same value if they are attached to the same root and so these datalinks should also share the same type information.

- Type of operations: the type information available for an operation is described by a signature. A *signature* consists of one or more *lines*. A line is made up of a sequence of input types and a sequence of output types. Each input and output of the operation is matched by a type expression which describes the set of classes from which the matching input or output can be an instance. The expression:

  ```
  <boolean> × <boolean> × <number> → <boolean> × <number>
  <boolean> × <boolean> × <number> → <boolean> × <number>
  ```

  could be two lines of the type signature of an operation with three inputs and two outputs where `<boolean>` denotes the boolean type and `<number>` the type of real and integer values in Prograph. (The range of valid type expressions will be discussed in section 5.6).

The case-wide inference is divided into two phases: the initialisation phase and iterative analysis phase.

5.5.2.1 Initialisation phase

During the set-up phase, the type information attached to the datalinks and the signatures attached of the operations are initialised.

The type of the datalinks is initialised to the most general type, except for the datalinks connected to the roots of the input bar, which are initialised according the results of the analysis of the previous case.

The signatures attached to the operations calling a primitive method are looked-up or can be built "on the fly" for some operations such as default `Get` and `Set`, `Match`, `Constant` and the `Init` operations. For an operation calling a user defined method, if the signature of this method has not been previously inferred, the current analysis is suspended. The missing signature is inferred independently from the current context; the suspended analysis may then resume. The analysis is said to be *monovariant* (the signature inferred for a method may be used for operations occuring in different cases).

5.5.2.2 Iterative analysis

The iterative analysis requires three successive passes over the nodes of the case graph: one forward, one backward and forward a second time. Recall that the graph of operations and datalinks is sorted into a linear execution sequence (see 2.2.1). The three passes skip the `Input` and `Output` operations (often refered to as the input and output bars). The forward pass follows the execution sequence from the first operation after the `Input` operation until the last operation before the `Output` operation, the backward pass follows the reverse sequence. The purpose of each pass can now be described:

- The first forward pass infers the types for the outputs of the operation from the types of its inputs and the signature of the operation.
- The backward pass proceeds against the flow of data. It infers the type of the inputs of the operation from the type of its outputs and its signature. It also computes some information useful to type the following case.
- The second forward pass detects whether the types of the outputs of the case depend on the types of the inputs of the case.

A line can then be constructed that summarises the type information gathered during the case analysis with the input and output types of the case.

5.5.3 Implementation outline

Visual dataflow forcefully exposes program structures and the code for the inference mechanism provides a good outline of the algorithm.

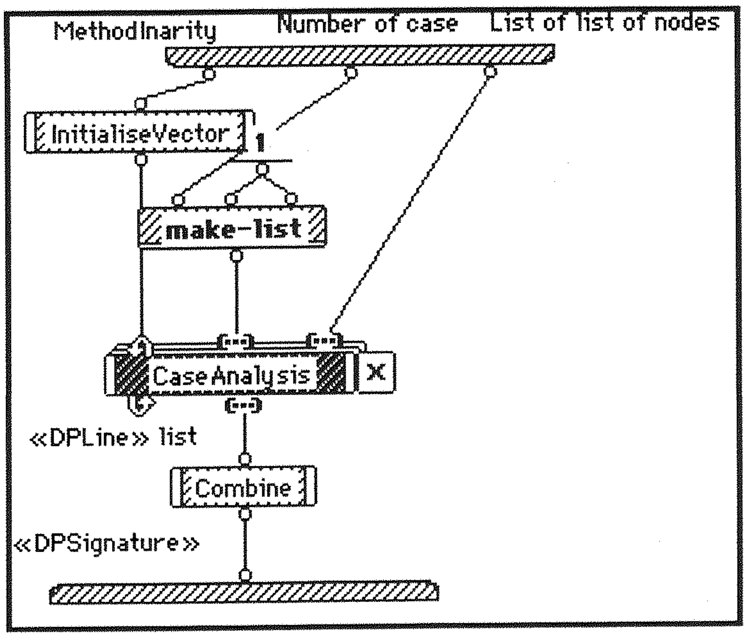


Fig. 5.8: Analysis of a method

Fig. 5.8 shows the analysis applied to a method. On the rightmost input of the case, the method is passed as a sequence of cases which in turn are represented as sequences of operations (also called nodes). The sequence of operations within a case follows their execution order. The local method `InitialiseVector` constructs a sequence of input types for the first case of the method. Most of the processing takes place in the `Case Analysis` local method. Each iteration of `Case Analysis` produces a sequence of input types for the next case on the left root of the operation and a line to describe the input and output types of the case analysed on the right root of the operation. When all the cases have been analysed, their respective lines are combined in the `Combine` local method to produce the signature of the method.

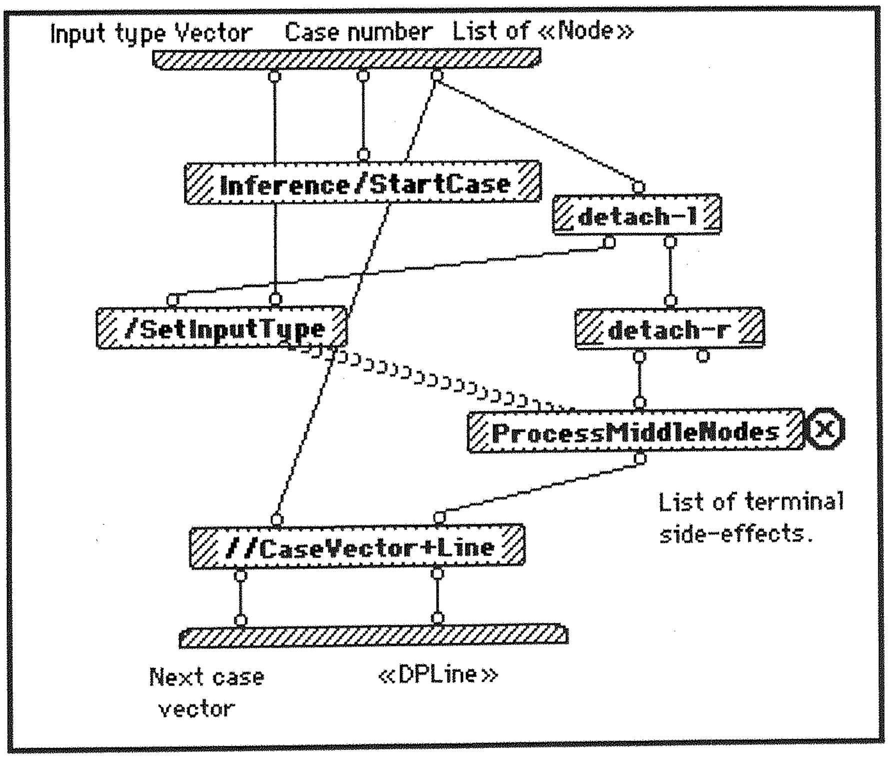


Fig. 5.9: The `CaseAnalysis` local method

Fig. 5.9 shows the implementation of the `CaseAnalysis` method. The `Inference/StartCase` operation performs some housekeeping activities. The `Input` and `Output` operations (respectively the first and the last elements in the sequence of nodes) are removed from the sequence and `SetInputTypes` operation sets the input types for the case. The `ProcessMiddleNodes` operation executes the different phases of the analysis on the operations between the `Input` and `Output` operations. The `CaseVector+Line` operation constructs the line with the input and output types for the case as well as the sequence of input types for the following case.

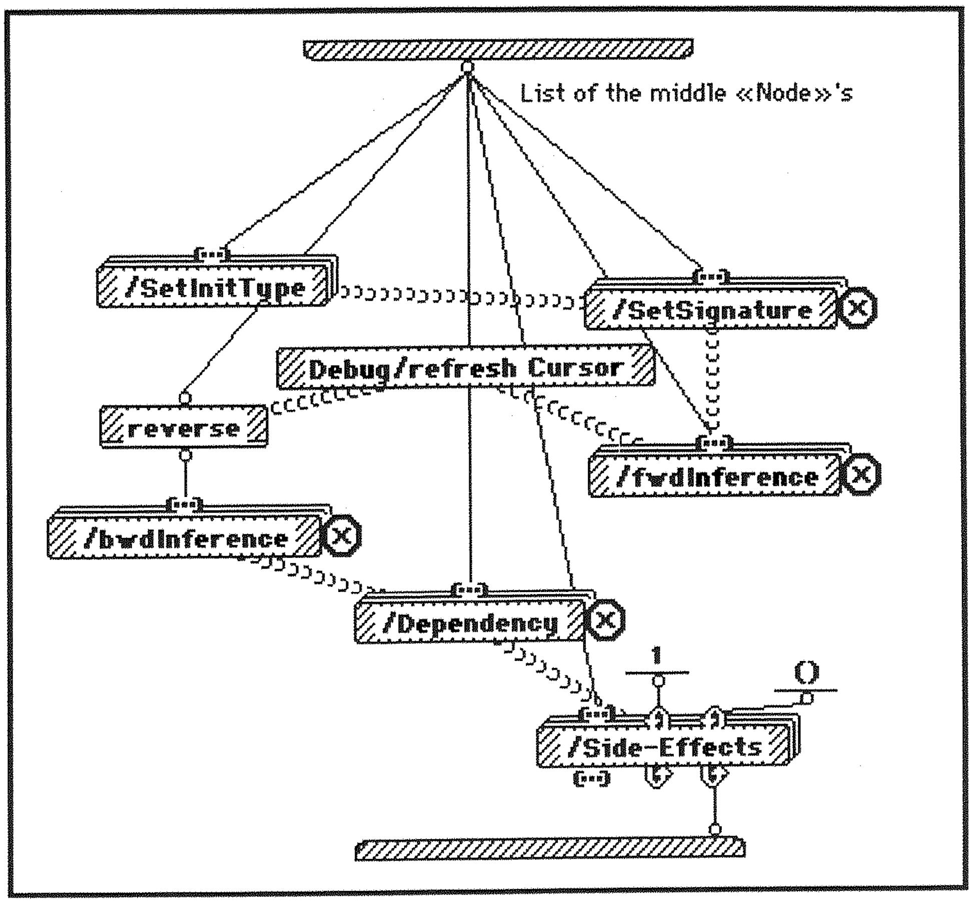


Fig. 5.10: The `ProcessMiddleNodes` method.

The operations of the case of `ProcessMiddleNodes` (fig. 5.10) carry out the different stages of the case wide inference. The `SetInitType` operation initialises the type information for the datalinks which are not connected to the roots of the `Input` operation. The `SetSignature` operation constructs the signature of the operations and the `fwdInference` and `bwdInference` operations perform the forward and backward analysis over the operations of the case currently analysed. `Dependency` checks the existence of a dependency between the input and output types of the case.

The `Side-Effects` operation is concerned with the effect signature of the operations of the case (effect inference is the subject of the next chapter of this thesis).

5.5.4 Properties of the algorithm

There are two properties of interest for a type inference algorithm:

- *Soundness* is the guarantee that if a program has been typed by the inference algorithm, it cannot fail because of a type error. Soundness is paramount when the motivation for type inference is type safety. The inference algorithm proposed for Prograph may infer a type signature for methods whose excution would result in a run-time type error. The algorithm proposed for Prograph also rejects type correct code.

• *Completeness* is the ability of an inference algorithm to infer the most general type of an expression. No claim is being made about the completeness of the type inference algorithm proposed for Prograph.

5.6 Prograph Types

Type inference in Prograph tries to achieve conflicting goals:

• The effect analysis requires concrete type information to describe the effects induced by a subcomputation. Similarly, in order to reduce the uncertainty caused by dynamic binding, it is necessary to produce concrete type information. As in [Suzuki 1981], [Johnson 1986] and [Palsberg and Schwartzbach 1991], the type of an object is the set of classes of which the object can be an instance. The notion of subtype and subset are equivalent.

• A method-wide inference must yield the most general signature. Sets of classes do not suffice to express all the possible uses of the method, there might exist dependencies between argument types or between argument types and return values. Dependencies between argument types often result from the use of an arithmetic or relational primitive. Dependencies between the input and output types of methods are often the result of operations that operate on lists or return one or several of their arguments. The proposed type system for Prograph allows explicit description of the dependencies between the input and output types of methods but not between input types.

5.6.1 Class hierarchy

Prograph distinguishes between Prograph data types and user-defined classes. It is, for example, not possible to create a subclass of `integer`. However, this distinction is not relevant for type inference and both the Prograph data types and user-defined classes can be considered as classes.

Classes are organised in a lattice (fig. 5.11). It is necessary to introduce a few extra classes to be able to construct the lattice. The top element of the lattice is called the `Universal` class and the bottom element `Bottom`. The left part of the lattice is fixed and consists of the Prograph data types. The right part is application dependent and consists of the user defined classes that are inserted below the `UDC` (`User Defined Class`) class. For the purpose of the effect analysis, it is not necessary to distinguish between the `real` and `integer` types and the two are indistinctively represented by the `number` class.

The ordering of the elements of the lattice is based on the *isSubclassOf* relation. The lattice respects the Prograph model of single inheritance except for the `Bottom` class which is the subclass of all classes.

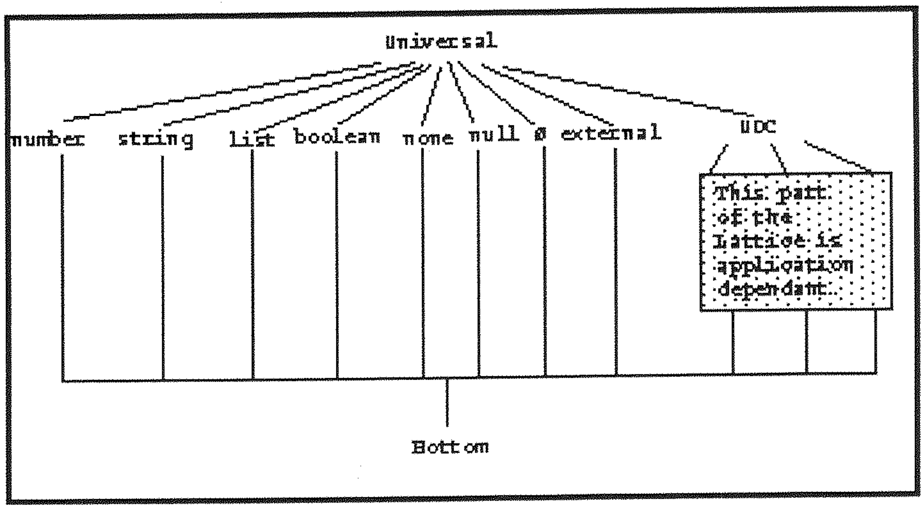


Fig. 5.11: The class lattice

5.6.2 Type

Types are sets of classes. To facilitate the analysis, these sets should be easy to describe and to manipulate. A suitable type representation should allow for parametrised types (e.g. lists).

A type can be one of the following alternatives.

5.6.2.1 Single Type

The type of a data object is the set of possible classes of which this data object can be an instance. The most trivial set is `{a, Bottom}` where `a` is a class. Such a set is denoted by `<a>` and is called a *single type*. However, because of inheritance in object-oriented languages, it is often necessary to designate not only a single class but the single class and all its subclasses, `<a+>` denotes the sublattice whose vertex is the class `a`.

5.6.2.2 String Type

`Get` and `Set` operations accept a string as a reference to a class. The value of the string is potentially useful information to describe effects.

The *string type* is a specialisation of the single type. A string type also holds the value of the string. Other type inference approaches often find it necessary to introduce such ad hoc types (e.g. strategic types in [Kind and Friedrich 1993]). The main motivation for introducing string types is to type the inputs of `Get` and `Set` operations more precisely.

It would have been possible to include a sublattice under the `string` abstract class. The elements of the sublattice would be the names of the user-defined classes and the ordering would be defined by the subclassing relation. However, a dynamic solution, where the ordering between two string values is computed when needed, has been preferred.

As with single types, it is necessary to distinguish between the string and the "subtypes" of the string. `"a"` designates the pair `{"a",Bottom}` with `"a"` a string whose value is a. `"a"+` designates the set formed by the string `"a"`, the strings whose values are the names of all the subclasses of `a` and the `Bottom` element.

5.6.2.3 List Type

A *list type* is by definition the pair `{List, Bottom}` but it is also parametrised by the type of the elements of the list object.

(τ) denotes a list whose elements have the type τ. Prograph allows lists to be heterogeneous.

5.6.2.4 Union Type

A single type is not always enough to represent the set of possible classes for a data object. A *union type* is the union of an arbitrary number of types. $[\tau_1 | \tau_2]$ is an example such a set. Union types cannot be nested. The analysis will not differentiate between heterogeneous lists and the union type of homogeneous lists, so $[(\tau_1) | (\tau_2)]$ must be expressed as $([\tau_1 | \tau_2])$.

5.6.3 Type dependencies

A *type dependency* expresses the fact that the output type of a polymorphic method may depend on one or several of the input types of the method. Type dependencies appear on the right hand side of a signature line.

A type dependency can be thought of as a function of the input types. When the dependency is evaluated, references to the input types are substituted with the actual types to compute a type for the output.

The evaluation of a type dependency is described by an `Evaluate` function. The signature of the `Evaluate` function is:

$$\texttt{Dependency} \times \texttt{Type*} \rightarrow \texttt{Type}$$

From the dependency and the sequence of input types, `Evaluate` produces the type of the output value.

The `Inverse` function of a dependency computes the input types in a signature line using the value of the dependency, the signature of `Inverse` is:

$$\texttt{Type} \times \texttt{Dependency} \times \texttt{Type*} \rightarrow \texttt{Type*}$$

`Inverse` takes the type which corresponds to the value of the dependency, the dependency and the sequence of input types and returns the sequence of updated input types.

Type dependencies are not only used in the lines of an operation signature. As will be explained later, they can also be attached to the datalinks of a case in order to detect whether the type of the output of the case depend on the type of the inputs of the case.

Five type dependencies are available. They can be composed in order to express any possible dependency.

5.6.3.1 Input

The *input* type dependency expresses the most trivial type dependency between an input and an output. The type of the output is the same as the type of the designated input numbered from the left. In the line of an operation signature, `Id(1)` means that the type of the output is the same as the type of the first input of the operation. When attached to a datalink, `Id(1)` means that the type of the value on this datalink is the type of the first input of the case.

The input dependency provides the same functionality as Milner's type variables as shown below:

$$\texttt{<Universal+>} \rightarrow \texttt{Id(1)}$$

is equivalent to the following function type in Milner's type system:

$$\alpha \rightarrow \alpha$$

It can also be used to express bounded universal polymorphism:

`<a+>` $\rightarrow$ `Id(1)`

is equivalent to Kind's qualified type variables in the line:

$$\alpha^{<a>} \rightarrow \alpha^{<a>}$$

5.6.3.2 Element

The *element* dependency returns the type of the element of the list coming onto the designated input. `E(1)` means that the type of the output is the type of the elements of the list on input 1.

For example, the primitive `detach-r` detaches the rightmost element of its input list. The element dependency is needed for the type signature of `detach-r`.

The line:

`(<Universal+>)` $\rightarrow$ `E(1)`

is equivalent to the following function type in Milner's type system:

α `list` $\rightarrow \alpha$

5.6.3.3 List

The *list* dependency designates that the output is a list whose elements have their type dependent on the type of the inputs.

For example, the primitive method `pack` can take an input `a` and returns a list with `a` as a single element. List is used in the signature of `pack`. `L(`δ`)` denotes the list dependency applied to the type dependency δ.

The line:

`<Universal+>` $\rightarrow$ `L(1)`

is equivalent to the following function type in Milner's type system:

$\alpha \rightarrow \alpha$ `list`

5.6.3.4 Union and Intersection

The *union* and *intersection* dependencies, denoted `U` and `I` respectively, return the union or the intersection of several types and type dependencies.

The primitive `find-instance` takes a list of objects, an attribute name and a value and returns the index in the list of the first object for which the named attribute has the required value, the object itself is returned as the second output of the primitive. If no object is found, `0` is returned for the index and `NULL` for the found object. The type of the second output of `find-instance` is described by the dependency:

```
U(I(E(1) <UDC+>) <null>)
```

5.6.4 Operations on types and dependencies

To evaluate and invert type dependencies during the iterative analysis, it is necessary to compute the union ($\cup$) and intersection ($\cap$) of types and/or of type dependencies

Such computations rely heavily on the predicates $\subseteq$ and ?.

$\subseteq$ is defined for any pair of types and/or type dependencies. It is equivalent to set inclusion. However if the intersection of the two type expressions cannot be computed, $\subseteq$ returns `FALSE`.

`Id(1)` $\subseteq$ `<Universal+>` = `TRUE` because whatever type `Id(1)` is evaluated to, `<Universal+>` will include it, by definition.

`Id(1)` $\subseteq$ `<number>` = `FALSE`, because the overlap between `<number>` and `Id(1)` cannot be computed, this overlap depends on the type to which `Id(1)` is evaluated.

`<number>` $\subseteq$ `<none>` = `FALSE` because the two types do not overlap.

A second predicate, ?, is also defined for any pair of types and/or type dependencies. This predicate returns true when the intersection of two sets cannot be computed e.g. `<number>?E(1)` = `TRUE`.

The use of the union and intersection operations is illustrated by the following examples:

`<boolean>` $\cup$ `<number>` = `[<boolean> | <number>]`

`<boolean>` $\cap$ `<number>` = `<Bottom>`

`(<boolean>)` $\cup$ `(<number>)` = `([<boolean> | <number>])`

`(<boolean>)` $\cap$ `(<number>)` = `<Bottom>`

Where a is a user defined class:

`<a>` $\cup$ `<UDC+>` = `<UDC+>`

`<a>` $\cap$ `<UDC+>` = `<a>`

`<a>` $\cup$ `Id(1)` = `U(<a> 1)`

`<a>` $\cap$ `Id(1)` = `I(<a> 1)`

5.6.5 BNF for type expressions

The set of valid type expressions is given using the Backus-Naur form (BNF):

```
SingleType::<a> | <a+>

StringType:: "a" |"a"+

SimpleType::SingleType|StringType

ListType :: (SimpleType) | (UnionType)| (ListType)

UnionType:: [SimpleType | SimpleType+]
| [ListType | SimpleType+]

n:: Integer

InputDependency :: Id(n)

Element Dependency :: E(n)

UnaryDependency :: InputDependency | ElementDependency

ListDependency :: L(n) | L(Union Dependency) |
L(IntersectionDependency)

UnionDependency :: U(UnaryDependency UnaryDependency+)
| U(UnaryDependency+ SimpleType+)
| U(UnaryDependency+ ListType)
| U(UnaryDependency+ SimpleType+ ListType)
| U(IntersectionDependency IntersectionDependency+)
| U(IntersectionDepency+ SimpleType+)
| U(IntersectionDependency+ListType)
| U(IntersectionDepency+ SimpleType+ ListType)
| U(UnaryDependency+ IntersectionDependency+)
| U(UnaryDependency+ IntersectionDepency+ SimpleType+)
| U(UnaryDependency+ IntersectionDependency+ ListType)
| U(UnaryDependency+ IntersectionDepency+ SimpleType+ ListType)
```

```
IntersectionDependency :: I(UnaryDependency+)
| I(UnaryDependency+ SimpleType)
| I(UnaryDependency+ ListType)
```

5.7 Operation Signatures

The outline of the type inference algorithm explained that the analysis of the cases of a method is divided into two distinct phases: the initialisation phase and the iterative analysis phase. During the intialisation phase, a signature is constructed for every operation of the case (except for the `Input` and `Output` operations). This signature depends on the nature of the operation. The rules to derive the signatures of the operations are given in the following subsections.

5.7.1 Simple operation

A simple operation can be a call to a primitive or a user-defined method. Primitives' signatures cannot be inferred, they must be available so that they can be used to type any operation calling a primitive. The signatures of the primitive methods are explained in 5.7.2.

In the case of user-defined methods, the signature of the operation will depend on the sort of reference used (i.e. *universal, data-determined, context-determined* or *explicit* reference).

5.7.1.1 Call with a universal reference

With a universal reference, the signature of the operation is the signature of the universal method called. If the signature is not available it must be inferred separately.

5.7.1.2 Call with a context determined reference

An operation with a context determined reference can only be found in the case of a method defined by a class. The operation calls the method applicable to the class of the method which contains the operation. The signature of the applicable method is used as the operation signature.

When the operation with a context determined reference is super annotated, the signature of the operation is the signature of the method applicable to the superclass of the class of the method which contains the operation.

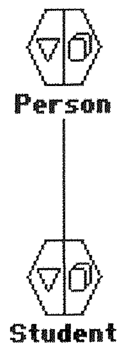
5.7.1.3 Call with a data-determined reference

With a data-determined reference the name of the operation does not suffice to determine which method is going to be called at run-time. As in [Suzuki 1981] and [Kind and Friedrich 1993], the solution is to construct the signature of the operation by joining the signatures of all the methods that may be called by the operation. The set of the methods that may be called comprises all the simple class methods with the name and arity of the calling operation and possibly a universal or a primitive method with the name and the arity of the calling operation.

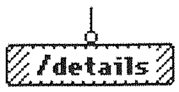
For each method potentially called, the leftmost input type (that is the type of the *receiver* in object-oriented terminology) in each line of the method's signature must be restricted to the set of classes to which the method is applicable. Restricting the type of the receiver means computing the intersection of the type in the line of the method with the set of classes to which the method is applicable.

This rule is quite simple to understand: if a method is to be called by an operation, the leftmost input of this operation must be an instance of a class to which the method is applicable. A parallel can be drawn between this rule and the edge conditions in [Palsberg and Schwartzbach 1991].

The rule is illustrated with a simple example. The `Person` class defines a method called `details`. The `Student` class is a subclass of the `Person` class.



The `Student` class redefines the `details` method. An operation calling the `details` method occurs in the case of a method:



At this stage of the analysis it is not possible to say whether the `details` method defined for the `Person` class or the `details` method defined for the `Student` class will be called. The signature of the `details` operation consists of the combined signatures of the two methods. The signature of the `details` method for the `Person` class is:

$$\texttt{<Person+>} \rightarrow$$

(Note that an instance of the `Student` class would be a valid argument for the `details` method defined for the `Person` class).

The signature of the `details` method defined for the `Student` class is:

$$\texttt{<Student>} \rightarrow$$

However, when constructing the signature of the details operation, it must be remembered that the `details` method defined for the Person class will be called if the type of the receiver contains the class `Person` or subclasses or `Person` which do not redefine the `details` method. As the immediate subclass of `Person`, the `Student` class redefines the `details` method, the type of the receiver must be `<Person>`. The `details` method redefined by the `Student` class will be called only if the receiver of the operation has the type `<Student>`. The signature of the `details` operation comprises the line of the signature of the `details` method defined for the `Person` class (the leftmost type of the line is restricted to `<Person>`) and the line of the signature of the `details` method defined for the `Student` class (the leftmost type of the line is restricted to `<Student>`):

$$\texttt{<Person+>} \cap \texttt{<Person>} \rightarrow$$
$$\texttt{<Student>} \cap \texttt{<Student>} \rightarrow$$

which is:

$$\texttt{<Person>} \rightarrow$$
$$\texttt{<Student>} \rightarrow$$

5.7.2 Primitive method signatures

The signatures for the primitive methods are stored in a repository and can be looked up using the name of the primitive. During the initialisation phase of the case-wide type inference, the signature of a primitive method is retrieved from the repository and associated with the operation that may call the primitive method.

Some primitives have optional inputs or outputs. For example, the primitive `(in)` takes a list, a data item and optionally the value of a start index as inputs and returns the index of the first occurrence after the start index of the data item in the list or 0 if the item is not found. The signature of `(in)` requires two lines. The first line describes the primitive `(in)` as a method with two inputs and one output, the second line describes an operation with three inputs and one output.

The signature of `(in)` is:

`(<Universal+>)` × `<Universal+>` → `<number>`

`(<Universal+>)` × `<Universal+>` × `<number>` → `<number>`

When the signature of the `(in)` operation is initialised, the line where the number of inputs does not match the number of inputs of the calling operation is discarded.

Other primitives may also have an arbitrary number of inputs or outputs. The + (number addition) primitive is a good example. The variable number of inputs cannot conveniently be represented by multiple lines as the number of inputs may vary between 2 and 256. A new concept must be introduced, that of a *Varity*. A varity term (denoted by ...) in a line means that the left hand side or the right hand side of the line may be extended by duplicating the type next to the varity term.

The signature of + is:

`<number>` × `<number>` × ... → `<number>`

When the signature of the + operation is constructed during the initialisation phase, the varity term must replaced with the required sequence of `<number>` types to match the arity of the calling + operation in the case.

5.7.3 `Get` and `Set` operations

Attributes are supported by a finite set of classes. Therefore, the name of an attribute accessed or modified gives a useful indication of the type of the leftmost argument of the `Get` or `Set` operation.

If a default `Get` or `Set` operation is called then the type of the receiver includes all the classes defining or inheriting the attribute and the string types with values that are the names of these classes. The inference algorithm does not attempt to keep track of the types of the class and instance variables.

A `Get` or a `Set` operation may also have a data-determined reference. The signature is obtained by joining the signatures of the default and user-defined methods that may be called. As for a simple operation with a data-determined reference, the leftmost type of the lines must be narrowed to the set of classes (and the names of these classes) to which a user-defined `Get` or `Set` method is applicable.

The signature of a user-defined `Get` or `Set` method is inferred independently.

5.7.4 Instance generator

The signature of an operation instantiating a new object can be constructed on the fly if a default instance generator is called:

```
<none> → <Z>
((<Universal+>)) → <Z>
```

where Z is the name of the operation. In the second line, the list of lists corresponds to the optional list of (attribute name, attribute value) pairs that can be passed to the `Init` operation.

If a custom instance generator has been defined, its signature has to be inferred independently.

5.7.5 Persistent operations

Very little type information can be inferred from a persistent `Get` or a persistent `Set` operation as the algorithm does not record the type of persistent values. In the case of a persistent `Set` operation, the type of the input is set to `<Universal+>`; for a persistent `Get`, the type of the output is also set to `<Universal+>`.

5.7.6 `Local` operations

The signature of a `Local` operation is obtained by inferring the signature of the local method attached to it.

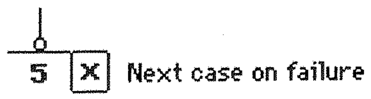
5.7.7 `Constant` operations

The type signature of a `Constant` operation is a line with no input and the type of the constant value as output.

5.7.8 `Match` operations

A `Match` operation can have different controls attached to it: `NextCase`, `Finish`, `Terminate`, `Fail` and `Continue`. The control can be triggered when the match fails, or on the contrary when it succeeds. For the purpose of the analysis, it would be interesting to keep track of the type that would trigger the activation of any control. However, this would result in an increased complexity. The algorithm keeps track only of the type of the value that might trigger a NextCase control. This type, called NextType, is a property of the datalink connected to the terminal of the Match operation. The rules are:

• If the NextCase control is activated on a failed match, the signature consists of a single line whose argument type is that of the value that must be matched and no return type; NextType is set to <Universal+> for the datalink coming into the Match operation. This case is illustrated below:



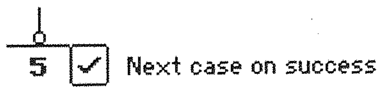
The execution will resume at the input operation in the following case if the value on the incoming datalink of the Match operation is not equal to 5. That is any value other than 5, and by extension a value of any type (including number) can trigger the control. Therefore NextType is set to <Universal+>. If the execution is to proceed in the current case, the value on the datalink must be 5 and the signature of the Match operation should be:

$$\texttt{<number>} \rightarrow$$

• For a NextCase control activated on a successful match, the signature constructed for the Match operation is:

$$\texttt{<Universal+>} \rightarrow$$

NextType is set to the type of the value to be matched. In the following example:



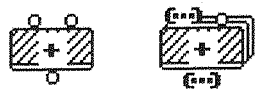
NextType would be set to <number>.

This rule becomes slightly more complex when the same value flows into several Match operations. This situation occurs when the datalinks connected to the terminals of different Match operations are connected to the same root. The NextType of the datalinks is set to the union of the types propagated by the differnt Match operations (this rule is explained in greater detail in 5.8.2.4).

5.7.9 Signature of multiplex operations

The clean separation between an operation and the different multiplex annotations that can be applied to the terminal and roots of an operation or the operation itself makes the typing of an annotated operation relatively easy.

In the case of an operation with list annotated terminals and roots, the type signature of the unannotated operation can be specialised by converting the type of list terminals and list roots into list types or list type dependencies. The transformation is illustrated by the following example:



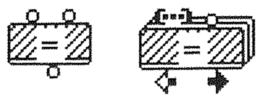
The signature of `a` is:

`<number> × <number> → <number>`

The transformation of the signature of `a` yields for `b` the signature:

`(<number>) × <number> → (<number>)`

The typing of a partition annotated operation is slightly more complex. There exists a dependency between the type of the list being partitioned and the types of the *fail* and *pass* lists.



The signature of `a` is:

`<Universal+> × <Universal+> → <boolean>`

it becomes for `b`:

`(<Universal+>) × <Universal+> → L(U(E(1)<ø>)) × L(U(E(1)<ø>))`

where `<ø>` is a type such that `(<ø>)` is the empty list type.

Other multiplex annotations do not affect the signature of the operation.

5.8 Type inference algorithm

This section explains in greater detail the inference of a method signature. The first subsection describes the start of the method-wide analysis. Most of the analysis occurs in the scope of the individual cases of the method and the second subsection covers the different steps of the case analysis. The synthesis of the method signature from the results of the case analyses is then explained. The last subsection presents the analysis of recursive methods.

5.8.1 Method wide analysis

The inference is applied to one method at a time. It is necessary to be able to identify precisely the method to which the type analysis must be applied. Name overloading and the existence of different types of methods (`Set`, `Get`, `Simple`, `Init` and `Local`) requires a combination of three components to identify a method.

A *method identifier* identifies a method using the following *ClassName/MethodName/MethodType* triplet.

- *ClassName* is the name of the class to which the method belongs, for a universal method, the `Universal` keyword is used instead.
- *MethodName* is the name of the method, `Init` methods are designated by the «» characters. The name of a local method is constructed from the name of the containing method and the name of the local method (if it has one).
- *MethodType* distinguishes among the various method types: `Set`, `Get`, `Simple` and `Local` (there is no `Init` type because custom `Init` methods can be distinguished by their name).

The inference mechanism maintains a stack of method identifiers during the analysis. When the analysis is applied to a method, its method identifier is pushed onto the stack. The method identifier on the top of the stack corresponds to the analysis currently active, and all the identifiers occurring in the stack are those of the methods for which analysis has been suspended. Upon completion of the analysis of a method, its identifier is popped from the top of the stack. The stack of method identifiers serves two purposes, it detects possible rescursion in the method currently analysed and, in case of failure, helps to localise at which point the inference failed.

5.8.2 Case wide analysis

Most of the computations to infer the type signature of a method take place in the scope of the individual cases of the method. This subsection describes in detail the different stages of the case analysis:

- The initialisation phase

- The forward analysis
- The backward analysis
- The computation of the `NextType` info for the datalinks of the case
- The computation of the type dependencies between the inputs and outputs of the case.

5.8.2.1 Initialisation phase

During the initialisation phase, the information attached to the datalinks and the signatures of the nodes are set up.

The information inferred about the value flowing on a datalink or a set of datalinks is described by a tuple: (`Type`, `NextType`, `Dependency`). The purpose of each field is now explained:

- The value of `Type` is an approximation of the type of the data object on the datalink.
- The value of `NextType` is the type the data object should have if a `NextCase` control is to be activated.
- `Dependency` keeps track of the dependencies between the types of the objects on the graph.

During initialisation, seven categories of datalinks can be distinguished. The first four categories are defined by the possible combination of two parameters: connection of the datalink to the input bar and connection of the datalink to the terminal of a `Match` operation. For the purpose of the analysis, it has been necessary to introduce three extra categories of datalinks, the first one is called `NotConnectedTerminal` and handles unconnected operation terminals. The second extra category, called `NotConnectedRoot`, is required to deal with a root which has no datalink connected to it. The third extra category is used for the roots of the `Input` operation which are not connected.

The rules used to construct the signatures of the operations have been described in section 5.7.

The table below shows the values of the properties attached to the datalinks after the initialisation phase:

	Type	NextType	Dependency
Not connected to Match Connected to Input	τ	NoInfo	Id(i)
Connected to Match Connected to Input	τ	τ'	Id(i)
Not connected to Match Not connected to Input	<Universal+>	NoInfo	–
Connected to Match Not connected to Input	<Universal+>	τ'	-
NotConnectedTerminal	<none>	NoInfo	-
NotConnectedRoot	<Universal+>	NoInfo	_
NotConnectedRoot of Input operation.	τ	NoInfo	-

τ , the value of Type for a datalink connected to the input bar can be:

- <Universal+> if the case being analysed is the first case of the method.
- the NextType of the matching input in the previous case if NextType is not NoInfo (NoInfo means that there is no information available to type the next case)
- the Type of the matching input in the previous case otherwise.

τ', the NextType of a datalink connected to a Match operation is set during the construction of the signature of the Match operation (according to the rules described in 5.7.8).

Id(i) is the Dependency value for a datalink connected to the input bar where i is the position of the input in the sequence of the inputs of the case (1 is the leftmost input of the case). Id(i) means that the type of the datalink connected to the input bar is the type of the input of the case.

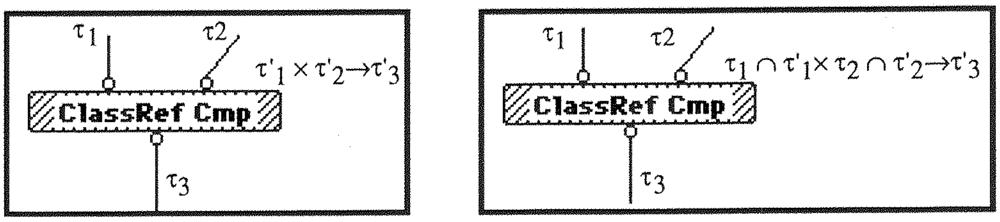
5.8.2.2 Forward Analysis

After the initialisation phase, the signature of each operation comprises one or more lines. The role of the forward analysis is to infer the value of `Type` for the outgoing datalinks (i.e the type of the value flowing on the datalink) from the value of `Type` for the incoming datalinks and the signature of the operation.

For each operation of the case (except the input and the output bars), following the execution order, the forward analysis is performed in two stages:

- The update of the lines of the signature of the operation
- The update of the values of `Type` for the outgoing datalink of the operation.

The pairwise intersections of the types of the incoming datalinks, (the values of `Type` for the incoming datalinks, τ_1 and τ_2 in fig. 5.12) and the matching input types in the line (τ'_1 and τ'_2 in fig. 5.12) are computed. If, for one pair, the intersection yields `<Bottom>`, the line is *disqualified* as a whole, otherwise, the input types in the line slots are replaced by the intersection set ($\tau_1 \cap \tau'_1$ and $\tau_2 \cap \tau'_2$ are the two intersection sets in fig. 5.12). This update is repeated for each of the line of the signature.



τ_1
$\tau 2$
$\tau'_1 \times \tau'_2 \rightarrow \tau'_3$
ClassRef Cmp
τ_3
τ_1
$\tau 2$
$\tau_1 \cap \tau'_1 \times \tau_2 \cap \tau'_2 \rightarrow \tau'_3$
ClassRef Cmp
τ_3

Fig. 5.12: Intersection of the incoming types with the input types of the lines.

The purpose of the second stage is to update the value of `Type` for the outgoing datalinks of the operation.

Each line propagates a new type for each outgoing datalink of the operation. If an output slot in a line contains a type dependency, the type to be propagated is the result of the evaluation of the type dependency, otherwise it is the type stored in the output slot of the line.

Eventually, the updated value of `Type` for each outgoing datalink is the union of the types propagated by the different lines for that datalink.

Fig. 5.13 illustrates the most trivial case, a signature with a single line whose output slot contains a type. τ_3 is replaced with τ'_3 in fig. 5.13.

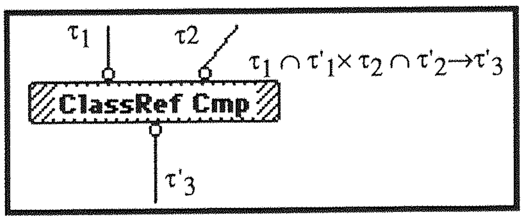

τ_1
$\tau 2$
$\tau_1 \cap \tau'_1 \times \tau_2 \cap \tau'_2 \rightarrow \tau'_3$
ClassRef Cmp
τ'_3

Fig. 5.13 Propagation of the output types.

If after update, the signature of the the operation shown in fig. 5.12 was:

$$\tau_1 \cap \tau'_1 \times \tau_2 \cap \tau'_2 \rightarrow \delta_1 \text{ (}\delta_1 \text{ is a type dependency).}$$

τ'_3, the updated value of `Type` for the outgoing datalink of the operation would be:

$$\texttt{Evaluate}(\delta_1, (\tau_1 \cap \tau'_1, \tau_2 \cap \tau'_2))$$

To illustrate the most general case, if after its update the signature of the operation shown in fig. 5.12 comprised the two following lines:

$$\tau_1 \cap \tau'_{1a} \times \tau_2 \cap \tau'_{2a} \rightarrow \delta_{1a}$$
$$\tau_1 \cap \tau'_{1b} \times \tau_2 \cap \tau'_{2b} \rightarrow \tau'_{3b}$$

τ'_3, the updated value of `Type` for the outgoing datalink of the operation would be the union of the value of the type dependency δ_{1a} and the type τ'_{3b}:

$$\texttt{Evaluate}(\delta_{1a}, (\tau_1 \cap \tau'_{1a}, \tau_2 \cap \tau'_{2a})) \cup \tau'_{3b}$$

In summary, at the end of the forward analysis, the value of `Type` for each datalink of the case has been updated once, except for the datalinks connected to the input bar.

5.8.2.3 Backward Analysis

The role of backward analysis is to infer the value of `Type` on the input datalinks from the values of `Type` on the output datalinks and the signature of the operation. The likely presence of type dependencies in the output slots of the signature lines makes the backward analysis more complex than the forward analysis.

For each operation of the case (except the input and the output bars), following the reverse execution order, the backward analysis is performed in two stages:

- The update of the lines of the signature of the operation

- The update of the value of `Type` for the incoming datalinks of the operation.

Each line in the signature is updated as follows. In each line, each output slot contains either a type or a type dependency.

If the output slot of the line contains a type, the intersection of that type with the type of the matching outgoing datalink is computed and becomes the type stored in the output slot of the line (this is similar in principle to the first stage of the forward analysis, except that the slots whose types are being updated are now on the right hand side of the line).

If the output slot of the line contains a type dependency, the pairwise intersection of the last computed value of the type dependency ($\texttt{Evaluate}(\delta_1, (\tau'_1, \tau'_2, \tau'_3))$ in fig. 5.14) and the type of the corresponding datalink (τ_4 in fig. 5.14) must be computed and the intersection becomes the new value of the type dependency (τ'_4 in fig. 5.14).

As with forward analysis, an intersection producing a `<Bottom>` result disqualifies the entire line.

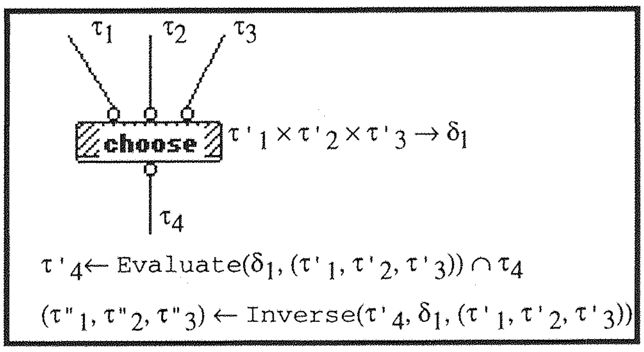

τ_1
τ_2
τ_3
choose
$\tau'_1 \times \tau'_2 \times \tau'_3 \rightarrow \delta_1$
τ_4
$\tau'_4 \leftarrow \texttt{Evaluate}(\delta_1, (\tau'_1, \tau'_2, \tau'_3)) \cap \tau_4$
$(\tau''_1, \tau''_2, \tau''_3) \leftarrow \texttt{Inverse}(\tau'_4, \delta_1, (\tau'_1, \tau'_2, \tau'_3))$

Fig. 5.14: Update of the line of the signature of the operation

The next step in the update of each line is to obtain the revised input types of the line.

If the output slots of the line contain only types, the input types of the line are left unmodified.

If the output slots of the lines contain type dependencies, the change in the value of the type dependencies (τ'_4 is the new value of dependecy δ_1 in fig. 5.14) must be reflected in the input types of the line. The application of the `Inverse` function to a dependency produces an updated sequence of input types for the line ($(\tau''_1, \tau''_2, \tau''_3)$ in fig. 5.14). If the same line has several output slots containing type dependencies, the same number of sequences of updated input types will be produced. The sequences are combined into one by computing the union of the matching elements of the sequences. The following example summarises the update of the lines of the signature of an operation during the backward analysis. A line contains two types dependencies δ_1 and δ_2:

$$\tau'_1 \times \tau'_2 \times \tau'_3 \rightarrow \delta_1 \times \delta_2$$

The inversion of δ_1 produces the sequence of input types $(\tau''_{1.1}, \tau''_{1.2}, \tau''_{1.3})$ and the inversion of δ_2 produces the sequence of input types $(\tau''_{2.1}, \tau''_{2.2}, \tau''_{2.3})$, the two sequences are combined to construct the updated line:

$$\tau''_{1.1} \cup \tau''_{2.1} \times \tau''_{1.2} \cup \tau''_{2.2} \times \tau''_{1.3} \cup \tau''_{2.3} \rightarrow \delta_1 \times \delta_2$$

After all the lines of the signature of the operation have been updated, the changes must be propagated to `Type` values of the incoming datalinks of the operation.

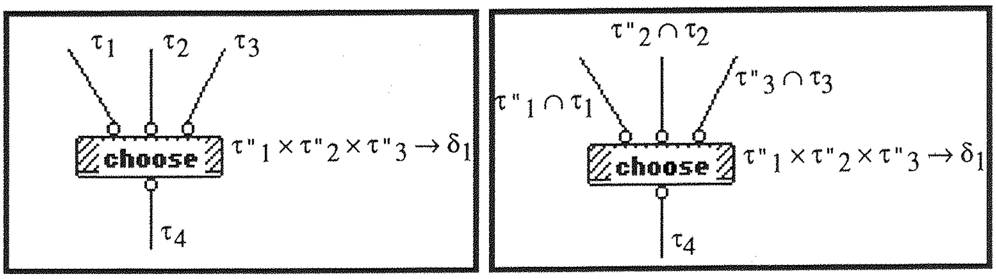


Fig. 5.15 a&b: Propagation of the types on the input datalinks.

If the signature of the operation contains several lines, the matching input types of the lines are combined using the union operator to produce a sequence of update types for the incoming datalinks of the operation.

In order to understand how the types on the incoming datalinks of the operation should be updated, it must be remembered that all the datalinks connected to the same root share the same `Type` value. The update input types are propagated upward by computing, for each datalink, the intersection of the current type of the datalink (τ_1 to τ_3 in fig 5.15.a) and the matching update input type (τ''_1 to τ''_3 in fig. 5.15.b).

The need to compute the intersection to propagate types upward is illustrated by the example in fig. 5.16.

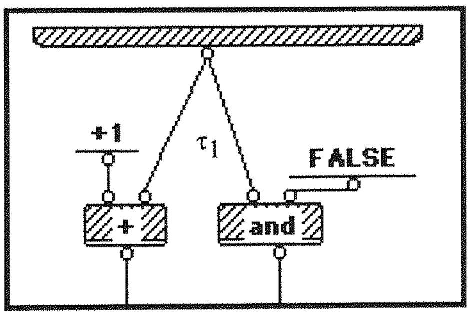


Fig. 5.16: Type incorrect code fragment.

The fragment of code shown in fig. 5.16 is not type-correct. During the forward analysis, τ_1 is set to `<Universal+>`. If the backward analysis processes the `and` operation first, τ_1 will become `<Boolean>`. When processing the + operation, the analysis must propagate the information that the + operation requires an input of type `<number>`. The intersection of `<number>` and `<boolean>` produces `<Bottom>` and the analysis fails.

In summary, the backward analysis updates the value of `Type` for all the datalinks except for those connected to the `Output` operation.

The combined effect of the forward and backward analyses is that the value of `Type` for each datalink of the case has been updated at least once (moreover, the value of `Type` of all the datalinks of the case that are connected to neither the roots of the `Input` operation nor the terminals of the `Output` operation has been updated twice). The optimal number of passes to yield an approximation of the types is likely to depend on the topology of the dataflow graph. The Kaplan and Ullman algorithm [Kaplan and Ullman 1980] reiterates forward and backward passes until a fixed point is reached; the drawback of this approach is its computational cost. The approach chosen for Prograph favours a lesser computational cost at the expense of the precision of the analysis.

5.8.2.4 Computing `NextType`

The purpose of `NextType` is to gather type information about the input values of a case (cf. 5.7.8). There may exist a correlation between the type of the inputs of a method and the sequence of cases visited during its execution. The operation to which the `NextCase` control is attached may not be directly connected to the input bar. It is therefore necessary to compute the value of `NextType` for all the datalinks on the graph.

The value of `NextType` can be a type or a `NoInfo` token. The value of `NextType` for each datalink is computed during the backward pass of the case analysis. The procedure described below is applied to all the operations (except the input and output bars) on the dataflow graph of the case.

In each line of the signature of the operation, each output slot contains either a type or a type dependency.

If the slot contains a type, no further action is necessary.

If the slot contains a type dependency and the value of `NextType` for the matching outgoing datalink is `NoInfo`, no further action is required. Otherwise, the type dependency is inverted with the value of `NextType` for the outgoing datalink to produce an update sequence for the `NextType` values of the incoming datalinks. Each element of the update sequence is either a type or a `NoInfo` token. If several update sequences are produced by the same and/or different lines of the signature of the operation, they are combined by performing the union of their matching elements (a `NoInfo` token acts as a neutral element for the union).

As for the explanation of the backward analysis, it is important to remember that all the datalinks connected to the same root share the same `NextType` value. The types (or `NoInfo` tokens) in the combined update sequence are propagated upward by computing for each datalink the union of the value of `NextType` with the value of the matching element of the sequence. The need to compute the union to update the value `NextType` is illustrated by the example in fig. 5.17

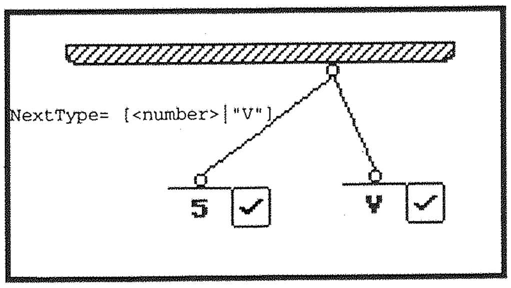


Fig. 5.17: Union of `NextType` values

In the example shown, the computation will resume in the following case if the input is the integer value 5 or is the string `"V"`. Thus, the `NextType` value for the two datalinks connected to the input must be the union type of the string `"V"` and `number`, `[<number> |"V"]`.

5.8.2.5 Computing intra-case type dependencies

The purpose of the last property attached to the datalinks of the graph, the `Dependency` property, is to record dependencies between the input types and the output types of the case being analysed. The value of `Dependency` is computed during a second forward pass over the nodes of the case.

The procedure described below is applied to all the operations of the case (except the `Input` and `Output` operations) following the execution order.

In each line of the signature of the operation, each output slot contains either a type or a type dependency.

If the output slot contains a type, this type becomes the value of `Dependency` for the matching outgoing datalinks.

If the output slot contains a type dependency, this dependency must be *composed*. Composing a data dependency means that all references to operation inputs that occur in the dependency are replaced with the value of `Dependency` for the corresponding incoming datalink. If all the referenced `Dependency` values are types, the substitution will logically yield a type, otherwise it yields a composed type dependency. The result of the substitution becomes the value of `Dependency` for the matching outgoing datalink.

The computation of the `Dependency` value is illustrated by the following example (fig. 5.18). δ_1, δ_2 and δ_3 are the `Dependency` properties attached to the three datalinks of the graph.

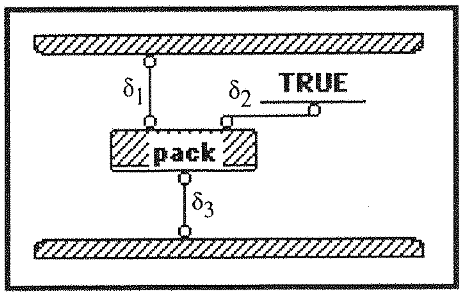


Fig. 5.18: Computation of the `Dependency` property

The signature constructed for `pack` during the initialisation phase is:

`<Universal+>` × `<Universal+>` → `L(U(1 2))`

δ_1 is `Id(1)` and δ_2 is `<boolean>`. When δ_3 is computed, the reference to the input `1` and `2` in the output type dependency of `pack` are substituted with the values of δ_1 and δ_2 and the value obtained for δ_3 is:

```
L(U(1 <boolean>))
```

When the signature of an operation comprises more than one line, the type dependencies are composed line by line and the value of `Dependency` for each outgoing datalink of the operation is obtained by performing the union of the composed dependencies.

The computation of the type dependencies across the case is the last pass over the dataflow graph before the case analysis completes.

5.8.2.6 Construction of the line for the case

Once all the passes have been carried out, a line can be constructed to describe the input and output types of the case.

The `Type` values for the datalinks connected to the `Input` operation become the input types of the case.

The `Dependency` values (whether a type or a type dependency) for the datalinks connected to the `Output` operation are used directly to describe the types of the outputs in the case line.

It is also necessary to construct the sequence of input types for the following case. This sequence is constructed by examining the values of `NextType` and `Type` for the datalinks connected to the roots of the `Input` operation of the case currently analysed. For each root of the `Input` operation, two cases can be distinguished, either the `NextType` value is `NoInfo` or the `NextType` value is a type.

- If the `NextType` value for the datalink(s) connected to the root is a type, the `NextType` value is used in the sequence for the next case.
- Otherwise, when `NextType` is `NoInfo`, the `Type` value of the datalink(s) connected to the root is used in the sequence For a root of the `Input` operation with no datalink connected to it (a `NotConnectedRoot`, see 5.8.2.1), the `NextType value` is `NoInfo`.

5.8.3 Synthesis of the method signature

The signature of the method is obtained by combining the lines produced by the analysis of the cases of the method into a single line. Each slot of the combined line contains the union of the corresponding types and type dependencies from the various lines.

5.8.4 Handling recursion

Recursion in Prograph may occur in different guises. Most simply, a universal method is recursive if it is called again directly or indirectly by an operation in one of its cases.

A more subtle form of a recursion is the consequence of Prograph object-orientation. An operation with a data-determined reference may appear in one of the cases of a method and have the same name as the method. The type of the receiver of the operation will determine whether the method is recursive or not.

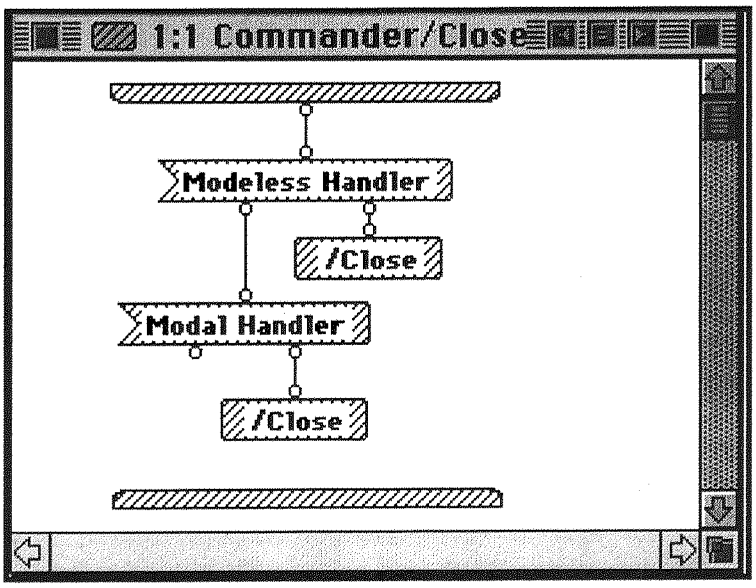


Fig. 5.20: Potentially recursive method.

In the example shown in fig. 5.20 (taken from the Application Building Classes), the method `Close` is in practice not recursive because the objects stored in the `Modeless Handler` and `Modal Handler` attributes are not instances of the `Commander` class. It is a common programming practice in Prograph to call, in one of the cases of a method, operations with the same name as the enclosing method to apply them to the objects stored in the attributes of the method receiver.

The detection of a recursive pattern in the analysis (that is, trying to apply the type analysis to the same method twice) is the primary motivation to keep track of the order in which the nested analyses are applied using a stack of method identifiers (see 5.8.1). Before starting the analysis of a method, the inference mechanism checks whether its method identifier is already on the stack of method identifiers. If this is the case, recursion has been detected. It can be direct recursion (if the matching method identifier is on the top of the stack or if it is separated from the top only by local methods' identifiers) or mutual recursion (several identifiers, other than local method identifiers, occur between the matching identifier and the top of the stack).

Instead of analysing it again, a dummy signature for the operation making the recursive call is constructed. The approximation used to construct the dummy signature is based on the following observations:

- If the method returns a value, the termination clause will have to be implemented as a separate case in which no recursive call occurs. Typing this case provides a first approximation of the signature of the method. Furthermore for a non-tail recursive method, the signature of the operation occuring between the recursive call and the `Output` operation helps to refine the approximation of the signature of the recursive method.

- A recursive method with no return value can be implemented with a single case. Recursion must be stopped by a control attached to an operation which checks that some conditions are met. The operation with the control must occur before the recursive call in the execution sequence, otherwise recursion will be infinite. The signature of the operation with the check provides an approximation of the input types of the recursive method.

- As in the example shown above, the apparent recursion does not result in a recursive program and the relevant type information to construct the operation signature comes from the signatures of the other methods that may be called.

The dummy signature for the recursive method has all its input and output types set to `<recursive>`. `<recursive>` is a pseudo type and acts as a neutral element for both type union and type intersection (`<recursive>` $\cap$ `a = a` and `<recursive>` $\cup$ `a = a`, with `a` being any type or type dependency). This reflects the intuition that the dummy signature carries no useful information and that it should interfere as little as possible with the inference:

- Type unions and intersections are computed during the case analysis and `<recursive>` should allow the analysis to obtain information from the context in which the recursive call occurs.

- Type unions are computed to construct the signature of a method from the results of the analyses of its cases. For a tail recursive method the relevant type information comes from the case with no recursive call.

5.9 Examples

5.9.1 A simple example

This example shows how type inference is applied to the method `IsEven?`. `IsEven?` has a single case, it takes an integer as input and returns `TRUE` if the integer is even, `FALSE` otherwise.

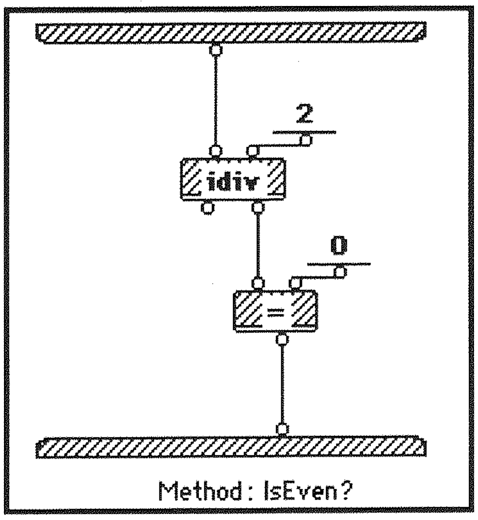


Fig. 5.21: The `isEven?` method

The following subsections describe the different phases of the type inference, where the properties of the datalinks are printed on the graph using the tuple format defined in 5.8.2.1, that is `(Type, NextType, Dependency)` and the operation signatures are printed next to the operations to which they are attached.

5.9.1.1 Initialisation phase

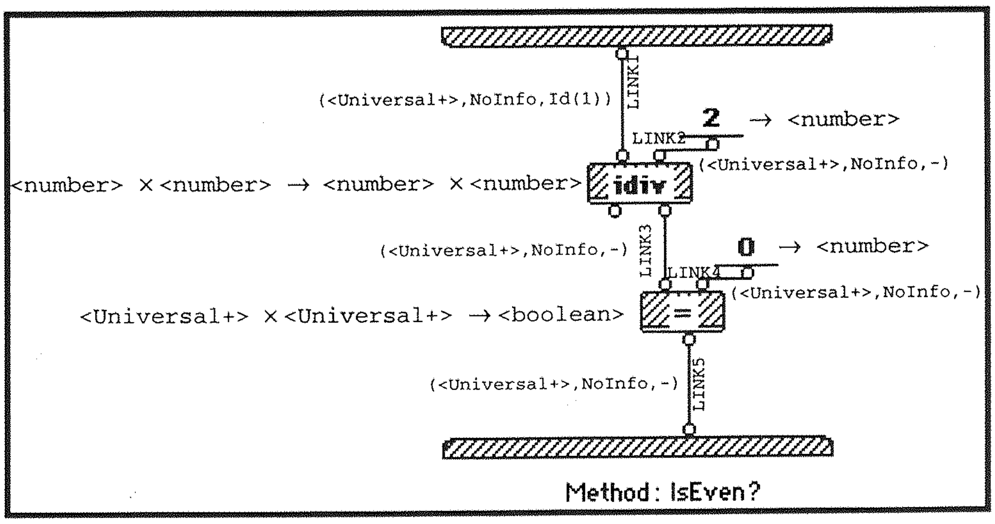


Fig. 5.22.a: The signatures of the nodes have been set.

During the set-up phase, a signature is associated with each operation on the dataflow graph, except for the `Input` and `Output` operations (fig. 5.22.a). The `Type` value for all the datalinks is set to `<Universal+>`, the `NextType` value for all the datalinks is `NoInfo`, the `Dependency` value for `LINK1` is `Id(1)` and it is not specified for the other datalinks of the case.

5.9.1.2 Forward analysis

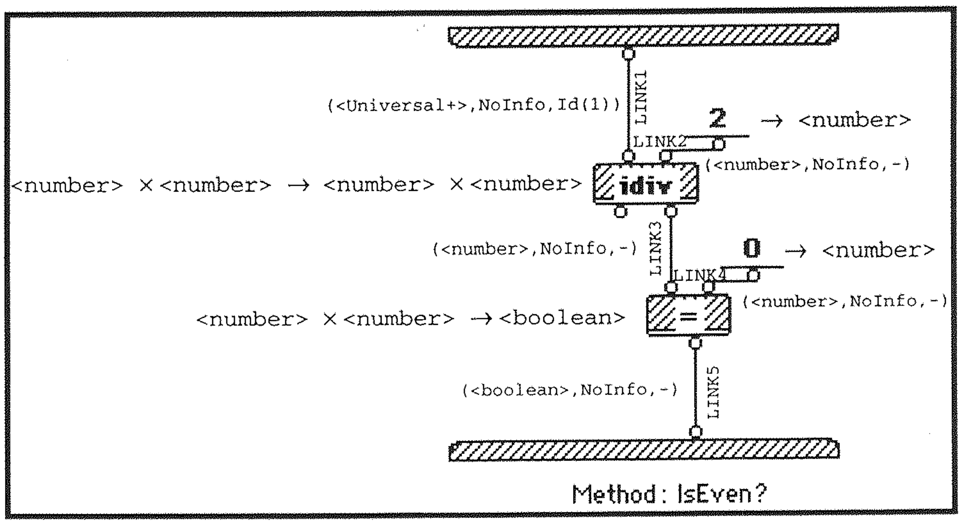


Fig. 5.22.b: After the forward analysis.

During the forward analysis `LINK1` remains unchanged and `LINK2` has type `<number>`. Since the signature of the operation `idiv` has only one line (which is compatible with the type on `LINK2`), `LINK3` must have the type

<number>. The intersection of the types on LINK3 and LINK4 with the input types of the = signature yields <number> for both inputs of the = operation. The Type value for LINK5 becomes <boolean>. The Type value for LINK1 remains <Universal+>.

5.9.1.3 Backward analysis

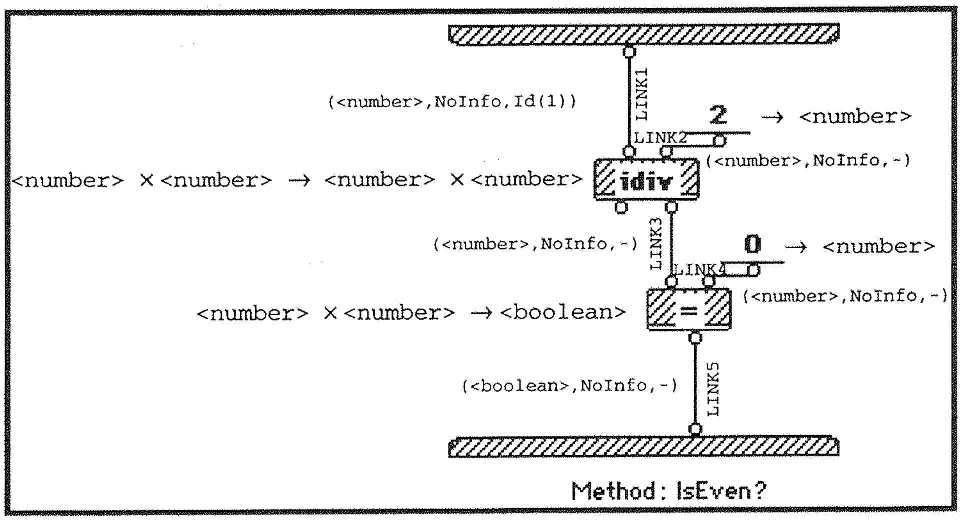


Fig. 5.22.c: After the backward analysis

During the subsequent backward analysis little changes. The signature of the idiv operation, however, implies that the Type value for LINK1 is <number>.

There is no Match operation in the case of the IsEven method, and the NextType values are left unchanged by the backward analysis.

5.9.1.4 Computation of `Dependency`

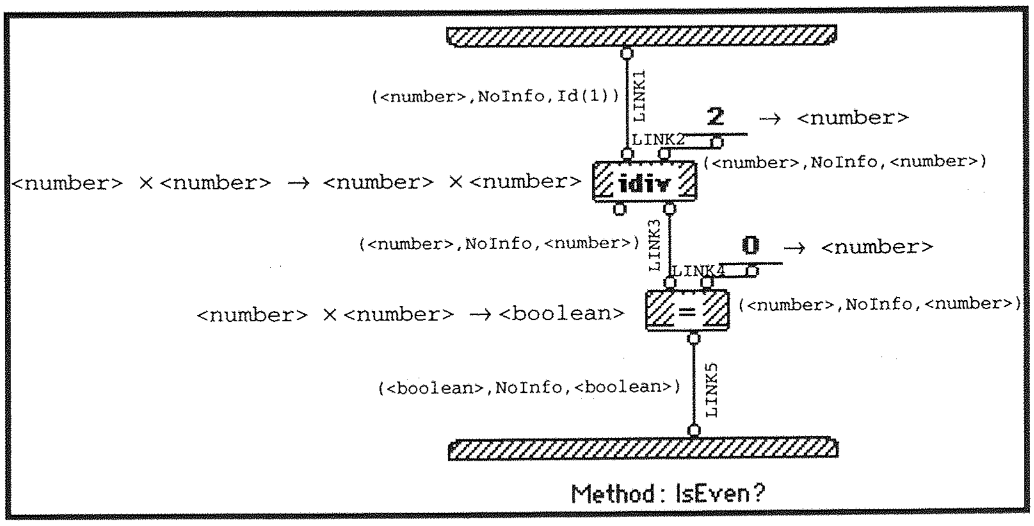

(<number>,NoInfo,Id(1))
LINK1
2 → <number>
LINK2
(<number>,NoInfo,<number>)
<number> × <number> → <number> × <number>
idiv
(<number>,NoInfo,<number>)
LINK3
0 → <number>
LINK4
<number> × <number> → <boolean>
=
(<number>,NoInfo,<number>)
(<boolean>,NoInfo,<boolean>)
LINK5
Method: IsEven?

Fig. 5.22.d: After the computation of `Dependency`

None of the operations of the case has a signature with a type dependency, as a result the `Dependency` value for all the datalinks except `LINK1` is a type.

The signature of the method can now be constructed. The input type is the `Type` value for `LINK1` and the output type is the `Dependency` value for `LINK5`. The signature is:

<number> → <boolean>

5.9.2 A recursive example

The method `Factorial` (see fig 5.23) computes the factorial of a positive integer.

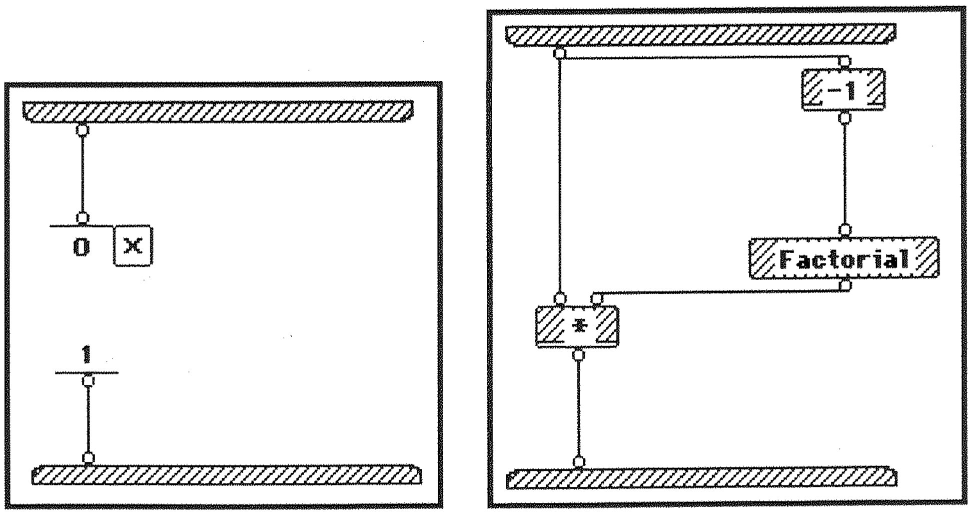


Fig. 5.23.a&b: Cases of the `Factorial` method.

5.9.2.1 Analysis of the first case

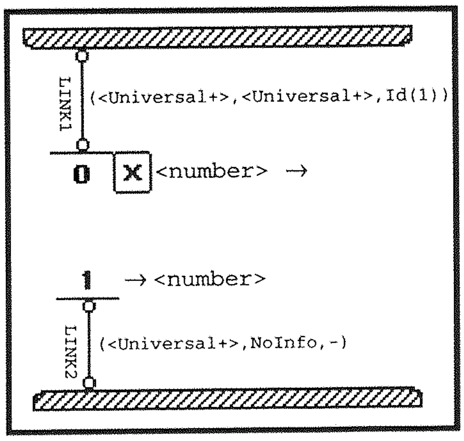


Fig. 5.24.a: After the initialisation phase

It must be noted that after the initialisation phase, the value of NextType for LINK1 is set to <Universal+> because LINK1 is connected to the terminal of a Match operation with a NextCase control activated on failure.

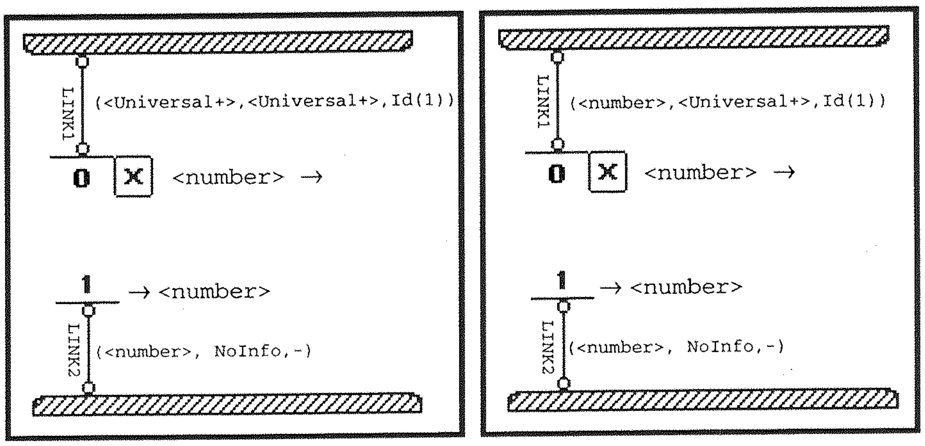


Fig. 5 24.b and c: Forward and backward analyses

The value of Type for LINK2 is updated during the forward analysis and the value of Type for LINK1 is updated during the backward analysis.

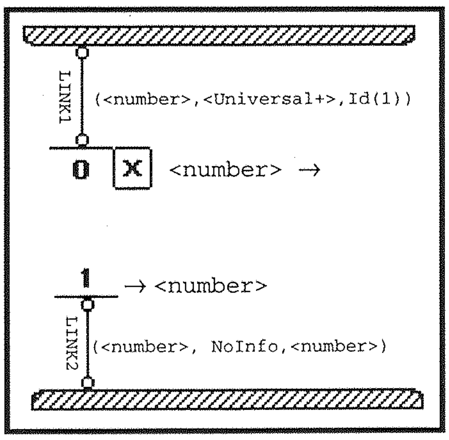


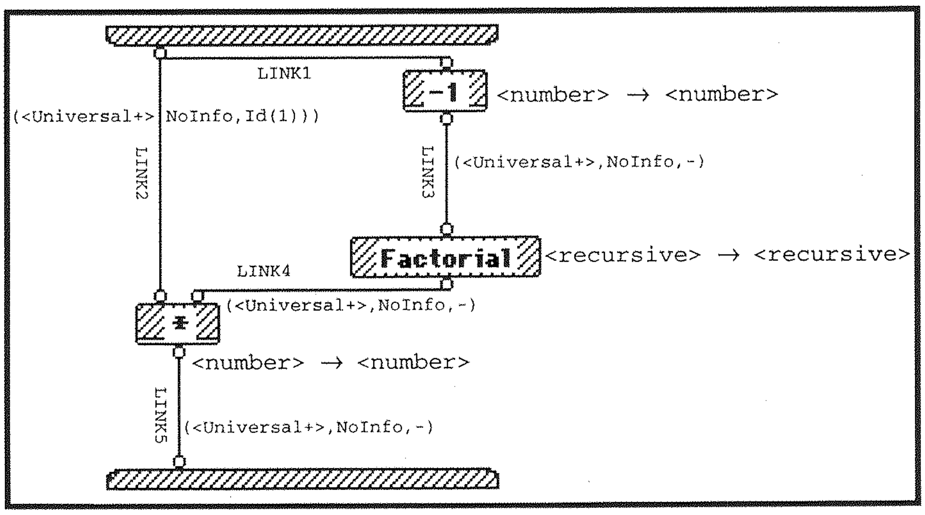
Fig. 5.24.d: Computation of the dependencies

After the computation of the dependencies, the line for the first case can be constructed:

<number> → <number>

and the type of the input of the next case is the value of `NextType` for `LINK1`, (`<Universal+>`).

5.9.2.2 Analysis of the second case



Fig, 5.25.a: After the initialisation phase

During the initialisation phase, the inference algorithm tries to construct the signature of the `Factorial` operation. However, the method identifier of the method currently analysed is `Universal/Factorial/Simple`, recursion is detected and the signature constructed for the operation is:

`<recursive>` → `<recursive>`

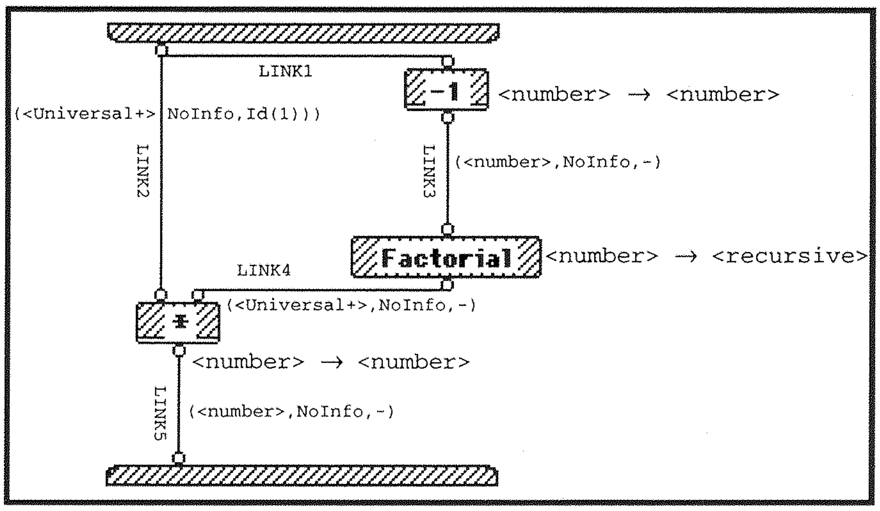


Fig. 5.25.b: After the forward analysis

After the forward analysis, the `Type` values of `LINK3`, `LINK4` and `LINK5` have been updated. The left-hand side of the signature of the `Factorial` operation has been updated as well.

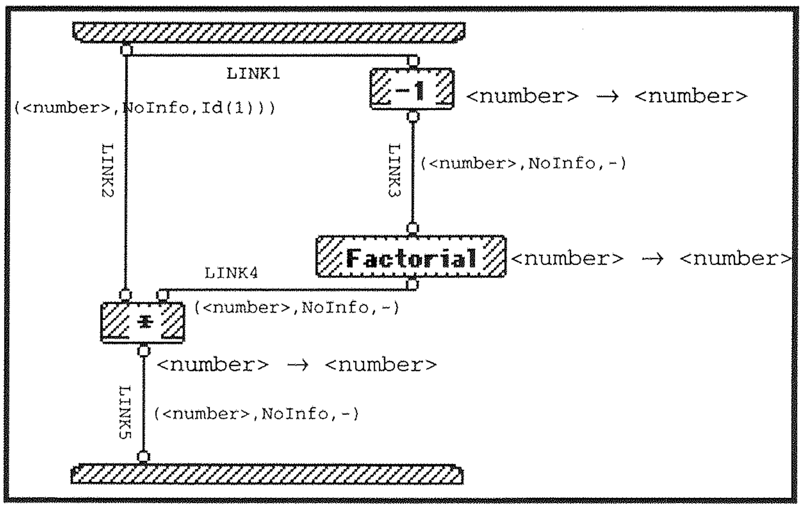


Fig. 5.25.c: After the backward analysis

After the backward analysis, the `Type` value shared by `LINK1` and `LINK2` and the right-hand side of the signature of `Factorial` are updated.

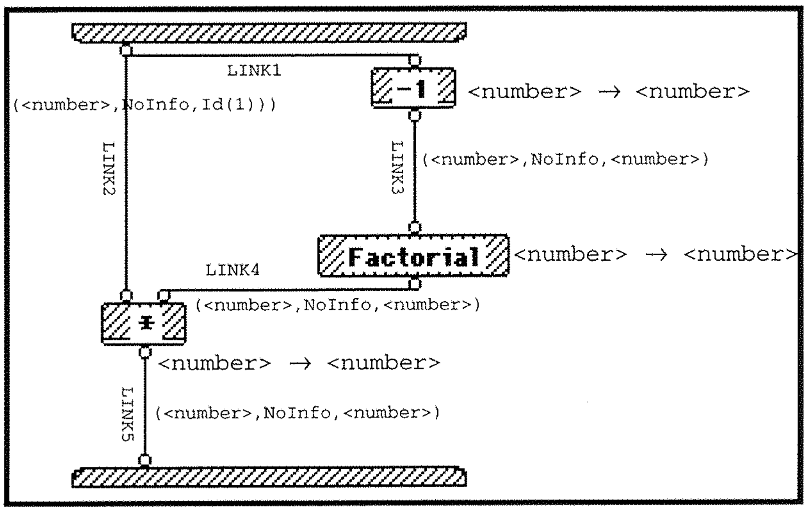


Fig. 5.25.d: Computation of the dependency.

There is no operation whose signature contains a type dependency. Therefore, for all the datalinks not connected to the input bar, the `Dependency` value is a type.

The line for the second case can be constructed:

$$\texttt{<number>} \rightarrow \texttt{<number>}$$

The signature for the whole method is constructed by combining the lines of the two cases. The line for the first case is:

$$\texttt{<number>} \rightarrow \texttt{<number>}$$

The line for the second case is also:

$$\texttt{<number>} \rightarrow \texttt{<number>}$$

and the combined line is:

$$\texttt{<number>} \cup \texttt{<number>} \rightarrow \texttt{<number>} \cup \texttt{<number>}$$

which is:

$$\texttt{<number>} \rightarrow \texttt{<number>}$$

5.10 Shortcomings of the type analysis

In subsection 5.5.4, it is stated that the type inference algorithm might fail to detect type errors and might also reject type correct code. These two situations are illustrated by concrete examples.

5.10.1 Failure to detect type errors

The type representation chosen for the analysis does not permit the expression of type dependencies between the inputs of an operation. A partial solution to this problem is to have a signature with several lines, a line for each valid combination of input types. The signature of the < primitive (shown below) is a good example.

The signature of the < primitive is:

```
<number> × <number> →<boolean>
<number> × <number> →
"" × "" →<boolean>
"" × "" →
```

However, during the forward and backward analyses, the type dependency embodied by the use of several lines is lost as the type of the inputs is the union of the matching input types of the lines. The loss of information is shown with the following (somewhat contrived) example. The relational operator is wrapped in a local method (fig. 5.26.a) and an invalid pair of arguments is passed to the local method (fig. 5.26.b).

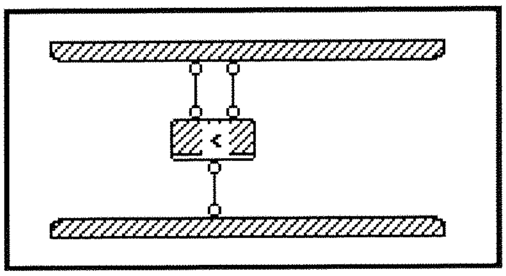


Fig. 5.26.a: < wrapped in a local method

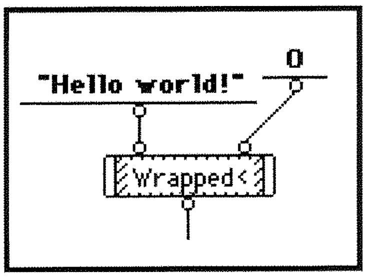


Fig. 5.26.b: Invalid arguments

The signature inferred for the local method is:

```
[<number> | ""] × [<number> | ""] → <boolean>
```

which is compatible with the pair of argument types. However the evaluation of this fragment of code should fail because the two arguments are of incomparable types. In the current version of Prograph, four primitives have dependencies between their input types: ≤, ≥, < and >.

The combination of the lines of the different cases of a method into a single line for the method signature has the same consequences as the combination of the lines of a primitive.

The example shown in fig. 5.27 is taken from the code of the Application Building Classes. A local `Get Value` is defined with three cases (see fig. 5.27.a, b and c). The `Get Value` operation can only call a class method because there exists no `Get Value` primitive or universal method.

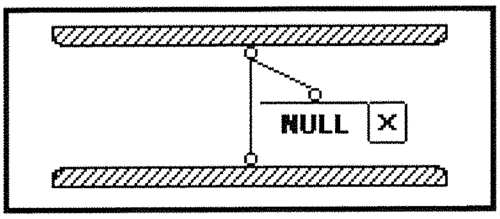


Fig. 5.27.a: First case of `Get Value`.

The line for case 1 is `<null>` → `Id(1)`

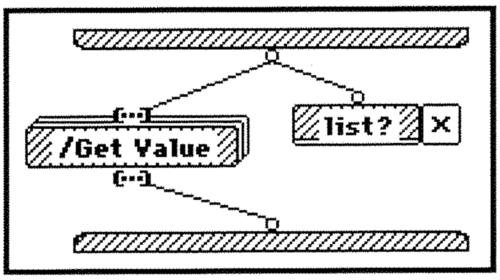


Fig. 5.27.b: Second case of `Get Value`.

The line for case 2 is:

`([...|...])` → `(<Universal+>)`

[... | ...] is the textual representation of a union type which is too large to be printed (this is because many classes define a `Get Value` method).

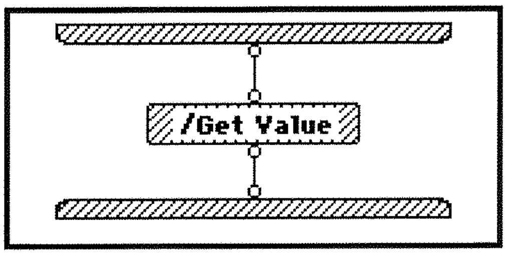


Fig. 5.27.c: Third case of `Get Value`.

The line for case 3 is:

`[...|...]` → `<Universal+>`.

The union type is a subset of the `<UDC+>` type (which means that the input of the third case has to be an instance of a user-defined class). Since it does not affect the explanation and to make it easier to understand, the line for case 3 is replaced with:

`<UDC+>` → `<Universal+>`

The combination of the three lines produces a signature for the method with the line:

```
[<null>|([...|...])|<UDC+>] → <Universal+>
```

During the passes of the inference, as a consequence of the misuse of the `Get Value` local method, the line may be specialised to:

```
<null> → (<Universal+>)
```

This line does not correspond to a valid execution of the local method. The error comes from the fact that the input types of the three lines and the output types are disjoint. Generally, $\alpha \cup \alpha' \rightarrow \beta \cup \beta'$ is equivalent to the two lines $\alpha \rightarrow \beta$ and $\alpha' \rightarrow \beta'$ if one of the following conditions is met:

- $\beta = \beta'$
- $\alpha \subset \alpha'$ and $\beta \subset \beta'$
- $\alpha' \subset \alpha$ and $\beta' \subset \beta$

However, checking the equivalence between the combined line and the sequence of lines from which the line has been obtained would be an expensive computation. Therefore a cruder approximation is used to combine the lines of the cases of the method.

Loss of information also occurs as the result of the approximation used for the signature of the primitives used to test the type of objects at run-time. For example `list?` (shown in the second case of `Get Value` above) succeeds and optionally returns `TRUE` if its input is a list, otherwise it fails or returns `FALSE`.

The signature of `list?` is:

```
<Universal+> → <boolean>
<Universal+>→
```

This signature does not keep track of the fact that an input type which is not a subset of `(<Universal+>)` will cause the operation to fail. When the primitive is encountered with a `NextCase` on failure control or `NextCase` on success control attached to it, the value of `NextType` for its input datalink will be set to `<Universal+>` in both cases.

5.10.2 Rejection of type correct code

The rule to construct the signature of a `Match` operation might lead to the rejection of type correct code. This is because the rule requires that for a `Match` operation with a `NextCase` on failure control, the `Type` value can only be the type of the value to be matched. The following code is not very useful because the `NextCase` control will always be activated, it is however correct:

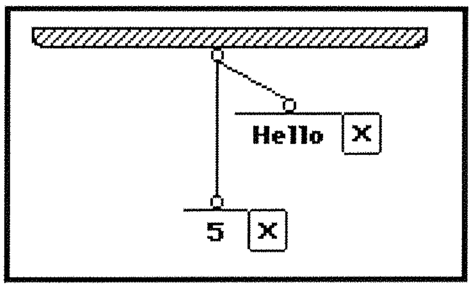


Fig. 5.28: Rejected type-correct code.

The signature of the `Match` string operation is:

"Hello" →

and that of the `Match` integer operation is:

<number> →

During the backward pass of the case analysis, the intersection of the input types of the two signatures must be computed because the same value is flowing into the two operations. The result is the `<Bottom>` type which causes the failure of the analysis. The analysis reports as a type error, code that, if executed, is likely to present a program error.

5.11 Summary

- Type inference is the ability to infer type information in the absence of type declaration. Type inference has been notably applied to object-oriented and functional languages.

- The purpose of type inference in Prograph is to reduce the uncertainty caused by dynamic binding and to gather some information for the effect analysis.

- A type is the set of classes of which a value can be an instance. A type dependency expresses a type as a function of another type.

- A method signature consists of one or more lines. A line describes the input and output types of an operation.

- The inference algorithm is applied to a method at a time. A method-wide inference can be decomposed into a sequence of case-wide inferences.

6 Effect inference and synthesis

The purpose of the effect analysis is to describe how the execution of an operation annotated for distribution may affect the arguments of this operation and global variables. The effect analysis proposed for Prograph proceeds in two steps: effect inference and effect synthesis. Effect inference is tightly integrated with type inference and produces a description of the effects of individual methods. Effect synthesis interprets the effect information available for an operation annotated for distribution.

This chapter is divided into four sections. The first section discusses the different motivations for undertaking effect analysis. The second section reviews some of the work undertaken in the field of effect analysis. The third section is devoted to the effect inference and fourth section covers effect synthesis in Prograph.

6.1 Purpose of effect analysis

Research work has focused on procedure-oriented and functional languages and information about effects serves several purposes.

Effect information is useful in several contexts:

• Optimising compilers rely on the results of effect analysis to perform various optimisations. Examples of optimisations include: *constant propagation*, the compiler can perform a significant amount of precomputation by propagating the constants through the program, *constant folding*, the compiler replaces operations with constant operands with their computed value. Effect information also provides knowledge about the lifetime of data objects and allows a more efficient management of memory such as the stack allocation of data objects instead of heap allocation.

• Parallelising compilers also exploit effect information [Bacon, Graham and Sharp 1994]. Effect analysis produces the set of the locations read and written by different subcomputations and the dependency analysis can detect dependencies by computing the intersection of the different sets. Knowledge of dependencies allows the parallelising compiler to partition and schedule computations into sets of concurrent tasks.

• The selection of test data for a program is another example of use of effect information [Rapps and Weyuker 1982]

• The maintenance and evolution of large software systems require tools that automate the production of documentation about the system. Effect information is useful to check that the software system evolves in a consistent way [Ryder 1989].

6.2 Related Work

This section reviews several research areas. These examples have been chosen to illustrate the various purposes of effect analysis, the techniques used and the languages to which it is applied.

6.2.1 Chow and Harrison

The analysis proposed in [Chow and Harrison 1992] is part of a multilingual parallelising compiler, the Miprac system. The aim of the analysis is to gather information on:

- side-effects
- data dependencies
- object lifetime
- unordered accesses

The analysis applies *abstract interpretation* to whole programs converted into MIL, an intermediate language used by the Miprac compiler. MIL provides three kinds of values: integers, locations and closures. Locations are initialised by a `create` operation, accessed by a `read` operation and modified by a `write` operation. Parallelism is expressed by a `cobegin` construct. `cobegin` spawns different processes to evaluate the expressions passed to it as arguments. The processes execute in a shared memory space. Before describing in further details the analysis, it is helpful to give some explanation of abstract interpretation.

6.2.1.1 Abstract interpretation

Abstraction interpretation [Cousot and Cousot 1977] is the evaluation of a program based on an *abstract semantics*. The abstract semantics defines an evaluation function for all the expressions of the language. The standard semantics domains are mapped into corresponding abstract domains. The purpose of abstract interpretation is that the abstract semantics makes the evaluation of programs more efficient than the evaluation based on the standard semantics, yet precise enough to record the information of interest to a particular analysis.

The rule of signs for the multiplication is the archetypal example of abstract interpretation. The set $\mathtt{Z}$ of the integers is mapped onto the set $\mathtt{Z}^{\#}$=`{plus, minus, zero}`. An abstraction function $\mathtt{abs}_{\mathtt{Z}}: \mathtt{Z} \rightarrow \mathtt{Z}^{\#}$ is defined:

$\mathtt{abs}_{\mathtt{Z}}(\mathtt{x})$ = `plus if x>0`

= `minus if x<0`

= `zero if x=0`

The abstract version of $*$, denoted by $*^{\#}$ may be expressed as:

`*#(plus, minus)= *#(minus, plus) = minus`

`*#(plus, plus) = *#(minus, minus) = plus`

`*#(a, zero)= *#(zero, a)= zero.`

The sign of the product of two integers is obtained while avoiding the cost of the multiplication.

Safeness of an abstract interpretation requires that for all elements of the concrete domain, the abstractions of the results of the concrete function applied to concrete domain elements are in the result set of the abstract function applied to the abstractions of concrete domain elements. This requirement is best illustrated by the fig. 6.1 (taken from [Field and Harrison 1988]) for the rule of signs:

$$\begin{tikzcd}
Z\text{x}Z \arrow[r, "*"] \arrow[d, "\text{abszxz}"'] & Z \arrow[d, "\text{absz}"] \\
Z\#\text{x}Z\# \arrow[r, "*\#"'] & Z\#
\end{tikzcd}$$

Fig. 6.1 Safeness of the rule of signs

The rule of signs is safe if $\mathtt{abs}_{Z} \circ * \subset *\# \circ \mathtt{abs}_{Z \times Z}$

Field and Harrison consider that the difficulty for an analysis based on abstract interpretation is to find abstract domains which provide useful information about a property of the program being analysed while guaranteeing safeness. A further problem in choosing the abstract domain is that the more complex the domain, the more computationaly intensive is the algorithm.

6.2.1.2 Description of the analysis

Chow and Harrison's analysis is developed in two steps. Firstly, an analysis is proposed for the concrete semantic domains. As a second step the concrete domains are abstracted and the safeness of the abstract interpretation is established. The remainder of this subsection gives an overview of the analysis in the concrete domains.

The execution of the program is described by a transition system. Each expression is labelled, lambda expressions are uniquely identified by *procedure labels* and cobegin branches by *cobegin branch labels.*

Procedure strings are sequences of procedure and cobegin branch labels. A function call is denoted by α^{d} (d for down) and a function return by α^{u} (u for up), where α is the procedure label of the function. Similarly entering a cobegin branch is denoted by η^{d} and exiting by η^{u}. Procedure strings capture the procedural/concurrency movement along the program execution. The use of the procedure strings is illustrated in fig. 6.2:

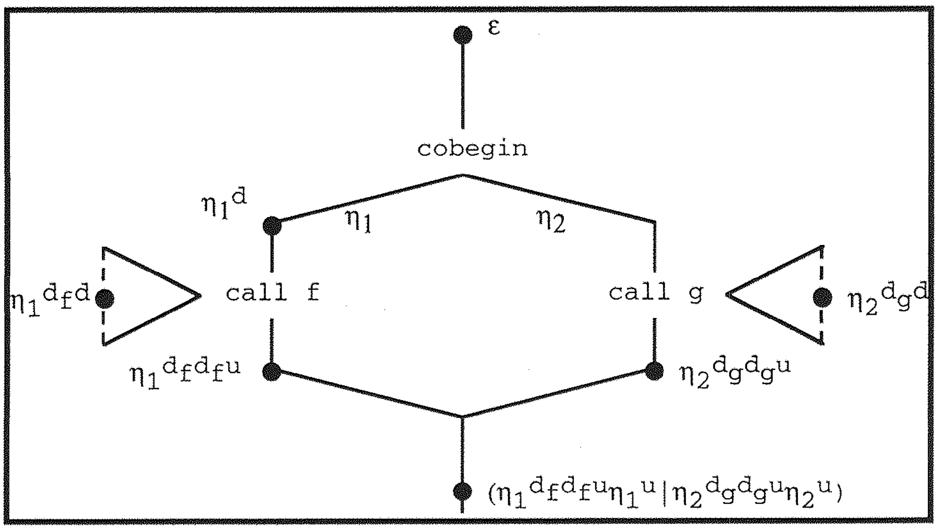


Fig. 6.2: An example of procedure strings

The graph in fig. 6.2 illustrates the spawning of two cobegin branches η_1 and η_2, η_1 calls the function `f` and η_2 the function `g`. After `f` and `g` return, the two cobegin branches merge.

The transition system models the evaluation of the program by recording a configuration for each program point; a configuration comprises the description of the processes currently active and that of the shared store. A procedure string is part of the process description. The *birth date* of a variable is the procedure string attached to the expression which created the variable. The birth date of a variable is saved with the variable identifier in the store.

Program properties are derived from the manipulation of procedure strings. The following example shows how procedure strings are used to check whether a variable outlives the evaluation of the function that created it (the rule shown below is only applicable to a sequential program, a more general rule which handles cobegin branches is also presented in [Chow and Harrison 1992]).

The procedure β creates the object `L` with a birthdate p_b. `L` is referenced by expression `r` with a procedure string p_r. The analysis computes the net procedural movement between the creation of `L` and the program point where it is referenced:

- θ', p_b subtracted from p_r is then computed.

- `Net(θ')`, the net movement of the string θ' is computed, that is, all matching pairs of the form $\alpha^d \alpha^u$ are removed from θ'.

If the net movement of θ' contains the term β^u, it can be inferred that the procedure β returns before `L` is referenced. In other words, the variable `L` outlives the procedure β which created it.

The object lifetime property can be illustrated by a simple example. A function α calls a function β, β calls the function γ and inside γ the object `L` is created. γ and β return before `L` is referenced. The birthdate of `L`, p_b, is: $\alpha^d\beta^d\gamma^d$ and p_r is: $\alpha^d\beta^d\gamma^d\gamma^u\beta^u$. and $\mathtt{Net}(p_r - p_b) = \gamma^u\beta^u$. From the analysis, it appears that `L` cannot be allocated on the stack frame of the procedure β and γ but the space for `L` can be freed after the return of α.

6.2.2 The FX effect system

The FX programming language [Gifford et al. 1987] is a functional language with imperative constructs. FX is targeted at parallel programming and in that context obtaining effect information would be useful to schedule expressions in parallel. [Lucassen and Gifford 1988] proposed a polymorphic effect system to infer the type and effects of expressions in a subset of FX called MFX (mini FX).

MFX is based on the higher-order *kinded* lambda calculus. The language distinguishes between ordinary lambda abstractions and polymorphic lambda abstractions. Interaction with the store is possible through the `NEW` operation which creates a new location, `GET` which accesses a location and `SET` which updates a location.

Inference proceeds in a modular and bottom-up fashion deriving information about the type and effects of expressions using inference rules. The type and effects are tightly integrated in expression *descriptions*. These descriptions are constructed from three basic *kinds*: *regions*, *effects* and *types*.

A *region* is the abstraction of a store area. A region expression can be:

- a region variable
- a region constant
- the union of one or more regions

The analysis distinguishes among three different effects, namely *allocation, read* and *write* effects. Effect expressions can be one of the following:

- An effect variable.
- `(ALLOC Region)` is the effect corresponding to the allocation of a memory reference.
- `(READ Region)` indicates an access to a memory location.
- `(WRITE Region)` corresponds to the update to a store location.
- `(MAXEFF Effect*)` constructs the union of zero or more effects.
- `PURE` denotes the absence of effects.

Type expressions can be one of the following alternatives:

- A type variable.
- The type of ordinary lambda abstraction is described by `(SUBR (Type) Effect Type)`, where `(Type)` is the list of argument types, the second `Type`, the return type and `Effect` is the latent effect of the lambda abstraction.
- `(POLY (DVAR:Kind) Effect Type)` describes the type of polymorphic abstractions. The `(DVAR:Kind)` term is the list of description variables in the description of the abstraction, thus reflecting the polymorphic nature of the abstraction.
- The type expression for location is `(REF Region Type)`.

One example given in [Lucassen and Gifford 1988] describes the type and effect signature of the function `twice`. The function `twice` takes a function of a single argument and composes that function with itself. The signature of `twice` is:

```
twice: (POLY (t:TYPE e:EFFECT) PURE (SUBR(SUBR:(t) e t) PURE (SUBR
(t) e t)))
```

The signature indicates that `twice` is a polymorphic abstraction. The signature contains a type variable `t` and an effect variable `e`, the polymorphic abstraction takes a function as its input and returns a function and the composition of the function induces no effect.

6.2.3 Type and effect inference in ML

Milner's original type inference algorithm can be applied to a purely functional subset of ML; however, efficiency concerns have motivated the addition of imperative constructs to the functional core of the language. The language provides *reference cells* and three operators are available to handle these cells: the `ref` operator applied to a value creates a reference cell with that value, the `!` operator accesses the content of a cell and the := operator updates the content of a cell.

Unfortunately, the extension of Milner's type system to a version of ML with imperative constructs is not trivial. The availability of references complicates the generalisation of type variables. The following example illustrates the difficulty:

```
let x = ref(fun(a)=a) in x:= (fun(n) = n+1); (!x) true
```

Applying Milner's type discipline, the type of x is $\alpha \rightarrow \alpha$ when the reference is created, with α generic and α is instantiated to `Int` and to `Boolean` successively. However, the evaluation of the expression above causes a type error: `(!x) true` tries to add `1` to `true`.

The typing rule for the let expressions does not reflect the sharing implied by references. The well typing of expressions requires that reference cells have only one type.

Various solutions have been proposed to control the generalisation of type variables in the presence of reference cells and a good survey of the various approaches can be found in [Wright 1993].

The solution advocated by [Wright 1991] is to approximate the *allocation effects* of an expression, that is the set of reference cells that may be allocated as a result of the evaluation of the expression. The *type effect* of an expression is the set of type variables that appear in the allocation effects of that expression. Type variables occurring in a type effect may not be generalised.

With the expression shown above, the bound expression `ref(fun(a)=a)` has type `ref` and effect $\{\alpha\}$, thus the type variable α cannot be generalised.

[Talpin and Jouvelot 1994] have developed a type and effect analysis to control the generalisation of type variables. The representation of types of effects follows that of [Lucassen and Gifford 1988]. The effect information is taken account of by the inference rules to decide whether a type variable can be safely generalised or not.

6.2.4 Effect analysis for test data selection

The work done by Rapps and Weyuker [Rapps and Weyuker 1982] illustrates how effect analysis can be used in the field of software testing. The effectiveness of program testing strongly depends on selecting a set of test inputs representative of the entire input domain. The selection of test data may be based on code coverage, one coverage measure is *branch* coverage (the number of branches traversed during the testing). Programs have a potentially very large number of execution paths and a realistic testing strategy can only test a limited number of paths. The results of the effect analysis are used to check that the input test data will cause the tested programs to cover program paths that satisfy a chosen path selection criterion.

The analysis is applied globally to programs specified in an intermediate level imperative language. The language is equipped with the following statement types:

- Start statement: `start`
- Input statement: `read` $x_1 \dots x_n$
- Assignment statement: `y:=f(`$x_1 \dots x_n$`)`
- Output statement: `print` $e_1 \dots e_n$
- Unconditional transfer statement: `goto m`
- Conditional transfer statement:

  `if p(`$x_1 \dots x_n$`) then goto m`
- Halt statement: `stop`

All program statements are labelled with an integer label. These labels define a total ordering on the statements. A *block* is a sequence of statements such that if the first statement is executed, all the statements in the block are executed. The program is represented as a graph whose nodes are labelled blocks. The edges of the graph result from transfer statements between blocks of instructions. A path is a sequence of nodes such that there is an edge between each pair of successive nodes. An example is shown in fig. 6.3 (taken from [Rapps and Weyuker 1982]).

```
1. start
2. read x, y
3. if y<0 then goto 6
4. pow:= y
5. goto 7
6. pow := -y
7. z:=1
8. if pow=0 then goto 12
9. z:=z*x
10. pow:=pow-1
11. goto 8
12. if y≥0 then goto 14
13. z:=1/z
14. answer:=z+1
15. print answer
16. stop
```

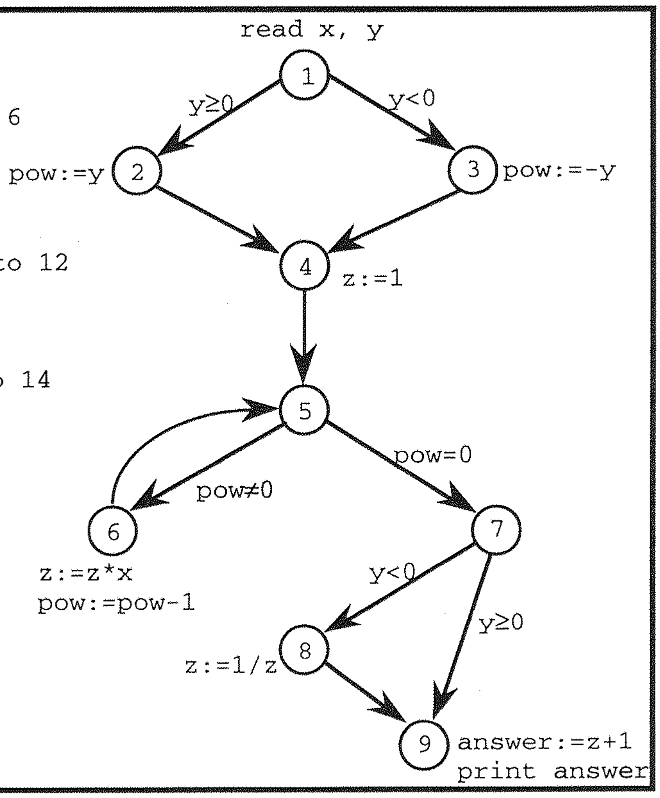


Fig. 6.3: A program and its corresponding graph.

The dataflow analysis classifies each variable occurrence as a definitional occurrence (called *def*), a computation-use occurrence (*c-use*) or predicate-use occurrence (*p-use*).

The def/use information is attached to the nodes and the edges of the program graph:

- Def and c-use sets are associated with each node. In the example, the c-use set for node 2 is `{y}` and its def set is `{pow}`.
- A p-use set is associated with each edge of the graph. For example, the edge (1,2) has the p-set `{y}`.

The analysis masks local def and local c-use. A definition of a variable is a local one, if all the computation-use occurrences of the variable appear in the same block as the definition of the variable. A local c-use is a c-use of a variable defined in the same block.

Nine different path selection criteria are defined, two examples are given below:

- A set P of complete paths of the graph G meets the *all-nodes* criterion if every node of G is included in P.
- A set P meets the *all-p-uses* criterion, if for every node of the graph and for every variable defined at the node, P includes a path from the variable definition to all the edges whose p-use set contains the given variable.

An ordering relation can be constructed over the criteria. Criterion c_1 includes criterion c_2 if a set P of complete paths that satisfies c_1 also satisfies c_2. The inclusion is strict if c_2 is satisfied but not c_1. If c_2 is included in c_1, then c_2 is said to be weaker than c_1.

The effect information is used to verify that a set of test data meets the test criteria selected by the user.

6.3 Effect inference

This section explains how the effects of a method can be inferred. Effects in Prograph are discussed. The inference is outlined and a suitable representation for effect information is discussed. The inference algorithm is described in greater detail and is illustrated by an example.

6.3.1 Motivation for effect inference in Prograph

The purpose of the effect inference in Prograph is to be able to describe how a method when called by an operation may affect its arguments and some global variables. This information is recorded in the *type signature* of the method.

6.3.2 Outline of the effect inference mechanism

Similarly to the effect inference systems of MFX and ML presented in the previous section, the effect inference proposed for Prograph proceeds in a modular fashion, that is the effect inference is applied to a method at a time and not to an application as a whole. Information about types and effects is tightly integrated. For the sake of clarity, type and effect inferences are treated in two different chapters in this thesis; however, the two inferences proceed together. As for the type inference, the effect inference is broken into a sequence of case wide inferences.

6.3.2.1 Case-wide inference

The effect inference distinguishes between the effect properties of the data objects called *affected data properties* and the properties of the operations, called *side-effects*.

Affected data properties are attached to the datalinks of the case being analysed. The purpose of these affected data properties is to summarise the sequence of side-effects necessary to obtain the value flowing on each datalink of the case.

A side-effect describes the way an operation accesses or updates its arguments or some global variables.

The *effect signature* of an operation is integrated with its type signature. For this purpose the lines comprising the signature will store some side-effects. The following notation is used for a line with type and side-effects:

$$\tau_1 \times \tau_2 \rightarrow \tau_3 \times \tau_4 \; \texttt{/*SE*/} \sigma_1 \times \sigma_2$$

The term `/*SE*/` separates the type information on its left and the side-effects on its right, σ_1 and σ_2 are two side-effects.

The affected data properties of the datalinks and the effect signature of the operations are constructed during the initialisation phase along with the type properties and the type signature. The effect signatures of primitive operations cannot be inferred, they must be available in a signature repository.

Type inference requires three successive passes over the operations of the case. Effect inference is performed as a fourth pass over the graph. It proceeds from the first operation after the `Input` operation until the last one before the output operation, following the execution order defined for the case. The purpose of this forward pass is to *compose* the side-effects of the operations of the case.

6.3.3 Effects in Prograph

This section discusses a representation of effect information in Prograph. Like the various works presented in the first section of this chapter, the effect representation should distinguish between read and write side-effects.

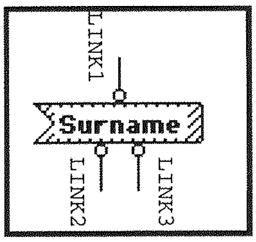


Fig. 6.4: A read side-effect

In fig. 6.4 the `Get` operation accesses the value of the instance attribute `Surname` from the instance flowing on `LINK1` (or possibly the default value of the class whose name is passed as a input to the `Get` operation). A read side-effect, σ, is used to describe the access performed by the operation.

The value flowing on `LINK1` is called the *argument* of side-effect σ and the value flowing on `LINK3` is said to be the *result* of side-effect σ. Side-effect σ records that its argument is passed through the first terminal of the `Get` operation with a reference to this terminal (a reference is an integer value). Side-effect σ must also record that its result is propagated through the second root of the `Get` operation with a reference to that root. A complete definition of the information required to describe σ will be given below.

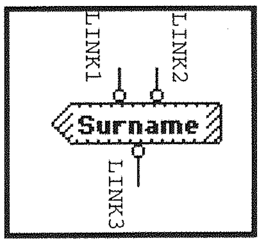


Fig. 6.5: A write side-effect

In fig. 6.5, the `Set` operation updates the value of the attribute `Surname` of a class or an instance (depending on the value on `LINK1`) with the value flowing on `LINK2` and the result is passed to `LINK3`. The update performed by the `Set` operation is described by a write side-effect, σ'.

The value flowing on `LINK2` is called the *update value* of the side-effect σ'. Side-effect σ' contains a reference to the first terminal of the operation as its argument, a reference to the second terminal as its update value and a reference to the root of the operation to pass its result.

Some side-effects do not return a result (e.g. write side-effects on persistents), such side-effects are called *terminal* side-effects.

During the case-wide effect inference, the side-effects of the operations are composed. Composition proceeds by *applying* the side-effects of the different operations of the case.

Applying a side-effect means substituting the references to operation inputs contained in the side-effect with:

- a reference to an input of the case

- an affected data property if the argument or the update value of the side-effect is itself the result of a side-effect.

After substitution of the input references, the side-effect selects the outputs of the operation through which it passes its result (in fig. 6.4, the read side-effect propagates its result through the second root of the `Get` operation) and propagates affected data properties through these outputs.

Composition and side-effect application are now described with two simple examples and are explained in greater detail in section 6.3.4.

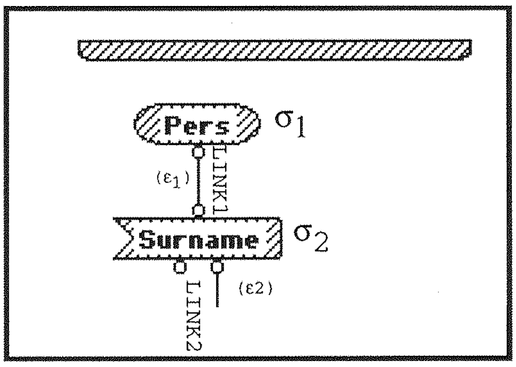


Fig. 6.6: Composition of side-effects

The `Get` persistent operation shown in fig. 6.6 extracts the value of the persistent `Pers` and this is described by the side-effect σ_1 in the signature of the `Get` persistent operation. When side-effect σ_1 is applied, it propagates the affected data property ε_1 on `LINK1` to show that the value on `LINK1` was extracted from the persistent `Pers`.

The `Get` operation extracts the value of the attribute `Surname` from its argument and this is described by the side-effect σ_2 in the signature of the

Get operation. When σ_2 is applied, the reference to the argument input (input 1) is replaced by the affected data property ε_1 propagated by side-effect σ_1. Thus side-effects σ_1 and σ_2 have been composed.

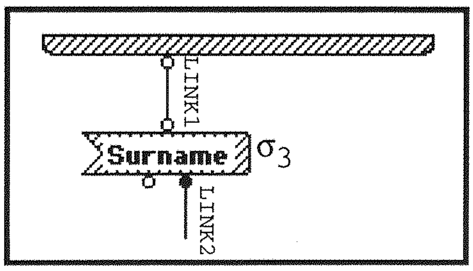


Fig. 6.7: Side-effect applied to an input of the case

In the second example shown in fig. 6.7, the argument of side-effect σ_3 is passed as an input to the case. When the side-effect σ_3 is applied, the reference to the input of the Get operation is replaced with a reference to the input of the case.

The different effect analyses presented in section 6.2 relied on low level representation of effects:

- the MFX system represents an effect as a READ or a WRITE operation on a region;

- in ML, effects can be a Get or a Set operation on a cell;

- in [Chow and Harrison 1992] as well as [Rapps and Weyuker 1982], an effect can be a read operation or a write operation on a variable.

The effect representation chosen for Prograph classifies effects in different *effect categories*. An effect category defines a set of side-effects and the affected data properties that result from the application of these side-effects.

All the effect categories are listed in the table below with an explanation about their purpose as well as the set of side-effects and affected data properties they define. It must be noted that when a category defines a terminal side-effect, it does not need to define the corresponding affected data property: a terminal side-effect when applied does not propagate an affected data property.

Effect Category	Purpose	Defines
Identity	This category describes the propagation of a data object without modification.	identity side-effect identity affected data property

Class	This category describes the effects induced by a `Get` or a `Set` operation on a class attribute.	class read side-effect
		class read affected data property
		class write side-effect
		class read affected data property

Instance	This category describes the effects induced by a `Get` or a `Set` operation on a class attribute.	instance read side-effect instance read affected data property instance write side-effect instance write affected data property
Instantiation	This category records the access to the value of class variables and/or default value of instance attributes when an instance of a user-defined class is created.	instantiation side-effect (terminal)
Local	This category indicates that a data object has been created in the scope of the current case.	local affected data property
Persistent	This category describes the effects induced by a `Get` or a `Set` operation on a persistent.	persistent read side-effect persistent read affected data property persistent write side-effect (terminal side-effect)
List	This category describes the effects induced by primitive operations manipulating lists.	list read side-effect list read affected data property list write side-effect list write affected data property
External	This category describes effects on external data structures.	external side-effect (terminal)

The side-effects and the affected data properties record information using the relevant combination of the following data items:

Item Name	Value	Purpose
`Argument`	integer/ affected data property	An integer value is a reference to the input of the operation which passes the argument of the side-effect. 0 indicates that the side-effect operates on a global variable (e.g. a persistent). When the side-effect is applied, the reference is replaced with one of the following: - an integer reference to an input of the case - an affected data property if the argument of the side-effect is the result of a previous side-effect.
`ArgumentType`	Type	This item records the type of the data item passed as argument to the side-effect.
`Action`	`Read/Write`	This item distinguishes between a read and a write side-effect.
`Data`	string	This item records some textual data to describe the side-effect.
`UpdateValue`	integer/ affected data property	An integer value is a reference to the input of the operation which passes the update value of the side-effect. When the side-effect is applied, the reference to the operation input is replaced with one of the following: - an integer reference to an input of the case - an affected data property if the update value of the side-effect is the result of a previous side-effect.
`Next`	a list of integers	This item lists the outputs of the operation through which the results of the side-effect are propagated. For a terminal side-effect, the value of `Next` is an empty list.

A textual representation is used so that side-effect and affected data property expressions can be easily parsed and printed. All expressions are of the form `Prefix(Item`$_1$ `Item`$_2$ `...)`, with the prefix used to encode the action and the category of the effect and the tuple containing the items relevant for a particular category of effects.

6.3.3.1 Identity effects

The most trivial side-effect is the *identity* side-effect, which means that the operation's output is the same as the one on the input. The identity side-effect records the following pieces of information:

Item name	Value	Explanation
`Argument`	integer	This is a reference to the input of the operation which passes the argument of the side-effect.
`ArgumentType`		The identity side-effect does not record the type of its argument.
`Action`	`Read`	An identity side-effect is a read side-effect.
`Data`		The identity side-effect does not record any textual data.
`UpdateValue`	0	The identity side-effect is a read side-effect and read side-effects do not take update values.
`Next`	sequence of integers	This is a reference to the outputs of the operations which propagate the result of the side-effect.

The textual representation for an identity side-effect is:

- `IDE(Argument Next)`

`IDE` is a short notation for **ID**Entity. The corresponding affected data property is represented with:

`-IDE(Argument)`

6.3.3.2 Effects on class attributes

Reading or writing a class attribute produces a class side-effect. The following information is recorded to describe a class side-effect:

Item name	Value	Explanation
`Argument`	1	The argument of a class side-effect flows into the first terminal of the `Get` or `Set` operation.
`ArgumentType`	Prograph type	This is the type of the data item passed as argument of the side-effect. This information will be used during the effect synthesis.
`Action`	`Read/Write`	A class side-effect can be a read or a write side-effect.
`Data`	string value	This is the name of the class attribute whose value is accessed or modified.
`UpdateValue`	0 for a read side-effect/ 2 for a write side-effect	A read side-effect takes no update value, hence 0/the update value for a write side-effect flows into the second terminal of the `Set` operation.
`Next`	(2) for a read side-effect/(1) for a write side-effect	The result of a read side-effect flows out from the second root of the `Get` operation/the result of the write side-effect flows from the first root of the `Set` operation.

A class read side-effect is represented with:

- `CAR(Argument Data Next)`

`CA` is a short notation for **C**lass **A**ttribute and the `R` stands for **R**ead. The corresponding affected data property is represented with:

- `CAR(Argument Data)`

The `ArgumentType` field is not printed in the effect representation.

A class write side-effect is represented with:

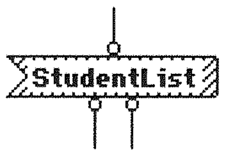
- `CAW(Argument Data UpdateValue Next)`

and the corresponding affected data property is represented with:

- `CAW(Argument Data UpdateValue)`

respectively. `W` stands for **W**rite.

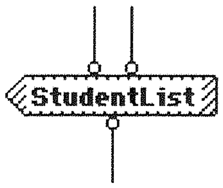
If `StudentList` is the name of a class attribute, the effect signature of the signature of the `Get` operation shown below:



is:

```
IDE(1 (1)) × CAR(1 "StudentList" (2))
```

For the `Set` operation shown below:



the effect signature is:

```
CAW(1 "StudentList" 2 (1))
```

6.3.3.3 Effects on instance attributes

Reading or writing an instance attribute is described with an instance side-effect. The information recorded to describe an instance side-effect is explained in the table below:

Item name	Value	Purpose
`Argument`	1	The argument of a class side-effect flows into the first terminal of the `Get` or `Set` operation.
`ArgumentType`	Prograph type	This is the type of the data item passed as argument to the side-effect. This information will be used during the effect synthesis.
`Action`	`Read/Write`	An instance side-effect can be a read or a write side-effect.
`Data`	string value	This is the name of the instance attribute whose value is accessed or modified.
`UpdateValue`	0 for read side-effect/ 2 for write-side -effect	A read side-effect takes no update value, hence 0/the update value for a write side-effect flows into the second terminal of the `Set` operation.
`Next`	(2) for a read side-effect/(1) for a write side-effect	The result of a read side-effect flows out from the second root of the `Get` operation/the result of the write side-effect flows from the first root of the `Set` operation.

Representations of side-effects and affected data properties on instance attributes are one of the following:

- `OAR(Argument Data Next)` (read side effect)

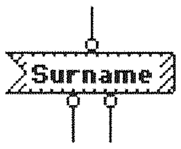
- `OAR(Argument Next)`

- `OAW(Argument Data UpdateValue Next)` (write side-effect)

- `OAW(Argument Data UpdateValue)`.

`OA` stands for **O**bject **A**ttribute.

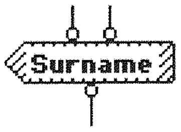
If `Surname` is the name of an instance attribute, the effect signature of the `Get` operation shown below:



is:

```
IDE(1 (1)) × OAR(1 "Surname" (2))
```

For the `Set` operation shown below:



the effect signature is:

```
OAW(1 "Surname" 2 (1))
```

6.3.3.4 Instantiation effects

Creating a new instance of a class also induces a side-effect. If the class of the new instance has class variables, the new instance will point to these variables and the instance variables will point to the default values defined for the class. The purpose of the *instantiation* side-effect is not to record the allocation of a new object but the access to the values of class variables and/or the default values of instance attributes. An instantiation side-effect is terminal.

Item name	Value	Purpose
`Argument`	0	An instantiation side-effect does not take an argument.
`ArgumentType`		This item is not relevant.
`Action`	`Read`	An instantiation effect is a read effect because it accesses information contained in the class to which the new instance belongs.
`Data`	string value	This is the name of the class to which the new instance belongs.
`UpdateValue`	0	An instantiation side-effect is a read side-effect: it does not have an update value.
`Next`	()	An instantiation side-effect is a terminal side-effect: it does not propagate an affected data property.

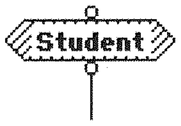
An `Init` operation taking a list of (attribute name, value) pairs on its input and is considered equivalent to an operation with an inject terminal. From a language point of view, such an `Init` operation should be considered to be an `Init` operation with several inject terminals, one for each pair in the list of (attribute name, value) pairs. But from the point of view of the effect inference, this approximation does not make any difference because the effect inference cannot built the effect signature of an operation with an inject terminal.

The instantiation side-effect is printed as:

`-ALR(Data ())`

`AL` stands for **Al**location.

The effect signature of the operation shown below:



is:

`ALR("Student" ())`

6.3.3.5 Local effects

Some data objects come into existence in the scope of the current case as the return value of a `Constant`, `Init` or primitive operation. However, there exists no side-effect to record the creation of data object (the purpose of the instantiation side-effect is only to record that the values of class attributes and/or the default values of instance attributes have been accessed).

Instead, a *local* affected data property can be created to indicate that a value on a datalink has come into existence in the scope of the current case. This affected data property is created only if a side-effect refers to the datalink to which the affected data property should be attached. In fig. 6.8, the write side-effect refers to the datalink connected to the root of the `Constant` operation.

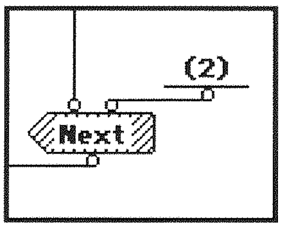


Fig. 6.8: Local affected data property

A local effect-data property is created and attached to the link connected to the root of the `Constant` operation and contains the following items of information:

Item name	Value	Purpose
`Argument`	0	The value 0 indicates that the data item "appeared" on the datalink to which the affected data property is attached.
`ArgumentType`		This item is not relevant.
`Action`	`Read`	A local affected data property is a read affected data property.
`Data`	string	This is a value which is constructed to identify the link to which the affected data property is attached.
`UpdateValue`	0	A local affected data property is a read property : it does not have an update value.

A local affected data property is printed as:

```
NEW("##")
```

`"##"` is printed instead of the `Data` item because the value of `Data` would be rather difficult to interpret.

6.3.3.6 Effects on persistents

A persistent side-effect results from the execution of a persistent `Get` or a persistent `Set` operation.

Item name	Value	Purpose
`Argument`	0	A persistent side- effect does not take an argument.
`ArgumentType`		This is not relevant.
`Action`	`Read/Write`	A persistent side-effect can be a read or a write side-effect.
`Data`	string	This is the name of the persistent whose value is accessed or set.
`UpdateValue`	0 for read side-effect/ 1 for write side-effect	A persistent read side-effect does not take an update value/ the update value of a persistent write side-effect flows into the first terminal of the `Set` persistent operation.
`Next`	(1) for read side effect/ ( ) for write side-effect	The result of the read side-effect is propagated on the first root of the persistent `Get` operation/ a persistent write side-effect is a terminal side-effect : it does not propagate a result.

Side-effects on persistents and their matching affected data properties are represented by one of the following expressions:

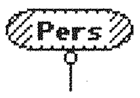
- `PER(Data Next)` (read side-effect)

- `PER(Data)`

- `PEW(Data UpdateValue Next)` (there is no write persistent affected data property)

`PE` stands for **Pe**rsistent.

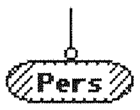
The effect signature for the `Get` persistent operation shown below:



is:

```
PER("Pers" (1))
```

The effect signature for the `Set` persistent operation shown below:



is:

PEW("Pers" 1 ())

6.3.3.7 Effects on lists

Effects on lists must be handled with special care. The difficulty of describing the effects induced by primitive operations on lists is compounded by the existence of a list annotation for the roots and terminals of operations.

Item name	Value	Purpose
`Argument`	integer	This is a reference to the input of the list primitive operation which passes the argument of the side-effect.
`ArgumentType`		A list side-effect does not record the type of its argument.
`Action`	`Read/Write`	A list side-effect can be a read or a write side-effect.
`Data`		This item is not relevant.
`UpdateValue`	0 for a read side-effect)/ integer value for a write-effect	A list read side-effect takes no update value/reference to the input of the list primitive operation which passes the update of the side-effect.
`Next`	list of integers	This item lists the outputs of the list primitive operation through which the results of the side-effect are propagated.

The representations for affected data properties and side-effects on lists are:

- `LIR(Argument ())` (list read side-effect)
- `LIR(Argument)`
- `LIW(Argument UpdateValue ())` (list write side-effect)
- `LIW(Argument UpdateValue)`

`LI` is the short notation for **LI**st.

The schematic in fig. 6.9 shows an example of the the internal working of the `attach-r` primitive.

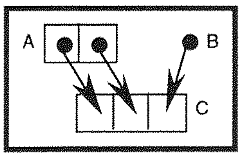


Fig. 6.9: Internal behaviour of `attach-r`.

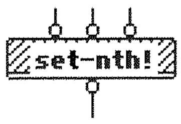
The references contained in the slots of list `A` are put in the first two slots of the newly created list `C`, reference `B` is put in the third slot of `C`. The effect signature of the `attach-r` primitive operation contains two side-effects:

- A list read side-effect: the extraction of the references stored in `A` can be described as a read side-effect with list `A` as its argument and list `C` as its result.
- An identity side-effect: the insertion of the references in list `C` is described using an approximation: list `C` can be identified with each of the values to which it points.

The effect signature of `attach-r` is:

`LIR(1 (1))` × `IDE(2 (1))`

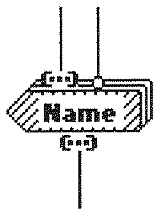
The only primitive method to cause a write side-effect on a list is the `set-nth!` primitive:



The signature of `set-nth!` is:

`LIW(1 2 (1))`

The effect signature of an operation with a `list` annotated terminal may have to be modified. If a side-effect has a reference to an input with a list annotation, the reference to this input must replaced by a list read affected data property. The argument of this affected data property is the original reference to the operation input. This substitution reflects the fact that the `Argument` or the `UpdateValue` of the side-effect has been extracted from the list flowing into input. In the operation shown below:



the leftmost terminal is annotated, so the reference to input 1 in the `Argument` of the side-effect must be replaced by a list read affected data property. The original effect signature of the `Get` operation is:

```
OAW(1 "Name" 2 (1))
```

The effect signature becomes:

```
OAW(LIR(1) "Name" 2 (1))
```

A list annotation on a root has no consequences for the effect signature of the operation.

6.3.3.8 External effects

External side-effects are used for the effect signature of the operations for which an effect signature cannot be inferred.

Item name	Value	Purpose
`Argument`	0	An external side-effect does not take an argument.
`ArgumentType`		This item is not relevant.
`Action`		This item is not relevant.
`Data`		This item is not relevant.
`UpdateValue`	0	This item is not relevant.
`Next`	list of integers	An external side-effect is a terminal side-effect: it does not propagate an affected data property.

No effect signature can be inferred for an operation that calls an external method. This decision is justified by the fact that a precise description of the side-effects induced by an external method would require a knowledge of the behaviour of all system calls defined for the Macintosh operating system. The inference mechanism takes the conservative view that the execution of any external method induces an external side-effect.

Also, the effect signature of an operation with an inject terminal cannot be known statically.

External side-effects are printed as:

`EXT(())`

6.3.3.9 Effect expressions and variable arity

Primitive methods may be called with variable numbers of terminals or variable numbers of roots. The signatures of all primitive methods are stored in a signature repository and they can be retrieved during the initialisation phase of the type and effect inference to construct the signatures of the operations that call primitive methods.

The same notation (...) is used for varity terms in the effect part as in the type part of the primitive signature. Two cases can be distinguished: the primitive method may be called with a variable number of terminals or with a variable number of roots.

- If the primitive has a variable number of terminals, the varity term appears at the top level of the effect signature and can be substituted with a side-effect. The substituted side-effect is the same as the one on the left of the varity term except for the value of its argument which is incremented by one for each extra terminal. The expansion rule is illustrated by the following example.

  The signature of the primitive `attach-r` in the signature repository is:

  `(<Universal+>) × <Universal+> × ... → L(U(E(1) 2 ...)) /*SE*/LIR(1 (1)) × IDE(2 (1)) × ...`

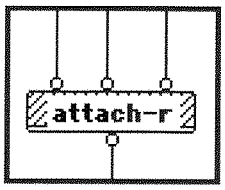


Fig. 6.10: The `attach-r` primitive

If `attach-r` is called by an operation with three terminals (see fig. 6.10), the signature computed for the operation during the initialisation phase will be:

`(<Universal+>) × <Universal+> × <Universal+>→ L(U(E(1) 2 3)) /*SE*/LIR(1 (1)) × IDE(2 (1)) × IDE(3 (1))`

• In the case of a primitive method with a variable number of roots, the varity term may appear in the `Next` information item of the side-effect because `Next` refers to the outputs of the operation. When constructing the signature of the operation calling the primitive method, the varity term is substituted with the sequence of the indices of the extra roots. The primitive `detach-r` can be called with a variable number of outputs. Its formal signature is:

`(<Universal+>) → L(U(E(1) <∅>)) × E(1) × .../*SE*/LIR(1 (1 2 ...))`

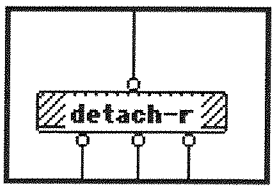


Fig. 6.11: The `detach-r` primitive

The signature constructed for `detach-r` with three outputs (fig. 6.11) is:

`(<Universal+>)→ L(U(E(1) <∅>)) × E(1) × E(1)/*SE*/LIR(1 (1 2 3))`

6.3.3.10 Operations on side-effects

As will be shown in the following subsections, the effect inference mechanism combines side-effects to eliminate duplicate information. This simplification is possible when some conditions are met.

Two affected data properties are equal if

- they belong to the same category of effects

and:

- the values of all their information items are equal

Affected data property ε_1 *overlaps* affected data property ε_2 if

- `Argument` of ε_1 is equal to ε_2 or `Argument` of ε_1 overlaps ε_2

Similarly, two side-effects are equal if:

- they belong to the same category of effects

and:

- the values of all their information items are equal.

Two side-effects complement each other if:

- they belong to the same category

and:

- the values of all their information items are equal except for the `Next` information item.

Side effect σ_1 overlaps side-effect σ_2 if:

– σ_2 is a terminal side-effect

and:

– ε'_2 being the affected data property obtained by truncating the `Next` information from the side-effect σ_2, the argument of σ_1 overlaps ε'_2.

Depending on the relation existing between a pair of side-effects (equality, complementarity, overlap or none of these), their combination will yield a different result. The results are presented in the following table:

Relation between σ_1 and σ_2	Result of the combination of σ_1 and σ_2
σ_1 equals σ_2	(σ_1)
σ_1 complements σ_2	(σ'_1) (σ'_1 is equal to σ_1 except for `Next` which contains the references held in `Next` of σ_1 and `Next` of $\sigma2$)
σ_1 overlaps σ_2	(σ_1)
σ_1 is overlapped by σ_2	(σ_2)
No relation between σ_1 and σ_2	$(\sigma_1\ \sigma_2)$

The possible relations between side-effects are illustrated by the following matrix. Each relation is read from the side-effect in the row to the side-effect in the column.

	`PER("Pers" ())`	`OAR(PER("Pers") "Surname" (1))`	`OAR(1 "Surname" (1))`	`OAR(1 "Surname" (2))`
`PER("Pers" ())`	is equal	is overlapped by	no relation	no relation
`OAR(PER("Pers") "Surname" (1))`	overlaps	is equal	no relation	no relation
`OAR(1 "Surname" (1))`	no relation	no relation	is equal	complements
`OAR(1 "Surname" (2))`	no relation	no relation	complements	is equal

6.3.4 Inference Algorithm

Like the type inference to which it is tightly integrated, the effect inference is applied to the successive cases of the method analysed.

6.3.4.1 Case-wide inference

The case-wide effect inference is divided into three stages:

- Initialisation of the affected data properties attached to the datalinks of the case and the effect signatures of the operations (at the same time as their type signatures).
- The composition of the side-effects of the case by a single forward pass over the operations of the case (after the three passes required by the type inference).
- The construction of the effect part of the line for the case.

The effect information attached to the datalinks of the case consists of a (possibly empty) list of affected data properties. As for the type information, all datalinks connected to the same root share the same list of affected data properties. For the initialisation of the affected data properties, two types of links are distinguished: those connected to the roots of the `Input` operation and those not connected. The effect information attached to the datalinks after initialisation is presented in the table below:

Link	Effect property after initialisation
Connected to the input bar	(`IDE(n)`), integer n is the position in the sequence of inputs of the case of the input to which the datalink is connected (1 is the rank of the leftmost input)
Not connected to the input bar	() (empty list)

The identity affected data properties attached to the links connected to the inputs indicate that the values flowing on these links are those of the inputs of the case.

Type and effect signatures of the operations of the case are constructed during the initialisation phase before the three passes of the effect inference (see section 5.5).

The forward pass of the effect inference iterates the composition routine over the operations of the case following their execution order. A list of terminal side-effects is maintained as the effect inference proceeds along the graph of the case. For each operation, the composition routine can be divided into three steps:

- During the first step, it is checked that the inputs of the operation for which there exist affected data properties are referenced by at least one side-effect of the operation. This reference may be in the `Argument` or in the `UpdateValue` information item of the side-effect. If there exists no reference to the input, the effect information will be lost because it is not propagated down the graph. To avoid this loss, any affected data property which is not referenced must be converted into a terminal side-effect and added to the list maintained by the inference mechanism.

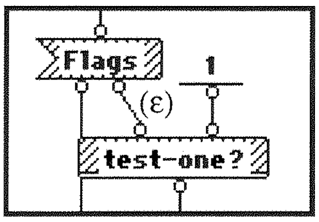


Fig. 6.12: Unpropagated affected data property

In the example of fig. 6.12, the primitive `test-one?` has no side-effect and the affected data property ε must be converted into a terminal side-effect.

• The side-effects of the operation are applied during the second step. Applying a side-effect means substituting the references to inputs with the affected data properties for the matching inputs. If the signature of the operation comprises several lines, the side-effects contained in the different lines are combined in order to eliminate duplication.

If there is no affected data property on the referenced input a local affected data property is created. For example, in fig. 6.13, as there is no affected data property attached to the input referenced by the `UpdateValue` item of the side-effect, a local affected data property must be created.

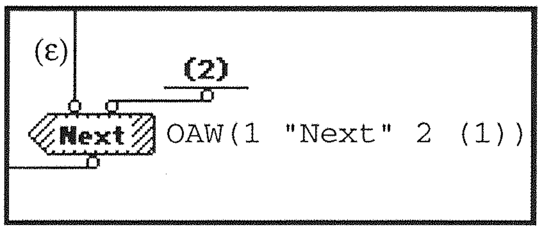
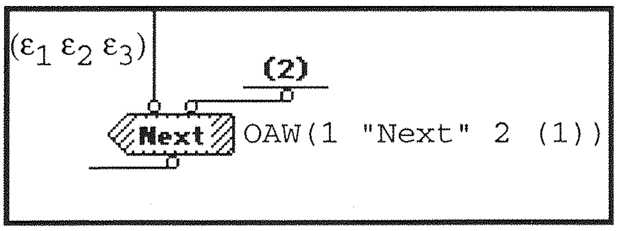


Fig. 6.13: Reference to a local value.

After application, the side-effect of the `Set` operation becomes:

```
OAW(ε "Next" NEW("##") (1))
```

If there are several affected data properties attached to a referenced datalink, the side-effect which refers to this input is duplicated so that there as many side-effects as there are affected data properties. In fig. 6.14, the `Argument` of the side-effect refers to a datalink with three properties, the side-effect is duplicated twice so that three side-effects can be applied to the three affected data properties.



Fig, 6.14: Side-effect applied to several affected data properties

After application the three side-effects become:

```
OAW(ε1 "Next" NEW("##") (1))
OAW(ε2 "Next" NEW("##") (1))
OAW(ε3 "Next" NEW("##") (1))
```

Likewise, if the `UpdateValue` of a side-effect refers to a datalink to which several affected data properties are attached, the side-effect is duplicated to match the number of affected data properties on the referenced datalink.

If the affected data property for a matching input is an identity affected data property, the input reference is not substituted with the identity affected data property but with the `Argument` of the affected data property. For example, if for the side-effect `OAR(1 "Surname" (2))`, the affected data property for the first input of the operation is `IDE(2)`, the substitution of the reference to input 1 will produce `OAR(2 "Surname" (2))` instead of `OAR(IDE(2) "Surname" 2)`. This rule is designed to keep side-effect expressions simple.

When the substituted input reference is the `Argument` of a class or an instance side-effect, the type of the referenced datalink becomes the `ArgumentType` of the side-effect.

In the case of a composed side-effect, the `Argument` or the `UpdateValue` of the side-effect may be an affected data property which records its `ArgumentType`. The update needs to propagate the `ArgumentType` through to the affected data properties whose `ArgumentType` is the type of the referenced datalink. For the composed side-effect:

```
OAW(OAW(OAW(1 "Surname" NEW("##")) "Name" NEW("##")) "DOB"
NEW("##") (1)).
```

The type of the value flowing into the first input of the operation to which the side-effect is attached is `<Student>`. The `ArgumentType` of `OAW(1 "Surname" NEW("##"))` is updated to `<Student>`. The type of the value to which the affected data property `OAW(1 "Surname NEW("##"))` is attached is also `<Student>` so the `ArgumentType` of the affected data property:

`OAW(OAW(1 "Surname" NEW("##")) "Name" NEW("##"))`

must also be updated to `<Student>`. The same reasoning applies to the side-effect itself and the `ArgumentType` of the side-effect must be updated to `<Student>`.

But for the side-effect `OAR(OAR(1 "Father") "Profession" (1))` only the `ArgumentType` of `OAR(1 "Father")` can be updated. The explanation is that `OAR(1 "Father")` is the affected data property of the value of the attribute `Father` of the instance coming onto the first input of the operation and there is no dependency between the type of an object (the instance on the first input of the operation) and the type of the value of an attribute of that object (the value of the `Father` attribute). Consequently, the `ArgumentType` of `OAR(OAR(1 "Father") "Profession" (1))` cannot be updated.

• The third step consists of propagating the affected data properties resulting from the application of the side-effects of the operation. Terminal side-effects are added to the list of terminal side-effects. The `Next` item of non-terminal side-effects refers to the outgoing datalinks to which an affected data property must be attached. If the output is not connected, the side-effect is converted into a terminal side-effect and added to the list. If the output is connected, the affected data property to be attached to it is obtained by truncating the `Next` information item of the side-effect (fig. 6.15).

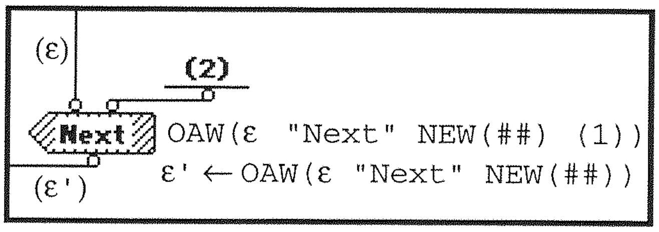


Fig. 6.15: Propagation of an affected data property

ε', the affected data property of the outgoing link of the `Set` operation results from the truncation of the side-effect of the operation.

The identity side-effect distinguishes itself in the way it propagates its affected data properties. If the `Argument` of the identity side-effect is an affected data property, this `Argument` is propagated in place of an identity affected data property itself. For example, the side-effect `IDE(PER("Pers") (1))` will propagate the `PER("Pers")` affected data property on the first output of the operation but the side-effect `IDE(1 (1))` would propagate the property `IDE(1)`. This rule aims at keeping affected data property expressions as simple as possible.

When the bottom boundary of the case graph is reached, the relevant information comprises the list of terminal side-effects and the affected data properties attached to the datalinks connected to the output operation.

All the affected data properties attached to an output datalink are converted into side-effects. The conversion of an affected data property into a side-effect requires that the value of each information item of the affected data property becomes the value of the corresponding information item of the new side-effect. The value of `Next` for the new side-effect is the list of the references of the outputs of the case to which the property is attached.

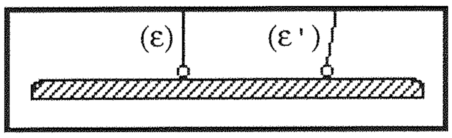


Fig. 6.16: Affected data properties reaching the output operation.

In fig. 6.16, ε is converted into a side-effect with the value `(1)` for the `Next` field and ε' with a side-effect with a `Next` field of `(2)`.

Since the same affected data properties may be propagated along different datalinks in the case it is likely that some of the side-effects gathered at the bottom of the case and some terminal side-effects will either overlap, be equal or complement each other and they are combined to eliminate duplication.

6.3.4.2 Method

The effects inferred for each case are in turn combined to obtain the method effect signature.

6.3.5 Handling recursion

The handling of the recursion follows the approach taken for the type inference and described in 5.8.7.

The dummy signature of an operation making a recursive call comprises no side-effect. Effect inference applied to a recursive method will be illustrated by an example in chapter 7.

6.3.6 Effect inference example

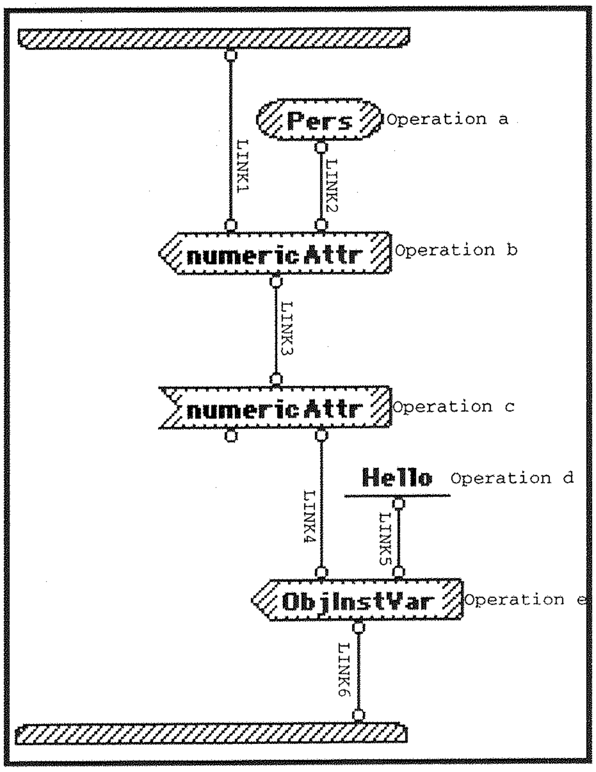


Fig. 6.17: Example of effect inference

The example in fig. 6.17 has been designed to illustrate the effect inference and does not correspond to any useful code. To understand the code better, it is necessary to describe a subset of the class hierarchy (fig. 6.18).

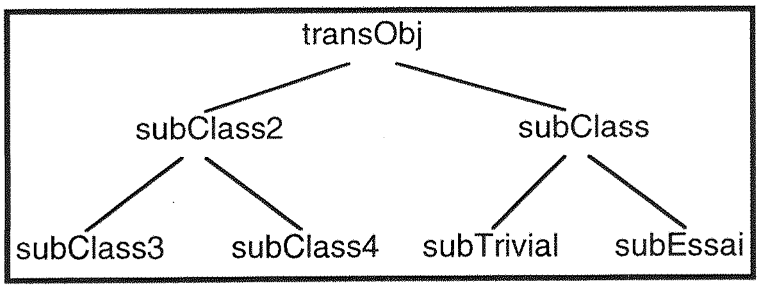


Fig. 6.18: Test class hierarchy

The class `transObj` defines `numericAttr` as a class attribute and `ObjInstVar` as an instance attribute.

6.3.6.1 Initialisation phase

The effect analysis proceeds in two phases. During the set-up phase, the type and effect signatures of the operations on the graph are constructed. The effect signatures of the operations are shown in fig. 6.19

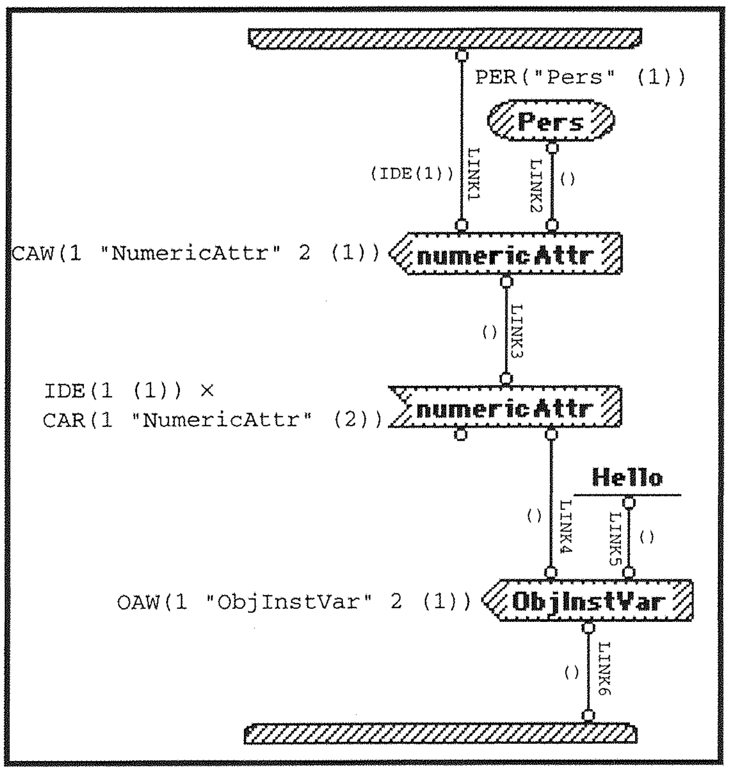


Fig. 6.19: After the initialisation phase

Class and instance side-effects record an `ArgumentType` value. This information is not shown in the textual representation of these side-effects, it is given in the table below:

Operation	Effect category/ action	`ArgumentType`
b	Class/ Write	`["transObj"+|<transObj+>]`
c	Class/ Write	`["transObj"+|<transObj+>]`
e	Class/ Write	`["transObj"+|<transObj+>]`

6.3.6.2 Composition of the side-effects

During the second step of the effect inference, the side-effects of the operations are composed during a forward pass over the case. The composition routine proceeds with three steps for each operation of the case following the execution order:

- Check that all the affected data properties attached to the incoming datalinks of the operation are referenced by the side-effects of the operation.
- Apply the side-effects of the operation.
- Propagate the affected data properties resulting from the application of the side-effects on the outgoing datalinks of the operation or add a side-effect to the list of terminal side-effects.

The inference starts with an empty list of terminal side-effects. The different steps of the composition routine are now detailed for each operation:

- The effect signature of `Operation a` contains one side-effect `PER("Pers" (1))`. As the operation has no input the first and second step of the composition routine can be ignored. The affected data property `PER("Pers")` is constructed and propagated onto `LINK2` of the case. No terminal side-effect is added to the list.
- The signature of `Operation b` contains one side-effect `CAW(1 "numericAttr" 2 (1))`. Both `LINK1` and `LINK2` have one affected data property attached to them and the side-effect has references to the two inputs to which the datalinks are connected. The side-effect is applied:

  - The reference to input 1 is replaced with the affected data property attached to `LINK1`.

  - The reference to input 2 is replaced with the affected data property attached to `LINK2`.

  - The `ArgumentType` of the side-effect is replaced with the type of `LINK1`.

  After its application, the side-effect becomes:

```
CAW(1 "numericAttr" PER("Pers") (1))
```

The affected data property `CAW(1 "numericAttr" PER("Pers"))` is propagated on `LINK3`. No terminal side-effect is added to the list.

• The signature of `Operation c` contains two side-effects, `IDE(1 (1))` and `CAR(1 "numericAttr" (2))`. `LINK3` has an affected data property attached to it but the input to which `LINK3` is connected is referenced by both side-effects.

The identity side-effect is applied:

- The reference to input 1 is replaced with the affected data property attached to `LINK3`.

After its application, the identity side-effect becomes:

```
IDE(CAW(1 "numericAttr" PER("Pers")(1))
```

The class read side-effect is applied:

- The reference to input 1 is replaced with the affected data property attached to `LINK3`.

- The `ArgumentType` of the side-effect is replaced with the type of `LINK3`.

The applied side-effect is:

```
CAR(CAW(1 "numericAttr" PER("Pers")) (2))
```

There is no link connected to the first root of the `Get` operation, so the affected data property to be propagated by the identity side-effect is converted into a terminal side-effect:

```
CAW(1 "numericAttr" PER("Pers")())
```

This terminal side-effect is added to the list of terminal side-effects.

The applied class read side-effect propagates the following affected data property on `LINK4`:

```
CAR(CAW(1 "numericAttr" PER("Pers")))
```

No terminal side-effect is added to the list.

• The signature of `Operation d` contains no side-effect and the operation has no input, so the three steps of the composition routine can be ignored. No terminal side-effect is added to the list.

• The signature of `Operation e` contains one side-effect `OAW(1 "ObjInstVar" 2 (1))`. There is an affected data property attached to `LINK4` and none attached to `LINK5`. The side-effect of the operation has a reference to the two inputs to which `LINK4` and `LINK5` are connected.

The instance write side-effect is applied:

- The reference to input 1 is replaced with the affected data property attached to `LINK4`.

- There is no affected data property attached to `LINK5`, so a local affected data property `NEW("##")` is created and attached to `LINK5` and replaces the reference to input 2 in the side-effect expression.

The applied side-effect is:

```
OAR(CAR(CAW(1 "numericAttr" PER("Pers"))) "ObjInstVar"
New("##") (1))
```

The side-effect propagates on `LINK6` the following affected data property:

```
OAR(CAR(CAW(1 "numericAttr" PER("Pers"))) "ObjInstVar"
New("##"))
```

No terminal side-effect is added to the list.

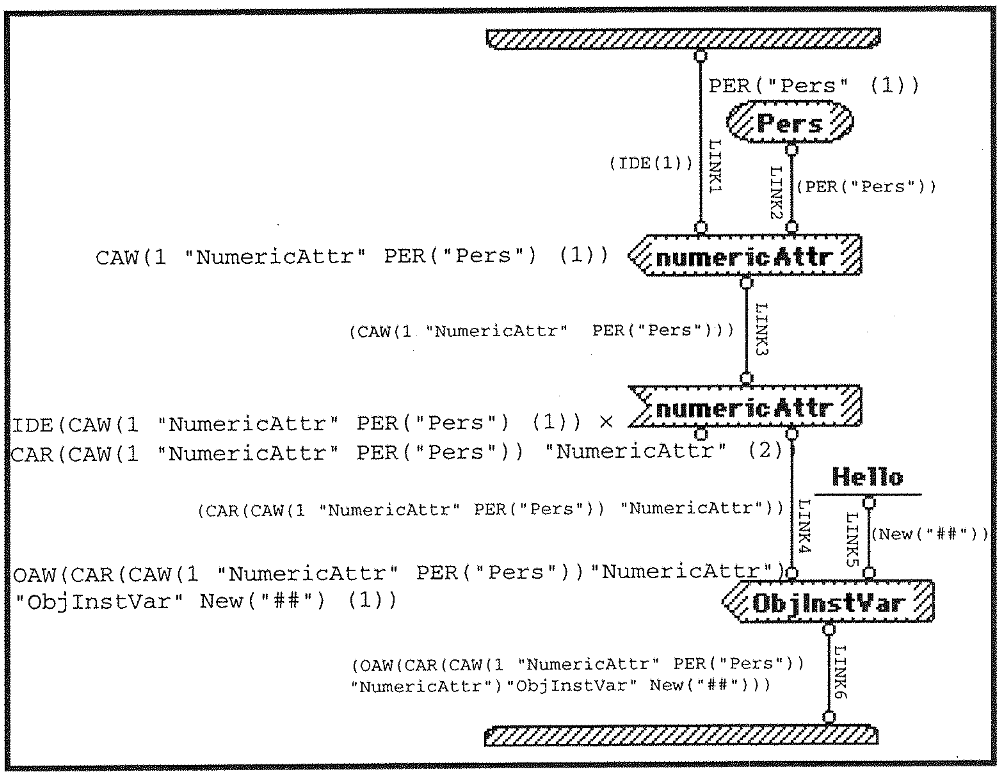

PER("Pers" (1))
Pers
(IDE(1))
LINK1
LINK2
(PER("Pers"))
CAW(1 "NumericAttr" PER("Pers") (1))
numericAttr
(CAW(1 "NumericAttr" PER("Pers")))
LINK3
IDE(CAW(1 "NumericAttr" PER("Pers") (1)) ×
CAR(CAW(1 "NumericAttr" PER("Pers")) "NumericAttr" (2))
numericAttr
Hello
(CAR(CAW(1 "NumericAttr" PER("Pers")) "NumericAttr"))
LINK4
LINK5
(New("##"))
OAW(CAR(CAW(1 "NumericAttr" PER("Pers"))"NumericAttr")
"ObjInstVar" New("##") (1))
ObjInstVar
(OAW(CAR(CAW(1 "NumericAttr" PER("Pers"))
"NumericAttr")"ObjInstVar" New("##")))
LINK6

Fig. 6.20: After the composition of the side-effects

When the analysis reaches the bottom of the graph, the list of terminal side-effects contains one terminal side-effect (added by the identity side-effect of `Operation c`):

```
(CAW(1 "numericAttr" PER("Pers") ()))
```

The affected data property attached to `LINK6` must be converted into a side-effect. Since `LINK6` is attached to the only output of the case, the value of `Next` property for the converted side-effect is `(1)`:

```
OAW(CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr")
"ObjInstVar" NEW("##") (1))
```

The terminal side-effect is overlapped by the side-effect converted from the affected data property of `LINK6` and can therefore be safely discarded.

The effect signature constructed for the method consists of the side-effect converted from the property of `LINK6`.

6.4 Effect Synthesis

The aim of the synthesis is to produce an approximation of the accesses and updates to the operation inputs and global variables that the execution of an operation annotated for distribution would induce. Using Palsberg and Schwartzbach's words [Palsberg and Schwartzbach 1991], the effect synthesis must tell *the whole truth* but may not tell *nothing but the truth*. This means that the approximation produced by the effect synthesis must be able to detect all the accesses and updates that may occur at run-time but it may also predict accesses and updates that will not occur at run-time.

6.4.1 Outline of the synthesis

The effect synthesis is only applied to the operations that have been annotated for distribution. The synthesis proceeds in three stages:

- Type inference is carried out in the case in which the operation annotated for distribution occurs.
- Effect inference is initiated on the case. During the forward pass of the effect inference, when the composition routine processes an operation annotated for distribution, all the side-effects of this operation are duplicated before being applied. The composition routine is applied as described in 6.3.6.2 to the original set of side-effects so that the effect inference can proceed.

  For the side-effects in the duplicate set, only their `ArgumentTypes` are updated (if required) but not their `Argument` or `UpdateValue`. The reason for doing so is that the synthesis must be applied only to the side-effects belonging the annotated operation and not to the side-effects composed with the side-effects of the operations occurring earlier in the case.
- The duplicate side-effects of the annotated operation are synthesised.

The information produced by the synthesis addresses two issues:

- Access to the global variables during the execution of an operation annotated for distribution.
- The analysis must record the updates performed on the arguments of the operation and on the global variables. An update on an argument or a

global variable may result from the *composition* of side-effects. In the case of nested structures, a structure is extracted from its containing structure and this can be seen as a read side-effect. However, if the extracted value is modified by another operation, the effective result will be viewed as an update of both the extracted and the containing structures. Although the update may not concern a slot of the containing structure, the modification is considered to affect it, by composition. The argument to justify this view is that the value of the containing structure is the graph whose highest vertex is the structure and that any modification to the graph is a modification of the value of the containing structure.

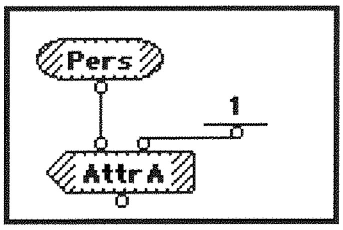


Fig. 6.21: Composition of effects.

Fig. 6.21 shows the example of a persistent which contains an instance of a class, the value of the persistent is read and the instance is modified. The execution of these two operations shown in fig. 6.21 leads to an update on the persistent as well as on the instance.

The concept of *route* is introduced to describe how a data object has become available in the case(s) of the method called by the operation annotated for distribution. The route of a value is the highest vertex of the graph followed to access the data.

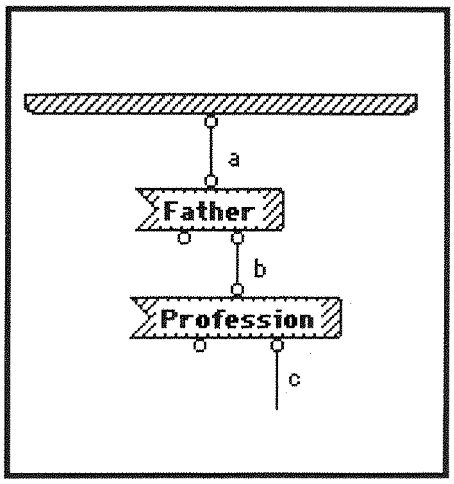


Fig. 6.22: The route of a value

Fig. 6.22 shows the case of a method, in this case the value of `c` has been reached via the input argument `a`, thus a possible route for the value is an input route.

A state operation describes an access or an update to a data structure (e.g. a class, a persistent, an instance or a list).

6.4.2 Routes

A value becomes available in the cases of the method called by an operation annotated for distribution in different ways. Each possible way defines a category of routes:

Route category	Description
`Local`	The value has been reached through a value instantiated locally.
`Persistent`	The value has been reached through a persistent value.
`Class`	The value has been reached through the value of a class attribute or the default value of an instance attribute.
`Input`	The value has been reached through the value of an input of the case.

Beyond its category, a route is described with two information items:

- `Data` is an integer or a string.

• `Depth` is an integer value which keeps track of the number of indirections necessary to reach the current value from the root of the graph.

The use of the `Depth` field is illustrated in fig. 6.23:

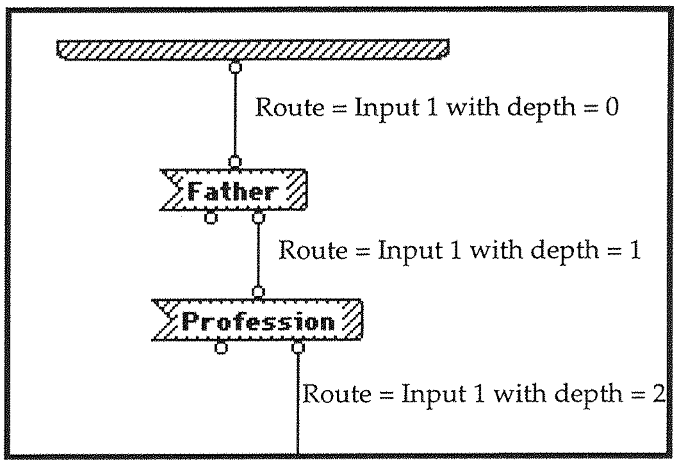

Route = Input 1 with depth = 0
Father
Route = Input 1 with depth = 1
Profession
Route = Input 1 with depth = 2

Fig. 6.23: Depth of a route

6.4.2.1 Class routes

A class route indicates that a data object has become accessible through a class. The textual representation for a class route is:

`C√(Data Depth)`

A route of depth zero represents the class itself. A value extracted from a class structure (either the value of a class attribute or the default value of an instance attribute) has a route of depth 1. The `Data` value of a class route is the name of a class and the notation `"a"+` means the class `a` and all the subclasses of `a`. In the example shown in fig. 6.24, `numericAttr` is the name of a class attribute.

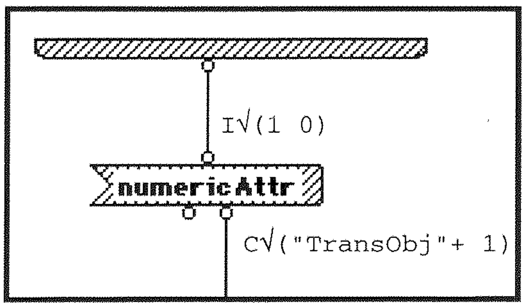


Fig. 6.24: A class route.

6.4.2.2 Input routes

An input route indicates that a data object has been passed as an argument to the operation annotated for distribution to be accessible in the case(s) of the method called by this operation. The textual representation for an input route is:

`I√(Data Depth)`

A route of depth 0 represents the value of the case input itself. The `Data` value of an input route is an integer which refers to an input of the operation annotated for distribution.

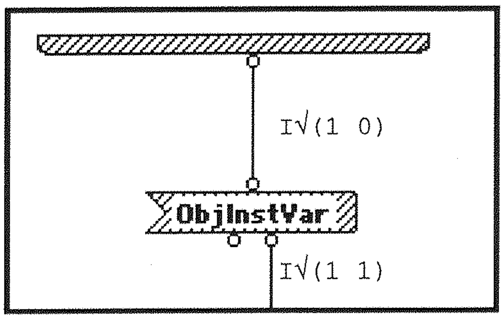


Fig. 6.25: Input routes

With the example shown in fig. 6.25, `ObjInstVar` is the name of an instance attribute. The input route on the outgoing datalink of the operation is one possible route for that value (the set of possible routes for that value will depend on the type of the input of the `Get` operation).

6.4.2.3 Local routes

A local route indicates that a data object has become accessible in the case(s) of the method called by the operation annotated for distribution through an object created during the execution of the called method. The textual representation for a local route is:

`L√("##" Depth)`

A local route of depth zero represents the newly created data object. The `Data` value of a local route is a string identifier built to identify the local route uniquely in the context of the current application.

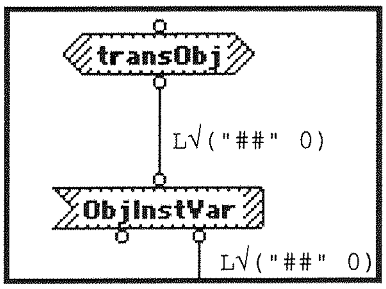


Fig. 6.26: Local routes

With the example shown in fig. 6.26, the local route on the outgoing datalink of the operation is one possible route for that value.

6.4.2.4 Persistent routes

A persistent route indicates that a data object has become accessible in the case(s) of the method called by the annotated operation through a persistent. The textual representation for a persistent route is:

`P√(Data Depth)`

A persistent route of depth zero represents the persistent and the value extracted from the persistent has depth of 1. The `Data` value for a persistent route is the name of the persistent through which the value has become available (see fig.6.27).

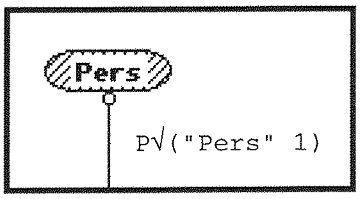


Fig. 6.27: A persistent route.

6.4.3 State operations

The concept of side-effect used for the effect inference and the concept of state operation seem to overlap but they do not entirely:

- A side-effect describes how an operation accesses or updates its inputs or some global variables to produce an output value. However, the information provided by a side-effect leaves some ambiguity about which data structure is accessed or updated. In the case of a side-effect on a persistent there is no ambiguity, the information recorded by the side-effect gives away which data structure is going to be accessed. An instance read or write side-effect is more ambiguous and depending on the `ArgumentType` of the side-effect, the operation will update either an instance structure or a class structure.

• The purpose of a state operation is to record an access or an update to a data structure. Whereas the focus of a side-effect was to describe how the access or the update was carried out (e.g. persistent read side-effect, instance write side-effect), a state operation is concerned with which data structure is accessed or updated. The correspondence between side-effects and state operation may seem one to one, for example, a persistent side-effect maps to a state operation. However, this is not the case for an instance side-effect which can be mapped to either a state operation on a class structure or a state operation on an instance structure.

Five categories of state operations are available, reflecting different ways of accessing or updating data structures in Prograph. External data structures can also be accessed or updated but they have not been included in the list because the effect synthesis does not consider them:

State operation	Purpose
Class state operation	Record the access or the update of the value of a class attribute or the default value of an instance attribute.
Instance state operation	Record the access or the update of the value of an instance attributes.
Allocation state operation	Record the creation of an instance of a class.
Persistent state operation	Record the access or the update of a persistent value.
List state operation	Record the access or the update of the value of a list element.

A state operation can be an access or an update state operation. Like a read side-effect, an access state operation has an argument and, like a write side-effect, an update state operation has both an argument and an update value.

All state operations are described using the relevant combination of the following data items:

Information item	Purpose
`Action`	Distinguish between an access or update state operation.
`Data`	Record some textual information such as the name of an attribute or of a persistent.
`Argument`	Store the route of the argument of the state operation in order to know how the data structure which is accessed or modified has become available in the case(s) of the method called by the annotated operation.
`UpdateValue`	Store the route of the update value of the state operation in order to know how the data structure which is accessed or modified has become available in the case(s) of the method called by the annotated operation.

6.4.3.1 Class state operations

Information item	Value	Explanation
`Action`	Access/ Update	
`Data`	String	The name of the attribute accessed or updated
`Argument`	Class route	Classes can be reached globally. The route of a class state operation is always a class route whose `Data` value is the name of a class and `Depth` is zero.
`UpdateValue`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure or the persistent before being passed as update value to the state operation.

A class route is needed as `Argument` because different classes may define an attribute with the same name. The information provided by `Data` is not sufficient to know which class data structure will be accessed or updated.

The textual representations for class state operations are:

```
C_access(Data Argument)
```

```
C_update(Data Argument UpdateValue)
```

6.4.3.2 Instance state operations

Information item	Value	Explanation
`Action`	Access/ Update	
`Data`	String	The name of the attribute accessed or updated.
`Argument`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure of the persistent before being passed as update value to the state operation.
`UpdateValue`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure or the persistent before being passed as update value to the state operation.

The textual representations for instance state operations are:

```
I_access(Data Argument)
```

```
I_update(Data Argument UpdateValue)
```

6.4.3.3 Allocation state operations

Information item	Value	Explanation
`Action`	Access	
`Data`	String	The name of the class to which the new instance belongs.
`Argument`	-	Not relevant
`UpdateValue`	-	Not relevant

The textual representation for allocation state operations is:

```
Alloc(Data)
```

6.4.3.4 Persistent state operations

Information item	Value	Explanation
`Action`	Access/ Update	
`Data`	String	The name of the persistent accessed or updated.
`Argument`	-	Not relevant.
`UpdateValue`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure or the persistent before being passed as update value to the persistent state operation.

A persistent state operation does not use `Argument` because the name of the persistent stored in `Data` is enough to know which persistent will be accessed or updated.

The textual representations for persistent state operations are:

```
P_access(Data)

P_update(Data UpdateValue)
```

6.4.3.5 List state operations

Information item	Value	Explanation
`Action`	Access/ Update	
`Data`	-	Not relevant.
`Argument`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure of the persistent before being passed as argument to the list state operation.
`UpdateValue`	Any route	The depth for a class route or a persistent must be ≥ 1 because the value must be extracted from the class structure or the persistent before being passed as update value to the list state operation.

The textual representations for list state operations are:

```
L_access(Argument)
```

```
L_update(Argument UpdateValue)
```

6.4.5 Synthesis algorithm

The synthesis is applied to all the side-effects of the operation annotated for distribution to produce a list of state operations.

6.4.5.1 Outline of the algorithm

A function called `Synthesise` is iteratively applied to the side-effects of the operation annotated for distribution. The list of state operations by one iteration of `Synthesise` is passed as the input list of state operations for the next iteration of the function as shown in fig. 6.28.

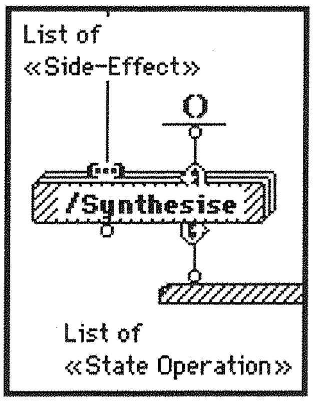


Fig. 6.28: Synthesis of the side-effects of an annotated operation

The type signature of the `Synthesise` function is:

`(Side-Effect + Affected Data Property + Integer) × State Operation* →`
`Route* × State Operation*`

The behaviour of the `Synthesise` function is best understood by looking at its implementation in Prograph (see fig. 6.29 a, b and c). Fig. 6.29.a and b show the synthesis of an input reference:

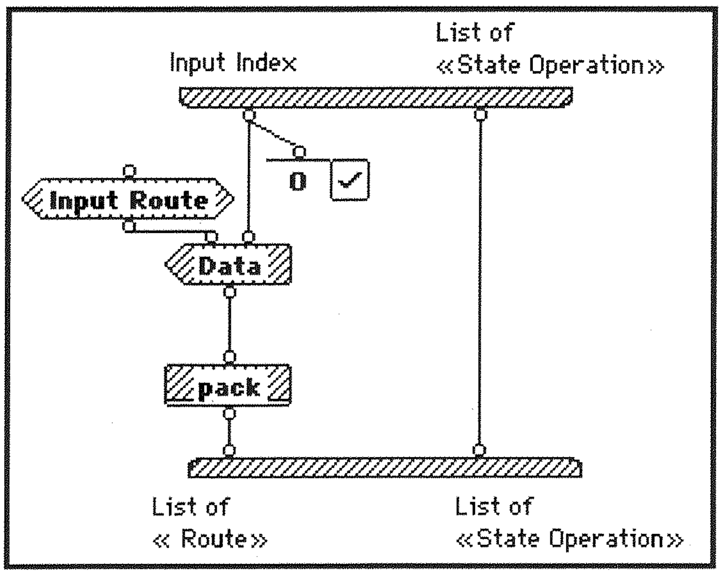


Fig. 6.29.a: Synthesis of an input reference other than 0

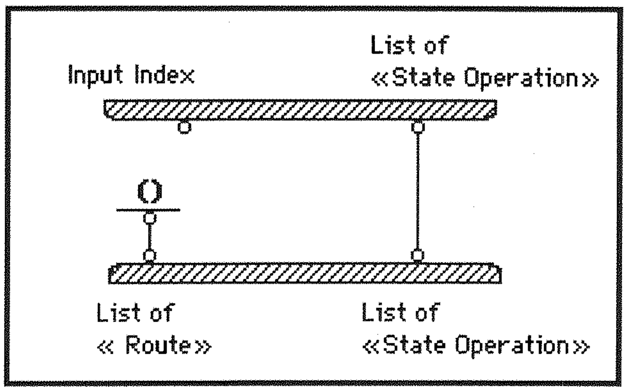


Fig. 6.29.b: Synthesis of a 0 input reference.

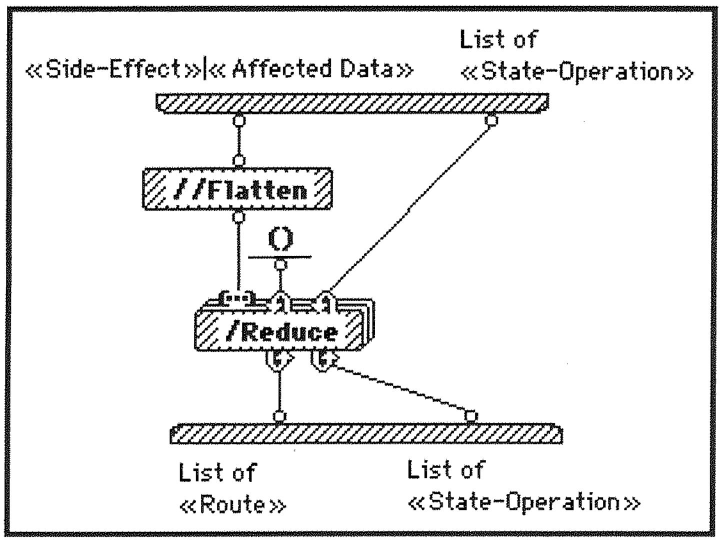

«Side-Effect»|« Affected Data»
List of «State-Operation»
//Flatten
()
/Reduce
List of «Route»
List of «State-Operation»

Fig. 6.29.c: `Synthesise` defined for a side-effect or an affected data property.

`Synthesise` takes a side-effect (or an affected data property) and a list of state operations as its arguments and proceeds in two steps:

- The `Flatten` operation produces a list representation of the side-effect or affected data property passed as argument to the `Synthesise` function.

- The `Reduce` operation is applied iteratively to the elements of the list representation of the side-effect.

The side-effect (or affected data property) to be synthesised may have a recursive data structure as `Argument` or `UpdateValue`. Recursion occurs when the `Argument` or the `UpdateValue` of the side-effect is an affected data property and the `Argument` of this affected data property is itself an affected data property (as the result of the composition of side-effects). The following side-effect has a recursive `Argument`:

```
OAR(OAR(PER("Info") "Father") "Surname") (1))
```

because the `Argument` of the side-effect is the instance read affected data property `OAR(PER("Info") "Father")` and the `Argument` of this affected data property is the persistent read affected data property `PER("Info")`.

The following side-effect has a recursive `UpdateValue`:

```
OAW(1 "Surname" OAR(PER("Father") "Surname") (1))
```

A side-effect is expanded into a list following the path of its `Argument` information item. If along the path an `UpdateValue` is a recursive affected data property, it is left untouched as it will be expanded at a later stage during the synthesis. The flattening is illustrated by the following example. The side-effect:

```
CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr" (1))
```

is expanded into a list as follows:

The side-effect becomes the first element of a list:

```
(CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr" (1)))
```

The argument of the side-effect (underlined in the expression above) is copied and put at the front of the list:

```
(CAW(1 "numericAttr" PER("Pers")) CAR(CAW(1 "numericAttr"
PER("Pers")) "numericAttr" (1)))
```

Again, the `Argument` of the first element of the list (underlined in the expression above) is copied and becomes the head of the list:

```
(1 CAW(1 "numericAttr" PER("Pers")) CAR(CAW(1 "numericAttr"
PER("Pers")) "numericAttr" (1)))
```

The reference to input 1 becomes the head of the list and the recursion stops. The recursive flattening of a side-effect or an affected data property into a list stops when the first element is a reference to an input of the operation annotated for distribution or the value 0. 0 is the implicit value of `Argument` for an affected data property or a side-effect which takes no argument (a persistent read affected data property or a local affected data property are two examples of affected data properties taking no argument).

The overall effect of `Flatten` is to convert a representation of side-effects based on a partial order into a total order. The consequences of this conversion will be discussed in 6.4.7.

The iterations of the `Reduce` function over the list of effects yield a list of routes and an updated list of state operations. The type signature of `Reduce` is:

`(Side-Effect + Affected Data + Integer)` × `Route*` × `State Operation*` → `Route*` × `State Operation*`

The `Reduce` function takes three arguments:

- The side-effect (or the affected data property or the input reference) currently being reduced.
- The list of route values to which the previous element in the list has been reduced. These routes are called the *parameter routes* of the `Reduce` function.
- The list of state operations maintained by the synthesis algorithm.

The semantics of the `Reduce` function depends on the category of the side-effect or of the affected data property to which it is applied. The return values of the `Reduce` function are:

- A list of route values, called the *return routes* of `Reduce`. If the `Reduce` is applied to a side-effect (which always corresponds to the last iteration of `Reduce`), it is not relevant to compute a list of route values and an empty list is returned instead.
- The list of state operations to which new state operations may have been appended.

6.4.5.2 Reduction rules

The reduction rules describe how the `Reduce` function updates the list of state operations and computes a set of return routes from the input reference (or the affected data property or the side-effect) currently being reduced, the list of parameter routes and the current list of state operations.

The purpose of the analysis developed in this work, and in particular of the effect synthesis which is the last stage of the analysis, is to provide information about effects to the distribution mechanism. The proposed analysis has been developed with as much independence as possible from the actual distribution mechanisms. The stage has now been reached however where some assumptions have to be made about the facilities available for distributed programming.

The effect synthesis takes the view that the distribution mechanism for Distributed Prograph will be built upon the facilities provided by the current version of Prograph. These facilities include a `to-bytes` primitive to pack values for transmission. `to-bytes` recursively flattens data of any complexity (instances of primitive data types or instances of classes) into a sequence of bytes and it also produces a *class translation map* to reconstruct the flattened data. It must be noted that the values of the class variables are not packed with the instances of the class. The flattened data and translation map may be transmitted over the network. Primitives have been written so that several communication protocols may be used from within the Prograph development environment.

The potential effects must be handled in different ways depending on the category to which they belong. The following matrix shows which effects are important. The columns distinguish between the structures affected and the rows between the action of the effect.

	`List`	`Instance`	`Class`	`Persistent`
`Access`	...	...	`X`	`X`
`Update`	`X`	`X`	`X`	`X`

... means that the effect requires no special action and X means that the effect must be dealt with properly.

Extracting a value from a list or an instance structure (but not the default value of an instance attribute) does not require special action because the value referenced by the slot of the instance or that of the list has been packed (by the `to-bytes` primitive) with the instance or the list and transmitted with it. Therefore, it is not necessary for the synthesis to record accesses on instance and list structures.

If a persistent value or a class variable is accessed during the execution of a remote operation, the value obtained may be out of date. The meaning of `X` in the above matrix is that the current facility is not enough to ensure the correctness of the execution and that this facility should be extended.

For all four categories of data structures it is important that all update state operations are properly recorded as the current facilities provide no mechanism to propagate updates across execution contexts.

Effects on external data structures have been omitted from the matrix above because the effect inference does not address external side-effects (see 6.3.3.8). If an external side-effect occurs in the effect signature of an operation annotated for distribution, the remote execution of this operation should be ruled out.

The reduction rules in the following sections specify how the `Reduce` function operates when applied to an affected data property or a side-effect of a given category.

- `Reduce` may create state operations to be added to the list of state operations.
- `Reduce` produces a set of return routes (when applied to a side-effect, the set of return routes is not relevant).

When a new state operation is created, the reduction rule specifies what the values of `Action`, `Data`, `Argument` and `UpdateValue` for the new state operation should be.

The reduction rule also specifies how the `Reduce` function computes a set of return routes. When the `Reduce` function is applied to a read affected data, the reduction rules require that the list of state operations is searched to find a *matching* update state operation. The intuition behind searching a matching state update is that a structure might be updated and then accessed during the execution of the method called by the operation annotated for distribution. This situation is illustrated by the following example:

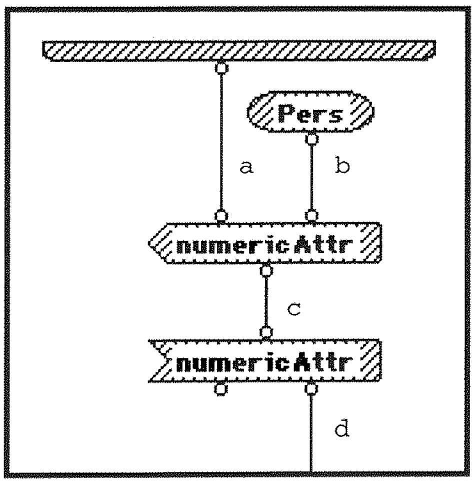


Fig. 6.30: Matching state-operations

The example in fig. 6.30 shows that the value of the attribute `numericAttr` is set. At a later stage, the value of the attribute `numericAttr` is accessed. When a matching update state operation is found, its `UpdateValue` route (the route of value `b` in the example above) should be element of the set of return routes produced by the reduction of the read affected data property (the routes of value `d`).

The following subsections define the reduction rules for the different elements that may occur in the flattened representation of an affected data property or a side-effect:

- Input reference/ identity side-effect
- Class affected data property/ side-effect
- Instance affected data property/ side-effect
- Instantiation side-effect
- Local affected data property
- Persistent affected data property/ side-effect.

Remembering that the reduction rule for a side effect is a simplified version of the reduction rule for the affected data property of the same effect category, the reduction rules will be explained for the affected data properties of the different categories, when applicable.

6.4.5.3 Input reference

An input reference is an integer value and represents one of the inputs of the case(s) of the method called by the operation annotated for distribution.

The reduction rule for an input reference proceeds as follows:

- No state-operation is added to the list of state operations.
- An input route (`I√(Data Depth)`) is created and returned as the set of return routes. The `Data` value of the route is the integer value of the input reference and its `Depth` is zero.

6.4.5.4 Class affected data property

The rule considers a read affected data property first and a write affected data property afterwards.

A class affected data property indicates that the value of a class attribute has been accessed or updated. The `ArgumentType` of the affected data property includes single types and/or string types. A single type is itself a set of classes and a string type is set of string values, some of these string values are class names. A class is *referenced* by the `ArgumentType` if at least one of the following two conditions is met:

- this class is an element of a single type included in the `ArgumentType`
- the name of this class is in a string type included in the `ArgumentType`.

In the set of classes referenced in the `ArgumentType`, another set of classes can be distinguished: the classes with no superclass referenced in the `ArgumentType`. These classes are called the *upper bounds* of the `ArgumentType` (using the inheritance relation as a partial ordering on the set of classes referenced by the `ArgumentType`).

The reduction rule for a read affected data property proceeds as follows:

- For each upper bound class in the `ArgumentType` of the affected data property, a new class route (`C√(Data Depth)`) and a new class access state operation (`C_access(Data Argument)`) are created. The value of `Data` for the class route is the name of the class referenced and its `Depth` is zero (the notation introduced in 6.4.3.1 may be used to include the classes referenced implicitly). The value of `Data` for the class access state operation is the name of the attribute accessed and its `Argument` is the class route.

• All the new class routes have their depth increased by one before being returned by `Reduce` as the set of return routes. The list of state operations is searched to find class update state operations with `Argument` routes matching one of the `Argument` routes of the newly created class access state operations. If matching update state operations are found, their `UpdateValue` routes are added to the list of return routes.

Unlike a class read affected data property, a class write affected data property has an `UpdateValue` which can be an input reference or an affected data property. The reduction of a class write affected data property proceeds as follows:

• A set of `UpdateValue` routes and an updated list of state operations are computed by applying the `Synthesise` function to the `UpdateValue` of the class write affected data property and to the current list of state operations. For each upper bound class in the `ArgumentType` of the affected data property, a new class route (`C√(Data Depth)`) is constructed. The `Depth` of each class route is zero and its `Data` value the name of the class referenced. The Cartesian product of the set of class routes and of the set of the `UpdateValue` routes is computed. For each pair in the product set a class update state operation (`C_update(Data Argument UpdateValue)`) is created with the class route as its `Argument` and the `UpdateValue` route as its `UpdateValue`. The `Data` value is the name of the attribute updated.

• The `Reduce` function passes its list of parameter routes unchanged as its return routes.

6.4.5.5 Instance affected data property

The rule for this category of affected data properties is the most complex of all the rules. This complexity results from the use in Prograph of strings as reference to classes.

The same affected data property may be reduced to a set of both class and instance state operations depending on the `ArgumentType` of the affected data property. The `ArgumentType` is divided into two subsets: a subset including the string types and a subset including the single types.

If the subset including the string types is not empty, this affected data property indicates that class structures may be accessed or updated. The property must be reduced to a set of class state operations in similar fashion to the reduction of class affected data properties described in 6.4.5.4. (the single types included in the `ArgumentType` are discarded before applying the reduction rule for the class affected data properties).

If the subset of `ArgumentType` including the single types is not empty, instance structures may be accessed or updated.

The rule for the reduction of an instance read affected data property is:

- `Reduce` does not add any state operation to its list of state operations because, as explained in 6.4.5.2, the effect synthesis should not record access on instances.

- All the parameter routes have their depth increased by one before being returned by `Reduce` as the set of return routes. The list of state operations is searched to find instance update state operations with `Argument` routes matching one of the parameter routes. If matching instance update state operations are found, their `UpdateValue` routes are added to the list or return routes.

The reduction of a write instance affected data property requires that:

- its `UpdateValue` is synthesised to produce a set of `UpdateValue` routes and an updated list of state operations. The Cartesian product of the set of parameter routes and of the set of `UpdateValue` routes is computed. For each pair in the product set an instance update state operation (`I_update(Data Argument UpdateValue)`) is created with the parameter route as its `Argument` and the `UpdateValue` route as its `UpdateValue`. The `Data` value of the state operation is the name of the instance attribute whose value has been updated.

- the parameter routes are returned as the return routes of `Reduce`.

6.4.5.6 Instantiation side-effect

An instantiation effect is a terminal side-effect with no `Argument` and therefore is always flattened into a list with a single element. The reduction of an instantiation side-effect requires that:

- an allocation state operation (`Alloc(Data)`) is added to the list of state operations. The `Data` value of the allocation side-effect is the name of the class to which the new instance belongs.

6.4.5.7 Local affected data property

The reduction of a local affected data property proceeds with the following steps:

- No state operation is created.
- A local route is created (`L√(Data Depth)`). The `Depth` of the route is zero and its `Data` value is the `Data` value of the local affected data property.

6.4.5.8 Persistent affected data property

The reduction of a read persistent affected data property proceeds with the following steps:

- A persistent access state operation is created (`P_access(Data)`). The `Data` value for this persistent access state operation is the name of the persistent.
- A persistent route (`P√(Data Depth)`) is created to be passed as a return route, its `Data` value is the name of the persistent and its depth is one. The list of state operations is searched to find persistent update state operations with `Data` matching the `Data` of the new persistent access state operation. If matching persistent update state operations are found, their `UpdateValue` routes are added to the list of return routes.

There exists no write persistent affected data property, as a persistent write side-effect is a terminal side-effect.

The reduction of persistent write side-effect proceeds as follows:

- the `UpdateValue` of the persistent write side-effect is synthesised to produce a set of `UpdateValue` routes and an updated list of state operations. For each `UpdateValue` route, a persistent update state operation is created (`P_update(Data UpdateValue)`) with as `Data` value the name of the persistent and as `UpdateValue` the `UpdateValue` route.

6.4.5.9 List affected data property

The reduction of a read list affected data property proceeds as follows:

- No state operation is created because, as explained in 6.4.5.2, the effect synthesis should not record access on lists.

- The parameter routes of `Reduce` are passed unmodified as its return routes. The list of state operations is searched to find list update state operations with `Argument` routes matching one of the input routes. If matching list update state operations are found, their `UpdateValue` routes are added to the list of return routes.

The decision not to increase the depth of the parameter routes before returning them as return routes is explained by the fact that the analysis does not distinguish between a list and the individual elements of that list (see 6.3.3.7).

The reduction of a write list affected data property proceeds as follows:

- The `UpdateValue` of the list write affected data property is synthesised to produce a set of `UpdateValue` routes and an updated list of state operations. The Cartesian product of the set of parameter routes and that of `UpdateValue` routes is computed. For each pair in the product set a list update state operation (`L_update(Argument UpdateValue)`) is created and added to the list with the parameter route as its `Argument` and the `UpdateValue` route as its `UpdateValue`.

- The parameter routes of `Reduce` are passed unmodified as return routes.

6.4.6 Synthesis example

The effect synthesis explained below builds on the example presented in 6.3.9.

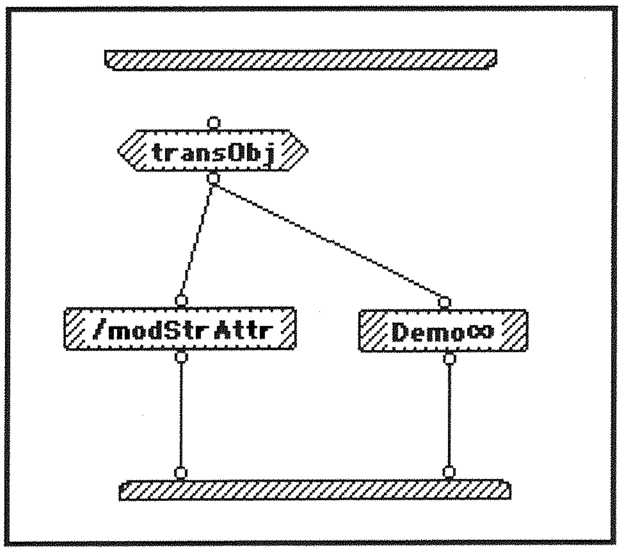


Fig. 6.31: Operation annotated for distribution

The operation `Demo` is annotated for distribution (see fig. 6.31) and calls the universal method whose implementation is shown in fig. 6.32. The ∞ symbol appended to the name of the operation indicates that this operation is annotated for distribution.

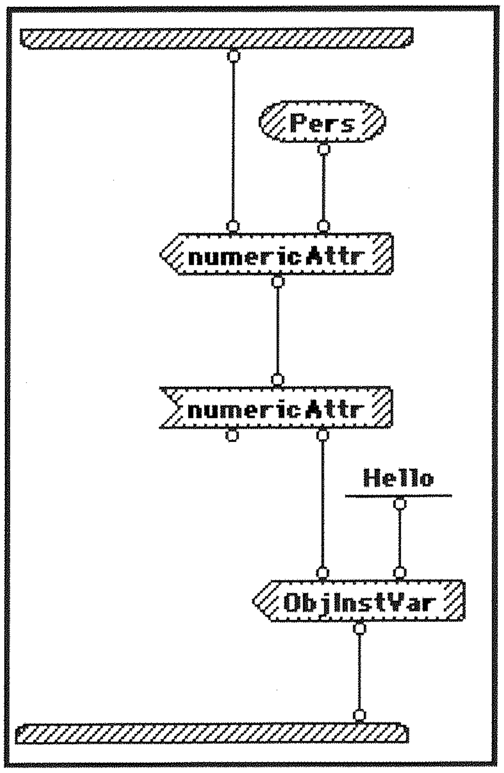


Fig. 6.32: Example of effect synthesis

The synthesis of the side-effects of the `Demo` operation proceeds as follows:

- The initialisation phase constructs the type and effect signatures of the operations and sets the type and affected data properties of the datalinks in the case in which the `Demo` operation occurs (see fig. 6.31).
- Type inference is carried out on the case. Fig. 6.33 shows the type information attached to the datalinks and the signatures of the operations after the case-wide type inference.
- The side-effect composition routine is applied to the operations of the case. When the composition routine reaches the `Demo` operation, the operation's side-effects are duplicated. The effect signature of `Demo∞` comprises a single side-effect (σ_2 in fig. 6.33):

```
OAW(CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr")
"ObjInstVar" NEW("##") (1)).
```

The `ArgumentType` of the composed affected data properties in the duplicate side-effect must be updated with the type of the datalink connected to the terminal of the operation `Demo∞`. The type of this datalink is `<transObj>`.

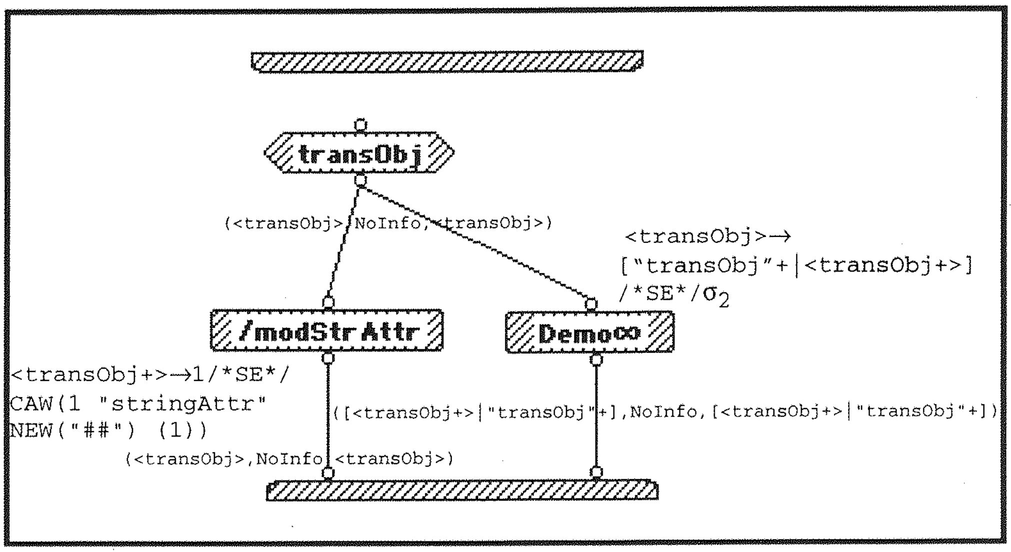

transObj
(<transObj>,NoInfo,<transObj>)
<transObj>→ ["transObj"+|<transObj+>] /*SE*/σ2
/modStrAttr
Demo∞
<transObj+>→1/*SE*/ CAW(1 "stringAttr" NEW("##") (1))
([<transObj+>|"transObj"+],NoInfo,[<transObj+>|"transObj"+])
(<transObj>,NoInfo,<transObj>)

Fig. 6.33: After the type inference

This type is propagated to the `ArgumentType` of the affected data property:

```
CAW(1 "numericAttr" PER("Pers"))
```

and propagated to the `ArgumentType` of the affected data property:

```
CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr")
```

However, the `ArgumentType` of the side-effect itself is not updated. This is because the `Argument` of the side-effect is the result of a `Get` operation on the `numericAttr` attribute and the type of the second output of a `Get` operation does not depend on the type of the input of this `Get` operation.

The first step of the synthesis is to flatten the side-effect into a list representation:

```
(1 CAW(1 "numericAttr" PER("Pers")) CAR(CAW(1 "numericAttr"
PER("Pers")) "numericAttr") OAW(CAR(CAW(1 "numericAttr"
PER("Pers")) "numericAttr") "ObjInstVar" NEW("##") (1)))
```

The `Reduce` function is applied to the four elements of the list. The synthesis starts with an empty set of state operations. The result of each reduction is presented in a table with the following entries:

- the input reference, affected data property or side-effect being reduced
- the `ArgumentType` of the affected data property or side-effect being reduced
- the list of parameter routes
- the current list of state operations
- the `UpdateValue` input reference or affected data property (only relevant when a write affected data property or a write side-effect is being reduced)
- the list of `UpdateValue` routes produced by the synthesis of the `UpdateValue` input reference or affected data property (only relevant when a write affected data property or a write side-effect is being reduced)
- the list of state operations after the synthesis of the `UpdateValue` input reference or affected data property (only relevant when a write affected data property or a write side-effect is being reduced)
- the list of return routes

- the list of state operations after the reduction.

The `Reduce` function is applied to the input reference 1:

- no state operation is added to the list of state operations.

- An input route is returned

Reduced input reference	`1`
Parameter routes	`()`
List of state operations	`()`
Return routes	`(I√(1 0))`
Updated list of state operations	`()`

`CAW(1 "numericAttr" PER("Pers"))` is reduced. The reduction proceeds in two steps:

- The `UpdateValue` of the affected data property must be synthesised first. The `Synthesise` function is applied to the `PER("Pers")` affected data property and the current list of state operations. The synthesis adds a persistent access state operation to the list of state operations and returns a single persistent route as the set of `UpdateValue` routes.

- A class update state operation is created. The `Argument` of the class update affected data property is a class route of depth zero. The `Data` value of the class route corresponds to the classes referenced in the `ArgumentType` of the affected data property:

```
C_update("numericAttr" C√("transObj" 0) P√("Pers" 1))
```

- The parameter routes of `Reduce` are passed as return routes.

Reduced affected data property	`CAW(1 "numericAttr" PER("Pers"))`
`ArgumentType` of the affected data property	`<transObj>`
Parameter routes	`(I√(1 0))`
List of state operations	`()`
`UpdateValue`	`PER("Pers")`
List of `UpdateValue` routes	`(P√("Pers" 1))`
List of state operations after synthesis of `UpdateValue`	`(P_access("Pers"))`
Return routes	`(I√(1 0))`
Updated list of state operations	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)))`

The affected data property `CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr")` is reduced.

The reduction rule requires that a state operation is created:

```
C_access("numericAttr" C√("transObj" 0))
```

When computing the return routes for the current affected data property, the `Reduce` function finds a matching update state operation:

```
C_update("numericAttr" C√("transObj" 0) P√("Pers" 1))
```

The `UpdateValue` of this matching state operation must be included in the set of return routes computed by `Reduce`.

Reduced affected data property	`CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr")`
`ArgumentType` of the affected data property	`<transObj>`
Parameter routes	`(I√(1 0))`
List of state operations	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)))`
Return routes	`(C√("transObj" 1) P√("Pers" 1))`
Updated list of state operations	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)) C_access("numericAttr" C√("transObj" 0))`

The side-effect `OAW(CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr") "ObjInstVar" NEW("##") (1)))` is reduced.

The first step in the reduction is the synthesis of the `UpdateValue` of the side-effect, `NEW("##")`. The synthesis adds no state operation to the list of state operation and returns a local route as the set of `UpdateValue` routes:

```
L√("##" 0)
```

The `ArgumentType` of the side-effect is `[<transObj+>|"transObj"+]`, that is the `Argument` of the side-effect can be either an instance or a string referring to a class. Consequently, `Reduce` must create both class and instance state operations.

The class state operation ignores the parameter routes of `Reduce`, instead a class route is created to be the `Argument` of a class update state operation:

```
C_update("ObjInstVar" C√("transObj"+0) L√("##" 0))
```

Two instance update state operations are added to the list of operations, one for each route in the set of parameter routes of `Reduce`:

```
I_update("ObjInstVar" P√("Pers" 1) L√("##" 0))
I_update("ObjInstVar" C√("transObj" 1) L√("##" 0))
```

Reduced side-effect.	`OAW(CAR(CAW(1 "numericAttr" PER("Pers")) "numericAttr") "ObjInstVar" NEW("##") (1)))`
`ArgumentType` of the affected data property	`[<transObj+>|"transObj"+]`
Parameter routes	`(C√("transObj" 1) P√("Pers" 1))`
List of state operations	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)) C_access("numericAttr" C√("transObj" 0))`
`UpdateValue`	`NEW("##")`
List of `UpdateValue` routes	`L√("##" 0)`
List of state operations after synthesis of `UpdateValue`	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)) C_access("numericAttr" C√("transObj" 0))`
Return routes	`Not relevant`
Updated list of state operations	`(P_access("Pers") C_update("numericAttr" C√("transObj" 0) P√("Pers" 1)) C_update("ObjInstVar" C√("transObj"+0) L√(## 0))) I_update("ObjInstVar" C√("transObj" 1) L√("##" 0)) I_update("ObjInstVar" P√("Pers" 1) L√("##" 0))`

The result of the synthesis is a list of state-operations:

```
P_access("Pers")
C_update("numericAttr" C√("transObj" 0) P√("Pers" 1))
C_access("numericAttr" C√("transObj" 0))
C_update("ObjInstVar" C√("transObj"+0) L√("##" 0))
I_update("ObjInstVar" C√("transObj" 1) L√("##" 0))
I_update("ObjInstVar" P√("Pers" 1) L√("##" 0))
```

6.4.7 Flow sensitivity

When a write affected data property or a write side-effect is reduced, the `Argument` of the affected data property or side-effect must have been synthesised before its `UpdateValue`. For example, if an operation annotated for distribution called the method case the case of which is shown in fig. 6.34, the effect signature of this annotated operation would be:

```
    OAW(CAR(1 "ObjectAttr") "ObjInstVar" PER("Info") ( ))
```

During the execution of the method called by the operation annotated for distribution, the `Get` persistent operation is executed just after the `Input` operation.

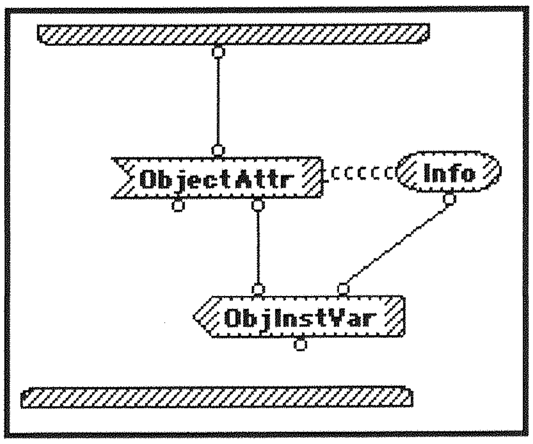


Fig. 6.34: Flow information

The side-effect is flattened into the following list:

```
(1 CAR(1 "ObjectAttr") OAW(CAR(1 "ObjectAttr") "ObjInstVar"
PER("Info") ( )))
```

The synthesis is executed in three steps. The reduction of the input reference is described in the table below:

Reduced input reference	`1`
Parameter routes	`()`
List of state operations	`()`
Return routes	`(I√(1 0))`
Updated list of state operations	`()`

The affected data property `CAR(1 "ObjectAttr")` is then reduced:

Reduced affected data property	`CAR(1 "ObjectAttr")`
`ArgumentType` of the affected data property	`<transObj+>`
Parameter routes	`(I√(1 0))`
List of state operations	`()`
Return routes	`(C√("transObj"+1))`
Updated list of state operations	`(C_access("ObjectAttr" C√("transObj"+0)))`

The side-effect `OAW(CAR(1 "ObjectAttr") "ObjInstVar" PER("Info") ())` is now reduced:

Reduced side-effect.	`OAW(CAR(1 "ObjectAttr") "ObjInstVar" PER("Info") ())`
`ArgumentType` of the affected data property	`[<transObj+>|"transObj"+]`
Parameter routes	`(C√("transObj"+ 1))`
List of state operations	`(C_access("ObjectAttr" C√("transObj"+ 0)))`
`UpdateValue`	`PER("Info")`
List of `UpdateValue` routes	`P√("Info" 1)`
List of state operations after synthesis of `UpdateValue`	`(C_access("ObjectAttr" C√("transObj"+ 0)) P_access("Info"))`
Return routes	`Not relevant`
Updated list of state operations	`(C_access("ObjectAttr" C√("transObj"+ 0)) P_access("Info") C_update("ObjInstVar" C√("transObj"+ 0) P√("Info" 1)) I_update("ObjInstVar" C√("transObj"+ 1) P√("Info" 1)))`

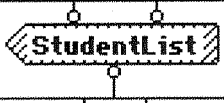
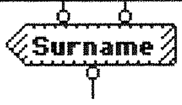
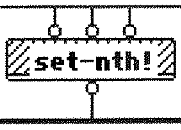
Although the `Get` persistent operation is executed before the `Get` class attribute (`ObjectAttr`) operation, the state operation describing the access performed by the persistent `Get` operation is going to be recorded after the state operation recording the access to the value of the class attribute. The `P_access("Pers")` was recorded *too late* with respect to the state operation `C_access("ObjectAttr" C√("transObj"+ 0))`. The recording of the persistent state operation was delayed because the `PER("Info")` affected data property was the update value of the `OAW(CAR(1 "ObjectAttr") "ObjInstVar" PER("Info") ())` side-effect.

One of the claimed properties of the effect synthesis is that it can detect locally created aliases. Such a situation occurs when an object is passed as the update value of a write side-effect and it is subsequently extracted by a read side-effect in one of the cases of the method called by the operation annotated for distribution.

The example above shows that the representation chosen for the side-effects results in loss of information about the execution order of the operations. There are two possible consequences:

- A write state operation is recorded too early. A local alias might be detected even if this alias cannot be created at run-time. As a result of finding a matching state operation, the `Reduce` function applied to a read affected data property will return an extra route which might become the argument of another state operation at a latter stage during the effect synthesis. The consequence of recording a write state operation too early is that the synthesis will record some unnecessary state operations, however the approximation remains safe because the set of state operations is a superset of the possible accesses or updates that may take place when the operation annotated for distribution is executed.
- A write state operation is recorded too late. The consequence of this is that the effect synthesis might fail to detect the creation of a local alias during the execution of the operation annotated for distribution and consequently fail to detect a way an operation annotated for distribution may access or update an input value or a global variable at run-time.

A possible solution to this shortcoming would be to force the synthesis to record write state operations as early as possible. This would entail the following modification to the representation of side-effects: all write side-effects must be terminal ones. The modified effect signatures are shown in the table below:

Operation	Side-effect	Current signature	Modified Signature
	Class write side-effect	`CAW(1 "StudentList" 2 (1))`	`CAW(1 "StudentList" 2 ()) × IDE(1 (1))`
	Instance write side-effect	`OAW(1 "Surname" 2 (1))`	`OAW(1 "Surname" 2 (1)) × IDE(1 (1))`
	List write side-effect	`LIW(1 2 (1))`	`LIW(1 2 (1)) × IDE(1 (1))`

With only terminal write side-effects, there would exist no write affected data property and the recording of write state operations would not be delayed because some write affected data property is composed in the update value of a write affected data property or a write side-effect. Moreover, the list of side-effects of the operation annotated for distribution could be sorted so that the write side-effects appear first in the list.

6.5 Summary

- The effect inference proceeds together with the type inference to gather information about the effects of a method.
- Effect information encompasses affected data properties which describe how data values are obtained and side-effects which are kept with the type information in the lines making up the signature of a method.
- Different categories of side-effects are for the different data structures that can be affected. A side-effect also has an action (read or write).
- Effect synthesis computes an approximation of the effects of the execution of an operation in a particular execution context.
- A state operation is the abstraction of an effect for the effect synthesis. A Route is the abstraction of a data value.
- The effect synthesis reduces the side-effects of an operation in a particular context into a list of state operations.

7 Experimenting with the analysis tool

The analysis described in chapter 5 and chapter 6 has been implemented. This chapter covers both the implementation of the analysis tool and the experimentation with it.

The first section of this chapter highlights some of the implementation details of the analysis. The next section looks at a complex example to describe the behaviour of the analysis. The third section comments on the applicability of the analysis and suggests some improvements. The fourth section explains how the results of the analysis can be interpreted. The last section suggests some ways of exploiting these results to support distribution.

7.1. The analysis tool

Since the access to the code of the implementation of Prograph is not available, it has not been possible to integrate the analysis with the interpreter. The analysis is run as a separate application within the interpreter. This section highlights some of the aspects of the implementation of the analysis tool.

The analysis is triggered by selecting an item on a pull-down menu and typing the method identifier of the method to be analysed. The code of the method to be analysed is stored in a file, a utility program (provided by Pictorius Inc.) extracts the code from the file and converts it into a form which is amenable to analysis.

7.1.2 Auxiliary data

The construction of type expressions, operations on type expressions and computation of operation signatures requires the construction of the set of superclasses or subclasses of a given class and the look-up of all the methods or all the attributes with the same name. In order to speed up these operations, the information has been organised in the form of indexed files:

- The class hierarchy is stored in a file and a class description consists of a (class name, class identifier) pair. Class identifiers are sequences of integers constructed in such a way that by comparing two class identifiers it can be easily worked out whether a class is a superclass of another.

Each class description is indexed by both its class identifier and its class name, allowing the look-up of a class name from the class identifier and vice-versa.

• Attribute information is stored in a separate indexed file. The name of the attribute is used as a key and the associated data consists of two lists. The first list contains the identifiers of the classes that define (but not the classes that inherit) the attribute as a class attribute and the second list contains the identifiers of the classes that define (but not inherit) the attribute as an instance attribute.

• [Suzuki 1981] proposed the idea of a look-up table to mimic the behaviour of Smalltalk's method look-up mechanism within the analysis. The table used for Prograph maps a method name to three lists of class identifiers. The first list contains the identifiers of the classes that define a simple method with the same name. The second and third lists are the lists of the identifiers of the classes that define `Get` and `Set` methods respectively. During the initialisation phase of a case-wide type and effect inference, the signatures of the operations of the case are set. To construct the signature of a simple, `Get` or `Set` operation with a data-determined reference, the inference mechanism looks up a method by name and method type (i.e. `Set,Get` or simple) and finds out the identifiers of the classes that (re)define a method with the required name and method type. To build the signature of an `Init` operation, the inference mechanism checks whether a custom initialisation method is defined for the class.

• The signatures of the primitives are stored in a repository and are indexed by the primitive name. The repository itself is constructed from the primitive signature files. These text files contain the textual representation of all the signatures of the primitive methods of a given category (e.g. list primitive methods, math primitive methods). The content of the signature files is parsed to construct the signatures put in the repository.

7.1.3 Restricted method despatching

During the course of execution the analysis may prompt the user, if an operation name is heavily overloaded, to discard the classes whose methods cannot be despatched. This can be seen as equivalent to requiring the user to annotate the type of methods' receivers and therefore incompatible with the idea of type inference. However, the information provided by the programmer is not stored and thus is of a less permanent nature than a type annotation and the analysis might become intractable without this user feedback.

7.1.4 Results and errors logging

The analysis proceeds until it is completed or it fails. Failures are most likely to occur during the type inference either during the setting of the operation signatures or during the forward and backward passes of the inference. In case of a failure, the analysis reports at what stage of the analysis the failure has occurred (i.e. initialisation phase, forward or backward analysis) and the location the faulty operation in the code currently analysed.

The proceeding of the analysis is monitored by recording:

- The start of an inference and the value of the system clock at the time.
- The end of an inference and the corresponding value of the system clock (the only purpose for the values of the system clock is to measure the time taken by the analysis to complete).
- The signature produced by the inference.
- The restrictions by the user to the set of classes to which the receiver of an operation with a data determined reference can belong.

Indentation is used to indicate the nesting of the inferences. This information is displayed in a textual form on the screen upon the successful completion of the analysis. The template for the displayed information is shown below:

Start Inference for M_a : T_Start_a

    Start Inference for M_b : T_Start_b
    Finish Inference for M_b : T_Finish_b
    Signature for M_b : Sig_b
    Start Inference for M_c : T_Start_c

```
        Receiver of operation reduced to C1, C2
                Start Inference for Md: T_Startd
                Finish Inference for Md: T_Finishd
                Signature for Md: Sigd
                Start Inference for Me: T_Starte
                Finish Inference for Me: T_Finishe
                Signature for Me: Sige
        Finish Inference for Mc: T_Finishc
        Signature for Mc: Sigc
Finish Inference for Ma: T_Finisha
Signature for Ma: Siga
```

However, in order to make it more readily understandable, the execution of an analysis can also be represented by a tree (which is drawn manually). The nodes of the tree correspond to inferences applied to methods and the links represent dependencies between the results of inferences. Special nodes are also inserted in the graph to indicate the points of the analysis at which the set of classes of an operation receiver has been restricted by the programmer. The successive inferences are carried out in a top-to-bottom, left-to-right order. The graphical equivalent to the analysis log shown above is shown in fig. 7.1:

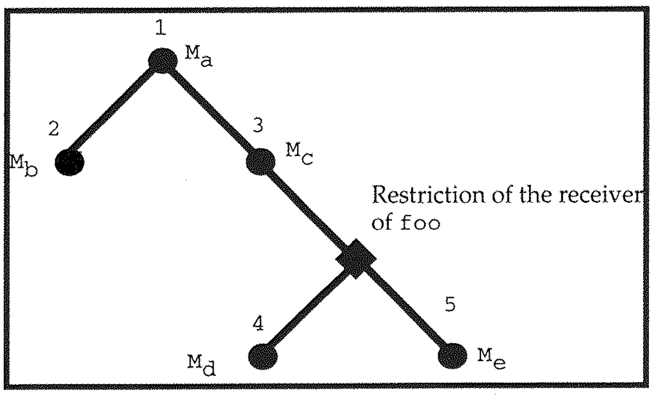


Fig. 7.1: Graphical interpretation of an analysis log.

Downward transitions (the transition from node 1 to node 2 in fig. 7.1, for example) occur during the signature set-up phase of the inference. If a signature is missing and no recursion is detected, the current inference is suspended and the missing signature is inferred.

7.1.5 Caching of intermediate results

Intermediate results are cached to speed up the inference. Cached information includes:

- Signatures of default `Get` and `Set` operations. The same `Get` and `Set` operations often occur in separate cases of a method. Caching their signature instead of building them on the fly for each occurrence of the same operation speeds up the analysis.
- The signatures of all the methods, class-based as well as universal.

The cache consists of a list of (method identifier, signature) pairs. For the signatures of default `Get` and `Set` operations, the *ClassName* component of the method identifier is `Universal`.

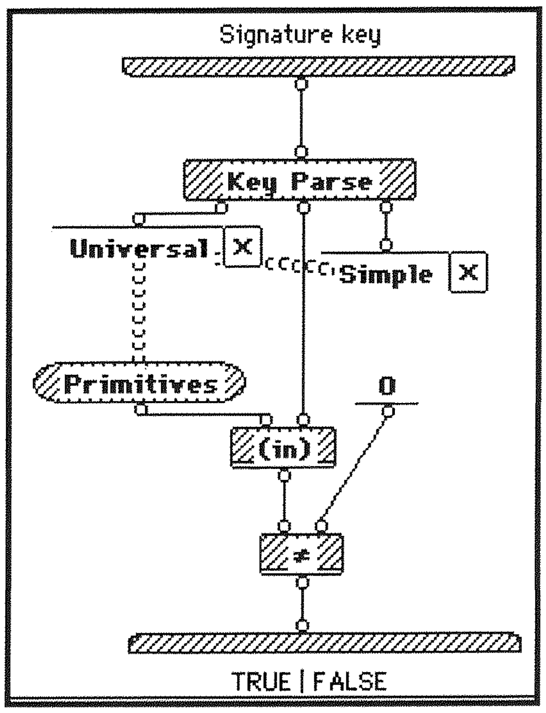
Computing the signature of an operation with a data-determined reference is expensive and the computed signatures are obvious candidates for caching. However, the inferred signatures are not cached because the user may restrict the type of the receiver in a given case of a method and the signature, while valid in the context of that particular case may be invalid in the context of another case.

Dynamic binding means that signatures may become invalid when some code is edited. It would be particularly difficult to keep track of which signatures should be invalidated when code is modified. The analysis takes the conservative view that the cached signatures are valid only for the duration of a session of the analysis program and all the cached information is discarded when exiting the analysis program.

7.2 Examples

7.2.1 Type and Effect Inference

The method chosen as the first example of the type and effect inference is a universal method called `IsPrimitive?` (fig. 7.2) and is used to check whether a method identifier is a reference to a primitive. `IsPrimitive?` calls the `Key Parse` universal method whose role is to break up a method identifier into its three separate components (i.e. *ClassName, MethodName* and *MethodType*). If the *MethodType* is `Simple`, `IsPrimitive?` looks for the method name in the list of the primitives' names stored in the `Primitive` persistent.



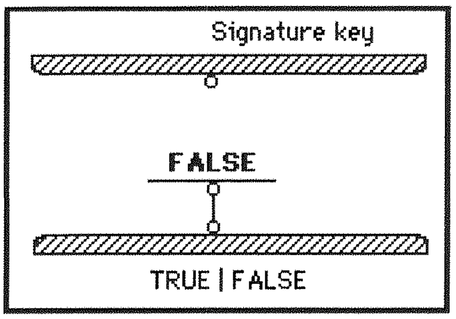


Fig. 7.2.a: First case of `IsPrimitive?` Fig. 7.2.b: Second case.

The information logged during the analysis is shown below:

```
##START ANALYSIS OF:Universal/Is Primitive?/Simple
2:31:49 pm
      ##START ANALYSIS OF:Universal/Key Parse/Simple
      2:31:51 pm
      ======================================
      ##FINISH ANALYSIS OF:Universal/Key Parse/Simple
      2:31:53 pm
      The signature is:
      ""=""*""*""
======================================
##FINISH ANALYSIS OF:Universal/Is Primitive?/Simple
2:31:56 pm
The signature is:
""=<boolean>
/*SE*/PER("Primitives" ())
```

From the signature inferred, it can be said that the method `Is Primitive?` takes a string as argument and returns a boolean value. `Is Primitive?` also induces a read side-effect on a persistent named `Primitives`.

7.2.2 A worst case example

The code to be analysed is part of the implementation of the effect inference algorithm. Each category of side-effect and affected data property is implemented as a separate class (fig. 7.3).

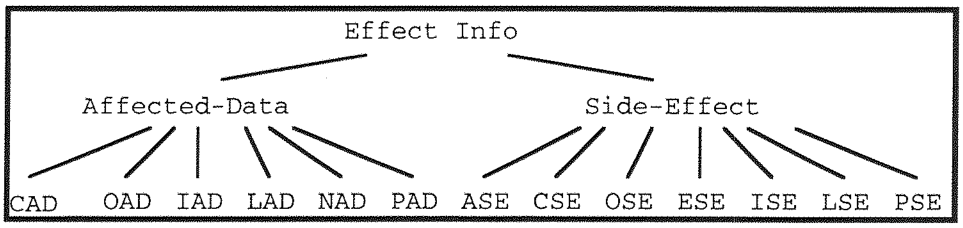


Fig. 7.3: The `Effect Info` class hierarchy

Each class defines a `to-string` method which produces a textual representation of the side-effect or affected data property passed as argument to the method. Each class also defines a `format` method to produce a formatting string for the textual representation of the effect.

The `to-string` method calls the version of `format` defined for its class and substitutes the formatting items in the formatting string with a textual representation of the values of each of the fields of the side-effect or affected data property. The `to-string` method defined for the class `OAD` is shown in fig. 7.4

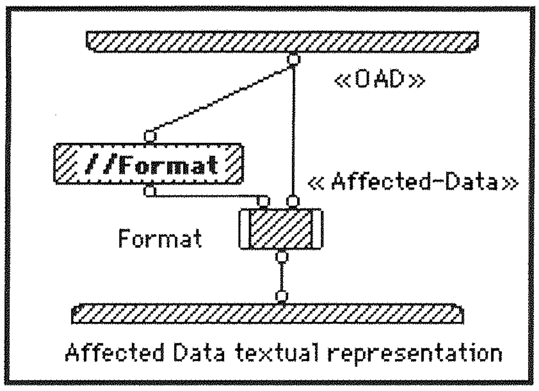


Fig. 7.4.a: `OAD/to-string/Simple`

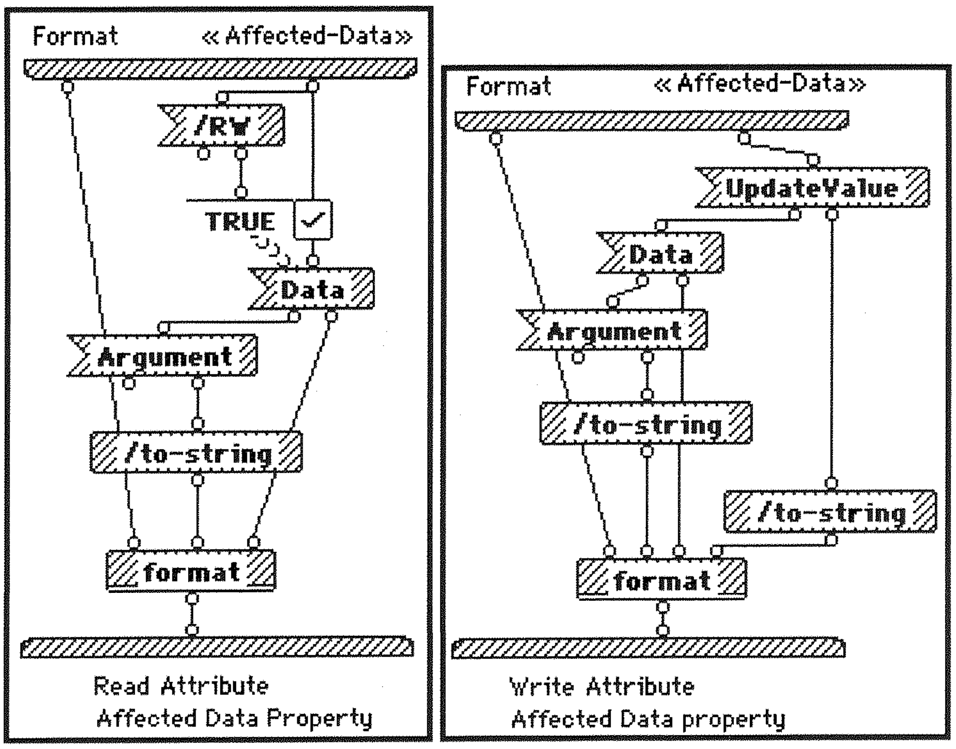


Fig. 7.4.b&c: Cases of the local of `OAD/to-string/Simple`

Although, the code for `to-string` is not extremely complex, producing a type and effect signature is complicated by the facts that affected data properties are recursive data structures and that the `to-string` name is heavily overloaded (36 classes implement a `to-string` method).

The graph shown in fig. 7.5 shows the progress of the analysis of the `to-string` method. The points of the analysis at which the set of classes of an operation receiver has been restricted by the programmer are also indicated. To simplify the diagram, the subtrees corresponding to the inferences of the signatures of the various `Format` methods have been omitted.

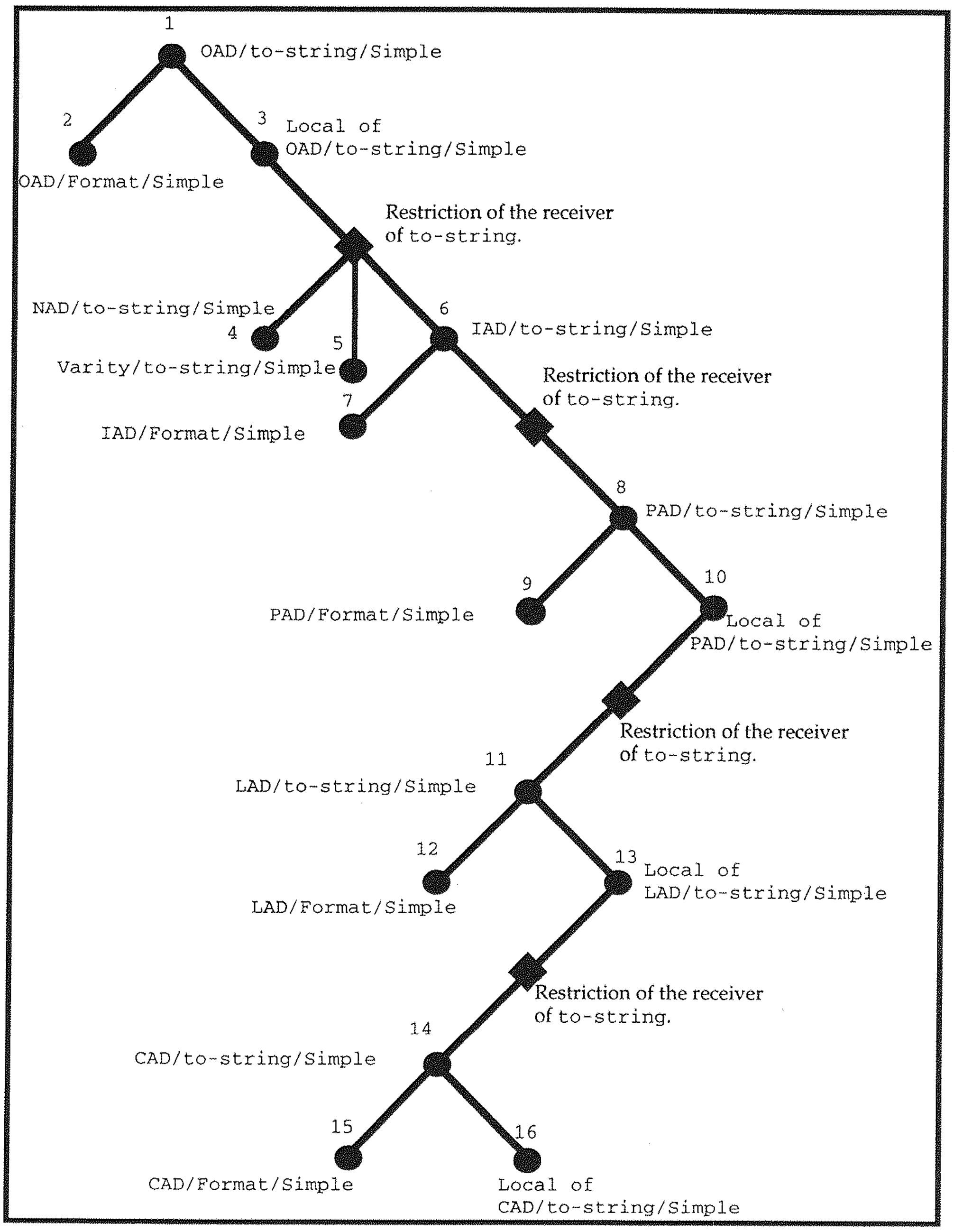

1 OAD/to-string/Simple
2 OAD/Format/Simple
3 Local of OAD/to-string/Simple
Restriction of the receiver of to-string.
4 NAD/to-string/Simple
5 Varity/to-string/Simple
6 IAD/to-string/Simple
7 IAD/Format/Simple
Restriction of the receiver of to-string.
8 PAD/to-string/Simple
9 PAD/Format/Simple
10 Local of PAD/to-string/Simple
Restriction of the receiver of to-string.
11 LAD/to-string/Simple
12 LAD/Format/Simple
13 Local of LAD/to-string/Simple
Restriction of the receiver of to-string.
14 CAD/to-string/Simple
15 CAD/Format/Simple
16 Local of CAD/to-string/Simple

Fig. 7.5: Tree representation of the analysis of `OAD/to-string/Simple`

The signature inferred by the analysis tool is:

`[<Side-Effect+>|<Effected Data+>|<Effect Info>]=""`

The type signature says that the argument of `OAD/to-string/plain` is an instance of `Effect Info` or one of its subclasses and a string is returned.

The effect signature contains no less than 72 side-effects and it would convey little to print all the expressions here.

A few comments can be made on the behaviour of the analysis:

- Although the complexity of the different `to-string` methods is roughly equivalent, the numbers of side-effects inferred for the different methods vary considerably as shown by the figures below:

method identifier	Number of side-effects
`OAD/to-string/Simple`	72
`IAD/to-string/Simple`	23
`PAD/to-string/Simple`	22
`LAD/to-string/Simple`	9
`CAD/to-string/Simple`	4

  By examining the tree representation of the analysis, one can see there is a correlation between the depth in the tree at which the inference is carried out and the number of side-effects inferred. The implementation of `to-string` defined for the `CAD` class is almost the same as the one defined for the `OAD` class but when trying to infer a signature for `CAD/to-string/Simple`, the inference mechanism detects the mutual recursion as soon as `to-string` is applied to the values extracted from the `Argument` and `UpdateValue` attribute of the instance of `CAD`. The reduction in the growth of the number of side-effects observed for `IAD/to-string/Simple` is due to the fact that for an instance of `IAD`, only the value of the `Argument` attribute is printed.

- The expansion occurs when the instance attributes `Argument` or `UpdateValue` are extracted from the first argument of the `to-string` method. The side-effect signature of `LAD/to-string/Simple` induced by the reading of the `Argument` and `UpdateValue` attributes are composed with those inferred for `CAD/to-string/Simple` as shown below:

  Side-effects for `CAD/to-string/Simple`:

```
OAR(1 "Flags" ())*
OAR(1 "UpdateValue" ( ))*
OAR(1 "Argument" ( ))*
OAR(1 "Data" ( ))
```

Side-effects for `LAD/to-string/Simple`:

```
OAR(1 "Flags" ( ))*
OAR(OAR(1 "UpdateValue") "Flags" ( ))*
OAR(OAR(1 "UpdateValue") "UpdateValue" ( ))*
OAR(OAR(1 "UpdateValue") "Argument" ( ))*
OAR(OAR(1 "UpdateValue") "Data" ( ))*

OAR(OAR(1 "Argument") "Flags" ( ))*
OAR(OAR(1 "Argument") "UpdateValue" ( ))*
OAR(OAR(1 "Argument") "Argument" ( ))*
OAR(OAR(1 "Argument") "Data" ( ))
```

The 1's (underlined with dots) in the side-effects of `CAD/to-string/Simple` are replaced by the read `UpdateValue` and `Argument` instance affected data properties (underlined with dots) in the side-effect signature of `LAD/to-string/Simple`.

In the signature of `OAD/to-string/Simple`, side-effects have five levels of nesting as in:

```
OAR(OAR(OAR(OAR(OAR(1 "Argument") "Argument") "UpdateValue")
"Argument") "Data" ( ))
```

• The analysis of the `to-string` method takes approximately 30 min to complete (interpreted on a Centris 610). It is instructive to look at how this time is spent. Whereas it takes about 0.3 % of the total analysis time to reach the bottom of the analysis graph (node 16), the upward transition from node 6 to node 3 takes about a third of the total analysis time. Although the logging of the analysis steps does not allow the break down of the cost of the different stages in the upward transition, clearly combining the lines of the different cases of a method is very expensive, especially the elimination of the redundant side-effects.

7.3 Applicability of the Analysis

7.3.1 Speed and memory use

The examples of section 7.2 (and in particular the example presented in 7.2.3) show the execution of the analysis may consume considerable computing resources, both in time and memory.

The analysis has been implemented in Prograph. Using Prograph to implement the complex algorithms required by the analysis demonstrates the expressiveness of the language and has allowed much flexibility to experiment and test during development. However, running the analysis as an interpreted application is expensive in terms of performance. The current implementation of the analysis executes at speeds which exclude the use of the analysis in a routine way which is transparent to the user. This handicap must be taken into account when integrating the analysis tool in the application development environment for Distributed Prograph.

For the usability of the analysis, it is important that it is reasonably robust (it must not crash) and either completes or fails within a reasonable time. The analysis algorithm has no built-in "circuit-breaker" but such functionality could be provided by taking into account the depth of the analysis graph. The deeper the graph becomes, the more likely it is that the analysis program will abort by reaching the limit of the available memory resources or that the time needed to obtain a result will become unacceptable. The circuit-breaker should report that the depth limit has been reached and fail the analysis.

The Prograph interpreter and editor also need to perform method and attribute look-ups as well as class hierarchy searches and probably maintain internally some data for the same purpose as the auxiliary data described in 7.1.2. Accessing the internal data maintained by the interpreter and using the associated operations to manipulate this data may be a more efficient solution than defining ad-hoc auxiliary data and operations to query the data.

7.3.2 Handling mutual recursion

The example presented in 7.2.2 shows how mutual recursion complicates the analysis. Two approaches are possible:

• Take the view that mutually recursive methods cannot be analysed and fail the analysis every time mutual recursion is detected. However, the way mutual recursion is detected is crude and it is likely that mutual recursion will be detected in situations where it does not occur at run-time. Overloading method and attribute names corresponds to well established object-oriented programming practices so mutual recursion may often be detected. Thus, this option might prove unnecessarily restrictive.

• Leave the programmer to give indications about the actual control flow of the method by restricting the receiver of operations with a data-determined reference. This option requires the programmer to have an in-depth knowledge of the code and goes against the principle of data abstraction. Taken to the extreme, this solution raises the question of the splitting of the tasks between the analysis and the programmer. As the degree of interaction required from the programmer increases, the benefits drawn from the analysis become less obvious.

7.3.3 Precision of the results

The design of the analysis always requires a balance to be struck between the accuracy and the speed of the analysis.

Some design decisions have been made and explained in the previous chapters, trading precision for speed:

• Case-wide type inference is carried out in two passes (forward and backward, the third pass is concerned only with type dependencies).

• The combination of the lines of the relational primitives.

• The combination of the lines of the different cases of the same method.

• In the case of mutual recursion, the signatures of the methods whose identifiers occur between two identical method identifiers are considered valid and are reused when they should be discarded

On the other hand, other decisions have leant toward precision at the expense of speed:

• The case in which the operation annotated for distribution appears is analysed before the side-effects of the annotated operation are synthesised. The expected benefits are that the type information inferred for the arguments of the operation annotated for distribution will help to narrow the approximation produced by the effect synthesis.

Other solutions have been implemented or propositions can be made to improve the precision of the analysis:

- The user is prompted to restrict the type of a receiver of an operation with a data-determined reference when this operation has a heavily overloaded name.

- The task of the effect synthesis is greatly complicated by the fact that the argument of a `Get` or a `Set` operation can be either an instance of a class or a string the value of which is the name of a class. The type of the operation argument does not matter when the attribute accessed or modified is a class attribute and the synthesis describes the effect induced by the `Get` or `Set` operation with the same state operation (a `class` state operation). But if an instance attribute is accessed or updated, the effect induced by the `Get` or `Set` operation will depend on the type of the argument and must be described by two different state operations. Passing a class reference to a `Get` or `Set` operation to obtain or modify the default value of an instance attribute is a very common practice in the Application Building Editors. However, this construct is less relevant for actual application code. Ideally, there should be two pairs of `Get` and `Set` operations in Prograph for different types of arguments: the `Get` and `Set` operations which take an instance as argument and the `Get` and `Set` operations which take a string as argument. Instead, the analysis could assume that a program is not going to access or modify default instance attribute values. The typing rules for `Get` and `Set` operations would have to be modified so that a string type is no longer a legal type for the first input of a `Get` or `Set` operation when this operation involves an instance attribute. No other changes would have to be made to the rest of the analysis.

7.4 Interpretation of the synthesis results

The results of the effect synthesis can be used to derive information about the effects induced by the execution of an operation annotated for distribution. The information of interest falls into four categories:

- Access to a global variable
- Update of a global variable
- Update of an operation argument
- Creation of an alias.

The information can be organised in a hierarchy (fig. 7.6).

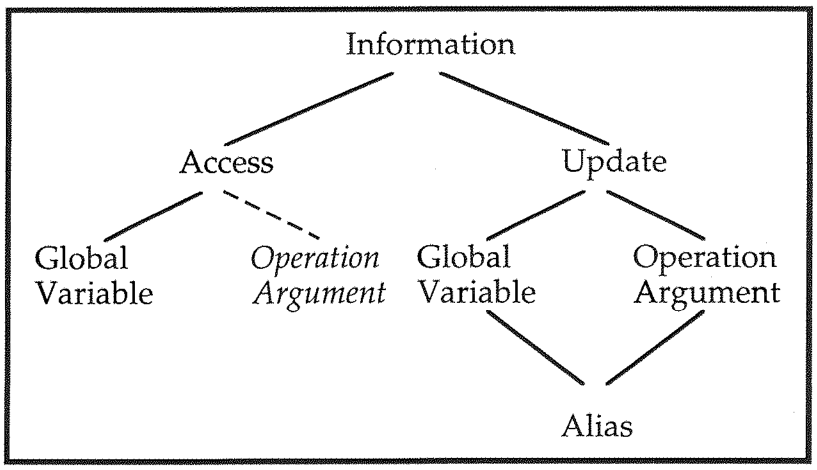


Fig. 7.6: Effect information hierarchy

Access to an Operation Argument appears for the sake of completeness, however this information is not recorded as explained in subsection 6.4.5.2. *Alias* is below both *Update of a Global Variable* and *Update of an Operation Argument* as the creation of an alias may result from a write state operation.

Information can be extracted by interpreting the state operations produced by the synthesis.

7.4.1 Access information

An access to a global variable occurs during the execution of an operation annotated for distribution if the list of state operations produced by the synthesis of the side-effects of the annotated operation contains:

- A persistent access state operation (`P_access(Data)`)

- A class access state operation (`C_access(Data Argument)`)
- An allocation state operation (`Alloc(Data)`)

7.4.2 Update information

An update to an input occurs when the list of the state operations produced by the synthesis of the side-effects of an operation annotated for distribution contains:

- An instance update state operation (`I_update(Data Argument UpdateValue)`) where the argument route is an input route (`I√(Data Depth)`).
- A list update state operation (`L_update(Argument UpdateValue)`) whose argument route is an input route (`I√(Data Depth)`).

An update to a global variable is characterised by the presence in the list of state operations of:

- A class update state operation (`C_update(Data Argument UpdateValue)`).
- A persistent update state operation (`P_update(Data UpdateValue)`).
- An instance update state operation (`I_update(Data Argument UpdateValue)`) where the argument route is a class route (`C√(Data Depth)`) or a persistent route (`P√(Data Depth)`).
- A list update state operation (`L_update(Argument UpdateValue)`) whose argument route is a class route (`C√(Data Depth)`) or a persistent route (`P√(Data Depth)`).

7.4.3 Alias information

Creating an alias means that, as the result of an update, an object is potentially referenced more than once.

The purpose of the `UpdateValue` route of an update state operation is to describe how the data object that will be pointed to after the update described by the state operation became available in the cases of the method called by the operation annotated for distribution. Examining the

UpdateValue route of an update state operation can provide useful information:

• An input UpdateValue route with a depth greater than or equal to one indicates the creation of an alias. This is because the object described by the route has been extracted from an input of the current case and, as a result of the update state operation, the extracted object is referenced at least twice. It is referenced once by the structure from which it was extracted and a second time by the structure that points to it after the update.

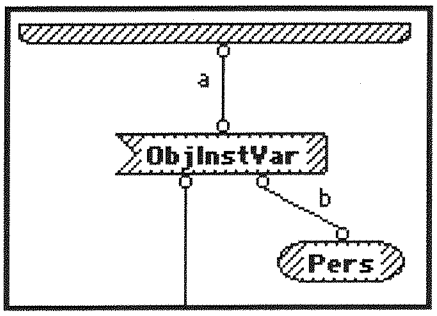


Fig. 7.7: Aliasing of an attribute value.

In fig. 7.7, after the execution of the persistent Set operation, b is pointed to by at least two structures: the persistent structure Pers and the structure from which b was extracted.

• An input UpdateValue route of depth zero may also indicate the creation of an alias because of two reasons. The first one is the reduction of read effects on lists. The reduction rule for a list affected data property requires that the Reduce function does not increment the depth of its parameter routes before passing them as return routes. A route with a depth of zero may describe an object extracted from a list passed as an argument to the current case. The second reason is that an input route of depth zero may also describe an object which was passed as an argument to the operation annotated for distribution. At the same time, this argument object may already be referenced outside the cases of the method called by the operation annotated for distribution (although there might be no other reference to the argument object, the analysis cannot rule out the creation of an alias).

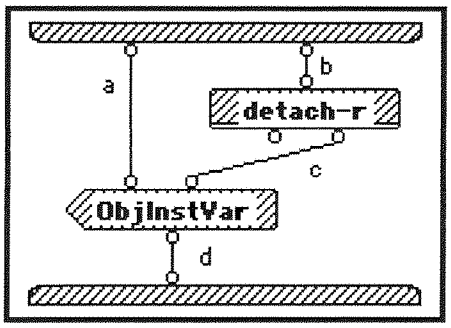


Fig. 7.8: Aliasing of the element of a list

In fig. 7.8, the value `c` is described by an input route of depth zero. After the `Set` operation, the value `c` is pointed to by both the list `b` and the instance a or the class whose name is the value of string a.

• A persistent or a class `UpdateValue` route shows that the object has been extracted from a persistent or a class structure before its reference is given to another structure during an update. Therefore, the object is now referenced at least twice.

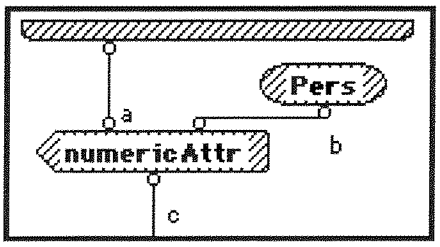


Fig. 7.9: Aliasing of a global variable.

In fig. 7.9 after the `Get` operation, the value extracted from the persistent `Pers` is referenced by both the persistent `Pers` and the instance a or the class whose name is the value of the string a.

• Objects created locally can also be aliased. The detection of these aliases requires a scan of the entire list of the state operations to find write state operations whose `UpdateValue` fields store identical `local` routes.

In the implementation of several distributed object-oriented languages, immutable data cannot be referenced across different object contexts. Instead, the immutable data objects are duplicated before being moved to another context. The consequence is that aliases to immutable data objects are always local. In Prograph, instances of primitive datatypes (with the exception of the `list` datatype) are immutable. It would be possible to extend the current effect inference mechanism so that the type of the `UpdateValue` slots of the effects is recorded.

7.5 Exploitation of the results for distribution

In the Distributed Prograph model, an operation annoted for distribution is exported from the originator context into a recipient context where it is executed. This section discusses three possible mechanisms for distribution and how these mechanisms can exploit the results of the effect synthesis.

7.5.1 Status quo

The first option uses the current facilities provided by Prograph (see section 2.4). The table in 6.4.5.2 shows that only accesses to instances and lists can be carried out using the distribution mechanism currently available in Prograph.

An operation annotated for distribution can be executed remotely only if the synthesis of the side-effects of this operation produces an empty list of state operations.

7.5.2 Access to global variables

The current distribution mechanism could be extended so that it becomes possible to send the value of global variables (i.e. persistents, class attributes and default values of instance attributes) from the originator context into the recipient context. It is assumed that, as in the current version of Prograph, these values can be transmitted in their full extent across contexts.

An operation annotated for distribution can be executed remotely if the list of state operations produced by the synthesis of the side-effects of the annotated operation contains only:

- access state operations (class access and persistent access state operations)
- allocation state operations (the instantiation of an object is considered as an access of the class attribute values and default instance attribute values of the class from which the object is instantiated)

- instance and list update state operations (`I_update(Data Argument UpdateValue)` and `L_update(Argument UpdateValue)` respectively) where both `Argument` and `UpdateValue` are local routes. This rule means that only objects created in the recipient context can be updated and that the update must not create an alias to an object located in the originator context.

The access state operations are used to determine which classes and which persistent values should be updated in the recipient context:

- for each class access state operation (`C_access(Data Argument)`), for all the classes referenced in the `Argument` of the state operation, the attribute named by the `Data` value of the state operation must have its value updated in the recipient context;
- for each persistent access state operation(`P_access(Data)`), the value of the persistent named by the `Data` value of the state operation must be updated in the recipient context;
- for each allocation state operation (`Alloc(Data)`), all the attributes of the class named by the `Data` value of the state operation must have their values updated in the recipient context.

7.5.3 Access and updates to operation inputs and global variables

A third option would be to design a mechanism which supports both accesses and updates to global variables and the arguments of an operation executed remotely. The mechanism should also provide global object identifiers so that replicas can be reconciled in the originator context after the remote execution of the operation annotated for distribution.

The result of the effect analysis could be used to decide which:

- Global variables must be updated in the recipient context as explained in the previous subsection.
- Inputs and global variables must be updated in the originator context, once the operation has been executed remotely.

The list of state operations produced by the synthesis of the side-effects of the operation annotated for distribution can be exploited in the following way:

• For each persistent update state operation (`P_update(Data Argument UpdateValue)`) the persistent named by the `Data` value of the state operation must have its value updated in the originator context after the execution of the operation annotated for distribution. However, the effect synthesis may pessimistically predict an update to a persistent value and this update may not occur at execution time. The value of the persistent must be updated in the recipient context before the execution of the exported operation. If, contrary to the prediction of the effect synthesis, the value of the persistent is not updated during the remote execution of the operation annotated for distribution, the net effect of the state-operation will be to cause a round trip of the persistent value from the originator context, through the recipient context and back to the originator context.

• For each class update state operation (`C_update(Data Argument UpdateValue)`), for all the classes referenced in the `Argument` of the state operation, the attribute named by the `Data` value of the state operation must have its value updated in the originator context after the execution of the operation annotated for distribution. For the same reasons as for a persistent value, the value(s) of the attribute(s) designated by the `Data` value and the `Argument` route of the class update state operation must be updated in the recipient context before the execution of the remote operation.

• For each list and instance update state operation (`L_update(Argument UpdateValue)` and `I_update(Data Argument UpdateValue)` respectively) with an input route as `Argument`, the value of the input of the operation annotated for distribution must be updated after the execution of the operation. Unlike classes whose attribute values can be updated individually, the update of an argument requires the complete argument to be sent back to the originator context.

• For each list and instance update state operation where `Argument` is a persistent route or class route (`P√(Data Depth)` or `C√(Data Depth)` respectively), the persistent or the class referenced by the persistent or the class route must be updated. The update of a persistent is performed as for a persistent update state operation. To explain how the update of the class should be performed, it must be noted that if a class route is passed as `Argument` to an instance or list update state operation, a class attribute or a default instance attribute must have been accessed beforehand in the case(s) of the method called by the operation annotated for distribution. This access is recorded with a class access state operation by the effect synthesis and the `Argument` route of this class access state operation refers to the same classes as the `Argument` route of the list or instance update state operation. Consequently, for each class access state operation (`C_access(Data Argument)`) with an `Argument` which refers to the same classes as the `Argument` route of the list update or instance update state operation, all the classes referenced in the `Argument` of the class access state operation, must have the value of their attribute named by the `Data` value of the class access state operation updated in the originator context after the execution of the operation annotated for distribution. A more efficient solution would be to have class routes to record not only the class from which a value was extracted but also the name of the attribute from which this value was extracted; it would be no longer necessary to search the list of class access state operations.

7.6 Summary

• The analysis tool is implemented as a separate application within the interpreter.

• Some examples show that the analysis deals imperfectly with heavily overloaded methods and recursion, in particular mutual recursion.

• Useful information (access or update of a global variable, update of a operation argument, creation of an alias) can be extracted from the effect synthesis results and exploited by the distribution mechanisms.

8 Conclusion

The last chapter of this thesis is divided into three sections.

The first section summarises the content of this thesis; the second section suggests how this research could be taken further. The last section reviews the contributions of this work.

8.1 Summary

This thesis has discussed the use of Prograph as a language for distributed programming. The ambition of Distributed Prograph is to extend the productivity that the current version of Prograph already offers for user-interface design and symbolic programming to distributed programming.

The dataflow model has been found to be a good model to express the potential for parallelism in Distributed Prograph. Operations in the case of a method are units of parallelism and distribution. However, the programmer keeps control of distribution with an annotation to indicate which operations should be distributed. The model hides from the programmer communication and distribution mechanisms.

Object-orientation presents some challenges for the implementation of the model notably that of state and behaviour consistency across several execution contexts. Behaviour consistency is the requirement that objects exhibit the same behaviour in different contexts.

An analysis has been developed to provide an approximation of the effects the execution of an operation might induce.

Types and effects are not orthogonal issues in Prograph. Instances of primitive data types cannot be updated, but for reasons of efficiency, instances of user-defined classes can be updated in place. Type information is useful to find more about effects and type inference is the first stage of the analysis. Object-orientation makes type inference more difficult than for other language paradigms because of dynamic binding, inheritance and data polymorphism. The purpose of inference is to reduce the uncertainty due to dynamic binding and to use type information for the effect analysis. The type inference algorithm designed and implemented for Prograph can be applied in a modular fashion to separate methods. The type system handles dependency between input and return types as well as variable arity but it has been decided not to tackle low-level operating structures (*externals* in Prograph terminology). The algorithm can type methods which exhibit a small level of polymorphism but calls to heavily overloaded methods can be handled only with some user assistance.

The effect inference extends the type inference algorithm to produce a type and effect signature for the methods. The choice of a representation for effect information is based on a study of the different effects a computation may have in Prograph and some trade-off between the precision and the tractability of the analysis.

Effect synthesis is the last stage of the analysis and uses the type and effect information gathered by the inferences to produce an approximation of how an operation annotated for distribution would access and update its arguments and global variables during its execution.

The purpose of the information produced by the analysis is twofold: it may assist the programmer in selecting the operations for distribution and could also be exploited for distributing the operations.

8.2 Future work

8.2.1 Integration of the analysis tool

The prototype developed in this work is implemented as an application executing within the interpreter, which requires the user to switch between the execution of the analysis and the interpreter. A better integration might make the tool more intuitive to use, in particular in the two following situations:

- When the type inference fails, it would be nice to be able to display the case in which the failure has occurred and highlight the operation for which the inference has failed.
- When typing an operation calling a heavily overloaded method, the analysis prompts the user to restrict the type of the receiver. It would be easier for the user to do it if the operation was highlighted on the screen.

A good integration also requires the choice of a notation to indicate parallelism. The interpreter/editor environment must be modified so that the operations can be annotated. In the current version of Prograph, operations can be annotated by selecting an item on the *Controls* pull-down menu of the interpreter/editor and it appears natural to add the new notation to the current list of items available in this menu.

The speed and the user involvement required precludes the application of the analysis transparently to the user. Instead the analysis should be triggered by selecting a case and an item from a pull-down menu (the *Info* menu of the interpreter would be a good candidate to insert the new item).

The last issue to be addressed is that of the storage of the analysis results. The current prototype displays the results of the effect synthesis in a textual form and then discards them, they should be saved so that they can be further exploited.

8.2.2 Exploitation of the results

The way the results of the effect analysis will be exploited depends on three different factors:

- The purpose of the information: the interest may lie in either correctness or performance. Correctness is concerned with the access to global variables, modifications of global variables and of operation arguments and dependencies between operations (this latter aspect is not addressed by the analysis). Performance is concerned with the ratio of computation over communication.

- The features of the distribution mechanisms have an impact on what operations can be safely executed remotely. For example, if the distribution mechanisms provide global object identifiers then aliasing should not be a problem. The features also have an impact on the evaluation of the performance as some of them may be less costly than others.

- User control: the results of the analysis can be used as a warning for the user who can then decide whether the operation should be distributed or not, whatever the consequences of this decision may be (the operation may not be executed correctly). The results of the analysis may be used by the compiler or the run-time support to decide whether the operation should be exported or not.

8.3 Contributions

An extension of the underlying model of sequential Prograph for parallel programming is discussed and compared to other models for distributed programming.

The current implementation of Prograph has been presented and other distributed language designs have been reviewed with the emphasis being put on the details that are relevant for the design of Distributed Prograph. The benefits the implementation could draw from effect information are discussed.

The analysis is broken into three stages:

- type inference which computes the types of the inputs and outputs of a method,
- effect inference which computes a description of the effects of a method
- effect synthesis which produces an approximation of the effects of an operation in a particular context.

Effect analysis has been widely applied to both imperative and functional languages but not to object-oriented languages.

The design of the type and effect inferences and the effect synthesis, together with the prototype implementations provide a good basis for future practical tools.

Bibliography

[Agesen 1996] Agesen, O. (1996). Concrete Type Inference: Delivering Object-Oriented Applications. *SMLI TR-96-52*, Sun Microsystems Laboratories.

[Agha 1990] Agha,G. (1990). Concurrent Object-Oriented Programming. *Communications of the ACM*, vol. 33 n°9, pp. 125-141.

[Alvisi et al. 1992] Alvisi, L., Amoroso, O., Babaoglu, O., Baronio, A., Davoli, R. and Giachini, L.A. (1992). Parallel Scientific Computing in Distributed Systems: The Paralex Approach. *Technical Report UBLCS-92-2*, Laboratory for Computer Science, University of Bologna.

[Babb 1984] Babb, R.G. (1984). Parallel Processing with Large-Grain Data Flow Techniques. *Computer*, vol 17, n°6., pp. 55-61.

[Backus 1978] Backus, J. (1978). Can programming be liberated from the von Neumann style? *Communications of the ACM*, vol. 21 n°8, pp. 613-641.

[Bacon, Graham and Sharp 1994] Bacon, D.F., Graham, S.L. and Sharp, O.J. (1994). Compiler Transformations for High Performance Computing. *ACM Computing Surveys*, vol. 26, n° 4, pp. 345-420.

[Bal, Steiner and Tanenbaum 1989] Bal, H.E., Steiner, J.G. and Tanenbaum A.S. (1989). Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, vol. 21, n° 3, pp. 262-313.

[Balter, Lacourte and Riveill 1994] Balter, R.., Lacourte, S., and Riveill, M. The Guide language. *The Computer Journal*, vol. 37 n°6, pp. 519-530.

[Bennett 1987] Bennett, J.K. (1987). The Design and Implementation of Distributed Smalltalk. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). Special Issue of ACM SIGPLAN Notices*, vol. 22, pp. 318-30.

[Birman 1993] Birman, K. (1993). The Process Group Approach to Reliable Distributed Computing, *Communications of the ACM*, vol. 36 n°12, pp. 36-53.

[Birrel and Nelson 1984] Birrel, A.D. and Nelson B.J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, vol. 2, n°1 pp. 39-59.

[Black et al. 1986] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object Structure in the Emerald System. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). Special Issue of ACM SIGPLAN Notices*, vol. 21, pp. 78-86.

[Briot and Guerraoui 1996]. Briot, J.P. and Guerraoui, R. (1996). A classification of Various Approaches for Object-Based Parallel and Distributed Programming. *Technical Report no 96-01*, Dept. of Information Science, University of Tokyo.

[Browne *et al.* 1994] Browne, J.C., Dongarra, J. J., Hyder, S.I., Moore, K. and Newton, P. (1994).Visual Programming and Parallel Computing. *Technical Report CS-94-229*,University of Tennessee

[BYTE 1996] BYTE (1996). Datapro Report, Wanted: Client/Server Expertise. *BYTE*, vol. 21, n°11 p.42

[Cann 1992]Cann, D. (1992). Retire Fortran? A debate rekindled. *Communications of the ACM*, vol. 35, n°8 pp.81-89.

[Cardelli 1987] Cardelli, L. (1987). Basic Polymorphic Typechecking. *Science of Computer programming*, vol. 8 pp.147-171.

[Cardelli 1995] Cardelli, L (1995). A Language with Distributed Scope. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, January 1995, pp.286-297.

[Cardelli and Wegner 1985] Cardelli, L. and Wegner, P. (1985). On understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, vol. 17, n° 4, pp. 471-522.

[Chow and Harrison 1992] Chow J.H. and Harrison W.L. (1992) Compile-Time Analysis of Parallel Programs that Share Memory. *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1992)*, pp.130-141.

[Cole 1989] Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press.

[Coulouris, Dollimore and Kindberg 1992] Coulouris, G., Dollimore, J. and Kindberg, T. (1992). *Distributed Systems: Concepts and Design- Edition2 draft material.* Addison-Wesley.

[Cousot and Cousot 1977] Cousot, P. and Cousot, R. Abstract interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL 1977)*, pp. 238-252.

[Cox 1996] Cox, P.T. (1996). *Private communication.*

[Cox and Mulligan 1985] Cox, P.T. and Mulligan, I.J. (1985). Compiling the graphical functional language PROGRAPH. *Proceedings of ACM Symposium on Small Systems*, pp.34-41.

[Cox and Pietrzykowski 1985] Cox, P.T and Pietrzykowski, T. (1985). Advanced programming aids in PROGRAPH. *Proceedings of ACM Sympos. on Small Systems*, pp.27-33.

[Cox and Pietrzykowski 1988] Cox, P.T. and Pietrzykowski, T (1988). Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. *Proceedings of the International Computer Science Conference 88*, pp. 695-704.

[Cox and Smedley 1996] Cox, P.T. and Smedley, T.J. (1996). A Visual Language for the Design of Structured Graphical Objects. *Proceedings of the IEEE Symposium on Visual Languages (VL '96)*, pp. 296-303,

[Darlington et al. 1995] Darlington, J. Gou, Y. To, H.W. and Yang J. (1995). Skeletons for Structured Parallel Composition. *Proceedings of the 15th ACM SIGPLAN Symposium onPrinciples and Practice of Parallel Programming SIGPLAN Notices*, vol.30, n°8, pp.19-28.

[Decouchant 1986]. Decouchant, D 1986. Design of a Distributed Object Manager for Smalltalk_80 System. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). Special Issue of ACM SIGPLAN Notices*, vol.21, pp. 444-452.

[DeRoure 1990] DeRoure, D.C. (1990). Experience with Lisp and Distributed Systems. *CSTR 90-21.*Department of Electronics and Computer Science, University of Southampton.

[Dollimore, Miranda and Xu 1991] Dollimore,J., Miranda, E. and Xu, W. (1991). The Design of a System for Distributing Shared Objects. *The Computer Journal*, vol. 34, n°6, pp. 514-521.

[Dollimore, Nascimento and Xu 1992] Dollimore, J., Nascimento, C. and Xu, W. (1992). Fine Grained Object Migration: Model, Mechanisms and Experience. *QMW CSL Report Number 571*

[Fasel and Keller 1986] Fasel, J.H. and Keller, R.M. (1986). *Introduction of the proceedings of the Santa Fe Graph Reduction Workshop*, Lecture Notes in Computer Science n° 279 Springer-Verlag.

[Feo, Cann and Oldehoeft 1990] Feo, J.T., Cann D.C. and Oldehoeft, R.R. (1990). A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, vol. 10 n°4 pp. 349-366.

[Field and Harrison 1988] Field, A.J. and Harrison P.G. (1988). *Functional Programming.* Addison-Wesley.

[Foster and Taylor 1990] Foster, I. and Taylor S. (1990). *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs.

[Freeh and Andrews 1995] Freeh, V.W. and Andrews, G.R.(1995). fsc: A Sisal Compiler for Both Distributed- and Shared-Memory Machines. *TR 95-01* University of Arizona.

[Garbinato, Guerraoui and Mazouni 1994] Garbinato B., Guerraoui R. and Mazouni K.: Distributed Programming in Garf. *Object Based Distributed Programming*, Lecture Notes in Computer Science n° 961, pp. 225-239, Springer Verlag.

[Gelernter and Carriero 1992] Gelernter, D., Carriero, N. (1992). Coordination Languages and their Significance. *Communications of the ACM*, vol. 35, n° 2, pp. 97-107.

[Giacalone, Mishra and Prasad 1989] Giacalone, A., Mishra, P., and Prasad, S. (1989). Facile: A Symmetric Integration of Concurrent and Functional Programming. *Proceedings of the 1989 TAPSOFT Conference*, Lecture Notes in Computer Science n° 352, pp. 184 -209, Springer-Verlag.

[Gifford et al. 1987]. Gifford, D.K., Jouvelot, P., Lucassen, J.M. and Sheldon, M.A. (1987). FX-87 Reference Manual. *MIT/LCS/TR-407*, MIT Laboratory for Computer Science.

[Glaser, Hankin and Till 1984] Glaser, H.W.,Hankin, T.L. and Til, D.(1984). *Principles of functional programming.* Prentice Hall International.

[Goldberg and Hudak 1986] Goldberg, B. and Hudak, P. (1986). Alfalfa: Distributed graph reduction on a hypercube multiprocessor. *Proceedings of the Santa Fe Graph Reduction Workshop*, Lecture Notes in Computer Science n° 279, pp. 94 -113, Springer-Verlag.

[Halstead 1984] Halstead, R.H. (1984). Implementation of MultiLisp: Lisp on a multiprocessor. *Proceedings of the ACM Conference on Lisp and functional programming* pp. 9-17.

[Hammond1994] Hammond, K. (1994). Parallel Functional Programming: An Introduction (invited paper). *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94).*

[Hammond et al. 1995] Hammond, K., Matson, J.S. Jr., Partridge A.S., Peyton Jones S.L., Trinder P.W (1995). GUM: a portable parallel implementation of Haskell. *Proceedings of the Workshop on the Implementation of Functional Languages '95,* pp. 259-280.

[Harrison 1989] Harrison W.L. (1989). The interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, vol. 2, n° 3/4, pp. 179-396.

[Hoare 1978] Hoare, C.A.W. (1978). Communicating sequential processes. *Communications of the ACM*, vol. 21, n° 8, pp. 666-677.

[Horwitz, Reps and Binkley 1988] Horwitz, S. Reps, T. and Binkley, D. (1988). Interprocedural slicing using dependen graphs. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Special Issue of ACM SIGPLAN Notices*, vol. 22, pp.35-46.

[Hudak 1984] Hudak, P. (1984). ALFL Reference Manual and Programmer's Guide. *Research Report YALEU/DCS/RR-322*, Yale University.

[Hudak 1986] Hudak, P. (1986). Para-functional programming. *Computer*, vol. 19 n°8, pp. 60-71

[Hudak 1989] Hudak, P. (1989). Functional Programming Languages. *Computing Surveys*, vol. 21 n° 3, pp. 360 -411.

[Johnson 1986] Johnson R.E (1986). Type-Checking Smalltalk. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). Special Issue of ACM SIGPLAN Notices*, vol. 21, pp. 318-330.

[Kaplan and Ullman 1980] Kaplan, M. and Ullman, J.D. (1980). A Scheme for the automatic inference of variable types. *Journal of the ACM*, vol. 27, n° 1, pp. 128-145.

[Keremitsis and Fuller 1995] Keremitsis E. and Ian J. Fuller, I.J (1995) HP Distributed Smalltalk: A Tool for Developing Distributed Applications, *Hewlett-Packard Journal*, vol. 46 n°2, pp. 85-92.

[Kind 1996]. Kind A. (1996). *Private communication.*

[Kind and Friedrich 1993] Kind, A. and Friedrich, H. (1993). A practical approach to type inference for EULisp. *Lisp and Symbolic Computation*, vol. 6, n° 1/2, pp. 159-176.

[Kranz, Halstead and Mohr 1989] Kranz D.A., Halstead R.H. and Mohr E.(1989). Mult-T: A high-performance parallel Lisp. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation. Special Issue of ACM SIGPLAN Notices*, vol. 24, pp. 81-90.

[Kristensen and Low 1995]. Kristensen A. and Low C. (1995). Problem-Oriented Object Memory: Customizing Consistency. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95). Special Issue of ACM SIGPLAN Notices*, vol. 30, pp. 399-413.

[LaLonde and Pugh 1990] Lalonde W.R. and Pugh J.R. (1990). *Inside Smalltalk.* Prentice Hall International.

[LaLonde and Pugh 1991] LaLonde, W. and Pugh, J.(1991).Subclassing ≠ Subtyping ≠ Is-a. *Journal of Object-Oriented Programming,* vol. 3, n° 5 pp. 57-62.

[LaLonde and Pugh 1996] Lalonde, W. and Pugh, J. (1996). Preparing to use the distributed facilty in IBM Smalltalk. *Journal of Object-Oriented Programming,* vol. 3, n° 5 pp. 44-48.

[Lea, Jacquemot and Pillevesse 1993] Lea, R.., Jacquemot, C. and Pillevesse, E. (1993). COOL: System Support for Distributed Programming. *Communications of the ACM,* vol. 36, n° 9, pp. 37-45

[Lee and Hurson 1994] Lee, B. and Hurson, A.R. (1994). Dataflow Architectures and Multithreading. *Computer,* vol. 27, No. 8, pp. 27-39.

[Lieberman 1986]. Lieberman, H. (1986). Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). Special Issue of ACM SIGPLAN Notices,* vol. 21 pp.214-223.

[Liskov 1988] Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM,* vol. 31, n°3, pp. 300-312.

[Liskov and Shrira 1988] Liskov, B. and Shrira, L. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distribute Systems. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Special Issue of ACM SIGPLAN Notices,* vol. 23, pp. 260-267.

[Lucassen and Gifford 1988] Lucassen, J.M. and Gifford D.K. (1988). Polymorphic effect systems. *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88),* pp. 47-57.

[Matwin and Pietrzykowski 1985]. Matwin, S. and Pietrzykowski, T. (1985). Prograph: a preliminary report. *Computer Language,* vol. 10, n° 2, pp.91-126.

[Milner 1978] Milner, R.. (1978). A theory of type polymorphism in programming. *Journal of Computer and Systems Science,* vol. 17, pp. 348-375.

[MPI forum 1993] Message Passing Interface Forum (1993). MPI: Message Passing Interface. *Proceedings of the Supercomputing '93 Conference,* pp. 878-883

[Nikhil, Pengali and Arvind 1986] Nikhil, R.S., Pingali, K. and Arvind (1986). Id nouveau. *GSG Memo 265,* MIT Laboratory for Computer Science.

[OMG 1996]. The Object Management Group (1996). The Common Object Request Broker: Architecture and Specification. *OMG Technical Document PTC/96-03-04.*

[OSF 1992] The Open Software Foundation (1992). The OSF™ Distributed Computing Environment. *OSF-DCE-PD-1090-4 White Paper*

[Oxhøj, Palsberg and Schwartzbach 1992] Oxhøj, N., Palsberg, J. and Schwartzbach, M. (1992). Making Type Inference Practical. *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP '92).* Lecture Notes in Computer Science n° 615, pp. 329-349, Springer-Verlag.

[Snyder 1991] Snyder, A. (1991). Modelling the C++ object model: an application of an abstract object model. *Proceedings of the Fifth European Conference on Object-Oriented Programming (ECOOP '91)*, Lecture Notes in Computer Science n° 512, pp. 1-20, Springer-Verlag.

[Steele and Hillis 1986] Steele, G.L. Jr. and Hillis, W. D. (1986). Connection Machine Lisp : Fine-grained parallel symbolic processing. *Proceedings of the ACM Conference on Lisp and functional programming*, pp. 279-297.

[Steele 1995] Steele, G.L Jr (1995). Parallelism in Lisp. *Lisp Pointers*, vol. 8, n°2, pp.1-14.

[Steenkiste and Hennesssy 1987] Steenkiste, P. and Hennessy, J. (1987). Tags and Type Checking in Lisp: Hardware and Software approaches. *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp.

[Sunderam 1990] Sunderam, V.S. (1990). PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice & Experience*,vol. 2, n° 4, pp. 315-339.

[Suzuki 1981] Suzuki N. (1981). Inferring types in Smalltalk. *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 187-99.

[Talpin and Jouvelot 1994] Talpin, J-P. and Jouvelot,P. (1994). The Type and Effect Discipline. *Information and Computation*, n° 111, pp. 245-296

[Trealeven, Brownbridge and Hopkins 1982] Trealeven, P., Brownbridge, D. and Hopkins, R.(1982). Data-driven and demand-driven computer architecture. *Computing Surveys*, vol. 14, n° 1, pp.93-143.

[Wegner 1986] Wegner, P. (1986). Classification in Object-Oriented Systems. *ACM SIGPLAN Notices*, vol. 21 n°10, pp.173-182.

[Wegner 1987] Wegner, P (1987). Dimensions of Object-Based Language Design. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). Special Issue of ACM SIGPLAN Notices*, vol. 22, pp. 168-182.

[Winder, Wei and Roberts 1992] Winder, R.., Wei, M. and Roberts, G. (1992). UC++: An Active Object Model for Parallel C++. *Research Note RN/92/115*, Department of Computer Science, University College London, 1992.

[Wright 1991] Wright, A.K. (1991). Typing references by effect inference. *Proceedings of the European Symposium on Programming (ESOP '91)*, Lecture Notes in Computer Science, n° 582, pp. 473-491, Springer-Verlag.

[Wright 1993] Wright, A.K. (1993). Polymorphism for Imperative Languages without Imperative Types. *Rice University Technical Report TR93-200.*