UNIVERSITY
of
GLASGOW

Computing Science

Ph.D. Thesis

Action Notation Transformations

Hermano Perrelli de Moura

Submitted for the degree of

Doctor of Philosophy

Action Notation Transformations
by
Hermano Perrelli de Moura

Submitted to the Department of Computing Science
on 15th November, 1993
for the degree of
Doctor of Philosophy

Abstract

ACTRESS is a semantics-directed compiler generation system based on action semantics. Its aim is to generate compilers whose performance is closer to hand-written compilers than the ones generated by other semantics-directed compiler generators.

ACTRESS generates a compiler for a language based solely on the language's action semantic description. We describe the process by which this is achieved.

A compiler for action notation, the formal notation used in action semantic descriptions, is the core of the generated compilers. We specify and implement a code generator for the action notation compiler. We also present the design and implementation of an action notation interpreter.

A conventional hand-written compiler eliminates, whenever possible, references to identifiers at compile time. Some storage allocation is often performed at compile-time too. We can see both steps as transformations whose main objective is to improve the quality of the object code. The compiler writer, based on his knowledge of properties of the source language, implements these "transformations" as best as he can. In the context of ACTRESS, where action notation can be seen as the intermediate language of every generated compiler, we adopt a similar approach. We introduce a set of transformations, called action transformations, which allow the systematic and automatic elimination of bindings in action notation for statically scoped languages. They also allocate storage statically whenever possible. We formalise and implement these action transformations. The transformations may be included in generated compilers. We show that this inclusion improves the quality of the object code generated by ACTRESS' compilers.

In general, action transformations are a way to do some static processing of actions. Transforming actions corresponds to partially performing them, leaving less work to be done at performance time. Thus, transformed actions are more efficient.

Binding elimination exposes the static and dynamic nature of bindings in action notation. This relates with the binding discipline one can find in a programming language. We study the binding discipline of action notation and we state a condition which identifies a statically scoped action. We extend this condition to a sufficient condition on the action semantics of a language which tells if the language is statically scoped. This condition can be implemented as an analysis to be performed by ACTRESS, at compiler generation time, to decide which transformations will be included in a generated compiler for the language.

Finally, we list possibilities for improvements and potential areas for further research.

Thesis Supervisor: Dr. David A. Watt
Title: Head of the Department of Computing Science

To my parents,
Ebel and Ceiça.

Pregão Turístico do Recife

Aqui o mar é uma montanha
regular redonda e azul,
mais alta que os arrecifes
e os mangues rasos ao sul.

Do mar podeis extrair,
do mar deste litoral,
um fio de luz precisa,
matemática ou metal.

Na cidade propriamente
velhos sobrados esguios
apertam ombros calcários
de cada lado de um rio.

Com os sobrados podeis
aprender lição madura:
um certo equilíbrio leve,
na escrita, da arquitetura.

E neste rio indigente,
sangue-lama que circula
entre cimento e esclerose
com sua marcha quase nula,

e na gente que se estagna
nas mucosas deste rio,
morrendo de apodrecer
vidas inteiras a fio,

podeis aprender que o homem
é sempre a melhor medida.
Mais: que a medida do homem
não é a morte mas a vida.

João Cabral de Melo Neto, 1955.

Acknowledgments

This thesis would not have come to reality without the great help of my supervisor David Watt. In our weekly meetings, David always participated with clear suggestions and ideas to improve my current work. I specially remember his care in the formulation of concepts, and his patience in answering my questions about compiler techniques. I will be always in debt to him.

A special thank to Deryck Brown, my colleague in the ACTRESS project, for all fruitful discussions we had, and the innumerable suggestions he gave to improve action transformations.

I would like to thank Philip Wadler and Simon Peyton-Jones (my second and third supervisors respectively) for their critical feedback on various stages of my research. Kyung-Goo Doh, Peter Mosses and David Schmidt, for the interesting discussions we had during their visits to Glasgow (and over e-mail occasionally). Gebreselassie Baraki, Nick Holt, John Launchbury, Martin Musicante and André Santos, for their useful comments and interest in my work. Thanks to Sílvio Meira for all motivation and support in the early days of the PhD process. Peter Mosses and Simon Peyton-Jones, as members of the examination committee, provided many interesting corrections and suggestions to improve this thesis.

Deryck Brown, Brian Matthews, Patrick Sansom and Duncan Sinclair for providing a nice environment in G141. Deryck and Duncan acted many times as my UNIX and X Window consultants!

CNPq (Brazilian Research Council) and CEF (Caixa Econômica Federal) provided the funding necessary to conduct the research which lead to this thesis (and to keep me and my family alive). Thanks to the Department of Computing Science of Glasgow University for a stimulating and friendly environment.

Some friends made our stay in Scotland a most enjoyable one. Thanks to Augusto Sampaio, for keeping the "Thesis Factory" spirit alive from Oxford. João Lopes was a nice company in our "saunas", which helped to relax from the stress of writing up. I am grateful to the Brazilian community in Glasgow who always gave us encouragement, sharing of problems, and organized some nice "festas", which made us feel more at home. Mark Rickards, an Englishman who loves Brazil, taught us to love the Scottish countryside. I specially recognize the help of Luiz Amado picking up Leonardo from school daily. Thanks Tchê!

I could not forget to mention the moral support and fondness of my family and family-in-law. In particular my sister Magda and aunt Lindalva.

Special thanks to Lindinalva, my mother in law. When half of the family was writing up, she gave all the support we needed to keep the home and family running.

Finally, my gratitude and love to Rose, Clarissa and Leonardo, my wife and children, for all their support, patience, encouragement and love. Particularly during the overloaded writing moments, when I had to be absent many times, they gave me all their understanding and solidarity. Big "xxxxx" to you all!

Hermano Moura

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is about the design, formalisation and implementation of action transformations. Experimentation with the transformations in the context of an action semantics based compiler generator reveals their effectiveness in transforming actions into more efficient actions.

1.1 Motivation

Action semantics is a formalism for the specification of programming languages, developed by Peter Mosses and David Watt [80, 110, 82]. The operational aspect of action semantics motivated some thoughts on the possibility of its use for (semantics-directed) compiler generation (Watt's conjecture). This was the start point of the design and implementation of ACTRESS, an action-semantics based compiler generator. The design of ACTRESS was a three-person task [16]. After the implementation of the preliminary version, possibilities for improvements were identified. The elimination of bindings and the allocation of storage at compile time appeared to be measures which could improve the quality of the object code generated by ACTRESS' compilers. This was the main motivation for the work presented here. Therefore we introduced *action transformations*, which proved to be a natural approach to the problem.

1.2 Scope and Objectives

Computer programs are complex objects. They are sentences of programming languages. A high-level programming language program has many different forms: an abstract form in some semantic model, a source form, its various forms during the compilation process, its final form as a machine-coded object. One can look at a program statically, and infer properties about it without the need to run it. On the other side, the dynamic world of a program can be very diverse: memory is allocated, control is transferred, the running can be eternal, etc. Analysing all these forms and worlds together in an unified framework is a challenging task.

Formal semantics of programming languages have helped us greatly to understand this world of programming languages and their programs. Designers, implementors and programmers benefit from this formal approach to programming languages. In particular, implementors can have a more systematic way to construct compilers, and their products are more reliable.

One can view a semantic description of a programming language from different points. At one extreme, it is viewed as the base against which a correctness proof for an implementation of the language should be given. At the other extreme, and this is the one explored in this thesis, it is viewed as the base from which an automatic implementation for the language can be obtained. Between these extremes, one can view a semantic description as a common base for discussions on extensions and enhancements to the language; or as a guide to a manual implementation for it.

However, the use of formal semantics to generate compilers, although feasible, has not achieved the objective of generating production quality compilers. In other words, *realistic semantics-directed compiler generation* is something yet to be achieved. Usually, object programs of generated compilers run two order of magnitude slower than the object programs of hand-crafted compilers.

ACTRESS' objective is to narrow this performance gap between hand-crafted compilers and generated compilers. It seems that this gap can be narrowed by the introduction of various forms of (static) analysis in the compiler generator. The technique of binding elimination, the main subject of this thesis, can be seen as an attempt to include in an automatic compiler generation system, in a systematic and formal way, part of the knowledge that the implementor uses when he is writing a compiler.

1.3 Organization

This presentation of our work is divided into four parts. The first part, comprising this and the following two chapters, introduces the work and puts it in context. Chapters 4, 5 and 6 are where the principal part of the work is fully described. Chapter 7 closes the presentation summarising and comparing the results. Two appendices complement the presentation. We describe now the main points of each chapter.

Chapter 2 is an overview of semantics-directed compiler generation. We identify main approaches, describe some systems and discuss the main problems of some of the current systems.

An introduction to action semantics is the content of Chapter 3. The syntax and semantics of ACTRESS' action notation is given. SPECIMEN, the programming language used for illustrative purposes throughout the thesis is introduced. Appendix A contains an informal description of SPECIMEN, and Appendix B contains its action semantic description.

ACTRESS is introduced in Chapter 4. We describe mainly the code generation process (action notation to C) and how an action semantic description is used to generate compilers. For illustration, we describe how a compiler for SPECIMEN is obtained. An interpreter for action notation is also briefly described.

Action transformations are described in Chapter 5. Binding elimination and static storage allocation are explained in detail, and many examples are given.

After reviewing binding notions present in programming languages and action notation, we present in Chapter 6 a condition which identifies statically scoped languages from their action semantic descriptions.

Finally, in Chapter 7, we discuss the effectiveness of action transformations in the ACTRESS context, we discuss the relationship of our approach to others, and we point out some possibilities for future extensions, improvements and further work.

Chapter 2

Semantics-Directed Compiler Generation

There are various ways of defining a programming language. However, the fundamental point is not the choice of one or another type of semantic definition. It is probably possible to derive a compiler from any kind (denotational, algebraic, operational) of semantic definition in a more or less easy way. The point is that one needs to use a formal semantics in a compiler generator.

M. C. Gaudel, 1981, in [37].

This chapter gives an overview of semantics-directed compiler generation. We start explaining what semantics-directed compiler generation is. Then we identify major approaches used in the design of semantics-directed compiler generation systems. The organization and features of some typical systems are covered. Finally we discuss some of the problems with current systems and what we think an ideal semantics-directed compiler generator would be.

2.1 Compilation and Interpretation

Compilation is the process of translating a program written in a *source language* (*source program*) to a program with an equivalent meaning in a *target language* (*object program*). A *compiler* is a program that performs this process. Consider a language $\mathcal{L}$, a program $\mathcal{P}$ of $\mathcal{L}$, and a (real) machine $\mathcal{M}$. At compile time, program $\mathcal{P}$ expressed in language $\mathcal{L}$ is translated to a program $\mathcal{P}$ expressed in $\mathcal{M}$ machine code. The compilation is carried out by a compiler typically running on machine $\mathcal{M}$ (it could be also a cross-compiler running on machine $\mathcal{M}'$). The compiler itself is expressed in $\mathcal{M}$ (or $\mathcal{M}'$) machine code. At run time, execution of the object program is carried out by machine $\mathcal{M}$.

It is worth to distinguish compilation from the *interpretation* which is used in the implementation of some programming languages. The interpreter executes instructions in the source program immediately as they are fetched. The main distinction between compilation and interpretation is the absence or presence, respectively, of the language processor at run time. In compilation, the compiler is discharged as soon as it generates an object program which can then be run on its own; in interpretation the object program needs the interpreter in order to run. Compilation also exhibits a clear notion of compile time (static phase) and run time (dynamic phase) of the program.

However the borderline between interpretation and compilation is not always so clear. Some systems use a mixture of them: compilation is used to translate source programs into instructions for an abstract machine — the object program — which will be run by an interpreter for the abstract machine. Although unlikely, we could even have the case where a compiler for a language compiles a source program by generating an interpreter and preserving the source program. The running of the "object program" is just the interpretation of the original source program by the generated interpreter.

We will use the term *compiler* for *a translator from source code to machine code, which can be run autonomously on a real machine*[1].

Another measure of how much a particular language processor is near compilation is the degree of *interpretive code* in its object programs. Ideally we would like to have no interpretive code in the object code as this means better quality object code. Notice that run-time language support, like garbage collection, is not considered as interpretive code.

2.2 The Compilation Process

Figure 2.1 illustrates the phases involved in the compilation process. The whole process has two main parts: the analysis part and the synthesis part. Each of these parts can be in turn divided into several phases.

[1]Notice that even in this case some interpretation is present at the machine microprogramming level.

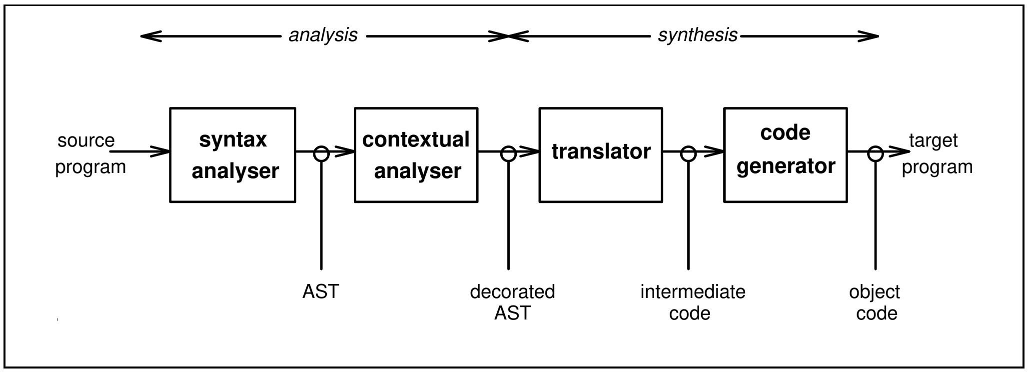


Figure 2.1: The basic phases of a compiler.

The analysis part comprises the *syntax analyser* (scanner and parser) and the *contextual analyser* (contextual constraint and type checker). The syntax analyser discovers the syntactic structure of a source program and (typically) builds an abstract syntax tree (AST) representation of it. The contextual analyser examines the abstract syntax tree in order to collect static information about the program. Tree nodes can then be decorated with this information resulting in a decorated abstract syntax tree. The part of a compiler which implements the analysis phase is also known as its *front end*.

The synthesis part basically comprises the translation phase and the code generation phase. The former translates the decorated abstract syntax tree into an intermediate representation of the program. This representation is convenient for conversion to target machine code. The *code generator* translates intermediate representations into final target programs. An object program consists of code that can be directly executed by the target machine (which can be an abstract machine). The object code is typically machine code, but it can also be assembly code or even a high-level language like C. The part of a compiler which implements the synthesis phase is also known as its *back end*.

This is not a complete and detailed scheme of the whole compilation process, but is enough for our presentation purposes. (For a more detailed presentation see [2].)

2.3 Compiler Generation

The manual construction of a compiler is a big and time consuming task. For example, the first PASCAL compilers took 6-12 man-months to be built. A compiler for ADA is likely to consume several man-years to be finished. Also, the task to assure the correctness of hand-crafted compilers is very demanding. Formal approaches to compiler correctness may be based on denotational semantics, algebraic semantics or structural operational semantics. (See [90] for an overview of compiler correctness proofs.)

Thus it is justifiable to search for methods and tools which can provide a more productive, automated, and systematic way to obtain compilers. Compiler correctness can also benefit from this approach. This is the aim of *compiler generation*. The decomposition of the compilation process means that the compiler generation problem can also be divided into smaller problems. Some of these subproblems are already satisfactorily solved. The decomposition also allows for the possibility of reusing compiler components. For example, we could have a family of compilers with a common front end and different back ends. Several *compiler writing tools* are now commonly used to generate parts of compilers automatically.

The LEX and YACC systems are examples of compiler writing tools. LEX [67] can be used to automatically produce a scanner from a regular-expression specification. YACC [49] automatically produces an (LR) parser from a grammatical description of the syntax of a language.

Front end generators typically process attribute grammars [59], which have proved useful for formally specifying the context-sensitive constraints of a programming language together with its context-free syntax. Linguist-86, GAG and HLP are examples of such systems [29, 30, 56, 96].

There are also some techniques that permit the generation of code generators [7, 34, 38]. The best systems generate code generators whose object code is comparable to the object code of hand-written code generators. However, some theoretical problems, such as proving that the translation (intermediate to object code) preserves the semantics of the source language, remain to be solved. As mentioned in [65], the correctness proof for code generators involves intractable congruence proofs, and these have not been given for practical code generators. In the case of automatically generated code generators, their correctness must be taken for granted.

2.4 Semantics-Directed Compiler Generation

The automatic generation of the translation phase is a more difficult task. This involves a strong connection to the formal semantics of the source language. The problem of automatically generating the translation phase based on the language's formal semantics constitutes a dynamic area of research called *semantics-directed compiler generation.*

In a broader view, the aim of semantics-directed compiler generation is to generate a compiler for a language $\mathcal{L}$ from $\mathcal{L}$'s syntactic and semantic formal description. This formal description is defined using some *formal notation.* Examples of formal notations are λ-notation and action notation. In some semantic methods, the meaning of a source program is a term in this formal notation. This term can be built and then translated into a target language. This formal notation should be a well known and general notation for description of programming languages, not a particular compiler specification language. It should be theoretically well founded, with a well-defined semantics.

This is an important point in the characterization of semantics-directed systems. The compiler generators and generated compilers can be designed in a way to exploit the properties of the formal notation, not the other way around. Again, analysis and transformation techniques can be built into the compiler generators and generated compilers based on the properties of the formal notation. It is important to make clear that many so-called compiler generators are merely tools to help the construction of specific compiler components. They are based on some particular techniques, not on a formal notation for description of programming languages, and/or the user has to provide some of the components of the generated compiler. Thus, there is a borderline (sometimes not so clear) between (compiler writing) tools and true semantics based compiler generators.

We can see many advantages of a semantics-directed compiler generation system:

- The precise intentions of the language designer are reflected by the formal description and transferred to the implementations.
- The compilers are generated automatically from formal descriptions, which are easier to write and debug than standard code for a compiler.
- If the compiler generator is correct, the generated compilers are correct with respect to the formal descriptions.

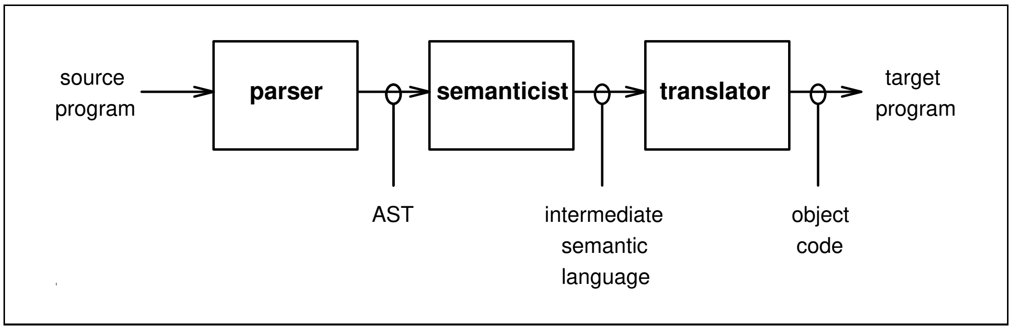


Figure 2.2: The inners of a generated compiler.

- The compiler generator provides a convenient tool for executing programs of a language as soon as a semantic description of the language is available.

Pleban's *language designer's workbench* concept [65] describes an environment centered on the existence of a formal semantic method (with nice pragmatic features) and a compiler generator based on it. In [112] Watt advocates a methodology of programming language design that exploits the ability to execute semantic descriptions: one only commits to the construction of a production compiler after some iterations over a *design-prototype-test* cycle entirely based on a formal semantics of the source language.

Figure 2.2 illustrates how the generated compiler can incorporate the semantics of the source language. The parser component is generated from the syntax description of the source language. The *semanticist* component incorporates the semantic description into the generated compiler. It takes the source program's abstract syntax tree (AST) and generates an intermediate representation of the source program. This intermediate representation is usually a term of the semantic meta-language, and represents the denotation of the source program. The *translator* translates the intermediate semantic representation to object code. In the case of a variable target language organization, the translator incorporates the semantics of the target language and the translation rules from terms in the source language to terms in the target language.

One could draw a comparison between the current state of compiler generation technology and the early days of programming language semantic methods. At that time, syntactic description methods were well known and semantic methods still in their infancy. There are many methods and tools widely available and used to generate the syntactical components of a compiler, but much remains to be done regarding semantic components. Although the latter is feasible it is not viable at present. Some semantics directed compiler generators deliver compilers whose object programs run three orders of magnitude slower than code generated by hand-written compilers [65].

2.5 Approaches to Compiler Generation

There are five main well known methods for defining semantics of programming languages: *operational semantics*, *axiomatic semantics*, *denotational semantics*, *algebraic semantics* and *action semantics*. Naturally, a lot of the work in semantics-directed compiler generation has been influenced by some of these methods.

As far as we know axiomatic semantics [32, 45] has not been used for compiler generation. As pointed in [65], the problem with this method is its inability to treat easily such common language features as side effects and scope; also, an axiomatic definition is designed to support reasoning about particular *properties* of programs, and it is hard to use these properties as the meaning of programs to build language implementations.

Denotational semantics, a well accepted method to describe semantics of programming languages, has inspired much work on semantics-based compiler generation, including the first system of this kind.

Surprisingly, traditional operational semantics has not been used for compiler generation. An operational semantics is very suggestive of an implementation. An interpreter can easily be defined from it. However, using new approaches to operational semantics, some interesting systems have been built.

Although the algebraic approach has been extensively used in algebraic specification of data types, it seems that it has not been applied to compiler generation. However some systems generate language tools such as editors, parsers and interpreters from an algebraic specification of the language. It is important to notice that algebraic techniques make available some useful mathematical knowledge that can help in areas like compiler correctness and compiler generator correctness.

Action semantics, as introduced in Chapter 3, although a new approach compared with the others, has already been used to build some systems.

Although *attribute grammars* and *partial evaluation* are not general methods for specifying programming languages, compiler generation has become an intensive application area for both techniques. In particular, attribute grammars have been used to create many compiler writing tools.

In this section we review briefly the principles behind each of these approaches and how they are used to build semantics-based compiler generators.

2.5.1 Denotational Semantics

In denotational semantics [79, 97], the semantics of a programming language is expressed as a mapping from syntactic phrases in the language to mathematical entities. In standard denotational semantics these mathematical entities are higher-order functions, and they are written using λ-notation [4]. The meaning of a program is a (higher-order) function which is obtained by the application of the semantic functions to the program's abstract syntax tree. In terms of compiler generation, this corresponds to a syntax-directed translation of the program's abstract syntax tree into λ-notation. A λ-expression constitutes the "target code" for a program, and can be "executed" by a *λ-expression evaluator*. Execution is performed by reduction of the λ-expression applied to the program's input.

Figure 2.3 shows the basic organization of a compiler generated by a denotational semantics based compiler generator. The generated compiler front end is a parser which maps a source program to its abstract syntax tree representation. The language's semantic description is incorporated in the *semantic function* phase (semanticist) which is a translator from an abstract syntax tree to a λ-expression denoting the source program. The λ-expression can be reduced at compile time using a reduction machine (*beta reductor*). The beta reductor applies the β reduction rules of the λ-calculus to evaluate a λ-expression. This improves the quality of the object code. At run time the λ-expression is supplied with its required arguments (program inputs), if any, and further reduced until a normal form is reached.

2.5.2 Operational Semantics

In traditional operational semantics, a language is defined by specifying a translation from the language to a defined abstract machine. This abstract machine can be very close to real hardware, in which case it can be simple to analyse and translate abstract machine code to real machine code, or high-level enough to facilitate the translation from the programming language. The abstract machine can be seen as an interpreter for the

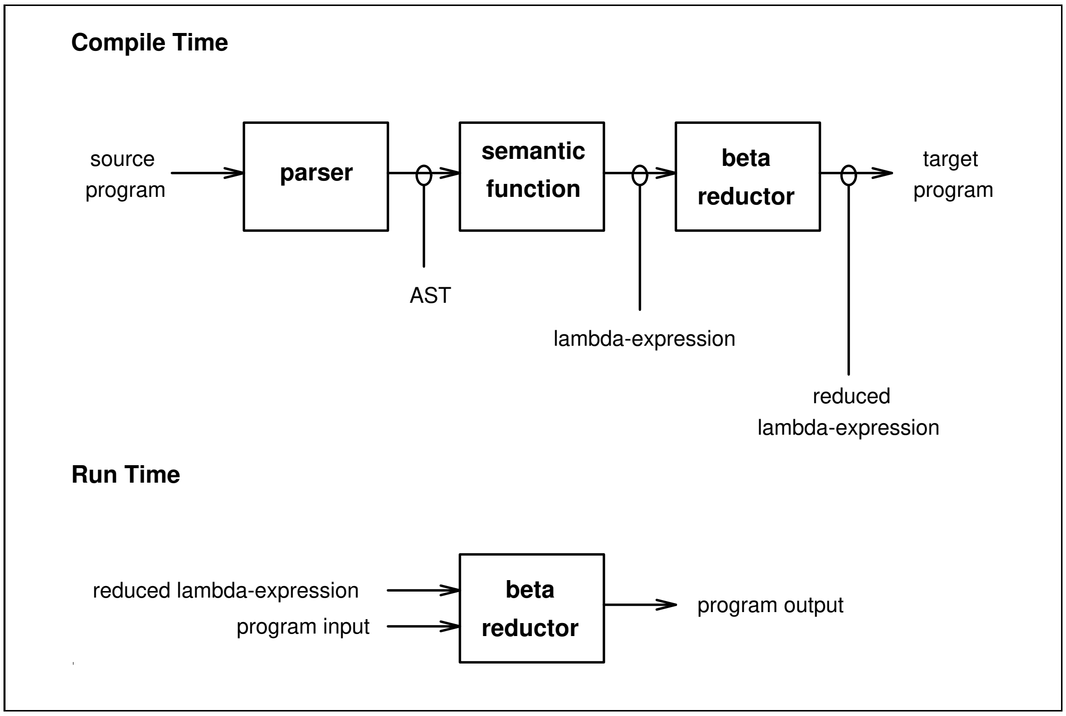


Figure 2.3: A compiler generated by a denotational semantics based system.

language, and the meaning of a program is the evaluation history that the interpreter produces when it interprets the program. The evaluation history is a sequence of internal interpreter configurations. An example of a definitional interpreter can be found in [63].

Plotkin's structural operational semantics [95] and Kahn's natural semantics [54] are two other interesting approaches to operational semantics. In the former, the semantics of a programming language is defined by a *transition system* whose steps describes the evaluation of a program in the language. In the latter, the semantics is defined by a mathematical relation between programs and results[2]. We will illustrate natural semantics here.

In general, a natural semantic definition provides axioms and inference rules that define the various semantic predicates to be defined on a language phrase. For example, we could have a rule of the form:

$$\mathcal{E}, \mathcal{S} \vdash E \Rightarrow v$$

which expresses that expression E evaluates to v in environment $\mathcal{E}$ and store $\mathcal{S}$.

[2]Sometimes the terms *transition semantics* and *relational semantics* are used to refer to structural operational semantics and natural semantics respectively [40, 20, 6].

There are probably various approaches to turn a natural semantics description into a compiler or interpreter. It can for example be compiled into PROLOG as explained in [54, 26]. Natural semantics can also be used to specify static semantics and translation [19].

Hannan and Miller have developed techniques for mechanically constructing provably correct implementations of programming languages based on operational semantics. Their work is based on the definition of *abstract machines* as term rewriting systems. In [43] they consider the transformation of a description given as a set of inference rules into abstract machines. In [42] is shown how the resulting abstract machines are transformed into compilers. A further translation to an even lower-level architecture, which is closer to machine code, is considered in [41].

2.5.3 Algebraic Semantics

Algebraic semantics has been used extensively for specification of abstract data types [114, 116, 5, 46]. An algebraic definition specifies some sorts, functionalities of some operations and some axioms (equations) over the operations. The meaning of an algebraic definition is an algebra (or a set of algebras).

Interpreting equations as left-to-right rewriting rules, an algebraic semantics specification can be *executed* by a term rewriting system [22].

Algebraic semantics can be used to specify a programming language. But in practice it is necessary to introduce auxiliary sorts such as environments and stores, auxiliary operations over these sorts, and auxiliary operations over the terms (phrases) of the programming language. The resulting definition is reminiscent of a denotational definition whithout higher-order functions.

2.5.4 Action Semantics

Action semantics is described in detail in Chapter 3. The denotations of program phrases are actions. An action can be performed. The execution of a program is represented by the performance of the action denoting the program. For example, the performance of the action may complete giving some value, which indicates a normal termination for the program.

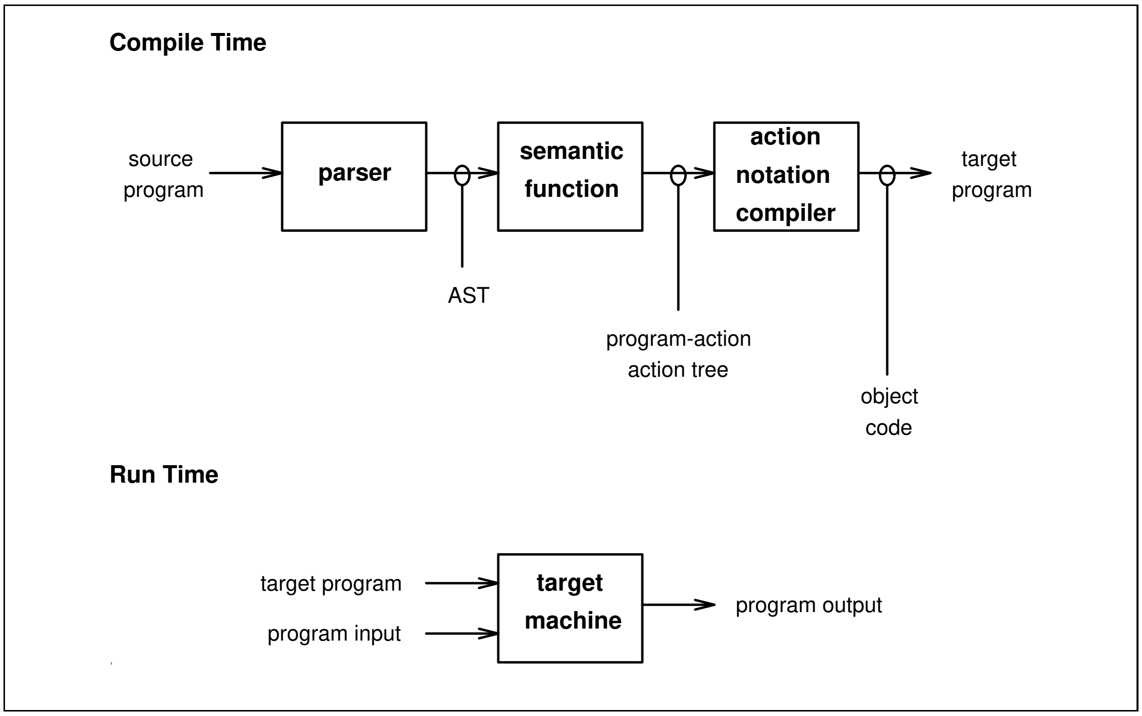


Figure 2.4: A compiler generated by an action semantics based compiler generator.

Figure 2.4 shows the general structure of a compiler generated by an action semantics based compiler generator. The central part of the system is an action notation compiler which translates an action to object code suitable for execution in some target machine. Notice, in Figure 2.4, that an action and an action notation compiler correspond, respectively, to a λ-expression and a λ-expression reductor in a denotational semantics based compiler generator (Figure 2.3).

2.5.5 Attribute Grammars

Attribute grammars were introduced by Knuth [59] as a way to incorporate (so called) semantics in a context-free grammar. They are mainly used to specify context-sensitive constraints of programming languages, like types and scoping rules, and code generation. The basic idea is that each grammar symbol (terminal or nonterminal) in the syntax tree has a fixed number of associated values, called *attributes*. Attributes represent information associated with the symbol, such as its type, symbol table, code sequence, value, and so on. Attributes may be evaluated as a program is parsed, or they may be evaluated after a syntax tree is constructed by the parser. The resulting syntax tree, augmented with attributes, represents the semantics of the program [31].

To obtain a type checker using an attribute grammar based system, for example, we first specify an attribute grammar with types as attributes and operations defining the type checking process. The system uses the specification to generate an attribute evaluator which implements the type checker.

2.5.6 Partial Evaluation

Although *partial evaluation* [50] is a program transformation technique, compiler generation is an important application of it.

A *partial evaluator* can be viewed as an interpreter that evaluates programs with partial input data. That is, if we give to a partial evaluator a program (*subject program*) and part of its input data, the partial evaluator will evaluate the program, using the known input data, given a new program as result (*residual program*). The residual program, when run with the rest of the input, will give the same result as would the subject program when run with its complete input data (this is the *correctness condition*). Thus, a residual program is a specialization of the subject program, with respect to its known input data.

Suppose we have an interpreter *int* for a language $\mathcal{L}$. Let p be an $\mathcal{L}$ program that needs some input data d to run. One could use *int* to run the program as follows:

$$int\ p\ d = r \tag{2.1}$$

where r is the result of running p with d as input data. Suppose now we use a partial evaluator (*mix*) as follows:

$$mix\ int\ p = t \tag{2.2}$$

that is, we specialized *int* with respect to a particular program p. Now, by the correctness condition, for all d:

$$t\ d = r \tag{2.3}$$

which is equivalent to saying that t is the compiled version of p.

The problem with (2.2) is that every time we need to compile a program p we need to partially evaluate *int*. We could avoid this by partially evaluating *mix* itself with respect to *int*:

$$mix\ mix\ int = comp \tag{2.4}$$

where *comp* is now a compiler for $\mathcal{L}$. Notice that, by the correctness condition:

$$comp\ p = t \tag{2.5}$$

To obtain a compiler using (2.4), the partial evaluator must be *self-applicable*, that is, it must have the property that it can partially evaluate itself. If we specialise *mix* with respect to itself, we obtain a compiler generator:

$$mix\ mix\ mix = cogen \tag{2.6}$$

which gives a compiler for $\mathcal{L}$ if we apply it to *int*:

$$cogen\ int = comp \tag{2.7}$$

This is the principle which allows us to use partial evaluation for compiler generation.

Notice that the user should provide an interpreter, which is easier to write than a compiler, to obtain a compiler for a language. The fact that partial evaluation can be used for semantics directed compiler generation stems from the fact that the interpreter can be obtained directly from a denotational semantic definition.

2.6 Compiler Generation Systems

There are many systems around which implement the various approaches to compiler generation. SIS [71, 73], PSP [92, 93], SPS [106], Kelsey and Hudak's system [57] and DML [94] are examples of denotational semantics based systems. The SAM system [115] uses a fixed semantic algebra as a mediator between the source language and the lambda-calculus [104]. TYPOL [25] is an example of a system based on natural semantics. PERLUETTE and ASF+SDF are examples of systems that use algebraic semantics techniques. CANTOR [91], ACTRESS [16] and Doh's system [27] are systems based on action semantics. DELTA [68], GAG [56, 55], HLP [96, 60, 61], LINGUIST-86 [29, 30] and MUG2 [35] are among numerous front-end generators developed using attribute grammar techniques. For

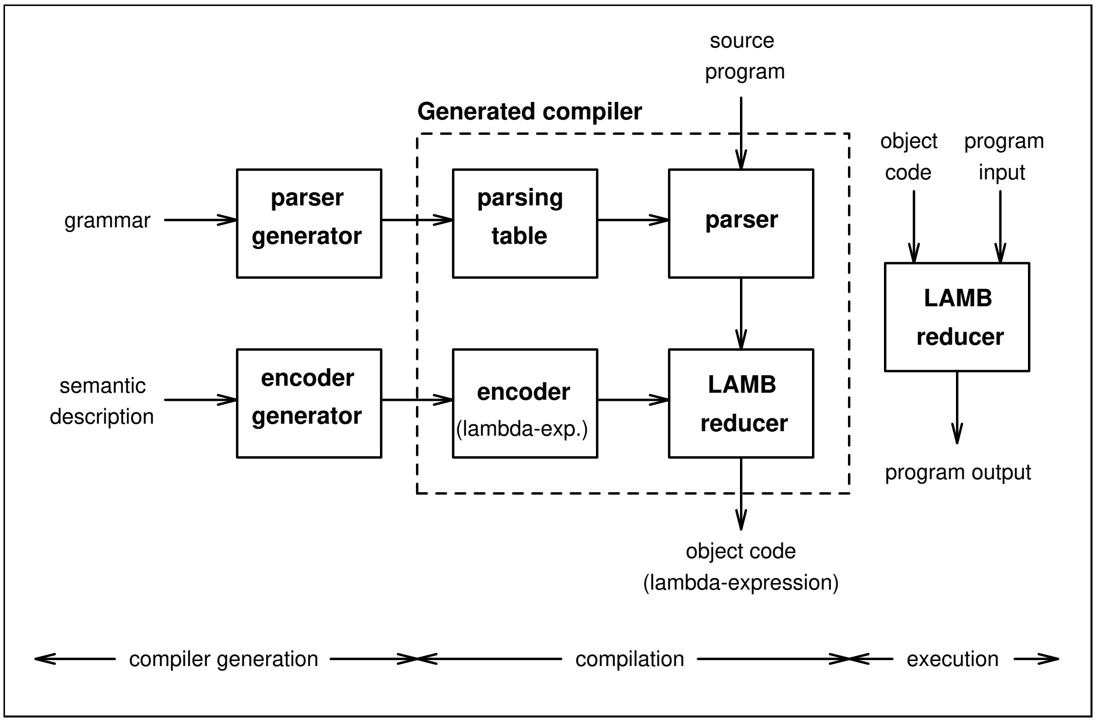


Figure 2.5: SIS – Semantics Implementation System.

a good survey of attribute grammars based systems see [21]. Mix [52] and Similix [11] are examples of partial evaluators. MESS [65], a system based on high-level semantics, is an example of a system that generates good quality compilers. In the sequel we describe some of these systems.

2.6.1 SIS

The Semantics Implementation System (SIS) was developed by Peter Mosses [71, 73]. It was the first denotational semantics based compiler generation system. The main components of the system are the parser, the parser generator, the encoder generator and the LAMB-reducer. Figure 2.5 shows the architecture of the system.

Semantic descriptions are written using DSL, a notation similar to the Scott-Strachey notation used in denotational definitions, and suitable for computer processing. DSL is a completely applicative notation [73]. The λ-notation used in SIS is called LAMB, which is in fact a sub-language of DSL. The encoder generator produces the code generator part of a compiler (*encoder*) from a denotational semantics of the source language, expressed in DSL. The encoder generator takes the parse tree of the semantic description, and produces basically an expression in λ-notation (called a LAMB-expression) denoting the specified semantic function. When this LAMB-expression (the encoder) is applied to the parse tree of a program, it produces a LAMB-expression denoting the semantics of the program (usually an input-output function). Actually, the produced LAMB-expression can be reduced at compile time and this reduced expression is taken as the generated object code.

To run a program we apply the generated λ-expression to the input (also a λ-expression). This application is evaluated (reduced to normal form) by the LAMB-reducer giving the output of the program. The LAMB-reducer is also used to evaluate applications of semantic functions to parse trees of programs. A call-by-need reduction strategy is used. SIS is written in BCPL.

An experiment with SIS is described in [9]. Some problems were reported as follows:

- There are some inefficiencies at compiler generation time, compile time and run time. The lexical analysers of SIS and of the generated compilers are based on context-free parsing methods rather than finite state techniques. The most critical source of inefficiency is the use of a reduction machine for the target machine of the "compiled code". The execution of loops causes the reducer to make many copies of the loop body during evaluation, thereby increasing the frequency of garbage collections.

- Error handling is not adequate. No syntax error recovery is provided at compiler generation time. Specifications written in DSL should be syntactically correct, otherwise SIS halts. The same applies to the parsers generated by SIS. No type checking is performed on the semantic equations. The language's abstract syntax must be specified three times: first in the description of the compiler front end, and subsequently (in a different form) as part of the definition of the static and dynamic semantics. However, no consistency check is included to ensure that the three specifications define identically structured abstract syntax trees.

2.6.2 Typol

Typol is a formalism that implements natural semantics [25, 26]. It can be used to specify static semantics, dynamic semantics, and translations. A natural semantic description is expressed and processed as a Typol program (ASCII representation). For example, a description of a type checker (a Typol program) is compiled into Prolog to create an executable type checker. Dynamic semantics and translations can be described and processed in a similar way [19]. Besides natural semantics rules, a Typol program also contains machinery for importing externally defined rules, functions, etc.

The Typol compiler includes a type checker and a code generator. Every abstract syntax term occurring in a rule is typed with its syntactic category. The type checking phase uses this information to verify a Typol description and generates an intermediate form [13]. After type checking, a Typol description can be compiled into Prolog code. A Prolog interpreter can then be used to execute the description. As described in [26], one of the main ideas in the design of the system was to keep the semantics of the rules independent of Prolog features, for example, the semantics of a Typol rule is independent of the order of the sequents in the numerator of the rule (as in natural semantics and logic). The current implementation uses MU-Prolog [86]. The type checker and the Prolog translator are written in Typol itself.

Typol runs on top of the Centaur system, an interactive programmming meta-environment, being one of the standard semantic formalisms provided by this system [13, 47].

2.6.3 Perluette

The Perluette system [23, 37, 36] is based on the specification of languages as abstract data types. It takes as its input three specifications: the source language definition, a description of the implementation choices, and the target language syntax. Figure 2.6 shows the architecture of the Perluette system.

The source language definition embodies the presentation of an algebraic data type which describes the properties of the language operations (statements, operators, etc). The semantic value of a program is a composition of some of these operations, that is, a term of the data type. The meaning of any program is stated via a set of algebraic semantic equations. The target language is specified as another algebraic data type. The translation is expressed as a representation of the *source data type* into the *target data type*.

The generated compilers work in three steps: the first step translates the source program to a term of the source data type; the second step translates this term to a term of

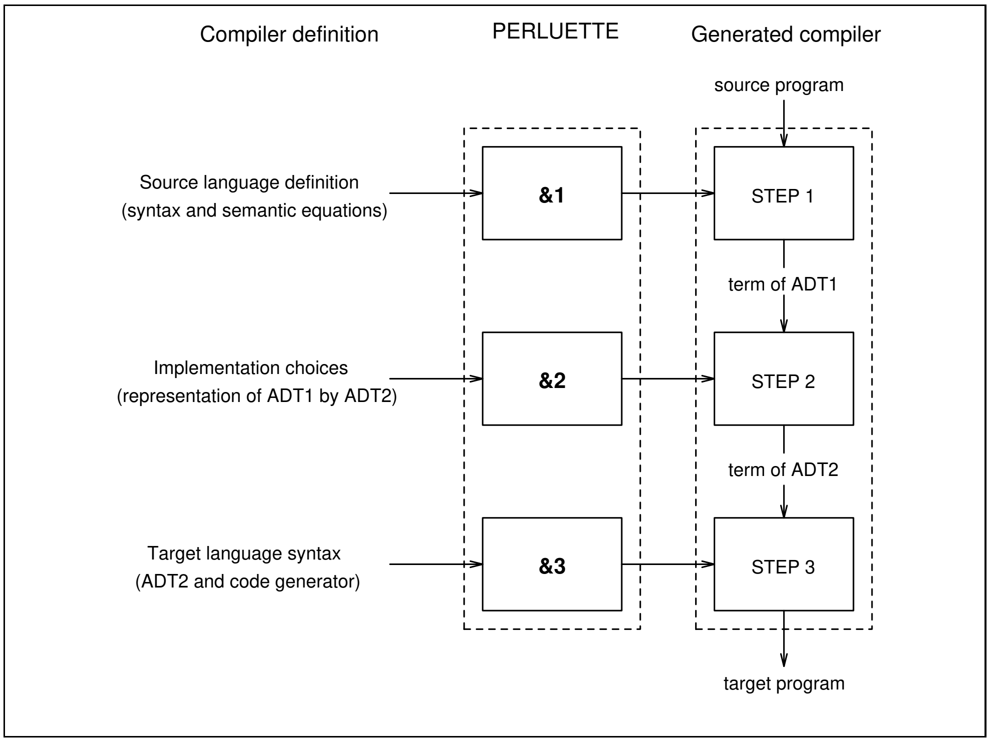


Figure 2.6: PERLUETTE.

the target data type; finally, the third step generates code from this translated term. The user must supply the code generator.

As described in [37] the semantic equations of the source language are specified using attributes. Attributes and the intermediate texts are expressed in LISP. Also, it is reported that the use of LISP is very natural as generated compilers perform term rewriting which can be expressed easily as a LISP evaluation.

As mentioned in [36], PERLUETTE takes into account only the "syntactic" part of the abstract data types associated with source and target languages. Also, according to [36], the axioms (the "semantic" part) of the data types are needed for the correctness proofs which were done by hand.

2.6.4 CANTOR

The CANTOR system generates compilers from action semantic descriptions of programming languages [90, 89]. The compiler generator component is written in PERL [105] and has as inputs the syntax and the semantics of the source language (these inputs are in the form of the actual LaTeX source of the semantic description, like the one that produced

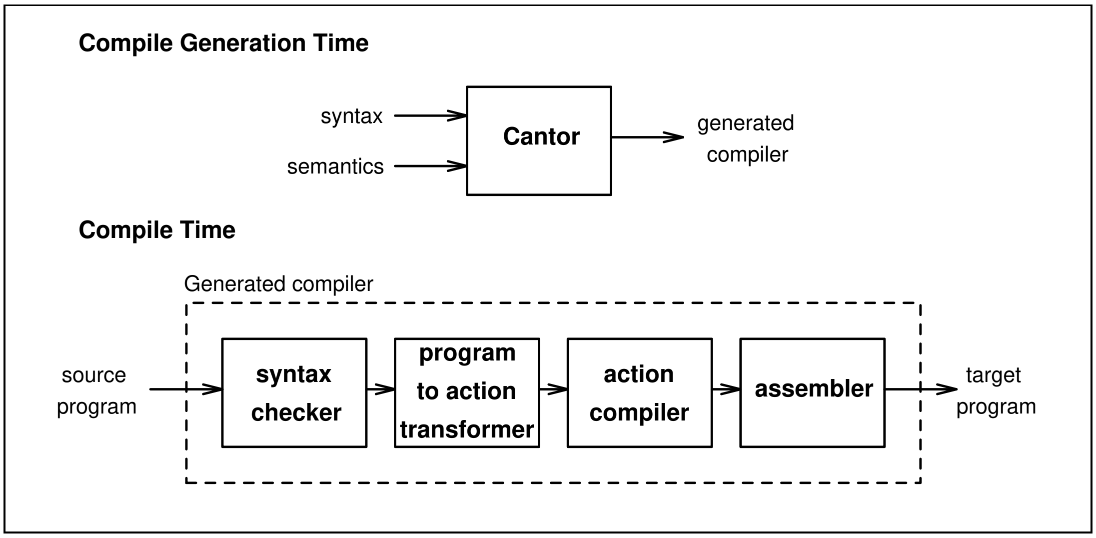


Figure 2.7: CANTOR.

Appendix B). The generated compiler emits code for an abstract RISC machine language (Pseudo SPARC), which is compiled into code for RISC processors. The generated compiler, written in SCHEME, consists of a syntax checker, a program-to-action transformer, an action notation compiler (a fixed part in the generated compiler) and a Pseudo SPARC assembler. Figure 2.7 shows the organization of CANTOR.

The main nice feature of CANTOR is that its correctness is proved. The proof is based on a natural semantics of action notation, a natural semantics of the abstract RISC machine, and the simplicity of the action notation compiler. It uses a variation of Despeyroux's proof technique [24].

As reported in [89] compilation time is very slow (circa 300 times slower than compilation time for hand-written compilers). Also the object code generated by CANTOR's compilers run two orders of magnitude slower than corresponding code produced by hand-written compilers. Experiments with CANTOR have included the automatic generation of compilers for a non-trivial subset of ADA and for HYPOPL [65].

2.6.5 GAG

From an attribute grammar specifying the static properties of a programming language, the GAG system [56] generates an attribute evaluator that implements the semantic analysis phase of a compiler.

The syntactic part of the attribute grammar is written in EXTENDED BNF [48]. The input attribute grammar is written in ALADIN (A Language for Attribute Definitions), a strongly typed applicative language. ALADIN is suited for specifying static language properties (for example, scoping and typing rules). Descriptions of code generation and optimizations are possible, but are not typical applications. The attributed tree, resulting from the analysis phase, is further processed by the later compiler phases.

As mentioned in [55] the specification of an attribute grammar should start with an analysis of the given or intended language and proceed in the following steps:

- specification of the context-free grammar;
- design of attribute types for the description of global language concepts like scope rules and types of objects;
- and design of the attribute rules and context dependent restrictions for each production together with functions for their computation.

Figure 2.8 shows how the GAG system is organized. The structure is comparable to the one found in conventional compilers. First the specified attribute grammar, written in ALADIN, is analysed (this analysis performs scanning, parsing and type checking for ALADIN). Then analysis of attribute dependencies and computation of tables that will control the attribute evaluator take place. Then the attribute evaluator performance (space and run time) is improved by several optimization techniques (like space reduction for attributes using lifetime analysis). Finally, the specification language ALADIN is translated into STANDARD PASCAL [55].

The system is implemented in STANDARD PASCAL [17] as well as the generated attribute evaluators. There are some facilities to embed the generated attribute evaluator in a compiler environment. Front ends have been generated for PASCAL, ADA and PEARL [87]. Performance figures show that the efficiency of generated front ends are very close to those of compilers using a tree as internal structure.

2.6.6 MIX

The first version of MIX, a self-applicable partial evaluator, was developed in 1984 at the University of Copenhagen [52]. Its subject language is MIXWELL, which is basically a subset of (pure) statically scoped LISP. MIX itself is written in MIXWELL. To generate a

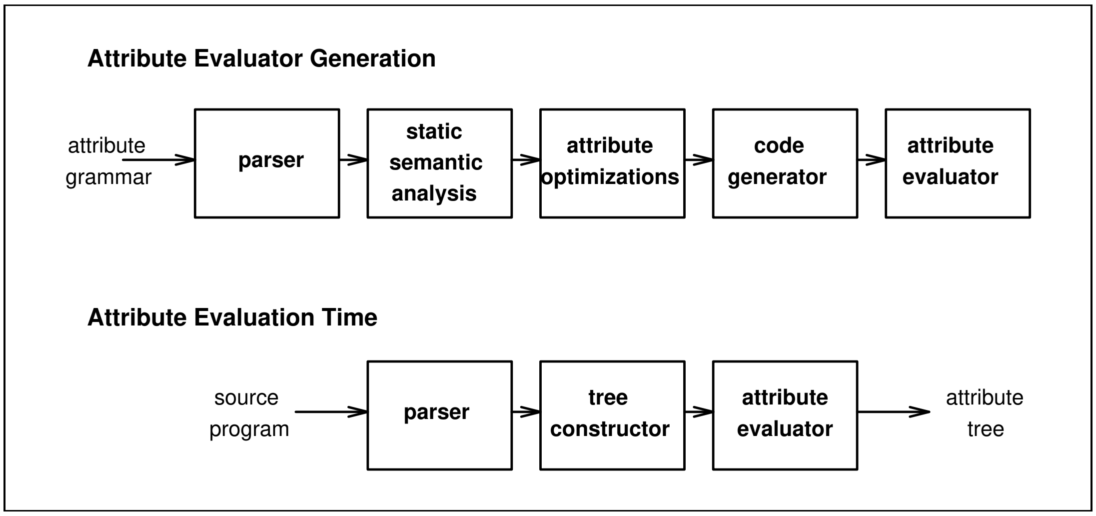


Figure 2.8: The GAG system.

compiler for a language $\mathcal{L}$, the user must provide an interpreter for $\mathcal{L}$, written in MIXWELL. By using Equation 2.7, the compiler is generated.

Partial evaluation in MIX comprises the following phases:

- binding time analysis,
- program annotation,
- function specialization,
- call graph analysis, and
- call unfolding and reduction.

Each phase can be understood as a transformation on the subject program, involving an analysis or a translation of it. The input to the binding time analysis is the subject program together with a description of which of the program's parameters will be available during partial evaluation. Binding time analysis is based on an *abstract interpretation* [1] of the subject program using the two-value domain $\{Static, Dynamic\}$ which reflects the condition of all variables in the program as static or dynamic. A static variable can be computed during partial evaluation, a dynamic variable cannot. So the aim of binding-time analysis is to analyse the program so annotations may be placed accurately.

The subject program together with the variable descriptions produced by the binding time analysis are used in the annotation phase which produces an annotated version of the subject program for use by the function specialization phase. *Function call annotation* is one of the annotations carried out in this phase: for example, a function call is marked as unfoldable if there is no risk of infinite expansion during function specialization, and residual otherwise [52].

The annotated version of the subject program — together with actual values of the subject program's parameters — is used in the function specialization phase to produce an intermediate residual program which consists of specialized versions of the subject program's functions.

The call graph analysis produces a list of function names from the intermediate residual program that are cutpoints of recursive call chains. These cutpoints will be used in the call unfolding phase to avoid infinite unfolding.

Finally, the call unfolding and reduction phase outputs the final residual program, obtained from the intermediate residual program, by unfolding calls and reducing the resulting expressions.

MIX has been used to generate compilers for various imperative and functional languages [52]. For a small imperative language reported in [52] (with assignment, a conditional, a while-loop, and with S-expressions as the only data type), the compilation by partial evaluation combined with a run of the target program is five times faster than the interpretation of the source program.

2.7 Other Compiler Generator Systems

There are some other semantics-directed compiler generators. Paulson's Semantics Processor (PSP) uses *semantic grammars* as its specification language. A semantic grammar is an attribute grammar that uses the domains and formulas of denotational semantics [93]. A traditional denotational definition includes a context-free grammar, and introduces a semantic function for every nonterminal symbol in the grammar, defined by cases (semantic equations) on the rules rewriting that nonterminal. In contrast, a semantic grammar embeds the semantic functions in attribute evaluation rules associated with the rules of the context-free grammar. As report in [93] the compiler generator is efficient enough to run experimental programs, but it is impractical for a production environment. A generated compiler for a subset of PASCAL compiles 25 times slower and its object code runs 1000 times slower than a hand-written compiler. PSP is written in PASCAL.

Mitchell Wand's Semantic Prototyping System (SPS) uses denotational semantics as its semantic formalism, but descriptions are expressed in a standard LISP syntax [106]. A typical language description, provided by the user, consists of definitions of types and auxiliary functions, and a translator (*transducer*) that includes the syntactic and semantic definition of the source language. The system has a type checker, which is used to debug semantic descriptions, and an interface to YACC which extracts the grammar from the transducer. The transducer is processed to produce a parser and an interpreter (a SCHEME function that traverses the parse tree) for the language. A program in the defined language may be run by piping it through the parser and then interpreting the parser's output. The system is written in SCHEME 84 [33], a dialect of LISP. There is reported a (CPU) time of 0.18 seconds to execute the 12-line program in [9] on a VAX 11/780 machine.

2.7.1 MESS

The MESS system generates compilers from *high-level semantic descriptions* [65]. The following presentation of MESS is based on the one found in [65]. High-level semantics is a *style* of semantic definition that overcomes the unsuitability of traditional denotational semantics for compiler generation. Its main features are: (a) the denotations are expressed in terms of a semantic algebra of action-based operators rather than the λ-calculus; (b) the operators of a semantic algebra are chosen to directly reflect both fundamental language concepts and fundamental implementation concepts — an efficient implementation of the operators can be obtained by interpreting them as templates of intermediate code for a code generator; (c) the semantic equations and the semantic algebra are defined in separate specifications called the *macrosemantics* and *microsemantics*, respectively; (d) the only information shared between macrosemantics and microsemantics is the signature of the semantic algebra defined by the microsemantics. This provides a modularity which guarantees the invariance of the macrosemantics under different interpretations of the algebra; (e) the separation between macrosemantics and microsemantics is also used to distinguish between the static and dynamic components of a language; (f) it is usually straightforward to add new operators to a semantic algebra which provides extensibility; (g) high-level specifications are written in a readable notation based on STANDARD ML.

A macrosemantics specifies a translation from abstract syntax trees to terms in a semantic algebra of actions called *prefix-form operator terms*, or POTs. This translation is a static computation since a POT represents the total dynamic effect of a program.

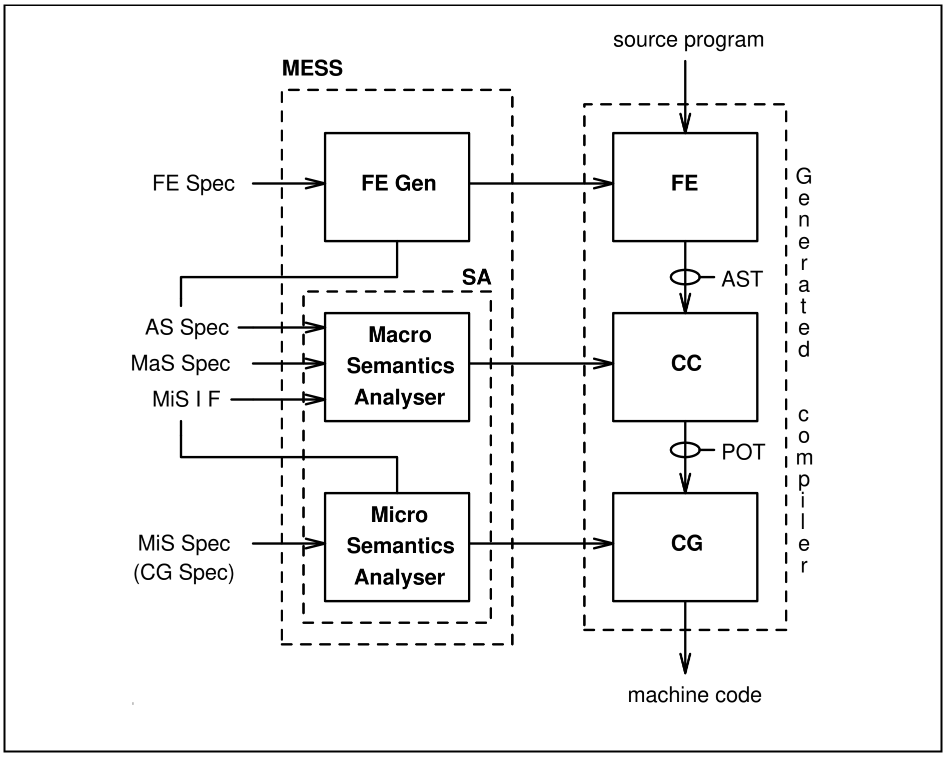


Figure 2.9: The MESS system.

In terms of realistic compilation, this means that a macrosemantics specifies all of the compiler computations involving with translating an abstract syntax tree to a POT intermediate representation. The microsemantics, then, provides a way of interpreting the POTs, for example by means of a code generator which regards the POTs as intermediate code (this would be a compiling interpretation for the semantic algebra).

Figure 2.9 shows the organization of the MESS system. The system is comprised of two parts, the front-end generator (FE Gen) and the semantics analyser (SA). From a specification of the source language's concrete syntax augmented with tree building rules (FE Spec), the front-end generator produces a compiler front-end (FE) that parses programs and builds abstract syntax trees. In addition, a specification of the abstract syntax (AS Spec) is generated. This is used by the semantic analyser to ensure the consistency of the abstract syntax expressions appearing in the front-end and macrosemantic specifications. The generated front-ends and the front-end generator itself are written in Pascal.

The semantics analyser processes both the macrosemantic and microsemantic specifications. From a microsemantic specification (MiS Spec) of a code generator (CG Spec), the microsemantic analyser generates a code generator (CG). This code generator translates

POTs to machine code. Another product of microsemantics analysis is the generation of a *microsemantics interface file* (MiS I F), which contains the specification of the signature of the semantic algebra (names of the actions domains and functionalities of the operators). A macrosemantic specification (MaS Spec), along with the microsemantics interface file, can be processed by the semantics analyser in order to generate a compiler core (CC). The compiler core, which is written in SCHEME, translates abstract syntax trees into POTs. The POTs are written as SCHEME s-expressions. The semantics analyser is itself written in SCHEME. The combination of the front-end, the compiler core and the code generator constitutes the generated compiler which translates source programs into machine code. A number of techniques for the generation of code generators from formal specification [7, 34, 38] may be used since the POTs are in prefix format.

2.7.2 ASF+SDF Meta-environment

The ASF+SDF Meta-environment [58] is based on the ASF+SDF formalism. It allows for rapid prototyping of algebraic specifications, expressed in the ASF+SDF formalism. The ASF+SDF formalism combines the "Algebraic Specification Formalism" [5] with the "Syntax Definition Formalism" [44]. By viewing signatures also as grammars, concrete syntax can be used for terms (e.g., in the equations). The formalism supports modularization, conditional equations, and built-in associative lists. The formalism is suited to provide specifications for arbitrary abstract data types (traditional algebraic specification), as well as definitions of any (formal) language (e.g. programming, query, text-processing, specification, etc).

From the signatures, parsers are generated, and from the equations, term rewriting systems are generated. Terms can be edited using syntax-directed editors. The ASF+SDF Meta-environment has an incremental implementation; if the specification is changed the prototyped tools are adapted rather than regenerated from scratch. This supports interactive developing and testing of specifications.

2.8 Comparison

A comparative study among the various semantics-directed compiler generator systems is a difficult task. Different formalisms, diverse built-in analysis, implementations on different machines, different example languages, etc, are some of the points that have to be analysed. A fair comparison would be against hand-written compilers and their object code (compile time, run time, size of object code, etc).

Table 2.1 summarizes to the best of our knowledge important information related to the systems presented in Section 2.6. In general all systems implement only a subset of the full formal notation. Many factors dictate this restriction. Some are simple syntactic restrictions, as in DSL/LAMB, others are adopted in order to obtain a subset amenable to compilation, as CANTOR's and ACTRESS' action notations. The use of a programming language (as SCHEME in SPS) as a description language should be avoided. Regarding implementation languages, we notice that the majority of systems make use of LISP-like languages. The reason for this seems to be that in LISP programs can be manipulated as data, an obvious advantage in compiler generation systems. A great variation on the kind of object code emitted by generated compilers is noticed. A good choice points in the direction of C, which improves the portability of generated compilers and poses no trade-off on efficiency. The evaluator used has a close connection with the emitted object code. Considering the systems provided with a type checker, only SPS, TYPOL, GAG and ASF+SDF do some type analysis on the semantic descriptions. The others have some type checking only in the formal notation level. CANTOR and MIX are the only systems with a correctness proof.

Among the systems in Table 2.1 we consider only SIS, PSP, PERLUETTE, MESS, CANTOR and ACTRESS as systems designed to be a semantics-directed compiler generator[3]. The others are semantic prototyping or compiler writing tools.

Figure 2.10 shows the history (so far) of the development of compiler generator systems. It seems that most (all) compiler generation systems follow a similar development path. Firstly the organization of the system is designed, and a prototype version is built to check the ideas and principles which influenced the organization adopted. Secondly, analyses are built into the system which improve the performance of generated compilers. This latter phase is sometimes more demanding and fruitful than the previous one.

[3] Although SIS was not specifically designed to be a compiler generator, we have considered it as so.

System	Formalism	Description	Implementation	Object Code	Execution	TC	CP	CQ
SIS	denotational s.	DSL/LAMB	BCPL	LAMB	CBN β-reduction	no	no	1000
PSP	semantic grammars	semantic grammar	PASCAL	SECD code	SECD machine	no	no	1000
SPS	denotational s.	SCHEME	SCHEME/UNIX	SCHEME functions	VSM	yes	no	1000
TYPOL	natural s.	TYPOL	LELISP	PROLOG	PROLOG interpreter	yes	no	N/V
PERLUETTE	ADT	N/V	LISP	N/V	N/V	N/V	N/V	N/V
ASF+SDF	algebraic s.	ASF+SDF	LELISP	N/V	N/V	yes	no	N/V
MESS	high-level s.	N/A	PASCAL/SCHEME	POT	various	no	no	1
CANTOR	action s.	AN subset	PERL/SCHEME	PSEUDOSPARC	SPARC/HPmachine	yes	yes	100
GAG	attribute grammars	ALADIN	PASCAL	PASCAL	N/A	yes	no	N/A
MIX	N/A	N/A	MIXWELL	MIXWELL		N/A	yes	N/V
ACTRESS	action s.	AN subset	STANDARD ML	S	S machine	yes	no	40

TC – a typechecker for specifications and/or formal notation is available.

CP – a correctness proof for the compiler generator is provided.

CQ – object code quality based on target programs runs: object code/hand-written compilers' object code.

AN – action notation.

N/A – not applicable.

N/V – not available.

Table 2.1: Compiler generation systems information table.

1975	SIS (Denotational Semantics)	Mosses
1980	Perluette (Abstract Data Types)	Gaudel & Deschamp
1982	GAG (Attribute Grammars)	Kastens et al
1982	PSP (Semantic Grammars)	Paulson
1984	SPS (Denotational Semantics)	Wand
1984	Ceres (Partial Evaluation)	Jones & Tofte
1988	Mix (Partial Evaluation)	Jones et al
1988	Typol (Natural Semantics)	Depeyroux et al
1989	MESS (High-level Semantics)	Lee & Pleban
1991	ASF+SDF (Algebraic Semantics)	Klint et al
1991	Actress (Action Semantics)	Brown, Moura & Watt
1992	Cantor (Action Semantics)	Palsberg

Figure 2.10: Compiler generation systems through the years.

2.9 Problems

We describe here some of the problems with the approaches to semantics-directed generation. Basically, these problems are related to the semantic formalism used and the difficulties in implementing a conventional compiler from them.

Denotational Semantics

The simple and elegant approach of denotational semantics has unsuitable properties for compiler generation [65]. In the sequel we present the causes of this unsuitability.

Denotational semantic descriptions are written using the λ-*notation*, with some syntactic sugaring. Hence, compiler generation systems based on denotational semantics are forced to emulate a λ-calculus reduction machine. This leads to inefficiences at compile-time and run-time. At compile-time, the use of a partial evaluator to perform the reductions worsens the compile-time performance [65, 66]. At run-time, even the reduced λ-expressions involve the handling of numerous closures. These closures are constructed explicitly, from anonymous λ-abstractions, by the reducer. Many of these abstractions survive the compile-time reductions, leading to closures at run-time. Although some progress has been made toward the efficient implementation of closures [62], a more relevant source of inefficiency is caused by the modeling of frequently accessed data structures, such as environments and stores, using composition of λ-abstractions. For example, the retrieval of an element of the store might involve the application of a large number of these closures.

A second problem with denotational semantics involves the *lack of separability* between the actual semantics of a language and the model-dependent details underlying it. Mosses observed this problem and proposed the use of abstract semantic algebras, where the model details are eliminated from semantic descriptions [75, 76]. Mosses cites several problems with denotational semantics descriptions: the fundamental concepts embodied within a programming language are rarely reflected explicitly in the semantics; the use of λ-notation means that formal reasoning about programs usually requires a difficult manipulation of higher-order functions; denotational descriptions require extensive rewriting with the addition of new language features. The lack of separability of denotational descriptions also makes extremely difficult to generate compilers that produce efficient target programs [65].

The *blurring of semantic distinctions* is another unsuitable feature of denotational descriptions pointed by Lee in [65]. He gives an example of a denotational description where no distinction is made between variables and formal parameters. However, realistic compilers implement these variables in different ways. A solution to this problem [70] brings the necessity of complicated congruence proofs.

Lee explains that the problem just mentioned also leads to the *overspecification* of some aspects of the language semantics. For example, the lack of distinction between local, nonlocal, and formal parameter variables might be viewed as a requirement that all three classes of variables be implemented in the same way. Specification of evaluation orders is another example where overspecification can badly influence the compiler construction. We do not see those examples as requirements of the semantic description, but clearly they complicate the task of semantics-directed compiler generation.

The *distinction of static and dynamic components of a language* is fundamental for a real implementation of it. Denotational descriptions also blur this aspect of programming language specifications. This causes problems like how to discover what components of the semantics can be statically evaluated.

Code generation from denotational semantics is closely related to code generation for functional languages. It seems that progress in the area of implementation of functional languages can be fruitful to denotational semantics based compiler generation. However, the problem lies not so much in the efficient implementation of λ-notation, as in the difficulty in recovering (automatically) useful concepts and information from denotational descriptions which allows the generation of an efficient compiler.

Although the denotational semantics based compiler generation systems do not generate realistic compilers, they are important because they demonstrate the feasibility of the compiler generation process.

Operational Semantics

In traditional operational semantics, although the derivation of *a particular implementation* (interpreters in general) is very straight forward, each new description defines a new machine or language on which to base the semantics, and no *general* method to generate compilers exists.

Structural operational semantics and natural semantics descriptions are concise, elegant, and independent of any underlying implementation detail. However, as pointed in [69], the use of quantifiers and various non-standard conventions means that there is no distinct method of implementing either of the forms. This requires that a precise "style" of writing language descriptions is required, along with a precise intended interpretation.

Attribute Grammars

Attribute grammar techniques are often used to specify tools for language processing, rather than as a formalism for giving semantics to programming languages. Depending on the language, one needs a lot of auxiliary data types and attributes such as state of registers, memory allocation, etc. As stated in [37], the work of the compiler designer is made easier, but the method is far from the main principle of implementation of programming languages directly from their semantic definitions (see page 100 of [37]).

Partial Evaluation

As mentioned in [64], some of the open problems in partial evaluation are as follows: termination is hard to guarantee; speedup is hard to predict; there are no cost/benefit analysis, which would allow us to know if the benefit gained from expanding a computation is worth the increase in size; binding time improvements are not completely automatic; semantic faithfulness not always easy to maintain, particularly in *less pure* languages. As said in [39] a lot of work remains to be done in the application of partial evaluation to generate "real" compilers: the generated code should be nearer machine code and many analysis and optimization techniques present in a hand-written compiler should be applied.

2.10 An Ideal Semantics-Directed Compiler Generator

What would be the ideal semantics-directed compiler generation system? The first requirement for a true semantics-directed compiler generator is the presence of a general, independent and widely accepted semantic formalism for defining programming languages. The semantic formalism must not be a compiler specification language. A compiler specification language is often biased towards a particular system and very low level. For example, it addresses many implementation details which are of no main concern for the language designer.

Experience suggests that, due to the generality of semantic formalisms, some restrictions are usually made on the formal notation, such as consider only a subset of the formal notation, when implementing a compiler generator based on the semantic formalism. This is fully acceptable as long as the class of languages that could be specified is not drastically reduced.

In a semantics-directed system compilation and run time behaviour of generated compilers and their object programs must be guided by the programming language semantics. The only input to the system should be the semantic definition. The user does not need to provide any additional components of the generated compilers or run time environment. A compiler writer puts in a compiler his knowledge of the source programming language, the target language, the target machine, etc. This knowledge is built into the compiler code and is responsible for the compiler's performance. This knowledge is very difficult to mimic in a compiler generator system, but some analysis can be included which improves the system's performance.

The performance of a compiler can be defined basically by three factors: compilation time, execution time and size of object programs. The same applies to compiler generation systems: compiler generation time, compilation time and size of generated compilers, and the execution time and size of their object programs. At present, and also for the purposes of the present work, compiler generation time is not an important issue, but it can well be in the future. Assessment (benchmarks) should be made against the performance of hand-written compilers for a fixed language.

Designing a compiler generator which accepts the full notation of the semantic formalism is not a trivial task. The designer should expect that the user of his system knows only about the semantic formalism. So, if a dialect of the formal notation is the input to the compiler generator, it must be designed syntactically and semantically as close as possible to the original formal notation.

The ideal semantics-based compiler generator would be one whose only inputs are the syntactic and the semantic description of the source language, and whose generated compilers have performance in all respects equal to or better than hand-written compilers for the same source language.

Some of the systems described in Section 2.6 are not true semantic-based compiler generators. They allow for executable specifications, which is different from turning the specification into a compiler. They may be classified as semantic-based interpreter generators. They are very useful for rapid semantic prototyping and have contributed to important techniques for implementation of the semantic formalism they are based upon.

2.11 Synopsis

Semantics-directed compiler generation approaches and systems are still an area of research. All the described systems, although usable, are academic-purpose systems. There is a lot to be done to see the generated compilers competing with hand-crafted compilers, and for semantics-directed compiler generation systems to achieve the utility of tools like `lex` and `yacc`.

In Chapter 4 we will present ACTRESS, our action semantics based compiler generator, and Chapters 5–6 will show how some analyses were built into the system to improve overall performance. But first, in Chapter 3, we will cover action semantics, the semantic formalism chosen.

Chapter 3

Action Semantics

In my belief that a large acquaintance with particulars often makes us wiser than the mere possession of abstract formulas, however deep, I have ended this paper with some concrete examples, and I have chosen these among the extreme designs of programming languages. To some readers I may consequently seem, by the time they reach the end of the paper, to offer a caricature of the subject. Such convulsions of linguistic purity, they will say, are not sane. It is my belief, however, that there is much of value to be learnt from the study of extreme examples, not least, perhaps, that our view of sanity is rather easily influenced by our environment; and this, in the case of programming languages, is only too often narrowly confined to a single machine. My ambition in this and other related papers, mostly so far unwritten, is to develop an understanding of the mathematical ideas of programming languages and to combine them with other principles of common sense which serve as correctives of exaggeration, allowing the individual reader to draw as moderate conclusions as he will.

Christopher Strachey, 1973, in [100].

Action semantics was developed originally by Peter Mosses [72, 74, 75, 76] with David Watt's collaboration [108, 109, 111]. After a long period of design (and redesign) and experimentation with action notation (the formal notation used in action semantics) action semantics became a well-defined method for specifying programming languages [80, 111, 110, 77, 82, 81]. We give in this chapter an introduction to action semantics, the semantic formalism used in ACTRESS. We start by giving the motivations which influenced the creation and design of action semantics. Then we introduce informally the programming language SPECIMEN which is our running example throughout the thesis.

The structure of action semantic descriptions, action notation and data notation are explained. We illustrate its use with parts of an action semantics of SPECIMEN. The action notation covered here, a subset of *standard action notation* [80], is called ACTRESS *action notation*. We conclude the chapter by comparing ACTRESS and standard action notations. Experiences with action semantics and references to more detailed presentations are listed at the end.

3.1 Inspiration

Denotational semantics is a powerful method for specifying programming languages. λ-notation is the formal notation used in denotational semantic descriptions: denotations are higher-order functions written as λ-expressions. Well-known computational entities like bindings, storage, etc are also defined using λ-notation. The use of an abstract and mathematical notation for description of programming languages, which was the main source of inspiration in the design of denotational semantics, turned out to be inconvenient for the specification of real and full-scale programming languages. Denotational semantics lacks some very desirable pragmatic properties. In [113] we find a list of these properties a programming language specification method should have:

- *Readability.* It is very nice to be able to discover some properties of the language by simple inspection of its semantic description. Also, this property makes the description accessible to all people with interest in the language (designers, implementors and programmers).

- *Modularity.* It is very useful and productive to use parts of an existing description when describing a new programming language or extending an already existing one. Modularity in formal descriptions improves reusability and modifiability. It also helps in breaking large descriptions into smaller and manageable components.

- *Abstractness.* The formalism should be abstract enough to free the designer from biasing towards any implementation alternative and to focus on important design issues.

- *Comparability.* It should be easy to compare different languages by looking into their formal descriptions. We should be able to see, for example, that the constructs x

and y, although syntactically different, are equivalent.

- *Reasonability.* The formalism should facilitate reasoning about programs written in the defined language.

But why action semantics? It is very natural, when giving a denotational semantics of a programming language, to come up with a set of *auxiliary operations* which identifies common concepts such as lookups of identifiers in environments and allocation of cells. These operations (defined in terms of λ-notation) and their values constitute an *algebra* [97]. If these auxiliary operations operate on values that are denotations of phrases of the programming language being defined (not only on subsidiary objects such as environments, states, etc) they constitute a *semantic algebra* [76]. This leads then to the notion of an *abstract semantic algebra*, which is "a semantic algebra where the operations are specified axiomatically, by giving the (usually algebraic) laws that they satisfy, instead of defining them explicitly in terms of λ-notation" [76]. Thus, one can think of the denotation of phrases now as *actions*, which carry out some computations when performed. These actions are formed using well-defined primitive actions and action combinators. If these *action operators* are chosen carefully they will correspond to fundamental concepts of computation like sequencing, iteration, selection, etc. This is the spirit of action semantics!

Action semantics achieves well all the desirable properties cited above:

- *Readability.* The notation used is verbose and suggestive, which improves readability of semantic descriptions. The correspondence to well-known computational concepts also contributes to the readability of action semantics descriptions.

- *Modularity.* The semantics of a programming language is given in terms of a small number of standard primitive actions and action combinators. This provides the possibility of reusing parts of previous action semantic descriptions when describing new languages. The *polymorphic* behaviour of the action operators (regarding the different kind of information they process) allows their use to specify languages fundamentally different using the same set of operators. Language descriptions can be organized in modules.

- *Abstractness.* Action semantics is operational in flavour, but not implementation-biased.

- *Comparability.* The use of standard primitive actions and action combinators also facilitates the semantic comparison of languages.

- *Reasonability.* The standard primitive actions and action combinators satisfy nice algebraic properties that can be used to reason about programs and allow us to carry out useful *transformations*. This property of action semantics is very much explored in this thesis.

3.2 The Programming Language SPECIMEN

SPECIMEN is the largest of a group of four imperative languages used as a case study in our research on semantics-directed compiler generation. (The other three are NANOSPECIMEN, MICROSPECIMEN and MILLISPECIMEN.) It has two types of abstractions: procedures and functions, which can be recursive; functions are higher-order and free of side effects, although they can access non-local variables; integers, booleans, and arrays are provided. The complete action semantic description can be found in Appendix B. Although a simple programming language, SPECIMEN embodies many important concepts found in the real world of programming languages. Also we use the whole of our action notation subset in its definition, which makes it broad enough for our purposes.

3.3 Structure of Action Semantic Descriptions

Action semantics, like denotational semantics, is compositional: the semantics of each language phrase is determined by the semantics of its subphrases. The difference is that the denotations of phrases are no longer higher-order functions: they are *actions*. Thus, action semantics can be regarded as denotational, where the denotations are actions.

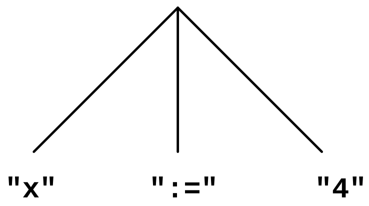
An action semantic description of a programming language is organized in three parts (modules): *abstract syntax*, *semantic entities* and *semantic functions*.

Abstract Syntax

The notation used for specify lexical, concrete, and abstract syntax is simply a form of *context-free grammar* plus *regular expressions*. For example, the abstract syntax for commands in SPECIMEN is defined as follows:

(1) Command = ⟦ Identifier “`:=`” Expression ⟧ |
⟦ Identifier “`[`” Expression “`]`” “`:=`” Expression ⟧ |
⟦ “`if`” Expression “`then`” Command ⟨ “else” Command ⟩$^?$ “end” ⟧ |
⟦ “`while`” Expression “do” Command “end” ⟧ |
⟦ “`call`” Identifier “`(`” ProcActuals “`)`” ⟧ |
⟦ “`local`” Declaration “in” Command “end” ⟧ |
⟨ Command ⟨ “;” Command ⟩* ⟩ .

The specification is a set of equations (we have shown only one equation). Each equation is similar to a group of productions in a BNF grammar: alternatives on the right hand side are separated by ‘|’. The last alternative illustrates the use of regular expressions: it specifies that a command can be a sequence of one or more commands separated by ‘`;`’. Also notice that terminal symbols are quoted, and their inclusion in the abstract syntax suggests what a concrete syntax could be. The occurrences of ‘⟦...⟧’ indicate the construction of nodes of a tree; for example, the abstract syntax tree for the command ‘x `:= 4`’ is ‘⟦x `:= 4`⟧’ and can be depicted as



In [80] (pages 50–51) a precise formal interpretation of the above grammar as an algebraic specification is given.

Semantic Entities

In this part we specify all semantic entities used in the semantic functions part. The standard semantic entities, primitive actions and action combinators, are already provided by the standard action notation and data notation (see Section 3.4). What we must specify here is the information to be processed by actions, as this can be very different depending on the programming language being defined; for example, an action might manipulate data of a sort defined in this part. This may involve extending data notation by defining new sorts of data, and specializing standard sorts, using *sort equations*[1]. We list below some equations found in the semantic entities part of SPECIMEN definition:

(1) primitive-value = truth-value | integer .

(2) value = primitive-value | function .

(3) cell = cell [truth-value] | cell [integer] .

(4) function = abstraction .

Sort equation (1) introduces the new sort 'primitive-value' in terms of the two standard data sorts 'truth-value' and 'integer'. '|' is the join (union) operator between sorts. The sort 'value' is specified in (2) as the join of a primitive value and the user-specified sort 'function'. An expression in SPECIMEN evaluates to an individual of sort 'value'. Sort equation (3) specializes the standard data sort 'cell' to the join of the two specified sort of cells: 'cell [truth-value]' and 'cell [integer]'. The first classifies all cells which can store an individual of sort 'truth-value'. The second classifies all cells which can store an individual of sort 'integer'. Sort equation (4) captures the fact that SPECIMEN functions are modelled as abstractions (see Section 3.4.6). The use of 'function' instead of 'abstraction' improves the readability of semantic equations; for example, a programmer consulting the semantic description would feel more confortable with 'function' than 'abstraction', although both are identical semantic entities.

Semantic Functions

Finally we specify a mapping between syntactic entities (trees) and semantic entities (actions) by means of semantic functions. Semantic functions are specified by semantic equations, in a form similar to denotational semantic descriptions. There should be one semantic equation for each form of programming language phrase. Each semantic function has only one argument.

For example, the semantic function for SPECIMEN commands is execute; the semantic equation for the assignment command is:

- execute _ :: Command → action .

[1] Also we usually define here some abbreviations for patterns that are frequently used in semantic equations. This helps to shorten semantic equations as well as to capture important concepts under the abbreviation name.

(1) execute ⟦ I:Identifier ":=" E:Expression ⟧ =
| evaluate E
then
| store the primitive-value in the cell bound to token-of I .

The first clause defines the *functionality* of execute: for every abstract syntax tree C for commands, 'execute C' is an action (semantic entity) which, when performed, expresses the behaviour of the command C. I and E are variables of (syntactic) sorts Identifier and Expression respectively. The vertical bars are used for grouping (parentheses can be used as well). 'evaluate E' is an application to an expression tree of the semantic function for SPECIMEN's expressions, and it gives an action as a result. One can read the semantic equation as plain English: *to assign the expression E to the identifier I, we first evaluate E then we store the resultant primitive value (an integer or a truth-value) in the cell bound to the identifier I!*

3.4 Action Notation

The formal notation used in action semantic descriptions is called *action notation*. In the following we present ACTRESS's action notation. We start by stressing the main concepts; we go on to explain some action notation constructs by given an informal account of each, its formal meaning and an example of its use in the context of SPECIMEN action semantics.

3.4.1 Concepts

Before we explore in detail each construct used in action notation, let us cover some basic and important concepts that underly the notation.

Actions

The main concept in action semantics is the concept of an *action*. An action is an entity that can be performed, receiving and producing data and/or changing storage. *Action combinators* combine actions into *compound actions*.

An *action performance* consists of a sequence of atomic steps made by a single *agent*. An agent can be thought as a processor or machine where the action is performed[2]. The

[2] In standard action notation, one can have more than one agent, which gives "true" concurrency on action performance.

performance of two sequences of steps (actions) can be: *sequenced*, when one sequence (action) is performed before the other; *interleaved*, when steps of both sequences (actions) are performed in an arbitrary order; or *exclusive*, when only one sequence (action) is performed. These concepts are clearly related with the control flow imposed by action combinators. Primitive actions are performed in a single step (one can see their performances as made in an *indivisible step*[3]).

A performance of an action (which may be part of an enclosing action) either[4]:

- *completes*, corresponding to normal termination; the performance of the enclosing action proceeds normally.
- *fails*, corresponding to abandoning the performance of an action. The enclosing action performs an alternative action, if there is one, otherwise it fails too.
- *diverges*, corresponding to nontermination. The enclosing action also diverges.

The syntax of ACTRESS action notation is shown in Figure 3.1. The semantics is specified using *semantic rules* (inference rules). The specification follows the natural semantic style [54]. The judgement

$$t, b, s \vdash a \rhd o, t', b', s'$$

states that the performance of action a with current transients t, current bindings b and current storage s, has the outcome status o, gives transients t', produces bindings b' and changes storage to s'. The outcome, as described informally above, can be *completed*, *diverged* or *failed*. We adopt the important convention that for any action a and *income* (t, b, s), if no semantic rule specifies otherwise, then

$$t, b, s \vdash a \rhd \mathit{failed}, \{\}, \{\}, s$$

holds. The triple (t, b, s), the action income, represents the *current information* available at the start of performance of an action. Transients t, a mapping between labels (nat-

[3] In fact, in standard action notation, the action 'indivisibly a' makes a to be performed as a single and indivisible step.

[4] In standard action notation, the performance of an action can also *escape*, corresponding to exceptional termination. The enclosing action is skipped until the escape is trapped.

$$
\begin{array}{lcll}
a & ::= & \mathsf{complete} \mid \mathsf{fail} \mid a_1\ \mathsf{and}\ a_2 \mid a_1\ \mathsf{and\ then}\ a_2 & (\mathit{basic}) \\
 & \mid & \mathsf{unfolding}\ a \mid \mathsf{unfold} \mid a_1\ \mathsf{or}\ a_2 & (\mathit{basic}) \\
 & \mid & \mathsf{give}\ y\ \mathsf{label}\ \#n \mid \mathsf{check}\ y \mid a_1\ \mathsf{then}\ a_2 & (\mathit{functional}) \\
 & \mid & \mathsf{bind}\ k\ \mathsf{to}\ y & (\mathit{declarative}) \\
 & \mid & \mathsf{furthermore}\ a \mid a_1\ \mathsf{hence}\ a_2 \mid a_1\ \mathsf{moreover}\ a_2 \mid a_1\ \mathsf{before}\ a_2 & (\mathit{declarative}) \\
 & \mid & \mathsf{store}\ y_1\ \mathsf{in}\ y_2 \mid \mathsf{deallocate}\ y & (\mathit{imperative}) \\
 & \mid & \mathsf{enact}\ y & (\mathit{reflective}) \\
 & \mid & a_1\ \mathsf{else}\ a_2 \mid \mathsf{recursively\ bind}\ k\ \mathsf{to}\ y \mid \mathsf{allocate\ a}\ s & (\mathit{hybrid}) \\
 & & & \\
y & ::= & \mathsf{the}\ s\#n & (\mathit{functional}) \\
 & \mid & \mathsf{the}\ s\ \mathsf{bound\ to}\ k & (\mathit{declarative}) \\
 & \mid & \mathsf{the}\ s\ \mathsf{stored\ in}\ y & (\mathit{imperative}) \\
 & \mid & y_1\ \mathsf{with}\ y_2 \mid \mathsf{closure}\ y \mid \mathsf{abstraction}\ a & (\mathit{reflective}) \\
 & \mid & y_1\ \mathsf{is}\ y_2 \mid \mathsf{if}\ y_1\ \mathsf{then}\ y_2\ \mathsf{else}\ y_3 \mid \mathit{op}\ (y_1, \ldots, y_n) & (\mathit{data\ operations}) \\
 & \mid & d & (\mathit{individual\ datum}) \\
 & & & \\
s & ::= & \mathsf{token} \mid \mathsf{bindable} & (\mathit{sorts}) \\
 & \mid & \mathsf{cell} \mid \mathsf{cell}\ [s] \mid \mathsf{storable} & \\
 & \mid & \mathsf{abstraction} & \\
 & \mid & \mathsf{datum} \mid \mathsf{distinct\text{-}datum} \mid \mathsf{integer} \mid \mathsf{truth\text{-}value} & \\
 & \mid & \mathsf{empty\text{-}list} \mid \mathsf{list} \mid \mathsf{list}\ [s] & \\
 & \mid & s\ \mathbf{|}\ s & \\
 & \mid & d &
\end{array}
$$

Figure 3.1: Syntax of Actress action notation.

ural numbers) and data, represent *transient information*, that is, data corresponding to intermediate results. Although we use the term "transients" for the mapping t, actually transients are the elements of the range of the mapping t. The labels are used to tag transients, so that one can refer to them later using the corresponding label. Bindings b, a mapping between tokens (strings) and data, represent *scoped information*, that is, bindings of tokens to data. Storage s, a mapping between cells and data, represents *stable information*, that is, data stored in cells. For example,

- $\{0 \mapsto \mathsf{true}, 3 \mapsto 34, 4 \mapsto \mathsf{cell0}\}$ are transients.
- $\{$"c" $\mapsto$ true, "dozen" $\mapsto$ 12, "x" $\mapsto$ cell2$\}$ are bindings.
- $\{\mathsf{cell0} \mapsto \mathsf{true}, \mathsf{cell1} \mapsto \mathsf{cell0}, \mathsf{cell2} \mapsto 9\}$ is a storage.

The various type of information give rise to the so-called *facets* of actions, according to the type of information the performance of an action affects:

- the *control facet*, in which the performance of an action is independent of any information.
- the *functional facet*, which affects transient information. Actions are *given* and *give* transients.
- the *declarative facet*, which affects scoped information. Actions *receive* and *produce* bindings.
- the *imperative facet*, which affects stable information. Actions *allocate* and *deallocate* cells of storage, and *change* the data stored in cells.

Sorts

Another important concept in action semantics is the concept of *sort*. A sort is a *choice* of individuals. An individual is an element of the sort. When one sort is a subsort of another, all individuals of the first sort are also individuals of the second sort. Sorts that are subsorts of each other are regarded as identical. The notation treats sorts themselves as abstract entities, and allows operations and relations to be applied to any arguments whatsoever — individuals, sorts, or even a mixture. In fact each individual is regarded itself as a sort that classifies just one entity. For example, one can write the following clauses:

(1) 1 : 1 .

(2) 1 : integer .

(3) $1 \leq$ integer .

(4) integer $\leq$ integer .

where, $d : s$ means d is an individual of sort s, and $s \leq t$ means that sort s is a subsort of sort t.

Sort symbols (such as 'integer') are treated as ordinary constants. The arguments and results of operations may be sorts as well as individuals. Operations are total, and monotone with respect to subsort inclusion; they may return the vacuous sort (nothing) to represent undefinedness. Moreover, sorts are treated as nondeterministic choices between the individuals that they classify; a singleton sort is thus treated as its unique individual. We will use the term *proper sort* to refer to a non-individual sort like 'integer' or 'truth-value'.

Yielders

Yielders are entities that can be evaluated to yield data during action performance. The result of the evaluation may depend on the current income to the action. The evaluation however does not affect the current information (transients, bindings and storage).

A constant is a special case of a yielder that always yields itself when evaluated.

The judgement

$$t, b, s \vdash y \rhd d$$

states that the yielder y yields (evaluates to) the individual datum d when evaluated in the presence of transients t, bindings b and storage s. The following important rules applies: for any yielder y and income (t, b, s), if no semantic rule specifies otherwise, then

$$t, b, s \vdash y \rhd \textsf{nothing}$$

holds. This is the only case where a yielder evaluates to a proper sort (the vacuous sort). For example, in the presence of income

$$\{0 \mapsto \mathsf{true}, 3 \mapsto 34, 4 \mapsto \mathsf{cell0}\}, \{\text{“c”} \mapsto \mathsf{true}, \text{“x”} \mapsto \mathsf{cell2}\}, \{\mathsf{cell0} \mapsto \mathsf{true}, \mathsf{cell2} \mapsto 5\}$$

- ‘the integer#3’ evaluates to ‘34’.
- ‘the integer#0’ evaluates to ‘nothing’.
- ‘the cell bound to “x”’ evaluates to ‘cell2’.
- ‘the integer stored in the cell bound to “x”’ evaluates to ‘5’.

Data

When performed, actions process individuals of data, selected from the current income, which is structured so access to individual items is possible.

An action term may contain a yielder composed by data constants and data operations. For example, in the presence of the income above,

- ‘successor (the integer#3)’ evaluates to ‘35’.
- ‘sum (the integer#0,8)’ evaluates to ‘nothing’.
- ‘not (the truth-value bound to “c”)’ evaluates to ‘false’.
- ‘difference (4,3)’ evaluates to ‘1’.

Abstractions (see Section 3.4.6) are considered as data. Many well-known mathematical entities like integers, truth-values, and lists are also available. Section 3.5 explains in more detail *data notation*.

3.4.2 Basic

Basic actions are concerned with *flow of control*.

The primitive action ‘complete’ is an indivisible action that always completes. It corresponds to a null action. It gives no transients; it produces no bindings; it does not change the storage. Rule 3.4 of Figure 3.2 specifies the behaviour of the ‘complete’ action.

The action ‘a_1 and then a_2’ performs a_1 followed by a_2. However, a_2 is only performed if a_1 completes. It corresponds to sequential performance. Both subactions are given the same transients and bindings as the compound action. When the subactions complete, the compound action gives all the transients that both subactions give (assuming the data are labelled disjointly) and produces all the bindings that both subactions produce (assuming the tokens are bound disjointly).

Rules 3.9, 3.10 and 3.11 of Figure 3.2 give the meaning of the '**and then**' combinator. The first rule applies when the performance of both subactions complete. The income to a_2 is equal to the income to a_1 except for storage which is the one after a_1 performance. This also implies the sequentiality of the performance: a_1 is performed before a_2. The *mergeable* predicate insists that the domain of its arguments (mappings) are disjoint. The merge operation ($\oplus$) returns the union of two mergeable mappings:

$$\{\} \oplus m_2 = m_2 \tag{3.1}$$

$$m_1 \oplus \{\} = m_1 \tag{3.2}$$

$$((k_1 \mapsto d_1) \cdot m_1) \oplus m_2 = (k_1 \mapsto d_1) \cdot (m_1 \oplus m_2),\ k_1 \notin dom\ m_2 \tag{3.3}$$

The second rule applies when the performance of a_1 fails or diverges. Finally, the third rule applies when a_1 completes and a_2 fails or diverges.

In ACTRESS action notation, 'a_1 **and** a_2' is semantically identical to 'a_1 **and then** a_2'[5].

Example 3.1. The semantic equation for execution of a sequence of commands in SPECIMEN is

(1) execute ⟦ C_1:Command ";" C_2:Command ⟧ = execute C_1 and then execute C_2 .

which precisely determines in which order C_1 and C_2 are performed. □

'**unfolding** a' is an action that performs a (the *unfolded* action), but whenever the dummy action '**unfold**' is reached, a is performed in place of '**unfold**'. It corresponds to iteration. Rule 3.12 gives the meaning of the '**unfolding**' action. 'a [**unfolding** a/**unfold**]' means replacing all free occurrences of '**unfold**' in a by '**unfolding** a'.

The compound action 'a_1 **or** a_2' is performed as follows: a_1 is performed. If the

[5] In standard action notation 'a_1 and a_2' is an action that performs both its subactions, with *arbitrary interleaving* of their indivisible subactions. It corresponds to arbitrary order of evaluation; an escape or failure causes any remaining parts of the subactions to be skipped. 'a_1 and then a_2' is a specialization of 'a_1 and a_2' when a_1 is completely performed before a_2.

(COMPLETE)

$$t, b, s \vdash \textsf{complete} \rhd \mathit{completed}, \{\}, \{\}, s \tag{3.4}$$

(FAIL)

$$t, b, s \vdash \textsf{fail} \rhd \mathit{failed}, \{\}, \{\}, s \tag{3.5}$$

(AND)

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \quad t, b, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \quad \mathit{mergeable}\ t_1\ t_2 \quad \mathit{mergeable}\ b_1\ b_2}{t, b, s \vdash a_1 \ \textsf{and}\ \ a_2 \rhd \mathit{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \tag{3.6}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \quad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1 \ \textsf{and}\ \ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.7}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \quad t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \quad o_2 \neq \mathit{completed}}{t, b, s \vdash a_1 \ \textsf{and}\ \ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.8}$$

(AND THEN)

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \quad t, b, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \quad \mathit{mergeable}\ t_1\ t_2 \quad \mathit{mergeable}\ b_1\ b_2}{t, b, s \vdash a_1 \ \textsf{and then}\ a_2 \rhd \mathit{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \tag{3.9}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \quad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1 \ \textsf{and then}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.10}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \quad t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \quad o_2 \neq \mathit{completed}}{t, b, s \vdash a_1 \ \textsf{and then}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.11}$$

(UNFOLDING)

$$\frac{t, b, s \vdash a\ [\textsf{unfolding}\ a/\textsf{unfold}] \rhd o', t', b', s'}{t, b, s \vdash \textsf{unfolding}\ a \rhd o', t', b', s'} \tag{3.12}$$

(OR)

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \quad o_1 \neq \mathit{failed}}{t, b, s \vdash a_1 \ \textsf{or}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.13}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{failed}, \{\}, \{\}, s_1 \quad t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2}{t, b, s \vdash a_1 \ \textsf{or}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.14}$$

Figure 3.2: Semantic rules for basic action notation.

performance of a_1 fails, a_2 is performed instead. The performance of each subaction has the same transients and bindings as the compound action. Storage for a_2 is the one after a_1 failed[6].

Example 3.2. We could use 'unfolding' and 'or' to specify the semantics of SPECIMEN's while command as follows:

execute ⟦ "while" E:Expression "do" C:Command "end" ⟧ =
  unfolding
  | | evaluate E
  | then
  | | | check (the truth-value is true) and then execute C and then unfold
  | | or
  | | | check (the truth-value is false) .

The evaluation of E gives a truth-value, which is passed to the 'or' action. If this truth-value is true than the first branch of the 'or' is performed, causing 'execute C' to be performed, followed by 'unfold'. Otherwise the second branch is performed and the while command terminates (the 'check' action, explained in the next section, guarantees this). Iteration is achieved by replacing the 'unfold' by the 'unfolding' action and performing the latter. □

Example 3.3. A common use of the 'or' combinator in action semantic descriptions is in the denotation of identifiers in expressions (r-value). For SPECIMEN we define:

evaluate I:Identifier =
  | give the value bound to token-of I
  or
  | give the primitive-value stored in the cell bound to token-of I .

If I is a constant or function identifier (that is, I is bound to a value), then the first subaction is performed and it will complete. On the other hand, if I is a variable identifier (that is, I is bound to a cell), the first subaction is performed and fails because a cell is not a value (an individual cell is not a subsort of value). This failure causes the second subaction to be performed and the result will be the primitive value stored in the cell. □

[6] In standard action notation the performance of each subaction has the same income as the compound action.

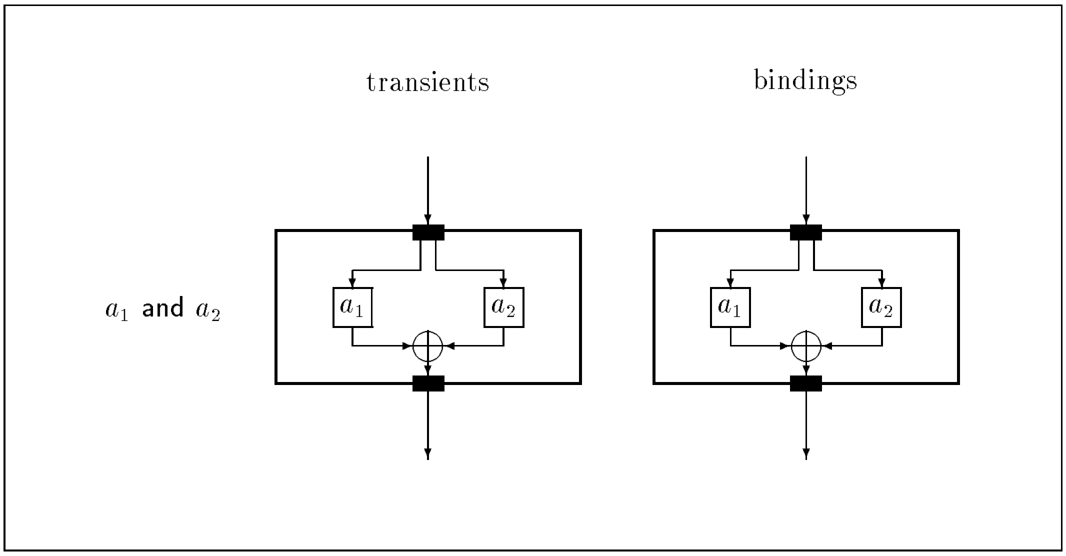


Figure 3.3: Action diagram for the 'and' combinator.

We will give a pictorial view of the flow of transients and bindings for some action combinators using action diagrams. Figure 3.3 shows an action diagram for the 'and' combinator. The symbol $\oplus$ represents the *merge* operation found in the semantic rules.

3.4.3 Functional

Functional actions are concerned with the processing of transient information. A functional action may provide some transient information, such as a particular labelled datum. Then we say that the action *gives* the datum. This datum may be an explicit operand of the action, or it may be the result of evaluating some yielder that refers to the current information (transient or other), or it may be the result of some data operation. The transients current at the start of an action are *given* to the action.

The functional action 'give y label $\#n$', where y is a yielder and n is a natural number, gives the individual datum yielded by y, labelled with n. It completes if y yields an individual datum, or fails if y yields nothing. Rule 3.15 states this formally. Notice that, according to our convention, the non-existence of a rule for the case where y yields nothing implies that the action fails in this case (see Section 3.4.1). 'give y' is an abbreviation for 'give y label $\#0$'.

The action 'check y', where y is a truth-value yielder, completes if y yields true, and fails if y yields false (or anything else). It can be used as a guard that a condition is true.

(GIVE)

$$\frac{t, b, s \vdash y \rhd d}{t, b, s \vdash \textsf{give } y \textsf{ label } \#n \rhd \mathit{completed}, \{n \mapsto d\}, \{\}, s} \tag{3.15}$$

(CHECK)

$$\frac{t, b, s \vdash y \rhd \textsf{true}}{t, b, s \vdash \textsf{check } y \rhd \mathit{completed}, \{\}, \{\}, s} \tag{3.16}$$

$$\frac{t, b, s \vdash y \rhd \textsf{false}}{t, b, s \vdash \textsf{check } y \rhd \mathit{failed}, \{\}, \{\}, s} \tag{3.17}$$

(THEN)

$$\frac{\begin{array}{c} t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t_1, b, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \\ \mathit{mergeable}\ b_1\ b_2 \end{array}}{t, b, s \vdash a_1 \textsf{ then } a_2 \rhd \mathit{completed}, t_2, b_1 \oplus b_2, s_2} \tag{3.18}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1 \textsf{ then } a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.19}$$

$$\frac{\begin{array}{c} t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t_1, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \\ o_2 \neq \mathit{completed} \end{array}}{t, b, s \vdash a_1 \textsf{ then } a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.20}$$

(THE)

$$\frac{d : r}{(n \mapsto d) \cdot t, b, s \vdash \textsf{the } r\#n \rhd d} \tag{3.21}$$

Figure 3.4: Semantic rules for functional action notation.

For example, the action

| check the truth-value and then give 0
or
| check not (the truth-value) and then give 1

gives 0 if true is given to it, or gives 1 otherwise.

'then' is the only functional action combinator: 'a_1 then a_2' is an action that performs a_1 followed by a_2. However, a_2 is only performed if a_1 completes. It corresponds to functional composition: a_1 is given the same transients as the compound action; if a_1 completes, a_2 is given the transients given by a_1; the compound action gives only the transients given by a_2. Both subactions receive the same bindings as the compound action. When the subactions complete, the compound action produces all the bindings that both subactions produce (assuming the bindings are for disjoint sets of tokens). This is stated formally in Figure 3.4.

'the $s\#n$' is a functional yielder, which yields the datum labelled with the label n in the current transients, restricted to the sort s. 'the s' is equivalent to 'the $s\#0$'. The yielder 'it' is an abbreviation for 'the datum#0'.

Example 3.4. Given the transients $\{0 \mapsto \text{false}, 3 \mapsto 42, 4 \mapsto \text{true}\}$:

- the action 'give the truth-value' gives the transient $\{0 \mapsto \text{false}\}$;
- the action 'give the integer#3 label #4' gives the transient $\{4 \mapsto 42\}$;
- the action 'give the integer' would fail.

Notice that the last action fails because of the restriction on the sort 'integer': the datum labelled 0 (false) is not an individual of sort 'integer'. □

Example 3.5. We give below an example of the semantics for SPECIMEN's if-then-else expression:

evaluate ⟦ "`if`" E_1:Expression "`then`" E_2:Expression "`else`" E_3:Expression "`end`" ⟧ =
| evaluate E_1
then
| | check (it is true) and then evaluate E_2
| or
| | check (it is false) and then evaluate E_3 .

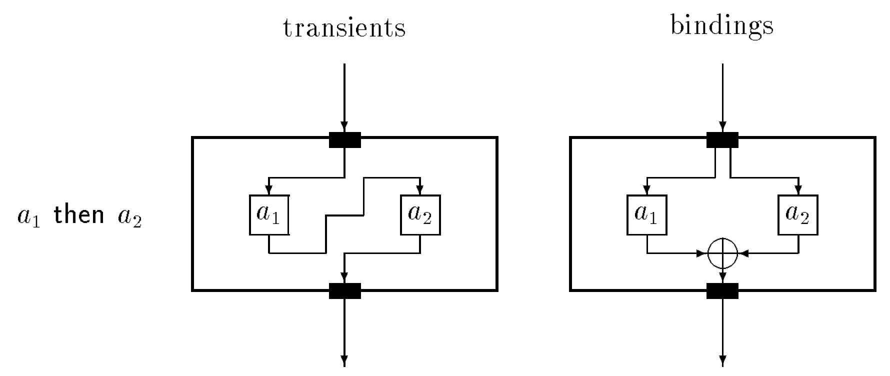


Figure 3.5: Action diagram for the '**then**' combinator.

We first evaluate the expression E_1, *then* pass its result to the second subaction of '**then**'. The '**check**' actions act like guards: if we choose to perform the first subaction of '**or**' and E_1 evaluated to true, then the '**check (it is true)**' action completes *and then* we evaluate E_2; if E_1 evaluated to false then '**check (it is true)**' fails and the second subaction of '**or**' will be performed. A similar interpretation applies if we choose to perform the second subaction of '**or**' first. □

Figure 3.5 shows the action diagram for the '**then**' combinator.

3.4.4 Declarative

Declarative actions produce bindings between tokens (usually identifiers) and data. The bindings produced by an action may depend on the current income to the action. The tokens in produced bindings however, may depend only on the income bindings and on the action itself. The declarative facet of actions is somewhat similar to the functional facet, with tokens used instead of numerical labels. The difference between the two facets lies in the fact that the bindings are scoped.

'**bind** k **to** y', the '**bind**' action, is an indivisible action that produces the binding of the token k to the datum yielded by y. Figure 3.6 shows the semantic rules for the '**bind**' action. The second antecedent insists that the datum yielded by y must be of sort **bindable**.

This sort classifies which individuals can be bound in a particular language.

"**furthermore** a" produces all the bindings received by itself, overlayed by the bindings produced by a. It can be used, for example, for specifying local declarations: the bindings established by the local declarations overlay global bindings for the same tokens. Note that it gives the transients given by a. Given mappings m and m', $m \oslash m'$ means the map obtained by overlaying map m on to map m':

$$\{\} \oslash m_2 = m_2 \tag{3.35}$$

$$m_1 \oslash \{\} = m_1 \tag{3.36}$$

$$((k \mapsto d_1) \cdot m_1) \oslash ((k \mapsto d_2) \cdot m_2) = (k \mapsto d_1) \cdot (m_1 \oslash m_2) \tag{3.37}$$

$$((k \mapsto d_1) \cdot m_1) \oslash m_2 = (k \mapsto d_1) \cdot (m_1 \oslash m_2), \text{ if } k \notin dom\ m_2 \tag{3.38}$$

'a_1 **hence** a_2' is an action that performs a_1 followed by a_2. a_2 is only performed if a_1 completes. The bindings received by the compound action are received by a_1. a_2 receives the bindings produced by a_1. The bindings produced by the compound action are the bindings produced by a_2. The functional facet is as for 'a_1 **and** a_2'.

'a_1 **moreover** a_2' is an action that performs a_1 followed by a_2. a_2 is only performed if a_1 completes. It corresponds to letting bindings produced by a_2 override those produced by a_1. Both subactions receive the bindings received by the compound action. The functional facet is as for 'a_1 **and** a_2'.

The yielder '**the** s **bound to** k' yields the datum to which the token k is bound in the current bindings, restricted to the sort s. It yields '**nothing**' if k is not bound.

Example 3.6. Given the bindings {**x** $\mapsto$ **false**, **y** $\mapsto$ **42**, **z** $\mapsto$ **true**}:

- the action '**give the truth-value bound to "x"**' give the transient {**0** $\mapsto$ **false**};
- the action '**give the truth-value bound to "y"**' fails.
- '**give the integer bound to "f"**' fails, because there is no binding for token "**f**".

□

The sort '**bindable**' classifies individuals which can be bound by the '**bind**' action. It is a subsort of '**datum**' and is specialized by the user of action notation. For example, for

(BIND)

$$\frac{t, b, s \vdash y \rhd d \qquad d : \mathsf{bindable}}{t, b, s \vdash \mathsf{bind}\ k\ \mathsf{to}\ y \rhd \mathit{completed}, \{\}, \{k \mapsto d\}, s} \tag{3.22}$$

(FURTHERMORE)

$$\frac{t, b, s \vdash a \rhd \mathit{completed}, t', b', s'}{t, b, s \vdash \mathsf{furthermore}\ a \rhd \mathit{completed}, t', b' \oslash b, s'} \tag{3.23}$$

$$\frac{t, b, s \vdash a \rhd o', t', b', s' \qquad o' \neq \mathit{completed}}{t, b, s \vdash \mathsf{furthermore}\ a \rhd o', t', b', s'} \tag{3.24}$$

(HENCE)

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b_1, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \qquad \mathit{mergeable}\ t_1\ t_2}{t, b, s \vdash a_1\ \mathsf{hence}\ a_2 \rhd \mathit{completed}, t_1 \oplus t_2, b_2, s_2} \tag{3.25}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{hence}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.26}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b_1, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \qquad o_2 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{hence}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.27}$$

(MOREOVER)

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \qquad \mathit{mergeable}\ t_1\ t_2}{t, b, s \vdash a_1\ \mathsf{moreover}\ a_2 \rhd \mathit{completed}, t_1 \oplus t_2, b_2 \oslash b_1, s_2} \tag{3.28}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{moreover}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.29}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \qquad o_2 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{moreover}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.30}$$

(BEFORE)

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b_1 \oslash b, s_1 \vdash a_2 \rhd \mathit{completed}, t_2, b_2, s_2 \qquad \mathit{mergeable}\ t_1\ t_2}{t, b, s \vdash a_1\ \mathsf{before}\ a_2 \rhd \mathit{completed}, t_1 \oplus t_2, b_2 \oslash b_1, s_2} \tag{3.31}$$

$$\frac{t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{before}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.32}$$

$$\frac{t, b, s \vdash a_1 \rhd \mathit{completed}, t_1, b_1, s_1 \qquad t, b_1 \oslash b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \qquad o_2 \neq \mathit{completed}}{t, b, s \vdash a_1\ \mathsf{before}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.33}$$

(BOUND)

$$\frac{d : r}{t, (k \mapsto d) \cdot b, s \vdash \mathsf{the}\ r\ \mathsf{bound\ to}\ k \rhd d} \tag{3.34}$$

Figure 3.6: Declarative action notation.

SPECIMEN, we define

$$\text{bindable} = \text{value} \mid \text{procedure} \mid \text{cell} \mid \text{array}$$

Example 3.7. The semantic equation below specifies how a constant declaration is elaborated in SPECIMEN.

elaborate ⟦ "`const`" I:Identifier ":" T:PrimitiveType "=" E:Expression ⟧ =
  | evaluate E
  then
  | bind token-of I to the primitive-value .

The expression E is evaluated first, and then the primitive value (a truth-value or an integer) resulting from this evaluation is bound to identifier I. □

Example 3.8. SPECIMEN block command execution is specified as follows:

execute ⟦ "`local`" D:Declaration "`in`" C:Command "`end`" ⟧ =
  | furthermore elaborate D
  hence
  | execute C .

The declaration D is elaborated first. The bindings received by the block command, overlayed by the bindings produced by the elaboration of D, are produced to the execution of C. □

Figure 3.7 shows the action diagram for 'moreover'. The symbol used to join the flow of bindings out of each subaction means that the bindings which come from a_2 overlay the bindings which come from a_1.

3.4.5 Imperative

Action notation adopts a simple model of storage: storage is organized as a collection of primitive *cells*, each of which may contain a single datum. A cell has to be *allocated* before it can be used to store data. The datum stored in a cell remains the same until a new datum is stored in the cell — or until the cell is deallocated. Imperative actions may inspect the current storage; they may also make *changes* to the storage by allocating and/or deallocating cells and/or by storing data in allocated cells.

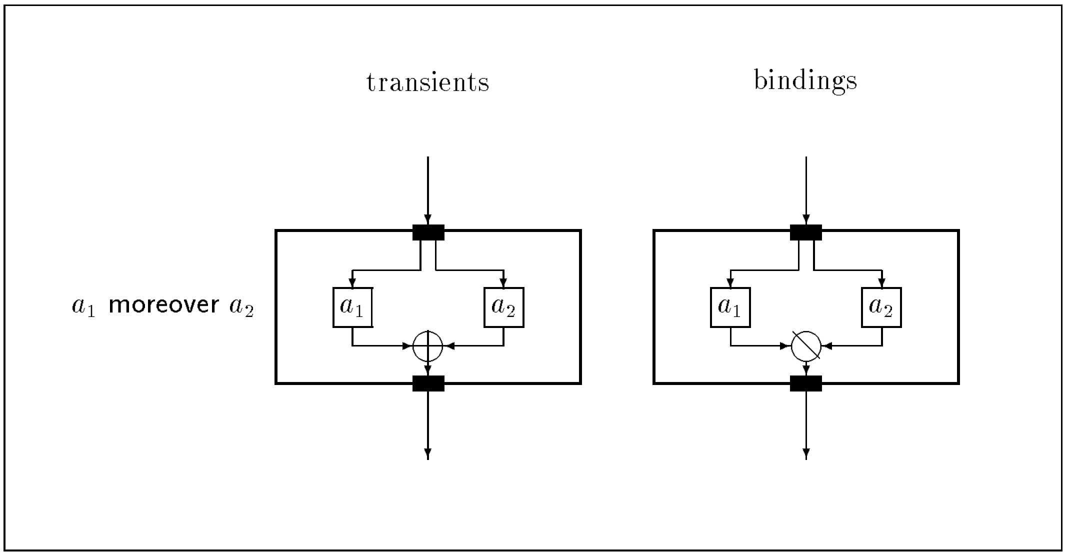


Figure 3.7: Action diagram for the '**moreover**' combinator.

(STORE)

$$\frac{t, b, s \vdash y_1 \rhd d \qquad t, b, s \vdash y_2 \rhd c \qquad s = (c \mapsto d_0) \cdot s' \qquad c : \mathsf{cell}\ [r] \qquad d : r}{t, b, s \vdash \mathsf{store}\ y_1\ \mathsf{in}\ y_2 \rhd \mathit{completed}, \{\}, \{\}, \{c \mapsto d\} \oslash s} \tag{3.39}$$

(DEALLOCATE)

$$\frac{t, b, s \vdash y \rhd c \qquad s = (c \mapsto d) \cdot s'}{t, b, s \vdash \mathsf{deallocate}\ y \rhd \mathit{completed}, \{\}, \{\}, s'} \tag{3.40}$$

(STORED)

$$\frac{t, b, s \vdash y \rhd c \qquad s = (c \mapsto d) \cdot s' \qquad d : r}{t, b, s \vdash \mathsf{the}\ r\ \mathsf{stored\ in}\ y \rhd d} \tag{3.41}$$

Figure 3.8: Semantic rules for imperative action notation.

The action '**store** y_1 **in** y_2' is an indivisible action that stores the datum yielded by y_1 in the cell yielded by y_2, failing if the cell is not allocated.

The primitive action '**deallocate** y' deallocates the cell yielded by y. (We look into the allocate action in Section 3.4.7.)

Storage lookups can be done using the yielder '**the** s **stored in** y'. It evaluates to the datum stored in the cell yielded by y, which must be of sort s. Notice (Figure 3.8) that the cell must be previous allocated.

Figure 3.8 gives the formal semantics for imperative action notation. Notice that we insist that an individual of sort s can only be stored in an individual cell of sort '**cell** [s]'

(the sort '**cell** [s]' classifies all cells which can be used to store individuals of sort s[7]).

The sort '**storable**' classifies all sorts of data that can be stored, and should be specified in the language definition. For example, for SPECIMEN we have specified:

storable = primitive-value

Actions and yielders are never storable! Notice however that we can indirectly store actions, using the '**abstraction**' operation (see Section 3.4.6).

Imperative actions are *committed actions*. This requirement is made to accommodate *single-threadedness* of storage, which guarantees that we need only a *single* copy of storage. This is because, for any two subactions, a_1 and a_2, if we perform a_2 after a_1, a_2 always uses the storage after a_1's performance, that is, there is no dependency of a_2 on the state of the storage before a_1's performance.

In standard action notation, the need for committed actions arises because of the '**or**' combinator. If the chosen alternative fails, the other alternative must be performed with the same initial storage. If the storage has been changed, the alternative action cannot be performed at all.

Example 3.9. The semantic equation for SPECIMEN's assignment command is:

```
execute [[ I:Identifier ":=" E:Expression ]] =
    | evaluate E
    then
    | store the primitive-value in the cell bound to token-of I .
```

That is, the expression E is evaluated and its result, a primitive value, is stored in the cell bound to identifier I. □

3.4.6 Reflective

Abstractions are a special kind of data that incorporate actions; an abstraction is formed by the '**abstraction**' operation. Abstractions are classified as data, although actions themselves are not. The *incorporated action* is performed when the abstraction is *enacted*: '**enact** y' is an action that performs the action incorporated in the abstraction yielded by evaluating

[7] Standard action notation does not have this operation as it would be non monotonic with respect to sort inclusion. In the ACTRESS context, however, this causes no practical difficulty.

(ENACT)

$$\frac{t, b, s \vdash y \rhd abstraction(a, t_0, b_0) \qquad t_0, b_0, s \vdash a \rhd o', t', b', s'}{t, b, s \vdash \textsf{enact } y \rhd o', t', b', s'} \tag{3.42}$$

(ABSTRACTION)

$$t, b, s \vdash \textsf{abstraction } a \rhd abstraction(a, \{\}, \{\}) \tag{3.43}$$

(WITH)

$$\frac{t, b, s \vdash y_1 \rhd abstraction(a, \{\}, b_0) \qquad t, b, s \vdash y_2 \rhd d}{t, b, s \vdash y_1 \textsf{ with } y_2 \rhd abstraction(a, \{0 \mapsto d\}, b_0)} \tag{3.44}$$

$$\frac{t, b, s \vdash y_1 \rhd abstraction(a, t_0, b_0) \qquad t, b, s \vdash y_2 \rhd d \qquad t_0 \neq \{\}}{t, b, s \vdash y_1 \textsf{ with } y_2 \rhd abstraction(a, t_0, b_0)} \tag{3.45}$$

(CLOSURE)

$$\frac{t, b, s \vdash y \rhd abstraction(a, t_0, \{\})}{t, b, s \vdash \textsf{closure } y \rhd abstraction(a, t_0, b)} \tag{3.46}$$

$$\frac{t, b, s \vdash y \rhd abstraction(a, t_0, b_0) \qquad b_0 \neq \{\}}{t, b, s \vdash \textsf{closure } y \rhd abstraction(a, t_0, b_0)} \tag{3.47}$$

Figure 3.9: Semantic rules for reflective action notation.

y. A single item of data can be supplied to an abstraction using the operation '**with**', and bindings by using the operation '**closure**'.

Figure 3.9 gives the rules for the above operations. '**enact** y' gives the transients, produces the bindings and makes change to the storage that the incorporated action of the abstraction yielded by y does. Notice that (rules 3.45 and 3.47) the 'with' and 'closure' operations have no effect when a datum and bindings respectively were previously supplied to the abstraction. '$abstraction(a, t, b)$' is the semantic value of an abstraction. It can be thought as a triple formed by the incorporated action, and transients and bindings to be provided to the incorporated action when the abstraction is enacted.

Example 3.10. Function declaration and application in SPECIMEN are defined below:

```
elaborate [[ "fun" I:Identifier "(" FS:FunFormals ")" ":" T:ValueType "="
             E:Expression ]] =
    recursively bind token-of I to closure abstraction
        | | furthermore elaborate-fun-formals FS
        | hence
        | | evaluate E then give the fun-result .
```

evaluate ⟦ I:Identifier "(" AS:FunActuals ")" ⟧ =
| evaluate-fun-actuals AS
then
| enact (the function bound to token-of I with the fun-argument-list) .

A function declaration binds the function identifier to an abstraction. We *close* the abstraction to ensure that all bindings received by the incorporated action are the ones current at binding time. The incorporated action binds the formal parameters to the arguments (which will be provided at enaction time) and evaluates the body of the function (E). For a function application, we first evaluate the actual parameters to give an argument list, and then perform (enact) the action incorporated in the abstraction bound to the function identifier I. The action is performed having the argument list as its current transient (or the argument list is given to the incorporated action). □

3.4.7 Hybrid

Hybrid actions are characterized by processing more than one kind of information (multifaceted actions), or are useful abbreviations. Figure 3.10 shows the formal specification of the hybrid actions.

'a_1 else a_2' performs either a_1 or a_2 depending on the truth-value labelled 0 in the transients given to it: if it is true a_1 is performed, if it is false a_2 is performed. 'a_1 else a_2' is an abbreviation for '(check (it is true) then a_1) or (check (it is false) then a_2)'.

'recursively bind k to y' produces a recursive binding. The evaluation of y can refer to the binding produced by the action itself. y must evaluate to an abstraction.

'allocate s' is an action that chooses an arbitrary free cell of sort s, stores the special datum 'uninitialized' in it, and gives the cell (thus it involves both the functional facet and the imperative facet). The sort s must be a subsort of cell.

Example 3.11. Given storage {cell0 $\mapsto$ false, cell2 $\mapsto$ 42}:

- the performance of 'allocate a cell' changes storage to {cell0 $\mapsto$ false, cell2 $\mapsto$ 42, cell4 $\mapsto$ uninitialized};
- the performance of 'allocate a cell then store 9 in the cell' changes storage to {cell0 $\mapsto$ false, cell2 $\mapsto$ 42, cell1 $\mapsto$ 9};

(ELSE)

$$\frac{\{\}, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1}{(0 \mapsto \mathsf{true}) \cdot t, b, s \vdash a_1 \ \mathsf{else}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{3.48}$$

$$\frac{\{\}, b, s \vdash a_2 \rhd o_2, t_2, b_2, s_2}{(0 \mapsto \mathsf{false}) \cdot t, b, s \vdash a_1 \ \mathsf{else}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{3.49}$$

(RECURSIVELY BIND)

$$\frac{t, \{k \mapsto d\} \oslash b, s \vdash y \rhd d \qquad d : \mathsf{abstraction}}{t, b, s \vdash \mathsf{recursively\ bind}\ k\ \mathsf{to}\ y \rhd \mathit{completed}, \{\}, \{k \mapsto d\}, s} \tag{3.50}$$

(ALLOCATE)

$$\frac{c : r \leq \mathsf{cell} \qquad c \notin \mathit{dom}\ s}{t, b, s \vdash \mathsf{allocate\ a}\ r \rhd \mathit{completed}, \{0 \mapsto c\}, \{\}, \{c \mapsto \mathsf{uninitialized}\} \oplus s} \tag{3.51}$$

Figure 3.10: Semantic rules for hybrid action notation.

- the performance of '**allocate a cell then (store 4 in the cell and then deallocate the cell)**' changes storage to $\{\mathsf{cell0} \mapsto \mathsf{false}, \mathsf{cell2} \mapsto 42\}$.

□

3.5 Data Notation

Finally, besides the computational entities covered in the previous sections (bindings, storage, abstractions, etc), action notation provides various familiar mathematical entities through its *data notation*. ACTRESS' data notation provides constants and operations for truth-values, integers and lists. They are specified algebraically in [80]. They are total, giving '**nothing**' for non-defined or ill-sorted arguments. Figure 3.11 summarises ACTRESS' data notation for lists.

3.6 Standard and ACTRESS Action Notations

Standard action notation is a very general and rich formal language. We explain and justify here the choices we had to make when we restricted standard action notation to our subset. This restriction was a structured way to approach the problem of generating

(1) empty-list : list

(2) list (_) :: datum → list

(3) concatenation (_,_) :: list, list → list

(4) head-of (_) :: list → datum

(5) tail-of (_) :: list → list

(6) length-of (_) :: list → integer

Figure 3.11: Data notation for lists.

compilers from action semantic descriptions, and it gave us a manageable subset. Future extensions towards standard action notation are possible (see Chapter 7).

To start with, communicative actions (actions *send* and *receive* messages, and *subcontract* tasks to other *agents*, processing *permanent* information) were left out of our subset. This part of action notation, used to describe concurrent aspects found in some programming languages, like processes, co-routines, tasks, etc, is not used in the description of the class of languages we have in mind at present.

Action semantic descriptions are modular, and a *meta-notation* for managing modules and its values does exist in standard action notation. For example, in Appendix B modules are sections, submodules are subsections, and so on; modules can be included in other modules, and precise rules of visibility exist. We did not discuss the semantics of modules because they are not important in the scope of this thesis. For a description of modules and their semantics see [80, 14].

General differences between standard action notation and ACTRESS action notation are given below:

- **Action performance.** Non-determinism, parallelism and interleaving of action performance were left out. Our 'or' action is deterministic. There is a single action performer, that is, at a particular time only a specific action a is being performed; for all actions 'a_1 *comb* a_2', where *comb* is an action combinator, the performance of each subaction is indivisible, that is, each subaction is performed in its totality before the performance of the other subaction starts. For example, the performance of a_1 and a_2 in 'a_1 and a_2' is not interleaved: we perform a_1 before starting the performance of a_2.

- **Typed subset.** Our subset is typed. We ruled out many actions that are perfectly acceptable in standard action notation. For example, we insist that the domain of transients and bindings given by a_1 and a_2 in the action 'a_1 or a_2' are the same. However we did not insist on a statically typed subset.

- **Transients.** In our subset transients are mappings between labels (natural numbers) and data. In standard action notation they are tuples[8]. Some properties are changed with this (for example, our 'and' is commutative), and the 'give' action has a different syntax in ACTRESS action notation. Also the standard action notation operation 'application _ to _' (which corresponds to 'with' in our subset) can give a tuple of datum to an incorporated action, instead of only a single individual datum.

- **Data operations on sorts.** Data operations operate only on individual data.

The following are primitive actions and action combinators that were not included in ACTRESS notation:

- **Escape actions.** Actions cannot escape in ACTRESS action notation. In standard action notation the primitive action 'escape' is used to signal abnormal computation. 'a_1 trap a_2' sets a_2 as the trap action to be performed when a_1 escapes.

- **Committed actions.** In standard action notation a committed action discards all alternatives except the one being performed, so any later failure will not cause an alternative (if there is one) to be tried (performed). In general, *commitment* is a property of action performance. For example, in standard action notation, all imperative actions are committed actions, that is, after an imperative action is performed any subsequent failure, for the chosen alternative, will cause the whole action to fail because the current alternative will not be tried (no *backtracking*). The ACTRESS subset does not have committing actions.

- **The primitive action 'choose'.** Not included because of its intrinsic non-determinism.

[8] In fact, standard action notation used the label-data mapping when we started the design of ACTRESS.

- **The primitive action 'rebind'.** Left out because of technical problems that it poses to sort inference.
- **Indirect bindings.** We do not treat indirect bindings.

The following actions are not included in standard action notation:

- **'a_1 else a_2'.** Adopted as a convenient abbreviation (see Section 3.4.7).
- **'deallocate y'.** The same as 'unreserve y' in standard action notation.

It is worth to stating that, although not as general as standard action notation, ACTRESS action notation is general enough to specify most of the features encountered in some real programming languages.

3.7 Experiences and References

The aim of action semantics is to obtain better pragmatic qualities than those of denotational semantics, and it has been used to give semantics to a variety of programming languages, including PASCAL [83], JOYCE [10], STANDARD ML [109, 85], BETA [88], CCS and CSP [18]. Action semantics has also been chosen as the specification language for the formal semantics of ANDF [103].

For technical details on *unified algebras*, which provides a framework for action semantics, see [78, 77].

Action notation has many nice algebraic properties [80] which will be explored in more detail in Chapter 5.

For a complete and detailed presentation of action semantics, including an account of its development (Chapter 19) and its operational semantics (Appendix C) see [80]. For a more gentle introduction see [110].

The complete static semantics for ACTRESS action notation together with a sort checker are given in [14]. We will say more about sort checking in the next chapter.

Chapter 4

ACTRESS

This chapter presents ACTRESS, a semantics-directed compiler generator based on action semantics. We start by describing the architecture and design decisions of the system, as well as an overview of the action notation compiler which constitutes ACTRESS' main module. Most of the chapter treats in detail the action notation code generator, the back end of the action notation compiler, and the two other ACTRESS modules, the actioneer generator and the action notation interpreter. We conclude with some preliminary benchmarks (which will also serve as a reference to the ones presented in Chapter 7), identifying some deficiencies and suggesting some improvements. ACTRESS is a result of joint work with David Watt and Deryck Brown [16]. In this chapter we describe in detail the parts of the system I have been responsible for.

4.1 Architecture

Since an action is the meaning of a program in action semantics, it is natural to think of a compiler for action notation as a first step towards an action semantics based compiler generator. In fact, the main ACTRESS module is ANC, an action notation compiler. Although ANC can be used to compile any action, we are particularly interested in the compilation of program actions, that is, actions that are denotations of programs. ANC can be understood as the following function composition:

$$anc \;=\; code \circ check \circ parse \tag{4.1}$$

The function *parse* takes a source program action and parses it producing an action abstract syntax tree (an action parse tree). We will use the term *action tree* for such a tree. *check* verifies whether the action is well-formed and well-sorted. Its output is a decorated action tree (more on this in Section 4.2.2). Finally, the *code* function implements a code generator for actions which takes the decorated action tree and translates it to a C program. The global effect is *the translation of the source action into a C program whose behaviour is equivalent, in some sense, to the source action performance.*

A second module of the ACTRESS system, the actioneer generator, will care for the incorporation of the semantics of the source language into the generated compiler:

$$actioneergen \;=\; gen \circ parse' \tag{4.2}$$

The actioneer generator input is an action semantic description of a programming language $\mathcal{L}$. From this it generates a program, the *actioneer for* $\mathcal{L}$, which will give meaning for $\mathcal{L}$'s programs. The function $parse'$, an extension of *parse* in Equation 4.1 which can also parse action semantic descriptions, parses the input and produces a parse tree for it. From this parse tree *gen* generates the actioneer for the source language. The actioneer for a source language takes a parse tree of a source program and composes the program action which corresponds to it. This can be achieved because the actioneer for a language incorporates the language's action semantics.

A third and optional module is an interpreter for action notation, ANI. This can be expressed by

$$ani \;=\; interp \circ parse \tag{4.3}$$

where *interp* is a function that interprets actions. ANI takes a source action and produces an outcome which represents the action performance.

To build a compiler for a language $\mathcal{L}$ using ACTRESS, we first need to generate a parser for $\mathcal{L}$. ACTRESS does not provide a parser generator and we borrow ML-YACC to do this job [101]. Suppose $syntax_{\mathcal{L}}$ is the syntax description for $\mathcal{L}$; then we can use the parser generator to obtain a parser for $\mathcal{L}$:

$$parse_{\mathcal{L}} \;=\; \textit{ml-yacc}\ syntax_{\mathcal{L}} \tag{4.4}$$

If $semantics_{\mathcal{L}}$ stands for $\mathcal{L}$'s action semantic description, then an actioneer for $\mathcal{L}$ can be obtained as follows:

$$actioneer_{\mathcal{L}} \;=\; actioneergen\ semantics_{\mathcal{L}} \tag{4.5}$$

Now we have the generated compiler for $\mathcal{L}$:

$$comp_{\mathcal{L}} \;=\; code \circ check \circ actioneer_{\mathcal{L}} \circ parse_{\mathcal{L}} \tag{4.6}$$

Compilation of an $\mathcal{L}$ program $\mathcal{P}$, to an object program $\mathcal{O}_{\mathcal{P}}$ can be expressed as

$$\mathcal{O}_{\mathcal{P}} \;=\; comp_{\mathcal{L}}\ \mathcal{P} \tag{4.7}$$

The organization adopted proved to be the right one for its simplicity and flexibility for future changes.

Except for the run-time support, ACTRESS is entirely implemented in STANDARD ML[1]. This choice was based in the good level of abstraction that a functional language provides, which improved the productivity and freed us to concentrate more in conceptual aspects than in implementation details. Also the robustness of STANDARD ML and its implemention was decisive when we chose it as our implementation language.

The choice of C as our target language gave a good level of abstraction, which freed us from idiosyncrasies of low-level languages, it gave to our object code a great degree of portability, and the efficiency penalty is not big when compared to assembly code.

We will detail in the next sections each one of ACTRESS's modules.

4.2 The Action Notation Compiler

A schematic diagram of ANC is shown in Figure 4.1. Its three basic components are described in the sequence.

[1] The New Jersey implementation was used [3].

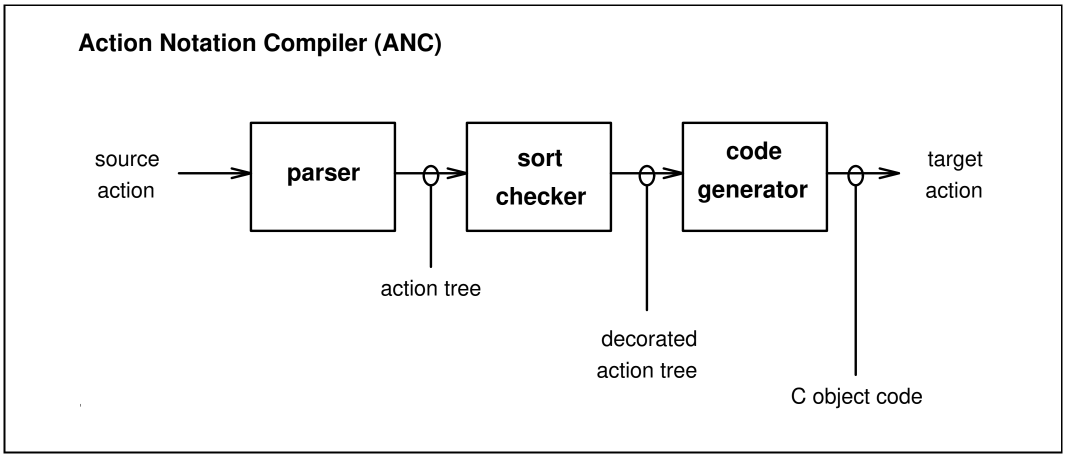


Figure 4.1: The action notation compiler (ANC).

4.2.1 The Parser

The *action notation parser* parses a source action and translates it to the corresponding action tree. It is an LALR(1) parser generated using ML-YACC. This is a standard component of ANC. Standard data operations and user-defined operations have a prefix syntax. Figure 4.2 shows a real input (ASCII format) to the parser, which is the program action for the SPECIMEN program shown in Figure 4.3. Source actions can be grouped using parentheses or vertical rules as in standard action notation.

The action tree is represented internally as a polymorphic STANDARD ML datatype. This representation, simple and elegant, allows for an uniform treatment of the action tree by the other compiler components.

4.2.2 The Sort Checker

The action notation sort checker is similar in function to an ordinary type checker, but it is in fact significantly more sophisticated. Its objective is to rule out ill-formed and ill-sorted actions (ill-formed actions are not ruled out by the parser because its grammar treats actions, yielders and data as terms). Figure 4.4 and Figure 4.5 shows examples of ill-formed and ill-sorted actions respectively.

The sort checker also decorates the action tree with *sort information*. This information will be useful to the code generator[2]. Figure 4.6 shows the syntax of the sort language

[2] Also it will play an important role in the transformer as explained in Chapter 5.

```
| | | give 1000000
| | then
| | | bind "n" to the value
| before
| | | | | give 0
| | | | then
| | | | | give the value label #1
| | | and
| | | | | allocate an cell
| | | | then
| | | | | give the datum label #2
| | then
| | | | bind "x" to the cell#2
| | | and
| | | | store the value#1 in the cell#2
hence
| | | give the value bound to "n"
| | then
| | | store the value in the cell bound to "x"
| and then
| | unfolding
| | | | | | | give the value stored in the cell bound to "x"
| | | | | | then
| | | | | | | give the value label #1
| | | | | and
| | | | | | | give 0
| | | | | | then
| | | | | | | give the value label #2
| | | | then
| | | | | give isGreaterThan(the value#1,the value#2)
| | | then
| | | | | | | | | | give the value stored in the cell bound to "x"
| | | | | | | | | then
| | | | | | | | | | give the value label #1
| | | | | | | | and
| | | | | | | | | | give 1
| | | | | | | | | then
| | | | | | | | | | give the value label #2
| | | | | | | then
| | | | | | | | give difference(the integer#1,the integer#2)
| | | | | | then
| | | | | | | store the value in the cell bound to "x"
| | | | | and then
| | | | | | unfold
| | | | else
| | | | | complete
```

Figure 4.2: Program action for the *loop* program.

```
program loop is
    const n : int = 1000000;
    var x : int := 0
in
    x := n;
    while (x > 0) do x := x − 1 end
end
```

Figure 4.3: The SPECIMEN *loop* program.

```
give 4 then successor (it)

bind "x" to give 3

allocate a cell then store 8 in the cell give true

bind "x" to 1 and abstraction (give 4)

enact (bind "x" to 3)
```

Figure 4.4: Ill-formed actions.

```
give 4 then give successor (the truth-value)

give 3 and give 4

bind "x" to 3 or bind "y" to 8

enact abstraction (give the integer) with true
```

Figure 4.5: Ill-sorted actions.

used to decorate the action tree. An action term is decorated with an action sort; a yielder term with a yielder sort and a datum term with a datum sort.

τ has information about the sorts of the transients required/given by an action, yielder or abstraction term. For example, if an action requires only a transient of sort $\mathsf{integer}$ labelled $\mathsf{3}$, τ (in the left side of the hook arrow) would be '$\{\mathsf{3} : \mathsf{integer}\}$'. We can see τ as a $\mathsf{natural}$-*DatumSort* mapping.

β is similar to τ except that it contains information about the sort of data bound to tokens (a token-*DatumSort* mapping). As an individual sort classifies just one element, the sort for an individual datum like $\mathsf{4}$ is $\mathsf{4}$. We write

$$a : (\tau, \beta) \hookrightarrow (\tau', \beta')$$

to say that action a has sort "$(\tau, \beta) \hookrightarrow (\tau', \beta')$" (or, more concretely, a's action tree is decorated with the correspondent sort). Below some examples:

$$\begin{aligned} \mathsf{give\ 1} :&\ (\{\},\{\}) \hookrightarrow (\{\mathsf{0} : \mathsf{1}\},\{\}) \\ \mathsf{bind\ \text{``x''}\ to\ true} :&\ (\{\},\{\}) \hookrightarrow (\{\},\{\text{``}\mathsf{x}\text{''} : \mathsf{true}\}) \\ \mathsf{give\ 1\ and\ bind\ \text{``x''}\ to\ true} :&\ (\{\},\{\}) \hookrightarrow (\{\mathsf{0} : \mathsf{1}\},\{\text{``}\mathsf{x}\text{''} : \mathsf{true}\}) \end{aligned}$$

In practice the sort checker traverses the action tree and decorates each node with sort information: action nodes with action sorts, abstraction nodes with abstraction sorts, and so on.

Sort information is represented internally by record types similar to those introduced by Wand in [107], and applied by Even and Schmidt in [98]. Each action sort consists of four record schemes, representing required and given transients, and required and produced bindings. For example, a record for transients might be:

$$\{\mathsf{0} : \mathsf{integer}, \mathsf{1} : \mathsf{truth\text{-}value}\}$$

and a record for bindings

$$\{\text{``}\mathsf{n}\text{''} : \mathsf{7}, \text{``}\mathsf{m}\text{''} : \mathsf{integer}, \text{``}\mathsf{z}\text{''} : \mathsf{cell}\ [\mathsf{integer}]\}$$

The sort discipline enforced by the sort checker has been specified using a set of infer-

$$
\begin{array}{rcll}
\mathit{Sort} & ::= & \mathit{ActionSort} & (4.8)\\
 & | & \mathit{YielderSort} & (4.9)\\
 & | & \mathit{DatumSort} & (4.10)\\
\\
\mathit{ActionSort} & ::= & (\tau,\beta) \hookrightarrow (\tau,\beta) & (4.11)\\
 & | & \mathsf{nothing} & (4.12)\\
\\
\mathit{YielderSort} & ::= & (\tau,\beta) \rightsquigarrow \mathit{DatumSort} & (4.13)\\
\\
\mathit{DatumSort} & ::= & \mathit{AbstractionSort} & (4.14)\\
 & | & \mathit{ProperSort} & (4.15)\\
 & | & \mathit{IndividualSort} & (4.16)\\
\\
\mathit{AbstractionSort} & ::= & (\tau,\beta) \rightsquigarrow (\tau,\beta) & (4.17)\\
\\
\mathit{ProperSort} & ::= & \mathsf{datum} & (4.18)\\
 & | & \mathsf{token} & (4.19)\\
 & | & \mathsf{integer} & (4.20)\\
 & | & \mathsf{truth\text{-}value} & (4.21)\\
 & | & \mathsf{list}\ [\mathit{DatumSort}] & (4.22)\\
 & | & \mathsf{cell}\ [\mathit{DatumSort}] & (4.23)\\
 & | & \mathit{DatumSort}\ \mathbf{|} \ldots \mathbf{|}\ \mathit{DatumSort} & (4.24)\\
 & | & \mathsf{nothing} & (4.25)\\
\\
\mathit{IndividualSort} & ::= & \mathit{Individual} & (4.26)\\
\\
\mathit{Individual} & ::= & \mathsf{true} \mid \mathsf{false} & (4.27)\\
 & | & \ldots \mid \mathsf{-1} \mid \mathsf{0} \mid \mathsf{1} \mid \ldots & (4.28)
\end{array}
$$

Figure 4.6: Syntax of sort information.

ence rules, somewhat analogous to the type inference rules for a programming language.

There are some cases where the sort checker modifies the action tree. The '**fail**' action can cause a simplification of the action tree, for example, the action '**fail or** a' is replaced by a. An action that is certainly ill-sorted is replaced by the action '**fail**'. Where the sort checker cannot guarantee that an action is well sorted, it must inform the code generator that some run-time sort check will be needed (see Section 4.3.2).

The sort inference algorithm used in the sort checker is based on the one given in [98]. The sort checker is described in detail in [14]. We gave here only a brief account for the purpose of understanding of our work.

4.2.3 The Code Generator

The decorated action tree produced by the sort checker is translated into C object code by the code generator. An action is translated to a C statement sequence; a yielder is translated to a C expression. In the generated code, transients and bindings are passed in registers. The code generator also makes use of sort information provided by the sort checker.

A run-time environment is defined providing data representation, data operations, sort-checking functions, storage management functions and auxiliary functions. In the next section we will describe code generation for action notation in detail.

4.3 Generating Code for Action Notation

The code generator translates a decorated action tree into a C program. It is specified by means of *translation rules*. As in the case of the semantics for ACTRESS action notation, the style of presentation here was inspired by natural semantics. Besides the translation rules, we present the run-time environment and we show how the implementation is close to the translation rules and what its current limitations are.

4.3.1 Translation Rules

The translation rules are similar in form to the ones used to give the semantics of ACTRESS's action notation. Pieces of C code in the rules are written in `this typewriter font`. Wherever this convention is not clear we use double quotes to distinguish them.

In general a variable is used instead of the actual C code with the value of the variable specified in the **where** part of the rule. Bidimensional layout is used to enhance the code readability, but we usually do not bother to use a string concatenation operator ($\smile$). The C code in turn can have some variables, for example, '`_d`d' stands for '`_d4`' when variable d is instantiated to 4 (variables are written in *italics*). Run-time functions are written using capital letters, for example, this is a `_RUN_TIME_FUNCTION`.

Judgements

Two types of judgements are present. The first one is used in the translation of an action term. It has the form

$$\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, (d_x, b_x)) \vdash a \twoheadrightarrow (c, S, A', b', f', r', i', (d'_x, b'_x)) \tag{4.29}$$

where a is the action subject to translation and c (together with S) is the corresponding C object code. More precisely, c is a C statement (a string) resulting from the translation of a; and S is a set of strings where each one is a C function resulting from the translation of an abstraction in a. The other variables are as follows:

- $\mathcal{K}$ is sort information from which we can infer the sort of a.

- In the generated C program transients are kept in registers called *d-registers* (C variables in fact). During translation the code generator maintains a mapping from labels to these d-registers to locate any transient needed; a d-register is identified by a positive integer. We call this mapping a *d-register assignment*; during translation for each action there is a d-register assignment A into the action and a d-register assignment A' out of the action. Notice that the range of A' is a set of d-registers where the transients given by a are stored.

- A set of bindings is kept in a single *b-register*. While each d-register contains only one individual datum, a b-register can contain any number of bindings (each binding is a token-datum pair). For an action a, b is a b-register which contains the bindings required by a and b' the b-register which contains the bindings produced by a. b is just a natural number; the value 0 is used to indicate that the action requires or produces no bindings (a dummy register).

- A_u is an auxiliary d-register assignment, called the *unfold d-register assignment*, used in the translation of 'unfolding' and 'unfold' actions.

- U_d is a set of d-registers (a set of positive integers) which indicates which d-registers cannot be used during the translation of a.

- U_b is a set of b-registers (a set of positive integers) which indicates which b-registers cannot be reused during the translation of a. An empty set indicates that any b-register can be used. After the translation of a program action a, b' will be the b-register where the bindings produced by a are stored.

- f, a natural number, is used to propagate the value of the current failure label. This will be only used to code the 'fail', 'or' and 'check' actions. In the case of the 'fail' action it will provide the label where the program flow should jump to. The C label correspondent to a failure label is uniquely generated by the translation of an 'or' action.

- The current repeat label r is only used to translate 'unfolding' and 'unfold' actions. The translation of 'unfolding a' generates the repeat label, and the translation of a free 'unfold' inside a uses its value as the point it has to jump to.

- l, a boolean, is the *fail action context* needed for determining the context of the 'fail' action during its translation (see the translation of 'fail' later in this section).

- i is a natural used to generate unique names for the C functions resulting from the translation of abstractions, for example, if, at some point during translation, the value of i is 4, the next abstraction will be translated to the C function `_abs4`.

- d_x and b_x are natural numbers indicating the maximum number of d-registers and b-registers used so far in the translation; they work as a *high water marks*. Their final values d'_x and b'_x are needed to code the C declarations for the registers. (We use C sometimes to stand for (d_x, b_x).)

The second judgement is used for yielders' translation:

$$\mathcal{K}, (A, b, i) \vdash y \twoheadrightarrow (c, i, S) \tag{4.30}$$

$$\frac{\mathcal{K}, ([\,], 0, [\,], \{\}, \{\}, 0, 0, \mathit{true}, 1, (0,0)) \vdash a \twoheadrightarrow (c', \{s_1, \ldots, s_{i'-1}\}, A', b', f', r', i', (d_x, b_x))}{\mathcal{K} \vdash a \blacktriangleright c} \tag{4.31}$$

where $c =$

```
#include "runtime.h"
#include "runtime.c"
s1
...
s(i'-1)
DATUM _d1, ... , _d(dx);
BINDINGS _b1, ... , _b(bx);
int main ()
{
c'
exit(0);
_failure0:
exit(1);
}
```

Figure 4.7: Program action translation rule.

where c is a C-expression resulting from the translation of the yielder y. In general S is the empty set, except when y contains a term of the form '**abstraction** a'. All the other variables are as in the judgement for action terms.

Program Action

Figure 4.7 shows the rule used for the translation of the program action. This rule tells how we start up the translation process. The variables are initialized which establish our initial translation hypothesis: the action requires no transients and no bindings (empty input d-register assignment and input b-register 0); the unfold d-register assignment is empty; all d-registers and b-registers are free; failure and repeat label are 0; the '**fail**' action context flag is *true*; no abstraction was previously translated; and no d-registers or b-registers were used. We also assume that a gives no transients and produces no bindings. This rule is applied only once.

The generated C program, c, consists of the runtime environment (see the `#include`'s in Figure 4.7), a C function for each abstraction in a ($s_1, \ldots, s_{i'-1}$), C declarations for d-registers and b-registers, and the C statement resulting from the translation of a which will form the body of `main` (c'). The failure label initial value is 0 so if the program action fails, there will be a jump to label `_failure0` and the program terminates with an exit status equal to 1 (abnormally). Otherwise it will terminate normally (exit status 0). Notice how the high water marks d_x and b_x are used to code the C declarations for the d-registers and b-registers used in the object code for a.

Basic

Figures 4.8 and 4.10 show the translation rules for basic action notation.

As would be expected, the '`complete`' action translates to the C null statement. This translation is specified by Rule 4.32. We use '_' as a place holder for a variable that is not needed for the translation of a term.

Two (mutually exclusive) rules specify the translation of '`fail`'. They are distinguished by the fail action context flag: if it is true, it is because we are inside an '`or`' action or at the program action top level, so we inspect the current failure label (f) and generate a jump to it (`goto _failure`f`;`); otherwise we are inside an abstraction top level, so we just exit the C function returning 1 (abnormally exit). As we will see later, the translation of the '`enact`' action generates C code that handles these abnormal (and normal) exits.

In the translation of 'a_1 `and` a_2' the subactions are translated sequentially, the left subaction first. This fact is specified in Rule 4.35 by the fact that we need A_1, b_1, f_1, r_1, i_1 and C_1 for the translation of a_2, and these values are only available after the translation of a_1.

The translation of a_1 must not reuse any d-register still to be used (read) by the translation of a_2 (U_d). Similarly, in the translation of a_2 we must not reuse any d-register used (written) by a_1 ($U_d \cup \mathit{range}\ A_1$, in the second antecedent of Rule 4.35). b_1 and b_2 stand for the b-registers containing the bindings produced by a_1 and a_2 respectively.

The third antecedent caters for the binding produced by the whole action. Some code might be needed here to do the merge of the bindings produced by a_1 and a_2. For example, if a_1 produces bindings in `_b1` and a_2 produces no bindings, there is no need for any code (`;`) and the output b-register will be `_b1`. If both actions produce bindings we have to rely on a run-time C function to merge the bindings (`_OVERLAY_BINDINGS`), whose result will be stored in a free b-register. Figure 4.9 specifies the overlay bindings judgement used in the translation rule of some composite actions like '`and`'. Notice that although we use the name *overlay bindings* — and in fact this is the semantics of the run-time C function —

(COMPLETE)

$$\_, (\_, \_, \_, \_, \_, f, r, \_, i, C) \vdash \mathsf{complete} \twoheadrightarrow (\text{“;''}, \{\}, [\,], 0, f, r, i, C) \tag{4.32}$$

(FAIL)

$$\_, (\_, \_, \_, \_, \_, f, r, true, i, C) \vdash \mathsf{fail} \twoheadrightarrow (\text{“}\texttt{goto \_failure}f;\text{''}, \{\}, [\,], 0, f, r, i, C) \tag{4.33}$$

$$\_, (\_, \_, \_, \_, \_, f, r, false, i, C) \vdash \mathsf{fail} \twoheadrightarrow (\text{“}\texttt{return(1);}\text{''}, \{\}, [\,], 0, f, r, i, C) \tag{4.34}$$

(AND)

$$\frac{\begin{array}{c}\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1 \twoheadrightarrow (c_1, S_1, A_1, b_1, f_1, r_1, i_1, C_1) \\ \mathcal{K}, (A, b, A_u, U_d \cup range\ A_1, U_b \cup \{b_1\}, f_1, r_1, l, i_1, C_1) \vdash a_2 \twoheadrightarrow (c_2, S_2, A_2, b_2, f_2, r_2, i_2, C_2) \\ (b_1, b_2, U_b) \overset{ov}{\vdash} (c_3, b')\end{array}}{\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1 \ \mathsf{and}\ a_2 \twoheadrightarrow (c, S_1 \cup S_2, A_1 \oplus A_2, b', f_2, r_2, i_2, C_2)} \tag{4.35}$$

where $c = c_1$
c_2
c_3

(AND THEN)

$$\frac{\begin{array}{c}\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1 \twoheadrightarrow (c_1, S_1, A_1, b_1, f_1, r_1, i_1, C_1) \\ \mathcal{K}, (A, b, A_u, U_d \cup range\ A_1, U_b \cup \{b_1\}, f_1, r_1, l, i_1, C_1) \vdash a_2 \twoheadrightarrow (c_2, S_2, A_2, b_2, f_2, r_2, i_2, C_2) \\ (b_1, b_2, U_b) \overset{ov}{\vdash} (c_3, b')\end{array}}{\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1 \ \mathsf{and\ then}\ a_2 \twoheadrightarrow (c, S_1 \cup S_2, A_1 \oplus A_2, b', f_2, r_2, i_2, C_2)} \tag{4.36}$$

where $c = c_1$
c_2
c_3

(UNFOLDING)

$$\frac{\begin{array}{c}\mathcal{K} \vdash \mathsf{unfolding}\ a : (\tau, \beta) \hookrightarrow (\tau', \beta') \\ A'_u = A \ominus dom\ \tau \\ \mathcal{K}, (A, b, A'_u, U_d, U_b, f, r+1, l, i, C) \vdash a \twoheadrightarrow (c', S', A', b', f', r', i', C')\end{array}}{\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash \mathsf{unfolding}\ a \twoheadrightarrow (c, S', A', b', f', r', i', C')} \tag{4.37}$$

where $c = \texttt{repeat\_}r + 1\texttt{:}$
c'

(UNFOLD)

$$\frac{\begin{array}{c}\mathcal{K} \vdash \mathsf{unfold} : (\tau, \beta) \hookrightarrow (\tau', \beta') \\ (A_u \ominus dom\ \tau, A \ominus dom\ \tau, U_d, d_x) \overset{rd}{\vdash} (c', d'_x)\end{array}}{\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, (d_x, b_x)) \vdash \mathsf{unfold} \twoheadrightarrow (c, \{\}, [\,], 0, f, r, i, (d'_x, b_x))} \tag{4.38}$$

where $c = c'$
`goto repeat_r;`

Figure 4.8: Basic translation rules.

(OVERLAY-BINDINGS)

$$(0, b, \_) \overset{ov}{\vdash} (``;", b) \tag{4.39}$$

$$(b, 0, \_) \overset{ov}{\vdash} (``;", b) \tag{4.40}$$

$$\frac{b' = new\ U_b \qquad b_1 \neq 0 \qquad b_2 \neq 0}{(b_1, b_2, U_b) \overset{ov}{\vdash} (``\mathtt{\_b}b' \mathtt{= \_OVERLAY\_BINDINGS(\_b}b_1\mathtt{, \_b}b_2\mathtt{);}", b')} \tag{4.41}$$

Figure 4.9: The overlay bindings rules.

it is in fact here a merge operation (see the semantic rule for '**and**'). But as the action is well-sorted (tokens produced by the subactions are disjoint) the effect of the overlay operation is the same as that of a merge operation. We have used the presence or absence of a b-register to decide at translation time if an action produces bindings; alternatively we could have made use of sort information.

In the translation of '**unfolding** a' (Rule 4.37) the code generator generates a repeat label and proceeds to translate a. Now, for every free tail-recursive occurrence of the '**unfold**' action inside a it generates a jump to the repeat label (Rule 4.38). We use the fact (enforced by the sort checker) that all transients and bindings required by the '**unfold**' action are of the same sort as those required by the enclosing '**unfolding**' action. The unfold d-register assignment (A'_u) records the d-register assignment for the '**unfolding**' action, so that the translation of its free '**unfold**' actions can generate, if needed, C code to rearrange the d-registers before the jump to the repeat label. For example, the translation of the (artificial) action

give 5 then unfolding (give sum (the integer,1) then unfold)

produces

```
_d1 = _MAKE_INTEGER(5);                      (1)
_repeat_1:                                   (2)
_d2 = _SUM(_d1, _MAKE_INTEGER(1));           (3)
_d1 = _d2;                                   (4)
goto _repeat_1;                              (5)
```

where the rearrangement code is at line 4.

Notice in the rules how the sort information was essential for the translation. As A

(OR)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f+1,r,true,i,C)\vdash a_1 \twoheadrightarrow (c_1,S_1,A_1,b_1,f_1,r_1,i_1,C_1)\\ \mathcal{K},(A,b,A_u,U_d,U_b,f_1,r_1,true,i_1,C_1)\vdash a_2 \twoheadrightarrow (c_2,S_2,A_2,b_2,f_2,r_2,i_2,(d_{x_2},b_{x_2}))\\ (A_1,A_2,range\ A_2,d_{x_2}) \overset{rd}{\vdash} (c_3,d'_x) \qquad (b_1,b_2) \overset{rb}{\vdash} c_4\end{array}}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,false,i,C)\vdash a_1\ \mathsf{or}\ a_2 \twoheadrightarrow (c,S_1\cup S_2,A_1,b_1,f_2,r_2,i_2,(d'_x,b_{x_2}))} \tag{4.42}$$

where $c = c_1$
`goto end_`$f+2$;
`failure_`$f+1$:
c_2
c_3
c_4
`end_`$f+2$:

$$\frac{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,false,i_i,C)\vdash a_1\ \mathsf{or}\ a_2 \twoheadrightarrow (c',S',A',b',f',r',i',C')}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,true,i_i,C)\vdash a_1\ \mathsf{or}\ a_2 \twoheadrightarrow (c',S',A',b',f',r',i',C')} \tag{4.43}$$

Figure 4.10: Translation rules for the 'or' combinator.

contains all the labels (transients) received by 'unfolding a' (not only the ones required) we need to restrict the mapping using τ whose domain contains exactly the labels required by the action ($A \ominus dom\ \tau$). For simplification we consider that all occurrences of the 'unfold' action are tail recursive. This is enforced by a test at translation time.

Figure 4.10 shows the translation rules for the 'or' combinator. In the translation of 'a_1 or a_2', the d-registers assigned to the transient data given by a_1 must be the same as those assigned to the transient data given by a_2. The same applies to b-registers. This raises the need for a rearrangement of d-registers and b-registers as the translation of the subactions may given different d-register and b-register assignments. The rearrangement is necessary so that, regardless of what subaction is actually performed, it leaves its transients and bindings in the same registers. The code generator takes a_1 as the reference and rearranges the registers that come out of a_2 in a way to match that of a_1. For example, the translation of

(give 1 or (give 2 then give 4)) then give successor (it)

produces

_d1 = *_MAKE_INTEGER*(1); (1)
goto *_end_2*; (2)

$\_failure\_1$: (3)
$\_d1 = \_MAKE\_INTEGER(2);$ (4)
$\_d2 = \_MAKE\_INTEGER(4);$ (5)
; (6)
$\_d1 = \_d2;$ (7)
$\_end\_2$: (8)
$\_d2 = \_SUCCESSOR(\_d1);$ (9)
; (10)

The code at line 7 is the rearrangement code for d-registers (there is no need for b-register rearrangement in this example). Line 4 to line 6 correspond to the translation of the second subaction of 'or'. The translation of the 'or' is the only place in the translation process where a failure label is introduced — apart from the main program failure label and incorporated actions' top level.

Functional

Figure 4.11 shows the translation rules for functional action notation.

The translation of 'give y label $\#n$' generates code to store the value of y in a newly-allocated d-register. Notice the d-register assignment out of the action ($[n \mapsto d]$). Notice that we have defined a special case in the translation of the 'give' action. When y is the yielder 'the $s\#m$', there is no need to generate any code, we need only to update the d-register assignment accordingly to the relabelling of the transient (from '$m \mapsto d$' to '$n \mapsto d$' in Rule 4.45).

The translation of 'check y' corresponds to the C 'if' statement, where the conditional expression corresponds to the translation of y (see Rule 4.46). If y evaluates to false then the control jumps to the current failure label. Otherwise the next command is executed.

The translation of 'a_1 then a_2' is the same as the translation for 'and' except that the d-register assignments out of the subactions are not merged: the one out of a_2 is used instead, which agrees with the semantic rules of the combinator (see Figure 3.4 in Chapter 3).

In the translation of 'the $s\#n$', the code generator inspects the d-register assignment for the d-register assigned to label n, and gives this register as a result. Now, for example, the action

(give 30 label #1 and give true label #2) then give the integer#1

(GIVE)

$$\frac{\mathcal{K}, (A, b, i) \vdash y \twoheadrightarrow (c_y, i_y, s_y) \qquad (d, U'_d) = new\ U_d}{\begin{array}{c}\mathcal{K}, (A, b, \_, U_d, \_, f, r, \_, i, (d_x, b_x)) \vdash \textsf{give}\ y\ \textsf{label}\ \#n \\ \twoheadrightarrow (c, s_y, [n \mapsto d], 0, f, r, i_y, (greater\ d_x\ d, b_x))\end{array}} \tag{4.44}$$

where $c =$ `_d`d `=` c_y`;`

(GIVE-SPECIALIZED)

$$\begin{array}{c}\mathcal{K}, ([m \mapsto d, \ldots], \_, \_, \_, \_, \_, f, r, \_, i, C) \vdash \textsf{give the}\ s\#m\ \textsf{label}\ \#n \\ \twoheadrightarrow (";", \{\}, [n \mapsto d], 0, f, r, i, C)\end{array} \tag{4.45}$$

(CHECK)

$$\frac{(A, b, i) \vdash y \twoheadrightarrow (c_y, i_y, S_y)}{\mathcal{K}, (A, b, \_, U_d, \_, f, r, \_, i, C) \vdash \textsf{check}\ y \twoheadrightarrow (c, \{\}, [\,], 0, f, r, i, C)} \tag{4.46}$$

where $c =$ `if (!`c_y`.datum.truth_value) goto failure_`f`;`

(THEN)

$$\frac{\begin{array}{c}\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1 \twoheadrightarrow (c_1, S_1, A_1, b_1, f_1, r_1, i_1, C_1) \\ \mathcal{K}, (A, b, A_u, U_d \cup range\ A_1, U_b \cup \{b_1\}, f_1, r_1, l, i_1, C_1) \vdash a_2 \\ \twoheadrightarrow (c_2, S_2, A_2, b_2, f_2, r_2, i_2, C_2) \\ (b_1, b_2, U_b) \overset{ov}{\vdash} (c_3, b')\end{array}}{\mathcal{K}, (A, b, A_u, U_d, U_b, f, r, l, i, C) \vdash a_1\ \textsf{then}\ a_2 \twoheadrightarrow (c, S_1 \cup S_2, A_2, b', f_2, r_2, i_2, C_2)} \tag{4.47}$$

where $c = c_1$
c_2
c_3

(THE)

$$\mathcal{K}, ([n \mapsto d, \ldots], b, i) \vdash \textsf{the}\ s\#n \twoheadrightarrow (\text{“}\texttt{\_d}d\text{”}, i, \{\}) \tag{4.48}$$

Figure 4.11: Functional translation rules.

translates to

$$
\begin{array}{lll}
\_d1 & = & \_MAKE\_INTEGER(30); \\
\_d2 & = & \_MAKE\_TRUTH(true); \\
; & & \\
; & & \\
; & &
\end{array}
\begin{array}{r}
(1)\\(2)\\(3)\\(4)\\(5)
\end{array}
$$

Notice that '**give the integer#1**' was translated using the special case for the '**give**' action (Rule 4.45).

Declarative

Figure 4.12 shows the translation rules for declarative action notation.

Tokens are translated into C strings. The '**bind**' action (Rule 4.49 is translated to a run-time function (`_BIND`) which will make the actual binding when the program is run. This binding is stored in a new b-register which is then passed ahead in the translation process (b').

The translation of 'a_1 **hence** a_2' (Rule 4.51) is similar to the translation of '**and**'. There are two main differences though: when the code generator translates a_2 the tied b-register is only the one where a_1 has stored its produced bindings; and no overlay binding code is necessary because the bindings produced by the whole action are the ones produced by a_2 (they are stored in $\_\mathtt{b}b_2$).

In the translation of '**the** s **bound to** y' the code generator uses a run-time function, `_BOUND`, which will look up what is bound to the token c_y in the set of bindings b (Rule 4.54). We had to make use of the run-time function because a set of bindings is kept in a single b-register instead of an individual b-register for each binding. Now, for example, the action

(bind "x" to 15 and bind "y" to 30) hence bind "z" to successor (the integer bound to "x")

translates to

$$
\begin{array}{lll}
\_b1 & = & \_BIND(\texttt{"x"},\_MAKE\_INTEGER(15)); \\
\_b2 & = & \_BIND(\texttt{"y"},\_MAKE\_INTEGER(30)); \\
\_b3 & = & \_OVERLAY\_BINDINGS(\_b2,\_b1); \\
\_b1 & = & \_BIND(\texttt{"z"},\_SUCCESSOR(\_BOUND(\texttt{"x"},\_b3)));
\end{array}
\begin{array}{r}
(1)\\(2)\\(3)\\(4)
\end{array}
$$

(BIND)

$$\frac{\mathcal{K},(A,b,i) \vdash y_1 \twoheadrightarrow (c_1,i_1,S_1) \qquad \mathcal{K},(A,b,i_1) \vdash y_2 \twoheadrightarrow (c_2,i_2,S_2) \qquad (b',U_b') = new\ U_b}{\begin{array}{c}\mathcal{K},(A,b,\_,\_,U_b,f,r,\_,i,(d_x,b_x)) \vdash \textsf{bind } y_1 \textsf{ to } y_2 \\ \twoheadrightarrow (c, S_1 \cup S_2, [\,], b', f, r, i_2, (d_x, greater\ b'\ b_x))\end{array}} \tag{4.49}$$

where c = `_b`b' `= _BIND(`c_1`,`c_2`);`

(FURTHERMORE)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a \twoheadrightarrow (c',S',A',b',f',r',i',C') \\ (b,b',U_b) \overset{ov}{\vdash} (c'',b'')\end{array}}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash \textsf{furthermore } a \twoheadrightarrow (c,S',A' \oslash A,b'',f',r',i',C')} \tag{4.50}$$

where $c = c' \smile c''$

(HENCE)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \twoheadrightarrow (c_1,S_1,A_1,b_1,f_1,r_1,i_1,C_1) \\ \mathcal{K},(A,b_1,A_u,U_d \cup range\ A_1,\{b_1\},f,r,l,i_1,C_1) \vdash a_2 \twoheadrightarrow (c_2,S_2,A_2,b_2,f_2,r_2,i_2,C_2)\end{array}}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \textsf{ hence } a_2 \twoheadrightarrow (c,S_1 \cup S_2,A_1 \oplus A_2,b_2,f_2,r_2,i_2,C_2)} \tag{4.51}$$

where $c = c_1 \smile c_2$

(MOREOVER)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \twoheadrightarrow (c_1,S_1,A_1,b_1,f_1,r_1,i_1,C_1) \\ \mathcal{K},(A,b,A_u,U_d \cup range\ A_1,U_b \cup \{b_1\},f,r,l,i_1,C_1) \vdash a_2 \twoheadrightarrow (c_2,S_2,A_2,b_2,f_2,r_2,i_2,C_2) \\ (b_1,b_2,U_b) \overset{ov}{\vdash} (c_3,b')\end{array}}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \textsf{ moreover } a_2 \twoheadrightarrow (c,S_1 \cup S_2,A_2 \oslash A_1,b',f_2,r_2,i_2,C_2)} \tag{4.52}$$

where $c = c_1 \smile c_2 \smile c_3$

(BEFORE)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \twoheadrightarrow (c_1,S_1,A_1,b_1,f_1,r_1,i_1,C_1) \\ \mathcal{K},(A,b_3,A_u,U_d \cup range\ A_1,\{b_1\} \cup \{b_3\},f_1,r_1,l,i_1,C_1) \vdash a_2 \twoheadrightarrow (c_2,S_2,A_2,b_2,f_2,r_2,i_2,C_2) \\ (b,b_1,U_b) \overset{ov}{\vdash} (c_3,b_3) \qquad (b_1,b_2,U_b) \overset{ov}{\vdash} (c_4,b_4)\end{array}}{\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \textsf{ before } a_2 \twoheadrightarrow (c,S_1 \cup S_2,A_1 \oplus A_2,b_4,i_2,C_2)} \tag{4.53}$$

where $c = c_1 \smile c_3 \smile c_2 \smile c_4$

(BOUND)

$$\frac{\mathcal{K},(A,b,i) \vdash y \twoheadrightarrow (c',i',S')}{\mathcal{K},(A,b,i) \vdash \textsf{the } s \textsf{ bound to } y \twoheadrightarrow (\text{“}\texttt{\_BOUND(}c'\texttt{,\_b}b\texttt{)}\text{”},i',S')} \tag{4.54}$$

Figure 4.12: Declarative translation rules.

(STORE)

$$\frac{\mathcal{K},(A,b,i)\vdash y_1 \twoheadrightarrow (c_1,i_1,S_1) \qquad \mathcal{K},(A,b,i_1)\vdash y_2 \twoheadrightarrow (c_2,i_2,S_2)}{\mathcal{K},(A,b,\_,\_,\_,f,r,\_,i,C)\vdash \textsf{store } y_1 \textsf{ in } y_2 \twoheadrightarrow (c,S_1\cup S_2,[\,],0,f,r,i_2,C)} \tag{4.55}$$

where c = `*`c_2`.datum.cell = `c_1`;`

(DEALLOCATE)

$$\frac{(A,b,i)\vdash y \twoheadrightarrow (c_y,i_y,S_y)}{\mathcal{K},(A,b,\_,\_,\_,f,r,\_,i,C)\vdash \textsf{deallocate } y \twoheadrightarrow (c,\{\},[\,],0,f,r,i,C)} \tag{4.56}$$

where c = `_DEALLOCATE_THE_CELL(`c_y`);`

(STORED)

$$\frac{\mathcal{K},(A,b,i)\vdash y \twoheadrightarrow (c_y,i_y,S_y)}{\mathcal{K},(A,b,i)\vdash \textsf{the } s \textsf{ stored in } y \twoheadrightarrow (\text{``}{*c_y}\texttt{.datum.cell}\text{''},i_y,S_y)} \tag{4.57}$$

Figure 4.13: Imperative translation rules.

Imperative

Figure 4.13 shows the translation rules for imperative action notation.

Cells are represented by pointers to run-time-allocated cells of memory. (In Chapter 5 we will extend the code generator and cells will also be represented as elements of an array.) Each of these cells holds a datum (see Section 4.3.2 for a description of the run-time environment). Rule 4.55 specifies that the 'store' action is translated to a store operation of the value (a C expression) resulting from the translation of y_1 in the cell pointed by the value (another C expression) resulting from the translation of y_2. For example, the action

allocate a cell then store true in it

translates to

$$\begin{array}{ll} \_d1 \;=\; \_ALLOCATE\_A\_CELL(); & (1)\\ *\_d1.datum.cell \;=\; \_MAKE\_TRUTH(true); & (2)\\ ; & (3) \end{array}$$

where, as we will see later, `_ALLOCATE_A_CELL` is a run-time C function which allocates a piece of storage which can hold a datum.

In Rule 4.56, `_DEALLOCATE_THE_CELL` is a run-time C function which "deallocates" cell

c_y. In fact, we cannot deallocate the cell at run-time. This is because the cell can still be "alive" in others parts of the action, so an access to the cell (after deallocation) will result in a failure. Consider, for example, the action

```
| | allocate a cell
| then
| | store 5 in the cell and give the cell
then
| | deallocate the cell
| and then
| | give successor (the integer stored in the cell)
```

which translates to

$$
\begin{array}{lr}
\_d1 \ = \ \_ALLOCATE\_A\_CELL(); & (1) \\
*\_d1.datum.cell \ = \ \_MAKE\_INTEGER(5); & (2) \\
; & (3) \\
; & (4) \\
; & (5) \\
\_DEALLOCATE\_THE\_CELL(\_d1); & (6) \\
\_d2 \ = \ \_SUCCESSOR(*\_d1.datum.cell); & (7) \\
; & (8) \\
; & (9)
\end{array}
$$

If *_DEALLOCATE_THE_CELL* (line 6) actually deallocates the cell "stored" in *_d1* then the expression '*_SUCCESSOR(*_d1.datum.cell)*' might evaluate to garbage instead of 'nothing'.

One solution would be to store a special mark (nothing) in the cell instead of actually deallocate it, and every time we need to access this cell we would have to check if the content of the cell is a valid one (different from nothing). If it is nothing we just fail (abort) the action. This has the disadvantage of the check for every cell access! Also we would need a garbage collector to actually deallocate a cell when there is no reference to it so cells could be reused.

The translation of 'the s stored in y' dereferences the cell which results from the translation of y (Rule 4.57).

Reflective

Figure 4.14 shows the translation rules for the 'enact' action. Figure 4.15 shows the translation rules for the 'abstraction', 'with' and 'closure' operations.

(ENACT)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,i)\vdash y \twoheadrightarrow (c_y,i_y,S_y)\\ (d',U_d') = new\ U_d \qquad (d'',U_d'') = new\ U_d' \qquad (b',U_b') = new\ U_b\end{array}}{\begin{array}{c}\mathcal{K},(A,b,A_u,U_d,U_b,f,r,l,i,(d_x,b_x)) \vdash \textsf{enact}\ y\\ \twoheadrightarrow (c,S_y,[0\mapsto d''],b',f,r,i_y,(greater\ (greater\ d_x\ d')\ d'',greater\ b_x\ b'))\end{array}} \tag{4.58}$$

where c =
```
_dd' = cy ;
if (_dd'.datum.abs->codeact) (_dd'.datum.abs->datum,
                              _dd'.datum.abs->bindings,
                              & _dd'',
                              & _bb'))
   goto _failuref;
```

Figure 4.14: Translation rule for the 'enact' action.

Before we examine the rules in more detail it is useful to explain how abstractions are represented. An abstraction is represented at run-time by a C structure with three fields. The first field contains a pointer to a C function which was obtained by the translation of the incorporated action. The second field contains datum which will be given to the incorporated action at enaction time. Finally, the third field contains bindings which can be given to the incorporated action at enaction time.

The enaction of an abstraction, 'enact y' corresponds to a call to the C function pointed to by the abstraction resulting from the evaluation of y (c_y). The datum and binding fields of that abstraction are passed to the C function as arguments. Also any datum and bindings given and produced respectively by the performance of the incorporated action are stored in a new d-register `_d`d'' and b-register `_b`b' (notice the d-register assignment and b-register out of the enact action). The 'if' statement handles cases where the incorporated action can fail.

In Rule 4.59, `_ABSTRACTION` is a C run-time function. It gives a C structure (see Section 4.3.2) representing the abstraction. `_abs`i is the name of a C function generated at translation time. (We use `_abs1`, `_abs2`, etc, as C names for these functions.) The translation of abstractions should also generate pieces of C code representing the incorporated action a (the C function body c_a). If a itself contains more abstractions (nested abstractions), these abstraction will generate other C functions. In the translation of 'abstraction a', the datum and bindings fields are empty. We assume that each incorporated action gives at most one individual datum (labelled n in the rule).

(ABSTRACTION)

$$\frac{\begin{array}{c}\mathcal{K} \vdash a : (\tau, \beta) \hookrightarrow (\tau', \beta') \\ A_a = \textbf{if } \tau{=}\{\} \textbf{ then } [\,] \textbf{ else } [0 \mapsto 1] \qquad b_a = \textbf{if } \beta{=}\{\} \textbf{ then } 0 \textbf{ else } 1 \\ U_{d_a} = \textbf{if } \tau{=}\{\} \textbf{ then } \{\} \textbf{ else } \{1\} \qquad U_{b_a} = \textbf{if } \beta{=}\{\} \textbf{ then } \{\} \textbf{ else } \{1\} \\ \mathcal{K}, (A_a, b_a, A_u, U_{d_a}, U_{b_a}, 0, 0, \mathit{false}, i+1, (0,0)) \vdash a \\ \twoheadrightarrow (c_a, S_a, [n \mapsto d], b'_a, f'_a, r'_a, i'_a, (d_x, b_x)) \\ c_3 = \textbf{if } \tau{=}\{\} \textbf{ then } \text{“;” } \textbf{else } \text{“\texttt{\_d1 = \_din;}”} \\ c_4 = \textbf{if } \beta{=}\{\} \textbf{ then } \text{“;” } \textbf{else } \text{“\texttt{\_b1 = \_bin;}”} \\ c_5 = \textbf{if } \tau'{=}\{\} \textbf{ then } \text{“;” } \textbf{else } \text{“\texttt{*\_dout = \_d}}d\text{;”} \\ c_6 = \textbf{if } \beta'{=}\{\} \textbf{ then } \text{“} * \texttt{\_bout} = \texttt{NULL;}\text{” } \textbf{else } \text{“\texttt{*\_bout = \_b}}b'_a\text{;”}\end{array}}{\mathcal{K}, (A, b, i) \vdash \mathsf{abstraction}\ a \twoheadrightarrow (\texttt{\_ABSTRACTION(\_abs}i\texttt{)}, i'_a, S_a \cup \{s\})} \tag{4.59}$$

where $s =$

```
int _absi (_din,_bin,_dout,_bout)
DATUM _din;
BINDINGS _bin;
DATUM * _dout;
BINDINGS * _bout;
{
DATUM _d1, ..., _ddx;
BINDINGS _b1, ..., _dbx;
c3
c4
ca
c5
c6
return(0);
_failure0:
return(1);
}
```

(WITH)

$$\frac{\mathcal{K}, (A, b, i) \vdash y_1 \twoheadrightarrow (c_1, i_1, S_1) \qquad \mathcal{K}, (A, b, i_1) \vdash y_2 \twoheadrightarrow (c_2, i_2, S_2)}{\mathcal{K}, (A, b, i) \vdash y_1 \ \mathsf{with}\ y_2 \twoheadrightarrow (\text{“}\texttt{\_WITH(}c_1\texttt{,}c_2\texttt{)}\text{”}, i_2, S_1 \cup S_2)} \tag{4.60}$$

(CLOSURE)

$$\frac{\mathcal{K}, (A, b, i) \vdash y \twoheadrightarrow (c_y, i_y, S_y)}{\mathcal{K}, (A, b, i) \vdash \mathsf{closure}\ y \twoheadrightarrow (\text{“}\texttt{\_CLOSURE(}c_y\texttt{, \_b}b\texttt{)}\text{”}, i_y, S_y)} \tag{4.61}$$

Figure 4.15: Translation rules for ‘abstraction’, ‘with’ and ‘closure’.

```
int _abs0(_din, _bin, _dout, _bout)
        DATUM _din; BINDINGS _bin; DATUM *_dout; BINDINGS *_bout;
{
        DATUM _d1; BINDINGS _b1;
        _b1 = _bin;
        _d1 = _SUCCESSOR(_BOUND("y", _b1));
        *_dout = _d1;
        *_bout = NULL;
        return (0);
_failure_0:
        return (1);
}

DATUM _d1, _d2, _d3; BINDINGS _b1, _b2;

int main()
{
        _d1 = _ABSTRACTION(_abs0);
        _b1 = _BIND("y", _MAKE_INTEGER(6));
        _d2 = _CLOSURE(_SORT_CHECK(_d1, 128L), _b1);
        if ((_d2.datum.abs->codeact)
                (_d2.datum.abs->datum,_d2.datum.abs->bindings,&_d3,&_b2))
                        goto _failure_0;
        exit(0);
_failure_0:
        exit(1);
}
```

Figure 4.16: An example of abstraction translation.

The translation of the 'with' and 'closure' operations (rules 4.60 and 4.61 respectively) relies on the run-time function `_WITH` and `_CLOSURE` respectively (see Section 4.3.2).

Now, for example, the action

| give abstraction (give successor (the datum bound to "y"))
then
| bind "y" to 6 hence enact closure (the abstraction)

whose sort is

$$(\{\}, \{\}) \hookrightarrow (\{0 : \mathsf{integer}\}, \{\})$$

translates to the C object code shown in Figure 4.16. The enaction of the incorporated action corresponds to a call to `_abs1`. This function will return in `_d3` and `_b2` the datum and bindings given and produced by the action, respectively (in this case only a datum is given by the abstraction). □

(ELSE)

$$\frac{\begin{array}{c}\mathcal{K},([0 \mapsto d]\cdot A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \twoheadrightarrow (c_1,S_1,A_1,b_1,f_1,r_1,i_1,C_1)\\ \mathcal{K},([0 \mapsto d]\cdot A,b,A_u,U_d,U_b,f_1,r_1,l,i_1,C_1) \vdash a_2 \twoheadrightarrow (c_2,S_2,A_2,b_2,i_2,(d_{x_2},b_{x_2}))\\ (A_1,A_2,U_d,d_{x_2}) \overset{rd}{\vdash} (c_3,d'_x) \qquad (b_1,b_2) \overset{rb}{\vdash} c_4\end{array}}{\mathcal{K},([0 \mapsto d]\cdot A,b,A_u,U_d,U_b,f,r,l,i,C) \vdash a_1 \ \mathsf{else}\ a_2 \twoheadrightarrow (c,S_1 \cup S_2,A_1,b_1,i_2,(d'_x,b_{x_2}))} \quad (4.62)$$

where $c =$ `if d_`d`.datum.truth_value {` c_1 `} else {` c_2 `};`
c_3
c_4

(RECURSIVELY BIND)

$$\frac{\begin{array}{c}\mathcal{K},(A,b,i) \vdash y_1 \twoheadrightarrow (c_1,i_1,S_1) \qquad \mathcal{K},(A,b_3,i_1) \vdash y_b \twoheadrightarrow (c_b,i_b,S_b)\\ b' = new\ U_b \qquad (b,b',U_b \cup \{b'\}) \overset{ov}{\vdash} (c_3,b_3,U'_b)\end{array}}{\begin{array}{c}\mathcal{K},(A,b,\_,\_,U_b,f,r,\_,i,(d_x,b_x)) \vdash \mathsf{recursively\ bind}\ y_1\ \mathsf{to}\ y_b\\ \twoheadrightarrow (c,S_1 \cup S_b,[\,],b',f,r,i_b,(d_x,greater\ b'\ b_x))\end{array}} \quad (4.63)$$

where $c =$ b' `= _BIND(`c_1`,_MAKE_UNKNOWN());`
c_3
b_3`->datum =` c_b`;`
b' `= _BIND(`c_1`,`b_3`->datum);`

- y_b is an abstraction yielder.

(ALLOCATE)

$$\frac{d = new\ U_d}{\mathcal{K},(\_,\_,\_,U_d,\_,\_,\_,\_,i,(d_x,b_x)) \vdash \mathsf{allocate}\ y \twoheadrightarrow (c,\{\},[0 \mapsto d],\{\},i,(greater\ d\ d_x,b_x))} \quad (4.64)$$

where $c =$ `_d`d `= _ALLOCATE_A_CELL();`

Figure 4.17: Translation rules for hybrid action notation.

Hybrid

Figure 4.17 shows the translation rules for hybrid action notation.

In the translation of '**recursively bind** k **to** y_b' we assume that y_b refers to a closed abstraction. The translated code is executed in four steps (Figure 4.17):

- firstly we make a binding of token k to the special value '**unknown**' and store this binding in b';
- secondly we overlay this binding on the income bindings. The produced bindings are stored in a new b-register b_3;

```
int _abs0(_din, _bin, _dout, _bout)
        DATUM _din; BINDINGS _bin; DATUM *_dout; BINDINGS *_bout;
{
        DATUM _d1, _d2, _d3; BINDINGS _b1, _b2;
        _b1 = _bin;
        _d1 = _MAKE_INTEGER(4);
        _d2 = _BOUND("f", _b1);
        if ((_d2.datum.abs->codeact)
                (_d2.datum.abs->datum,_d2.datum.abs->bindings,&_d3,&_b2))
                        goto _failure0;
        *_dout = _d3;
        *_bout = _b2;
        return (0);
_failure0:
        return (1);
}

DATUM _d1, _d2; BINDINGS _b1, _b2;

int main()
{
        _b1 = _BIND("f", _MAKE_UNKNOWN());
        _b1->datum = _CLOSURE(_ABSTRACTION(_abs0), _b1);
        _b1 = _BIND("f", _b1->datum);
        _d1 = _BOUND("f", _b1);
        if ((_d1.datum.abs->codeact)
                (_d1.datum.abs->datum,_d1.datum.abs->bindings,&_d2,&_b2))
                        goto _failure0;
        exit(0);
_failure0:
        exit(1);
}
```

Figure 4.18: Example of translation of a ‘recursively bind’ action.

- the datum field of the binding for k in b_3 is then updated to the value resulting from the evaluation of y_b (a closed abstration);
- finally, we produce a new binding of k to the abstraction bound to k in b_3 (and store it in b').

The translation of the action

```
| recursively bind "f" to closure abstraction
| | give 4 then enact the abstraction bound to "f"
hence
| enact the abstraction bound to "f"
```

is shown in Figure 4.18.

(DATA OPERATION)

$$\frac{\mathcal{K},(A,b,i) \vdash y_1 \twoheadrightarrow (c_1,i_1,S_1) \quad \ldots \quad \mathcal{K},(A,b,i_{n-1}) \vdash y_n \twoheadrightarrow (c_n,i_n,S_n) \qquad rop = op\_table\ op}{\mathcal{K},(A,b,i) \vdash op(y_1,\ldots,y_n) \twoheadrightarrow (\text{“}rop(c_1,\ldots,c_n)\text{''}, i_n, S_1 \cup \ldots \cup S_n)} \tag{4.65}$$

(DATA)

$$\mathcal{K},(A,b,i) \vdash d \twoheadrightarrow (c,i,\{\}) \tag{4.66}$$

where c = `MAKE_INTEGER(`d`)` , if d is an integer
= `MAKE_TRUTH(`d`)` , if d is a truth-value
= `"`d`"` , if d is a token

Figure 4.19: Data notation translation rules.

In Rule 4.64 `_ALLOCATE_A_CELL` is a run-time C function which allocates a free cell (see Section 4.3.2 for an explanation of this function).

Data Notation

Figure 4.19 shows how the data operations and individual data are translated. *rop* is the textual name for the run-time function corresponding to data operation *op*.

For example, the action 'give sum (3,4)' translates to

```
_d1 = _SUM(_MAKE_INTEGER(3),_MAKE_INTEGER(4));        (1)
```

where '`_SUM`' is implemented as

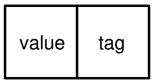
```
DATUM _SUM (x,y)
      DATUM x;
      DATUM y;
{
   DATUM z;

   z.datum.integer = x.datum.integer + y.datum.integer;
   z.tag = 4L;
   return z;
}
```

Notice that there is no run-time sort check on the arguments of `_SUM`. The result datum is tagged 4L (a C long integer) to indicate that it is of sort 'integer'.

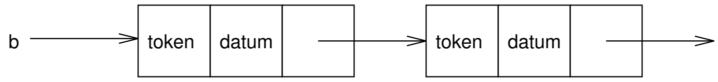
4.3.2 Run-Time Environment

One of the major objectives of the run-time environment is to give a representation for data. This is achieved by a C structure and a C union. Tags are used to do run-time sort checks. Schematically all data have the following representation (a C structure):



The value field (a C union) can hold an ordinary individual such as an integer, a truth-value, etc; a pointer to an abstraction in the case of an abstraction; a pointer to a datum in the case of a cell; and a pointer to a list in the case of a list. The tag field is a C long integer. The d-registers hold values with the above structure.

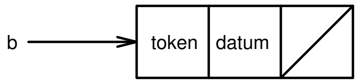
Bindings are represented as a linked list of token-datum pairs. A set of bindings is then just a pointer to the head of this list:



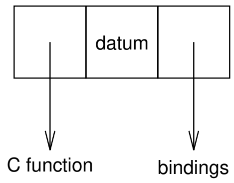
Tokens are C strings. The effect of the `_BIND` function is:

Before:

After:

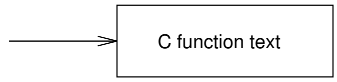


As we said before, an abstraction is represented as a C structure with three fields. The first one is a pointer to a C function, the one which resulted from the translation of the incorporated action. The second field is a datum which can be supplied by the '**with**' operation. The third field is a pointer to a set of bindings, the ones which can be provided by a '**closure**' operation. Schematically:

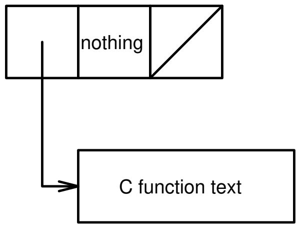


The structure above is created by the `_ABSTRACTION` run-time function which takes a pointer to a C function, and returns an abstraction (a C structure). The "datum" and "bindings" fields of the returned structure are initialized to `NOTHING` and `NULL` respectively. The structure is used in places where we want to assign an abstraction to a datum register (assign a C structure to a d-register). Schematically:

Before:



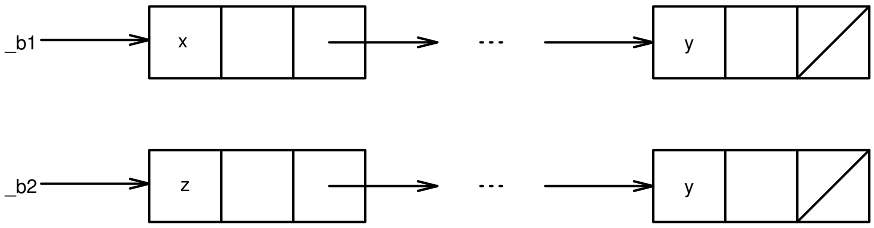
After:



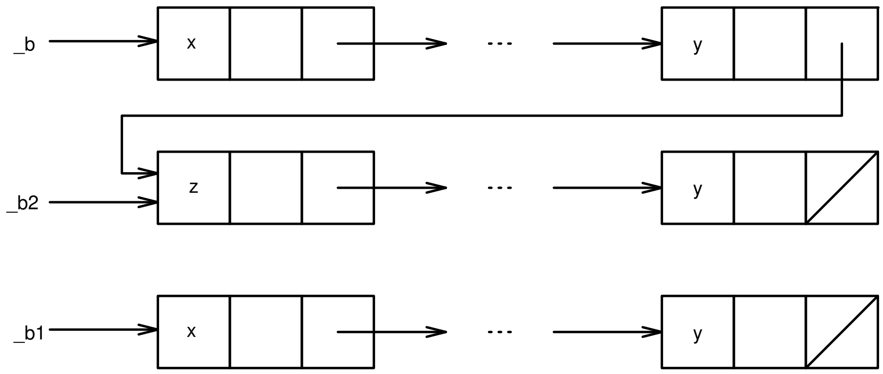
The '**with**' and '**closure**' operations are implemented as run-time operations. `_WITH` takes an abstraction and a datum, and gives an abstraction with the given datum in the datum field. If the given abstraction has already a proper datum in its field, `_WITH` will return the given abstraction. `_CLOSURE` takes an abstraction and a set of bindings, and gives an abstraction with the given bindings in the binding field. If the given abstraction has already a proper set of bindings in its field, `_CLOSURE` will return the given abstraction. Notice that as the `_BIND` function makes a new binding every time it is called, all previous bindings are preserved, and the `_CLOSURE` operation needs only to store a pointer to them in the third field of its argument abstraction.

An important run-time operation is `_OVERLAY_BINDINGS`. Schematically, this operation has the following effect (for `_b = _OVERLAYS_BINDINGS(_b1,_b2)`):

Before:



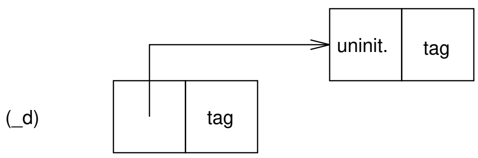
After:



Data notation operations (like sum, product, concatenation, etc) are also part of the run-time environment as well as all the other run-time C functions present in the translation rules (such as `_BIND`, `_ALLOCATE_A_CELL`, etc). The storage is implemented as a heap (the C `malloc` function is used to allocate cells). The effect of `_d = _ALLOCATE_A_CELL()` can be seen as:

Before:

After:



One important aspect of the run-time environment is the necessity of run-time sort checks. This is supported by a run-time sort checking function (*_SORT_CHECK*). Consider the yielder 'the $S\#n$' and suppose the sort information inferred for the input transients is

$$\{n : s\}$$

then, if

$s \leq S$	$\Rightarrow$	term well-sorted
s & S = nothing	$\Rightarrow$	term ill-sorted
otherwise	$\Rightarrow$	do run-time sort checking.

As we said before, the sort checker signals to the code generator points in the action tree where these run-time checks are needed. It provides also the sort that should be checked against at run time. For a particular datum d the run-time sort checking function simply checks d's run-time tag against the one provided by the sort checker. For example, the translation of

```
| give 2 or give true
then
| give the truth-value
```

requires a run-time sort check which will be present in the generated code as a call to *_SORT_CHECK*:

```
          _d1 = _MAKE_INTEGER(2);
          goto _end_2;
_failure_1:
          _d1 = _MAKE_TRUTH(true);
_end_2:
          _d2 = _SORT_CHECK(_d1, 2L);
```

if the value assigned to *_d1* is of sort 'truth-value' (2L) then *_SORT_CHECK* returns this value, otherwise it exits abnormally (*exit*(1)).

4.3.3 Implementation

The implementation of the action notation code generator is done by a top-down traversal of the decorated action tree. In fact it is very similar to the translation rules. We have two code functions (*code_action* and *code_yielder*) corresponding to our two judgements (judgements 4.29 and 4.30). Also, as each conclusion is unique we can use pattern matching to implement the rules without having to rely on any dirty trick. For a comparison we give in Figure 4.20 the STANDARD ML code for a part of the code generator which implements the translation rules of Figure 4.12. Sort information ($\mathcal{K}$), which is not relevant for the rules shown, is the third argument of the action tree constructors (*BIND_TO*, *HENCE* and *BOUND_TO*).

4.3.4 Limitations

- **Data given by abstractions.** We assume the following restriction on data given by abstractions (not present in standard action notation): an abstraction can only give a single datum. This simplifies the way the code generator implements the translation of abstractions. In practice this restriction does not introduce any problem because

(BIND)

```
| code_action (BIND_TO(y1,y2,_)) (A,b,_,_,Ub,f,_,_,i,(dx,bx)) =
  let
      val (b',_) = new Ub
      val C_b = code_breg b'
      val (c1,i1,S1) = code_yielder y1 (A,b,i)
      val (c2,i2,S2) = code_yielder y2 (A,b,i1)
      val c = C_b ^ " = _BIND(" ^ c1 ^ "," ^ c2 ^ ");"
  in
      (c,union S1 S2,emptymap,b',f,i2,(dx,greater b' bx))
  end
```

(HENCE)

```
| code_action (HENCE(a1,a2,_)) (A,b,Au,Ud,Ub,f,r,l,i,C) =
  let
      val (c1,S1,A1,b1,f1,i1,C1) = code_action a1 (A,b,Au,Ud,Ub,f,r,l,i,C)
      val (c2,S2,A2,b2,f2,i2,C2) =
          code_action a2 (A,b1,Au,union Ud (range A1),singleton b1,f1,r,l,i1,C1)
      val c = c1 ^ c2
  in
      (c,union S1 S2,merge A1 A2,b2,f2,i2,C2)
  end
```

(BOUND)

```
| code_yielder (BOUND_TO(s,y,_)) (A,b,i) =
  let
      val (c',i',S') = code_yielder y (A,b,i)
      val C_b = code_breg b
      val c = "_BOUND(" ^ c' ^ "," ^ C_b ^ ")"
  in
      (c,i',S')
  end
```

Figure 4.20: The implementation of some translation rules of Figure 4.12.

more than a single datum can be returned using a list. The same point applies for data given to abstractions.

- **Tail recursion for unfolding.** The code generator assumes that all occurrences of '**unfold**' are tail recursive. This condition is however tested by the code generator.

- **Source language data operations.** The user of ACTRESS must provide code for the source language data operations not pre-defined in ACTRESS data notation. Such code must be included in the run-time environment. Because a data operation table (see Rule 4.65) is used by the code generator to translate the name of data operations to the name of the corresponding C function which implements the operation, the user must update this table every time a new data operation is needed.

There are two cases where information provided by the sort checker was essential in the translation process: the translation of '**unfolding**' and '**unfold**' actions, and the translation of the '**abstraction**' operation. Also, the introduction of run-time sort checks is guided by information given by the sort checker.

4.4 The Actioneer Generator

The objective of the actioneer generator is to generate a program, the *actioneer for* $\mathcal{L}$, from the action semantic description of $\mathcal{L}$. The actioneer for $\mathcal{L}$ incorporates $\mathcal{L}$ action semantics into a generated compiler for $\mathcal{L}$. The semantic function r for an $\mathcal{L}$ program $\mathcal{P}$, present in the actioneer, when applied to the abstract syntax tree of $\mathcal{P}$, gives the program action for $\mathcal{P}$.

Figure 4.21 shows a piece of the actual input to the actioneer generator used for the generation of an actioneer for SPECIMEN (compare with the semantic description in Appendix B). Figure 4.22 shows the part of the actioneer for SPECIMEN which corresponds to the semantic equations of Figure 4.21 (STANDARD ML code). The generated actioneer is an ML program which defines a set of mutually recursive functions. Each function corresponds to a semantic function in the semantic description, and is defined on the ML datatype which represents SPECIMEN's abstract syntax. Notice that the semantic function **elaborate** is translated to the ML function *elaborate*. We assume that there is no overloading of semantic functions.

```
3.2.1   Elaborating Declarations

(1)     elaborate _ :: Declaration -> action .

(2)     elaborate [[ CONST I:Identifier T:Type E:Expression ]] =
                | evaluate E
                then
                | bind tokenOf I to the value .

(3)     elaborate [[ VAR I:Identifier T:Type E:Expression ]] =
                | | evaluate E then give the value label #1
                | and
                | | allocateForPrimitiveValue T then give the cell label #2
                then
                | | bind tokenOf I to the cell#2
                | and
                | | store the value#1 in the cell#2 .
```

Figure 4.21: Two semantic equations for SPECIMEN's declarations (actual input).

The implementation of the actioneer generator was very straightforward as long as we had defined an abstract syntax for action semantic descriptions.

4.5 The Action Notation Interpreter

The action notation interpreter [84], ANI, takes an action and interprets it giving an interpretation output. Interpretation of an action corresponds to its performance. The interpretation output is basically a triple representing transients and bindings given and produced respectively by the action, and the state of storage after action performance. Also an outcome status — *completed*, *escaped* or *failed* — is indicated. When an action diverges, its interpretation also diverges, which causes the interpreter to loop. Finally, the ouptut includes the commitment flag, *committed* or *uncommitted*. (The output can be sometimes an error message reporting some illegal condition arisen during interpretation).

4.5.1 Interpreting Actions

The interpreter, which includes the action notation abstract syntax, and an interpreting function that takes an action and interprets it, makes use of two auxiliary functions to interpret an action term: *step* and *propagate*. A *state* is formed by the action to be interpreted, together with the transients, bindings and storage to be received by that

```
and

(* elaborate :: Declaration -> action *)

elaborate (CONST (I, T, E)) = ActionAST.THEN (evaluate E,ActionAST.BIND_TO (t
okenOf I,ActionAST.THE (ActionAST.NAME ("value",()),0,()),()),())

|

elaborate (VAR (I, T, E)) = ActionAST.THEN (ActionAST.AND (ActionAST.THEN (ev
aluate E,ActionAST.GIVE (ActionAST.THE (ActionAST.NAME ("value",()),0,()),1,(
)),()),ActionAST.THEN (allocateForPrimitiveValue T,ActionAST.GIVE (ActionAST.
THE (ActionAST.NAME ("cell",()),0,()),2,()),()),()),ActionAST.AND (ActionAST.
BIND_TO (tokenOf I,ActionAST.THE (ActionAST.NAME ("cell",()),2,()),()),Action
AST.STORE_IN (ActionAST.THE (ActionAST.NAME ("value",()),1,()),ActionAST.THE
(ActionAST.NAME ("cell",()),2,()),()),()),())

|
```

Figure 4.22: The actioneer for SPECIMEN (part).

action. (The *initial state* is formed by the top level action, no transients, no bindings and an empty store). A *step* is an interpretation outcome; it is formed by the outcome status, transients, bindings, storage and the commitment flag.

The function *step* takes a state, interprets it and gives a step. For example, the following is the definition of *step* for the '**give**' action[3]:

$$
\begin{array}{l}
\textit{step State}\,(\textsf{give}\ y\ \textsf{label}\ \#n, t, b, s) = \\
\quad \textbf{let} \\
\qquad \begin{array}{ll}
f\ d & = \textit{Step}\,(\textit{Completed}, \{n \mapsto d\}, \{\}, s, \textit{Uncommitted}) \\
f\ \textit{Sort}\ (\textit{Individual}\ d) & = f\ d \\
f\ \textit{Nothing} & = \textit{Step}\,(\textit{Failed}, \{\}, \{\}, s, \textit{Uncommitted}) \\
f\ \textit{Sort}\ (\_) & = \textit{Step}\,(\textit{Error}, \texttt{proper sort in give action})
\end{array} \\
\quad \textbf{in} \\
\qquad f\ (\textit{evaluate}\ y\ t\ b\ s) \\
\quad \textbf{end}
\end{array}
$$

The function *evaluate* evaluates a yielder term, yielding a datum. The function f is used to handle the result of the evaluation of y; for example, if y evaluates to '**nothing**' (*Nothing*) the result of interpretation of the '**give**' action is $\textit{Step}\ (\textit{Failed}, \{\}, \{\}, s, \textit{Uncommitted})$. The step $\textit{Step}\ (\textit{Error}, \textit{msg})$ is exceptional; when it is reached the interpretation is aborted and message *msg* is printed.

[3]The presentation notation is inspired by ML syntax.

For states containing a compound infix action, we use the auxiliary function *propagate*, which takes two *stepped actions* composed by an infix combinator (a *stepped infix action*), composes them and gives a new step. For example, for the action 'a_1 then a_2', *step* is defined as:

$$
\begin{array}{l}
\mathit{step}\ \mathit{State}\,(a_1\ \mathtt{then}\ a_2, t, b, s) = \\
\quad \mathbf{let} \\
\qquad \mathit{Step}\,(r_1, t_1, b_1, s_1, c_1) = \mathit{step}\ \mathit{State}\,(a_1, t, b, s) \\
\quad \mathbf{in} \\
\qquad \mathbf{if}\ r_1 = \mathit{Completed} \\
\qquad\ \ \mathbf{then}\ \mathit{propagateThen}\ \mathit{Step}\,(r_1, t_1, b_1, s_1, c_1)\ (\mathit{step}\ \mathit{State}\,(a_2, t_1, b, s_1)) \\
\qquad \mathbf{else\ if}\ r_1 = \mathit{Escaped} \\
\qquad\qquad \mathbf{then}\ \mathit{Step}\,(r_1, t_1, \{\}, s_1, c_1) \\
\qquad\quad\ \ \mathbf{else}\ \mathit{Step}\,(r_1, \{\}, \{\}, s_1, c_1) \\
\quad \mathbf{end}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{propagateThen} \\
\quad \mathit{Step}\,(\mathit{Completed}, t_1, b_1, s_1, c_1) \\
\quad \mathit{Step}\,(\mathit{Completed}, t_2, b_2, s_2, c_2) = \mathit{Step}\,(\mathit{Completed}, t_2, \mathit{merge}\ b_1\ b_2, s_2, \mathit{commit}\ c_1\ c_2) \\
\\
\mathit{propagateThen} \\
\quad \mathit{Step}\,(\mathit{Completed}, t_1, b_1, s_1, c_1) \\
\quad \mathit{Step}\,(\mathit{Escaped}, t_2, b_2, s_2, c_2) \quad = \mathit{Step}\,(\mathit{Escaped}, t_2, \{\}, s_2, \mathit{commit}\ c_1\ c_2) \\
\\
\mathit{propagateThen} \\
\quad \mathit{Step}\,(\mathit{Completed}, t_1, b_1, s_1, c_1) \\
\quad \mathit{Step}\,(\mathit{Failed}, t_2, b_2, s_2, c_2) \qquad = \mathit{Step}\,(\mathit{Failed}, t_2, \{\}, s_2, \mathit{commit}\ c_1\ c_2) \\
\\
\mathit{propagateThen} \\
\quad \_ \\
\quad \_ \qquad\qquad\qquad\qquad\qquad = \mathit{Step}\,(\mathit{Error}, \mathtt{cannot\ propagate\ stepped\ actions})
\end{array}
$$

The *commit* function computes the commitment flag for a new step combining the commitment flags of two given steps as below:

$$
\begin{array}{lll}
\mathit{commit}\ \mathit{Uncommitted} & \mathit{Uncommitted} & = \mathit{Uncommitted} \\
\mathit{commit}\ \mathit{Committed} & \_ & = \mathit{Committed} \\
\mathit{commit}\ \_ & \mathit{Committed} & = \mathit{Committed}
\end{array}
$$

Finally, the *evaluate* function, as we said before, evaluates a yielder giving a datum. We show below how the yielder 'the $S\#n$' is evaluated:

$$
\begin{array}{ll}
\mathit{evaluate}\ (\mathsf{the}\ S\#n)\ t\ b\ s = & \textbf{let} \\
 & \quad S' = \mathit{evaluateSort}\ S\ t\ b\ s \\
 & \quad d = t\ \mathit{at}\ n \\
 & \textbf{in} \\
 & \quad \textbf{if}\ d\ \mathit{isOfSort}\ S'\ \textbf{then}\ d\ \textbf{else}\ \mathit{Nothing} \\
 & \textbf{end}
\end{array}
$$

Transients, bindings and storage (as well as the commitment flag) are propagated throughout the interpretation process. In the following paragraphs we explain in more detail the interpretation of some action notation terms.

Basic

The performance of the '**commit**' action completes, gives no transients, produces no bindings, does not change storage and changes the commitment flag to *Committed*. Thus:

$$\mathit{step\ State}\,(\mathsf{commit}, t, b, s) = \mathit{Step}\,(\mathit{Completed}, \{\}, \{\}, s, \mathit{Committed})$$

We have used some action notation laws ([80]) to implement the interpretation of some actions. For example, the law

$$\mathsf{diverge} = \mathsf{unfolding\ unfold}$$

was directly applied to interpret the action '**diverge**' as follows

$$\mathit{step\ State}\,(\mathsf{diverge}, t, b, s) = \mathit{step\ State}\,(\mathsf{unfolding\ unfold}, t, b, s)$$

The interpretation of '**unfolding** a' is equivalent to the interpretation of a with all unfold actions in a replaced by '**unfolding** a'. The replacement is done using the auxiliary function *unfold*:

$$\mathit{step\ State}\,(\mathsf{unfolding}\ a, t, b, s) = \mathit{step\ State}\,(\mathit{unfold}\ a\ a, t, b, s)$$

where *unfold* is defined as follows:

$$
\begin{array}{ll}
\mathit{unfold}\ \mathsf{unfold} & a = \mathsf{unfolding}\ a \\
\mathit{unfold}\ (\mathsf{unfolding}\ a_1) & a = \mathsf{unfolding}\ \ a_1 \\
\mathit{unfold}\ (\mathsf{indivisibly}\ a_1) & a = \mathsf{indivisibly}\ (\mathit{unfold}\ a_1\ a) \\
\mathit{unfold}\ (a_1\ \mathsf{or}\ a_2) & a = (\mathit{unfold}\ a_1\ a)\ \mathsf{or}\ (\mathit{unfold}\ a_2\ a) \\
\mathit{unfold}\ (a_1\ \mathsf{and}\ a_2) & a = (\mathit{unfold}\ a_1\ a)\ \mathsf{and}\ (\mathit{unfold}\ a_2\ a) \\
\ldots & \\
\mathit{unfold}\ (a_1\ \mathsf{before}\ a_2) & a = (\mathit{unfold}\ a_1\ a)\ \mathsf{before}\ (\mathit{unfold}\ a_2\ a) \\
\mathit{unfold}\ a_1 & a = a_1, \text{ where } a_1 \text{ is a primitive action}
\end{array}
$$

The action 'a_1 or a_2' is interpreted as follows. A randomly-generated number is used to determine which subaction should be interpreted. The other subaction is only interpreted (performed) if the chosen one fails and is uncommitted. This solution gives an interesting flavour of non-determinism for the 'or' combinator.

Reflective

An abstraction is represented by a value $Abstraction(a, t, b)$, which is basically a triple, where a is the incorporated action, t is transients, and b is bindings. These three fields are supplied by the 'abstraction', 'with', and 'closure' operations respectively:

$$evaluate\ (\mathsf{abstraction}\ a)\ t\ b\ s = Abstraction\,(a, \{\}, \{\})$$

$$\begin{array}{l}
evaluate\ (y_1\ \mathsf{with}\ y_2)\ t\ b\ s = \\
\quad \mathbf{let} \\
\qquad f\ Abstraction\,(a, t', b') \qquad\quad = (a, b', \{\}) \\
\qquad f\ Closure\,(a, t', b', s', brec') = (a, b', brec') \\
\qquad f\ d \qquad\qquad\qquad\qquad\quad\;\; = Step\,(Error, \texttt{not an abstraction}) \\
\qquad (a, b', brec') = f\ (evaluate\ y_1\ t\ b\ s) \\
\qquad d_2 = evaluate\ y_2\ t\ b\ s \\
\quad \mathbf{in} \\
\qquad Abstraction\,(\mathsf{give}\ d_2\ \mathsf{then}\ a, \{\}, REC\ (overlay\ brec'\ b')) \\
\quad \mathbf{end}
\end{array}$$

$$\begin{array}{l}
evaluate\ (\mathsf{closure}\ y)\ t\ b\ s = \\
\quad \mathbf{let} \\
\qquad f\ Abstraction\,(a, t', \{\}) \qquad\quad = Abstraction\,(a, t', b) \\
\qquad f\ Abstraction\,(a, t', b') \qquad\quad = Abstraction\,(a, t', b') \\
\qquad f\ Closure\,(a, t', b', s', brec') = Closure\,(a, t', b', s', brec') \\
\qquad f\ _ \qquad\qquad\qquad\qquad\quad\;\; = Step\,(Error, \texttt{not an abstraction}) \\
\qquad y' = evaluate\ y\ t\ b\ s \\
\quad \mathbf{in} \\
\qquad f\ y' \\
\quad \mathbf{end}
\end{array}$$

The 'enact' action is interpreted as follows:

$$\begin{array}{l}
step\ State\ (\mathsf{enact}\ y, t, b, s) = \\
\quad \mathbf{let} \\
\qquad f\ Abstraction\,(a, t', b') = step\ State\,(a, t', b', s) \\
\qquad f\ _ \qquad\qquad\qquad\qquad\; = Step\,(Failed, \{\}, \{\}, s, Uncommitted) \\
\quad \mathbf{in} \\
\qquad f\ (evaluate\ y\ t\ b\ s) \\
\quad \mathbf{end}
\end{array}$$

Hybrid

The interpretation of the action 'recursively bind k to y_b' is just to bind k to the closure value $Closure\,(y_b, t, b, s, r)$. The contents of the closure is the *unevaluated* abstraction yielder y_b, transients t, bindings b, storage s and a recursive binding r. The recursive binding component r is just a binding of k to a similar closure in which the recursive binding component is empty. (The effect is that the recursive binding component is a pointer to the binding.)

$$
\begin{array}{l}
step\ State\,(\textsf{recursively bind}\ y\ \textsf{to}\ y_b, t, b, s) = \\
\quad \mathbf{let} \\
\qquad f\ TokenE\,(k) = TokenE\,(k) \\
\qquad f\ \_ \qquad\qquad = Step\,(Error, \texttt{not a token in recursively bind action}) \\
\qquad token = f\ (evaluate\ y\ t\ b\ s) \\
\qquad r = \{token \mapsto Closure\,(y_b, t, b, s, \{\})\} \\
\qquad closure = Closure\,(y_b, t, b, s, r) \\
\quad \mathbf{in} \\
\qquad Step\,(Completed, \{\}, \{token \mapsto closure\}, s, Uncommitted) \\
\quad \mathbf{end}
\end{array}
$$

Every time a yielder evaluates to a $Closure\,(y_b, t, b, s, r)$ we evaluate y_b with income $(t, REC(overlay\ r\ b), s)$. The unrolling operation REC is defined as:

$$
\begin{array}{lll}
REC\,\{\} & = & \{\} \\
REC\,b & = & \mathbf{let} \\
 & & \quad d = domainOf\,b \\
 & & \quad r = rangeOf\,b \\
 & & \quad r' = map\ (recVE\ b)\ r \\
 & & \mathbf{in} \\
 & & \quad zip\ d\ r' \\
 & & \mathbf{end}
\end{array}
$$

where

$$
\begin{array}{ll}
recVE\ b'\ (Closure\,(y_b, t, b, s, r)) & = Closure\,(y_b, t, b, s, b') \\
recVE\ b'\ d & = d
\end{array}
$$

Another example of the use of laws in the interpreter is the implementation of the action 'allocate a cell' from the law

```
allocate c ≤ cell =
    indivisibly
        | | choose a cell [not in domain of current storage]
        | then
        | | reserve it and give it .
```

which was defined as below:

```
step State (allocate y, t, b, s) =
    step State (indivisibly (choose (y & cell-not-in (domain-of storage))
                             then
                             (reserve it and give it)),
                t, b, s)
```

where 'cell-not-in' is an auxiliary function that gives a cell that is not in a given set.

Data Notation

We also implemented many of the operations present in data notation (such as **sum**, **union**, etc). Some of them were used in the definition of the interpreting function itself, for example the function *merge* in the definition of *propagateThen*.

4.5.2 Limitations

Interleaving of action performance is not implemented. As a consequence actions like 'a_1 and a_2' are specialized to the case where the whole of a_1 is interpreted before the interpretation of a_2. For this reason, the behaviour of 'a_1 and a_2' is equivalent to 'a_1 and then a_2'. This can be a problem in cases where a_1 diverges and a_2 fails; in ANI such actions will diverge rather than fail (as it could happen if a more genuine interleaving interpretation were used). To minimize such effect we could choose randomly the subaction to be performed first. Due to the absence of interleaving the action 'indivisibly a' is the same as a.

Actions like 'choose a natural' is implemented, although it gives the same natural individual every time it is performed. In general, 'choose y' will give an individual of the sort yielded by the evaluation of y, if this is a subsort of distinct datum.

ANI implements only a fixed set of sorts, the standard ones that are subsort of datum. The sort 'cell [s]' is not implemented. User-defined sorts cannot be included in the present

datatype *Program*	= *PROGRAM* **of** *Identifier* ∗ *Declaration* ∗ *Command*	
and *Declaration*	= *CONST* **of** *Identifier* ∗ *Type* ∗ *Expression*	
	| *VAR* **of** *Identifier* ∗ *Type* ∗ *Expression*	
	| *PROC* **of** *Identifier* ∗ *Formals* ∗ *Command*	
	| *FUN* **of** *Identifier* ∗ *Formals* ∗ *Type* ∗ *Expression*	
	| *DECLSEQ* **of** *Declaration* ∗ *Declaration*	
and *Formals*	= *EMPTYFORMAL*	
	| *FORMAL* **of** *Formal*	
	| *FORMALSEQ* **of** *Formal* ∗ *Formals*	

Figure 4.23: A fragment of SPECIMEN's abstract syntax in STANDARD ML.

implementation. ANI reports an error when it tries to evaluate an unknown sort. Yielders such as

the (integer | truth-value)

are properly interpreted (or even 'the 1'!).

Communicative actions are not present in the current implementation.

4.5.3 Generating an Interpreter for SPECIMEN

Although ANI was designed to interpret arbitrary actions, it has proved to be a useful tool to interpret program actions. In order to achieve this, we use ANI in conjunction with the actioneer generator. We represent the abstract syntax of a language by a STANDARD ML datatype, which should be provided by the user. Figure 4.23 shows a fragment of SPECIMEN abstract syntax defined in such a way. To illustrate the use of ANI in conjunction with the actioneer generator we give a concrete example using SPECIMEN.

Suppose we want to interpret the factorial program shown in in Figure 4.24 which calculates the factorial of 10. Firstly we use a parser generator to build a parser for SPECIMEN. (We used ML-YACC for this purpose [101].) This parser must give an instance of the ML datatype as the representation of the parsed program. Secondly, we obtain an actioneer for SPECIMEN by application of the actioneer generator to SPECIMEN's action semantic description (Equation 4.5). Applying the parser to the source program gives the program's abstract syntax tree. Now we can apply the semantic function *run*, present in the actioneer, to this tree to obtain the program action for the *factorial* program. (We

```
program factorial is
   var
      y : int := 0;
   proc
      fact (n : int) =
         if (n = 0)
            then y := 1
         else
            call fact (n - 1);
            y := n * y
         end
in
   call fact (10)
end
```

Figure 4.24: The SPECIMEN *factorial* program.

could unparse the program action with a pretty printer for action notation, which would give an output like the program action for *loop* in Figure 4.2.) Finally, we use ANI to interpret the program action. The following is the interpretation outcome:

Outcome	Completed!
Transients	empty-transients
Bindings	empty-bindings
Storage	{ cell0 **-->** 3628800 }
	{ cell11 **-->** 0 }
	{ cell10 **-->** 1 }
	{ cell9 **-->** 2 }
	{ cell8 **-->** 3 }
	{ cell7 **-->** 4 }
	{ cell6 **-->** 5 }
	{ cell5 **-->** 6 }
	{ cell4 **-->** 7 }
	{ cell3 **-->** 8 }
	{ cell2 **-->** 9 }
	{ cell1 **-->** 10 }
Commitment	Committed action!

Notice that this shows the global effect of the program, that is, no transients are given, no bindings are produced and the factorial of 10 is stored in cell 0 (the cell allocated to hold the content of variable y). Also notice that 11 additional cells were allocated during the computation to hold the argument to the *fact* procedure. (These cells were not deallocated.) □

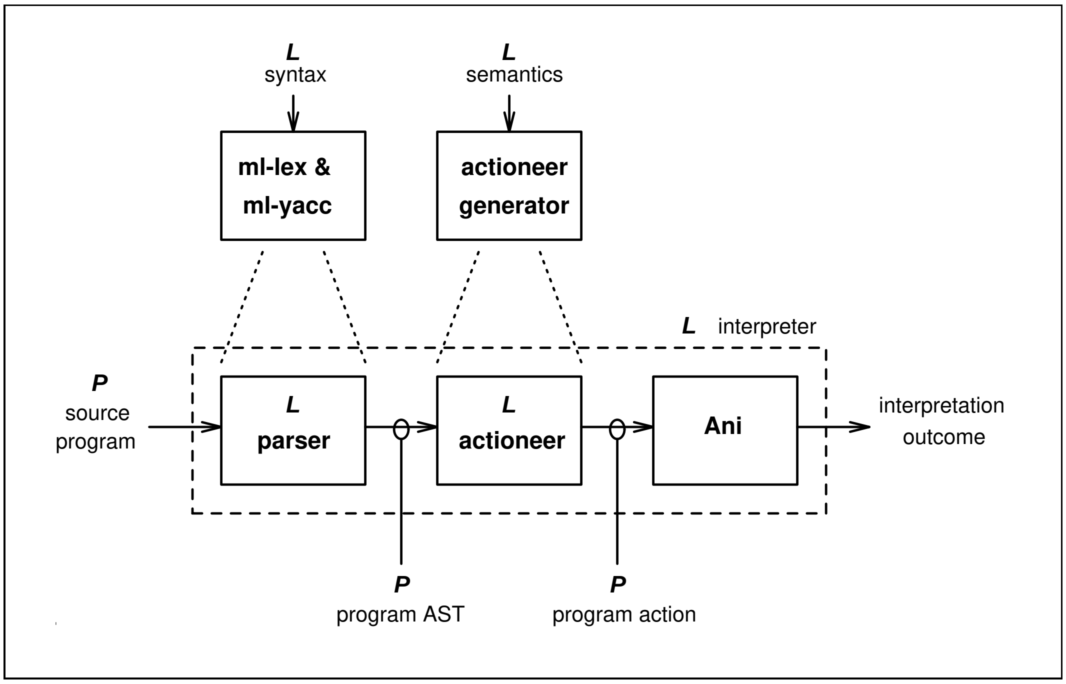


Figure 4.25: ANI, the actioneer generator and an interpreter for $\mathcal{L}$.

Figure 4.25 shows the architecture of the whole system. The current version implements most of the standard action notation[4]. It has a good user interface with good error messages. Some nice functions (tools) such as the unparsing function are provided. In Figure 4.26 we present a signature with some of the types and functions defined in the implementation. ANI has been used for some students in introductory courses to action semantics and we think it is a good tool to present to new *actioneers*. Besides SPECIMEN, we have defined an interpreter for a subset of STANDARD ML which includes higher-order functions.

4.6 Generating a Compiler for SPECIMEN

ACTRESS's generated compilers have four components. The following are the steps required to generate these components for SPECIMEN (or any other language):

- The first step is to provide an action semantic description for SPECIMEN. This will be the actual input to the actioneer generator as in Equation 4.5. In the current

[4] In fact we have two versions of the action notation interpreter: one for ACTRESS action notation and other for standard action notation.

```
signature Ani =
  sig
    type ast
    type datum and sort and transients and bindings and storage
    datatype state       = State of ast * transients * bindings * storage
    and commitment       = Uncommitted | Committed
    and step             = Step of outcomeStatus * transients * bindings * storage *
                                commitment
    and outcomeStatus    = Completed | Escaped | Failed
    val <                : sort * sort -> bool
    val &                : sort * sort -> sort
    val individualToSort : datum -> sort
    val sortToIndividual : sort -> datum
    val evaluate         : AST -> transients -> bindings -> storage -> datum
    val evaluateSort     : ast -> sort
    val propagateThen    : step -> step -> step
    val step             : state -> step
    val interp           : ast -> unit
  end
```

Figure 4.26: A signature with some of ANI's types and functions.

version of the actioneer generator, this input is an ASCII version of the description in Appendix B.

- A parser for SPECIMEN is obtained using the ML-YACC parser generator. At this stage, an abstract syntax for SPECIMEN in terms of an ML datatype must be given. Although this datatype could be automatically derived from the semantic description, we did not implement this derivation. The parser is the first component of the generated compiler.

- Now we can use Equation 4.5 to obtain an actioneer for SPECIMEN which will constitute the second component of the generated compiler. (Note that this and the previous step are the same steps as in generating an interpreter.)

- The third and fourth components are ANC's sort checker and code generator respectively.

The C object code produced by ACTRESS's compilers can be compiled using any standard C compiler (we have used GNU C compiler [99] in our experiments). One could see this as a fifth component of the generated compiler. Figure 4.27 shows the generated C object

```
DATUM _d1, _d2, _d3, _d4; BINDINGS _b1, _b2, _b3;

int main()
{
        _d1 = _MAKE_INTEGER(1000000);
        _b1 = _BIND("n", _d1);
        _d1 = _MAKE_INTEGER(0);
        _d2 = _ALLOCATE_A_CELL();
        _b2 = _BIND("x", _d2);
        *_d2.datum.cell = _d1;
        _b3 = _OVERLAY_BINDINGS(_b2, _b1);
        _d1 = _BOUND("n", _b3);
        *_BOUND("x", _b3).datum.cell = _d1;

_repeat_1:
        _d1 = *_BOUND("x", _b3).datum.cell;
        _d2 = _MAKE_INTEGER(0);
        _d3 = _IS_GREATER_THAN(_d1, _d2);
        if (_d3.datum.truth_value) {
                _d1 = *_BOUND("x", _b3).datum.cell;
                _d2 = _MAKE_INTEGER(1);
                _d4 = _DIFFERENCE(_d1, _d2);
                *_BOUND("x", _b3).datum.cell = _d4;
                goto _repeat_1;
        } else {
                ;                       /* complete */
        };
        exit(0);
_failure_0:
        exit(1);
}
```

Figure 4.27: Object code obtaind by compilation of the *loop* program.

code for the *loop* program of Figure 4.3. This is the output of an ACTRESS compiler which was generated from the semantic description for SPECIMEN given in Appendix B.

4.7 Improving ACTRESS

We measured the run-time of the *loop* program of Figure 4.3 against the run-time of a similar PASCAL program compiled with a hand-crafted compiler. Timing the running time with the UNIX `time` command we obtained:

SPECIMEN	`23.300u`	`0.090s`	`0:23.88`	`97.9%`	`0+99k`	`0+0io`	`0pf+0w`
PASCAL	`0.580u`	`0.150s`	`0:00.74`	`98.6%`	`0+164k`	`0+0io`	`0pf+0w`

That is, the object code generated by the ACTRESS SPECIMEN compiler runs approximately 40 times slower than the object code generated by the hand crafted PASCAL compiler.

In the next chapter we will address the subject of *action transformations*. The use of these transformations in ACTRESS will improve significantly the quality of the object code of its generated compilers.

Chapter 5

Binding Elimination

Assignment of a value to a variable is a feature found, in some form, in almost every high-level programming language. Its syntax varies slightly from one language to another: some use "`:=`" to indicate the assignment operation, others "`=`" or "`<-`"; some require the statement to begin with a keyword, such as "`SET`" or "`LET`"; and some require it to be terminated with a "`;`" or some other separator. But these syntactic differences are minor. There is much greater variation in semantics. In some languages, the storage area or cell referred to by a variable is fixed throughout the execution of a program, while in others it may vary in size, internal structure, or location at various times during execution, under either explicit or purely implicit programmer control. In some languages, one may obtain as a value a reference or pointer to a variable and manipulate it, including assigning a reference to one variable as the value of another, while in others such references are completely hidden from the programmer.

Neil Jones and Steven Muchnick, 1978, in [51].

This chapter presents a technique for binding elimination in action notation. We start by giving some motivation and by explaining how we identify known and unknown datum bound to identifiers. We then present how 'allocate' actions are classified, which corresponds to compile time storage allocation. After an intuitive introduction to transient elimination and binding elimination, we formalise both using what we call elimination rules. Some examples are given. The implementation is then described. Finally, we close the chapter by exploring relationships between the different formalisations for action notation, and by listing some improvements to the current work.

5.1 Motivation

In a conventional hand-written compiler for a statically scoped language, whenever possible identifiers are replaced by the values they denote at compile time. For example, a constant declaration associates a value to an identifier. This value can be, in general, *known* or *unknown* at compile time. If the value is known, the compiler just replaces every scoped occurrence of the constant identifier by its value and then eliminates the binding established by the declaration. If the value is unknown, we firstly replace the identifier-value association by an identifier-location and a location-value association. (Although the compiler does not know the value, it can determine the location, at compile time.) Finally, after replacing all scoped occurrences of the constant identifier by a location lookup, the identifier-location association (binding) can be eliminated. Note that a location (cell) was allocated at compile time. Moreover, probably there is no mention of this location in the original semantics of the constant declaration.

We call *binding elimination* the process by which identifier-value associations are eliminated from a program at compile time.

A variable declaration is also a good illustration of some aspects of binding elimination. Usually, a variable declaration (as in PASCAL) establishes a binding of an identifier to a memory location. If the compiler knows this location, it replaces all scoped *l-occurrences* of the identifier by the location; and all scoped *r-occurrences* by the contents of the location (in fact a location lookup). If the exact location is unknown, for example in the case of a variable local to a procedure, the compiler makes this unknown location into a known *relative* location and then, using this known relative location, replaces the scoped occurrences of the variable identifier as in the known case. In both cases the compiler can eliminate the original binding.

As a final illustration, consider a procedure declaration (as in PASCAL). The object code, which results from the translation of the procedure's body, is referenced by a location that is known at compile time (actually the location of the first instruction of the object code). So all subsequent occurrences of the procedure identifier (in procedure calls) are replaced by subroutine jumps to that location. After replacing all procedure identifiers, the identifier-procedure binding can be eliminated. Notice that, at run-time, when the actual jump is made, all the data accessed/manipulated by the procedure's code is in place and accessible through a pre-determined mechanism.

Our ultimate goal is total binding elimination, that is, the complete elimination of the declarative facet from the program action, for statically scoped languages. In terms of action notation, we can think of transforming each 'bind' action present in the program action: either by eliminating it or by transforming it into an imperative action. Also transforming every 'the s bound to k' which "corresponds" to the transformed 'bind' action.

Let us examine in more detail the object code generated by the SPECIMEN compiler obtained in Section 4.6.

Example 5.1. Extracts from the corresponding program action and object code for the *factorial* program of Figure 4.24 are in Figure 5.1 and Figure 5.2, respectively. The three binding registers in line 34 of Figure 5.2 are used to store bindings at run-time. The 'bind' actions for y (line 5 in Figure 5.1) and n (line 19 in Figure 5.1) make their way into the object code as calls to the run-time function *_BIND* (lines 38 and 7, respectively, of Figure 5.2). The call to *_OVERLAY_BINDINGS* in line 41 of Figure 5.2 resulted from the translation of the 'before' combinator in line 8 of the program action. The calls to *_BOUND*, in lines 10, 14, 19, 23, 24, 26 and 46, of Figure 5.2, represent binding lookups. All this run-time manipulation of bindings imposes a significant overhead on the object code. More importantly, it has no resemblance to the code generated by a hand-crafted compiler.

At the end of this chapter we will come back to this example, showing how the quality of the object code of Figure 5.2 (an the one in Figure 4.27) is improved after binding elimination. □

5.2 Action Transformations

Binding elimination is accomplished by *action transformations*. We call the *source action* the action that is the subject of transformations, and the *target action* the action obtained from the source action by application of one or more *transformation rules*. Thus, what one should do is a kind of simplification on the source action; for example, one could explore action notation algebraic properties (as presented in [80]) to obtain simpler actions. But, and most importantly, we need to introduce new transformation rules in order to do

```
| | | | give 0 then give the value label #1
| | | and
| | | | allocate an cell then give the cell label #2
| | then
| | | | bind "y" to the cell#2                                      (5)
| | | and
| | | | store the value#1 in the cell#2
| before                                                            (8)
| | recursively bind "fact" to closure abstraction
| | | | furthermore
| | | | | | give headOf(the proc-argument-list)
| | | | | then
| | | | | | | | give the value label #1
| | | | | | | and
| | | | | | | | | allocate an cell
| | | | | | | | then
| | | | | | | | | give the cell label #2
| | | | | | then
| | | | | | | | bind "n" to the cell#2                              (19)
| | | | | | | and
| | | | | | | | store the value#1 in the cell#2
| | | hence
| | | | | ...
| | | | then
| | | | | | give 1 then store the value in the cell bound to "y"
| | | | | else
| | | | | | | | | | | ...
| | | | | | | | | | and
| | | | | | | | | | | give 1 then give the value label #2
| | | | | | | | | then
| | | | | | | | | | give difference(the integer#1,the integer#2)
| | | | | | | | then
| | | | | | | | | give list(the datum)
| | | | | | | then
| | | | | | | | enact (the procedure bound to "fact" with the fun-argument-list)
| | | | | | and then
| | | | | | | | | | | | give the value bound to "n"
| | | | | | | | | | | or
| | | | | | | | | | | | give the primitive-value stored in the cell bound to "n"
| | | | | | | | | | then
| | | | | | | | | | | give the value label #1
| | | | | | | | | and
| | | | | | | | | | | | give the value bound to "y"
| | | | | | | | | | | or
| | | | | | | | | | | | give the primitive-value stored in the cell bound to "y"
| | | | | | | | | | then
| | | | | | | | | | | give the value label #2
| | | | | | | | then
| | | | | | | | | give product(the integer#1,the integer#2)
| | | | | | | then
| | | | | | | | store the value in the cell bound to "y"
hence
| | give 10 then give list(the datum)
| then
| | enact (the procedure bound to "fact" with the fun-argument-list)
```

Figure 5.1: Program action for the *factorial* program (extract).

```
int _abs0(_din, _bin, _dout, _bout)
        DATUM _din; BINDINGS _bin; DATUM *_dout; BINDINGS *_bout;
{       DATUM _d1, _d2, _d3, _d4, _d5, _d6; BINDINGS _b1, _b2, _b3;
        _d1 = _din; _b1 = _bin;
        _d2 = _HEAD_OF(_d1); _d3 = _SORT_CHECK(_d2, 4L);
        _d4 = _ALLOCATE_A_CELL();
        _b2 = _BIND("n", _d4);
        *_d4.datum.cell = _d3;
        _b3 = _OVERLAY_BINDINGS(_b2, _b1);
        _d2 = *_BOUND("n", _b3).datum.cell; _d3 = _MAKE_INTEGER(0);
        _d4 = _IS(_d2, _d3);
        if (_d4.datum.truth_value) {
                _d2 = _MAKE_INTEGER(1);
                *_BOUND("y", _b3).datum.cell = _d2;
        } else {_d2 = *_BOUND("n", _b3).datum.cell;
                _d3 = _MAKE_INTEGER(1);
                _d5 = _DIFFERENCE(_d2, _d3);
                _d2 = _LIST(_d5);
                _d3 = _WITH(_BOUND("fact", _b3), _d2);
                if ((_d3.datum.abs->codeact)
                        (_d3.datum.abs->datum,_d3.datum.abs->bindings,&_d5,&_b1))
                                goto _failure_0;
                _d2 = *_BOUND("n", _b3).datum.cell;
                _d3 = *_BOUND("y", _b3).datum.cell;
                _d6 = _PRODUCT(_d2, _d3);
                *_BOUND("y", _b3).datum.cell = _d6;
        };
        *_dout = _d1; *_bout = NULL;
        return (0);
_failure_0:
        return (1);
}

DATUM _d1, _d2, _d3; BINDINGS _b1, _b2, _b3;

int main()
{       _d1 = _MAKE_INTEGER(0); _d2 = _ALLOCATE_A_CELL();
        _b1 = _BIND("y", _d2);
        *_d2.datum.cell = _d1;
        _b2 = _BIND("fact", _MAKE_UNKNOWN());
        _b3 = _OVERLAY_BINDINGS(_b2, _b1);
        _b3->datum = _CLOSURE(_ABSTRACTION(_abs0), _b3);
        _b2 = _BIND("fact", _b3->datum);
        _b3 = _OVERLAY_BINDINGS(_b2, _b1);
        _d1 = _MAKE_INTEGER(10); _d2 = _LIST(_d1);
        _d1 = _WITH(_BOUND("fact", _b3), _d2);
        if ((_d1.datum.abs->codeact)
                (_d1.datum.abs->datum,_d1.datum.abs->bindings,&_d3,&_b1))
                        goto _failure_0;
        exit(0);
_failure_0:
        exit(1);
}
```

Figure 5.2: Object code for the *factorial* program.

further simplifications and, hopefully, to achieve the total elimination of the declarative facet. Naturally, each transformational step must be justified by a rule, which implies that the target action is, in some sense, semantically equivalent to its source action.

We assume that the action subject to transformation was previously sort-checked; thus all sort information that was derived by the sort checker is available. In fact this sort information plays a fundamental role in the transformations. By analysing sort information one can easily distinguish known values from unknown ones, as is explained in Section 5.3. We assume that sort checking is complete, that is, for every subaction a of the source action, the sort of a has been inferred by the sort checker. This sort is '**nothing**' for any action that must fail and $(\tau, \beta) \hookrightarrow (\tau', \beta')$ for any other action (where τ, β, τ' and β' are defined as in Section 4.2.2).

5.3 Known and Unkown Bindings

Depending on our compile-time (or static) knowledge of what datum (value) is bound to an identifier, a binding can be (statically) *known* or (statically) *unknown*. The transformations we apply to eliminate a known binding are different from the ones we apply when a binding is unknown. The identification of known and unknown bindings is made by a simple inspection of the sort information. We illustrate this with some examples.

Example 5.2. Consider the action '**give 1 then bind "z" to the integer**'. The sort information (decoration) for the second subaction is

$$(\{\mathsf{0} : \mathsf{1}\}, \{\}) \hookrightarrow (\{\}, \{\mathsf{z} : \mathsf{1}\})$$

which tells that the action produces a binding of 'z' to a value of sort 1. But the only individual x satisfying

$$x : \mathsf{1}$$

is 1. So the value bound to 'z' is *known* and *its value* is 1. □

Example 5.3. Now consider the incorporated action

| bind "x" to the integer
hence
| give successor (the integer bound to "x")

'x' might be a formal parameter of a function that gives the successor of its argument. Clearly, 'x' is bound to an unknown value. The sort information for the '**bind**' action is:

$$(\{0 : \mathsf{integer}\}, \{\}) \hookrightarrow (\{\}, \{\mathsf{x} : \mathsf{integer}\})$$

indicating that it produces a binding of 'x' to a value of sort integer. As many individuals x satisfy the assertion

$$x : \mathsf{integer}$$

that is, integer is a proper sort, the value bound to 'x' is *unknown*. □

Example 5.4. The 'bind' action in

| allocate a cell
then
| bind "x" to the cell and store 5 in the cell

has sort $(\{0 : \mathsf{cell}\}, \{\}) \hookrightarrow (\{\}, \{\mathsf{x} : \mathsf{cell}\})$. From this we can infer that 'x' is bound to an unknown cell (because cell is a proper sort).

Suppose, however, that we statically perform the 'allocate' action, so we obtain an individual cell, say c, as the result of the performance. Now the 'bind' action has sort $(\{0 : c\}, \{\}) \hookrightarrow (\{\}, \{\mathsf{x} : c\})$ and now 'x' is bound to a known cell. □

Example 5.5. If a token is bound to an abstraction, we treat it as if it were bound to an unknown datum. In

| bind "inc" to abstraction (give sum (the integer,1))
hence
| enact (the abstraction bound to "inc" with 4)

the 'bind' action has sort $(\{\}, \{\}) \hookrightarrow (\{\}, \{\mathsf{inc} : s\})$, where s is the abstraction sort

$$(\{0 : \mathsf{integer}\}, \{\}) \rightsquigarrow (\{0 : \mathsf{integer}\}, \{\}).$$

That is, 'inc' is bound to an abstraction (whose incorporated action expects an integer labelled 0 and empty bindings, and delivers an integer labelled 0 and no bindings). □

Known bindings are easy to eliminate. Action transformations turn unknown bindings into known bindings and then eliminate them.

5.4 Classifying Allocate Actions

The action '**allocate a cell**' allocates a cell dynamically, i.e., the cell is actually allocated when the action is performed. The allocated cell will be chosen from the currently free cells. Which cell will be allocated can depend on such factors as the region of memory where the program will be loaded, how active the system will be at allocation time, how the system memory manager works, etc. In this sense, the '**allocate**' action is a non-deterministic action. In general, all variables (in the sense of PASCAL variables) in a program action are allocated in the heap, no matter whether they are global or local or heap variables. However an ACTRESS compiler, like any conventional compiler, should use static and stack allocation wherever possible. Static and stack allocation are faster and cheaper than heap allocation. Also they assign known (relative) addresses to each variable at compile time, which is a pre-requisite for binding elimination.

Example 5.6. The '**allocate**' action in

```
| | allocate a cell
| then
| | store 4 in the cell and give the cell
then
| give sum (the integer stored in the cell,3)
```

can be statically allocated. Suppose we assume that '**allocate a cell**' allocates cell m. Then we could transform the source action above into

```
| store 4 in m
then
| give sum (the integer stored in m,3)
```

where we just replaced all references to the cell ('**the cell**') by m, an "eliminated" the '**allocate**' action because we had already performed it. (We will see later why and how the '**give**' action in the second line of the source action was eliminated.) □

We introduce a classification procedure which performs '**allocate**' actions statically whenever possible. This is achieved by replacing '**allocate a cell**' by '**give cell** (l, q)', where 'cell (l, q)' represents the q-th cell at *nesting level* l. The program action is at level 0. The occurrence of an abstraction defines a new nesting level.

In terms of action notation, we need not only to transform 'allocate a cell' into 'give cell (l, q)', but also to reserve the actual cell. For example, suppose that after classifying the 'allocate' actions, we know that a particular program action a needs 20 cells for its global variables (nesting level 0). Besides the transformation above, we need a further transformation that *prepends* an action which reserves the cells needed by a:

reserve 20 cells at level 0 and then a

where the action 'reserve n cells at level l' reserves n memory cells (or a memory block of size n) at nesting level l, and is defined by:

- reserve _ cells at level _ :: natural, natural $\rightarrow$ action .

(1) reserve 0 cells at level l = complete .

(2) reserve (successor (n)) cells at level l = | reserve n cells at level l
and
| reserve cell (l,n) .

For the purposes of compile-time computations (for example, binding elimination), 'cell (_,_)' can be treated as a literal representation of a cell (an individual known cell). In terms of action notation, 'cell (_,_)' can be seen as a yielder operation with functionality:

- cell (_,_) :: natural, natural $\rightarrow$ yielder [cell] .

which yields a cell computed at run time.

For simplicity we consider all cells as having the same sort 'cell'. The ANC code generator must be modified to understand this new operation. It must be informed on how many cells were allocated and at which level (see Section 5.11).

Consider an action a at level l and an 'allocate' action contained in a. Here a must be a program action or incorporated action. Then:

- If the 'allocate' action will be performed exactly once whenever a is performed, it can be replaced by 'give cell (l,q)', where 'cell (l,q)' is not used anywhere else in a.

- If the 'allocate' action will *not* be performed exactly once whenever a is performed, leave it unchanged.

In action notation it is rather easy to test whether a given subaction of a is performed exactly once when a is performed. For example, if 'a_1 **and** a_2' is performed once, then a_1 and a_2 are also performed once. This point about '**and**' applies equally to all the other action combinators except '**or**' and '**unfolding**'. If 'a_1 **or** a_2' is performed once, one of its subactions will generally not be performed at all. If '**unfolding** a' is performed once, its subaction will generally be performed several times.

So the classification procedure works as follows:

- We do not transform '**allocate**' actions in subactions of conditional actions ('**or**' and '**else**').
- We do not transform '**allocate**' actions inside '**unfolding**' actions.

For all other action combinators we can be sure that an '**allocate**' action will be performed exactly once.

Notice that, although an incorporated action may be performed more than once, the '**allocate**' actions inside it, and which do not fall in the two above cases, can be transformed.

During binding elimination, further cells might be statically allocated, which add to the number of cells allocated by the classification procedure.

Some programming languages provide mechanisms for dynamic memory allocation (like PASCAL's *new* or C's *malloc*). Although these are *genuine* dynamic memory allocations, we could still perform statically such allocations if one can guarantee that they are performed exacly once when the program is run. In general, the classification procedure will not transform them.

Example 5.7. Consider the following program in a version of SPECIMEN extended with PASCAL-like pointer types:

```
program pointer is
   varpointer p : integer
in
   while ...
      do
         ...;
         call new (p);
         ...
      end
end
```

The program action could look like the following:

```
| furthermore
| | allocate a cell then bind "p" to the cell
hence
| unfolding
| | ...
| | | allocate a cell then store pointer-to (the cell) in the cell bound to "p"
| | ...
```

The first '**allocate**' action can be transformed to use static allocation, but the one inside the '**unfolding**' action must remain as dynamic allocation:

```
| furthermore
| | give cell (0,3) then bind "p" to the cell
hence
| unfolding
| | ...
| | | allocate a cell then store pointer-to (the cell) in the cell bound to "p"
| | ...
```

Using other action transformations which will be introduced later, we can eventually transform this action to:

```
unfolding
| ...
| | allocate a cell then store pointer-to (the cell) in cell (0,3)
| ...
```

□

Although we can think separately about classification of '**allocate**' actions and binding elimination, they are specified using a single set of rules and implemented as a single pass over the decorated action tree. In the same pass we allocate the storage necessary for the classification of '**allocate**' actions, as well as storage (if any) necessary for binding elimination.

5.5 Transient Elimination

Suppose we want to simplify the source action

```
| give 1 then bind "x" to the integer
hence
| give the integer bound to "x"
```

Firstly we make the 'bind' action "consume" the transient given by 'give 1', and eliminate the 'give' action:

```
| complete then bind "x" to 1
hence
| give the integer bound to "x"
```

The elimination of the 'give' action is desirable as it became a *dead action* after the consumption of its transient. (This resembles the *dead code elimination* transformation used in conventional compilers.) Secondly, we can eliminate the 'bind' action and then replace all scoped occurrences of 'the _ bound to "x"' by 1:

```
| complete then complete
hence
| give 1
```

Thus, we have not only eliminated the 'bind' action but also a 'give' action from the source action. The elimination of the 'give' action is called *transient elimination.* In Section 5.8 we will specify how (partial) transient elimination is achieved together with binding elimination. (We shall see in Section 5.7 how we can do a further transformation and obtain 'give 1' as the final target action for the above action.)

5.6 Binding Elimination

Before the formalization of the transformations, we give some intuition on how the binding elimination process is carried out. We have to consider two cases in the design of the elimination rules:

- *A token is bound to a statically known datum.* If a token k is bound to a statically known individual datum d (like 5, *false*, or a particular cell), we can replace all occurrences of 'the s bound to k' by 'd & s'. (Note that 'd & s' can itself be simplified to d or 'nothing', depending on whether $d : s$ or not.)

- *A token is bound to a statically unknown datum.* If k is bound to a statically unknown datum of sort S (like a truth-value, an integer, a cell or an abstraction), we statically allocate a cell, say c, store the datum in the known cell c (this is done by replacing the 'bind' action by a 'store' action), and replace all scoped occurrences of 'the s bound to k' by 'the s stored in c'.

We describe now some particular examples which clarify the interaction between binding elimination and storage allocation. We will use the term *variable* in the sense used in PASCAL.

- Suppose k is a global variable, in which case, the program action contains an action that allocates a cell and binds it to k, e.g., '**allocate a cell then bind** k **to the cell**'. We can classify this '**allocate**' action by replacing it by '**give cell** $(0, q)$', where q is a natural and 0 is the nesting level of the '**allocate**' action. Now k is bound to a known cell, so we could replace all occurrences of '**the** s **bound to** k' by '**cell** $(0,q)$' and then eliminate the '**bind**' action for k. Thus lookup at this cell will be expressed as '**the** s **stored in cell** $(0,q)$' instead of '**the** s **stored in the cell bound to** k'.

- Suppose k is a procedure local variable, in which case, there is an incorporated action which contains an action that allocates a cell and binds it (as in the previous example). We can classify the '**allocate**' action by replacing it by '**give cell** (l,q)', where l is a positive integer representing the nesting level of the enclosing incorporated action, and q is a natural representing the next available local cell. Now k is bound to a known cell, so we could replace all occurrences of '**the** s **bound to** k' by '**cell** (l,q)' and eliminate the '**bind**' action for k. (As explained in Section 5.4, '**cell** (l,q)' can be seen as a compile-time known cell.)

- Suppose k is bound to a procedure. In this case we store the procedure in a known cell '**cell** (l,q)' and replace all occurrences of '**the abstraction bound to** k' by a storage lookup for the cell, that is, '**the abstraction stored in cell** (l,q)'.

- Suppose k is a procedure formal constant parameter. The token k is bound to an unknown given value at the beginning of the incorporated action (the action inside the abstraction that denotes the procedure). In this case we store the unknown given value in a known local cell '**cell** (l,q)', and replace all occurrences of '**the** s **bound to** k' by '**the** s **stored in cell** (l,q)', where l is the procedure nesting level. In this manner one can eliminate the binding for the formal parameter.

- Suppose k is a procedure formal variable parameter. The token k is bound to an unknown given cell at the beginning of the incorporated action. In this case we store the unknown given cell in a known local cell '**cell** (l,q)', and replace all occurrences of 'the cell bound to k' by 'the cell stored in cell (l,q)', where l is the procedure nesting level. In this manner one can eliminate the binding for the formal parameter.

As one can see from the examples above, there is a strong relation between binding elimination and storage allocation.

5.7 Action Notation Laws

Action notation has many nice algebraic properties. For example, the 'complete' action is the unit element for the 'and' combinator:

$$\textsf{complete and } a = a$$

In Figure 5.3 we present some action notation laws. Some of these laws correspond to the algebraic properties of action notation presented in [80] (Appendix B). Laws are distinct from the other transformation rules in the sense that they do not affect sort information. Some of them however are based on certain assumptions over sort information. This is reflected in the two judgements encountered in Figure 5.3:

$$a = a' \tag{5.1}$$

$$\mathcal{K} \vdash a = a' \tag{5.2}$$

The first judgement means that a is equivalent to a', that is, whenever we find a one can replace it by a' (and vice versa). Rules with this judgement are the ones which resemble the action notation properties as in [80]. The second judgement asserts the equivalence of a and a' based upon a restriction on the sort of a. For example, one can assert that 'complete then $a = a$' if the action a uses no transients. This is expressed by Rule 5.4. It is unusual to find an action like 'complete then a' in a program action (and even more unusual to find one in an action semantic description). However, the rule is useful because, even when the original program action does not contain one, such an action can arise after transformations, which can then be simplified according to the above rule. Moreover, in Actress, a program action that does not satisfy the antecedent of the rule is ill-sorted. Because the elimination rules (Section 5.8) preserve sort information (Section 5.12), we

(AND)

$$\text{complete and } a = a \text{ and complete} = a \tag{5.3}$$

(THEN)

$$\frac{\mathcal{K} \vdash a : (\{\}, \beta) \hookrightarrow (\tau', \beta')}{\mathcal{K} \vdash \text{complete then } a = a} \tag{5.4}$$

(FURTHERMORE)

$$\frac{\mathcal{K} \vdash \text{furthermore } a : (\tau, \{\}) \hookrightarrow (\tau', \beta')}{\mathcal{K} \vdash \text{furthermore } a = a} \tag{5.5}$$

(HENCE)

$$\text{rebind hence } a = a \tag{5.6}$$

$$\frac{\mathcal{K} \vdash a : (\tau, \{\}) \hookrightarrow (\tau', \beta')}{\mathcal{K} \vdash \text{complete hence } a = a} \tag{5.7}$$

$$\frac{\mathcal{K} \vdash a : (\tau, \beta) \hookrightarrow (\tau', \{\})}{\mathcal{K} \vdash a \text{ hence complete} = a} \tag{5.8}$$

(CLOSURE)

$$\frac{\mathcal{K} \vdash \text{closure } y : (\tau, \{\}) \rightsquigarrow s}{\mathcal{K} \vdash \text{closure } y = y} \tag{5.9}$$

Figure 5.3: Some action notation laws.

could drop the antecedent. We decided to keep it for generality though.

5.8 Elimination Rules

Transient elimination, binding elimination and classification of allocate actions are specified by a set of rules called *elimination rules*. Before we introduce the rules, we will give an example which helps to build our intuition.

Example 5.8. The SPECIMEN program

```
program intuition is
    var x : int := 5
in
    x := x + 1
end
```

has the (slightly simplified) program action

```
| | allocate a cell
| then
| | | bind "x" to the cell
| | and
| | | store 5 in the cell
hence
| store sum (the integer stored in the cell bound to "x",1) in the cell bound to "x"
```

as its denotation. Our objective is to eliminate the 'bind' action. Unfortunately 'x' is bound to an unknown cell. That is because the 'allocate' action is a dynamic action which finds an unreserved cell, reserves it and gives it *when the action is performed.* Surely, for this case, we could perform the 'allocate' action statically. If we know that no cell was previously allocated we could use (allocate) 'cell (0,0)':

```
| | give cell (0,0)
| then
| | | bind "x" to the cell
| | and
| | | store 5 in the cell
hence
| store sum (the integer stored in the cell bound to "x",1) in the cell bound to "x"
```

which by transient elimination can be transformed into

```
| | complete
| then
| | | bind "x" to cell (0,0)
| | and
| | | store 5 in cell (0,0)
hence
| store sum (the integer stored in the cell bound to "x",1) in the cell bound to "x"
```

Now clearly 'x' is bound to a known datum (cell (0,0)). We could eliminate the 'bind' action as long as we replace all scoped occurrences of 'the cell bound to "x"' by 'cell (0,0)'. As we shall see, a *substitution* will be used to record the eliminated bindings. So, we first eliminate the 'bind' action, and transform the second subaction of 'hence' knowing that every scoped occurrence of 'the _ bound to "x"' must be replaced by 'cell (0,0)':

```
| | complete
| then
| | | complete
| | and
| | | store 5 in cell (0,0)
hence
| store sum (the integer stored in cell (0,0),1) in cell (0,0)
```

Using now laws 5.4 and 5.3 we obtain:

```
| store 5 in cell (0,0)
hence
| store sum (the integer stored in cell (0,0),1) in cell (0,0)
```

But we still need to reserve 'cell (0,0)':

```
| reserve 1 cell at level 0
and then
| | store 5 in cell (0,0)
| hence
| | store sum (the integer stored in cell (0,0),1) in cell (0,0)
```

which is the final target action[1]. □

We introduce some notation and definitions now. A *substitution* is a mapping from yielder terms to yielder terms. We express a substitution as

$$[y'/y, \ldots]$$

where y'/y means that a yielder of the form y can be replaced by y'. We restrict the term y' to be either an individual datum d (an integer, a truth-value, or a cell), a yielder of the form 'the _ stored in c', or the special individual 'uneliminated'. An empty substitution is expressed as [ ].

For example, using substitution

$\mathcal{T}$ = [4 / the integer#2,
uneliminated / the integer#9,
true / the truth-value#5]

one can:

[1] To simplify the presentation, we are going to omit the cell reservations action in the target action from now on.

- transform 'give the integer#2' into 'give 4';
- transform 'give the integer#9' into 'give the integer#9', and
- transform 'store both (the truth-value#5,the truth-value#6)' into 'store both (true,the truth-value#6)'.

Using substitution

$\mathcal{B}$ = [cell (3,4) / the cell bound to "x",
1000 / the integer bound to "n",
uneliminated / the truth-value bound to "y",
the integer stored in cell (5,1) / the integer bound to "z"]

one can:

- transform 'bind "w" to the cell bound to "x"' into 'bind "w" to cell (3,4)';
- transform 'give successor (the integer bound to "n")' into 'give successor (1000)';
- transform 'give not (the truth-value bound to "y")' into 'give not (the truth-value bound to "y")', and
- transform 'store the integer bound to "z" in the cell' into 'store the integer stored in cell (5,1) in the cell'.

Definition 5.1 (Transient substitution) *A transient substitution* $[y'/y,\ldots]$ *is a substitution where every* y *is of the form* 'the _ $\#n$'.

Definition 5.2 (Binding substitution) *A binding substitution* $[y'/y,\ldots]$ *is a substitution where every* y *is of the form* 'the _ bound to k'.

Mathematically, transients and bindings substitutions are label-term mappings and token-term mappings, respectively. So, for example, one can write for the substitutions above:

- *dom* $\mathcal{T} = \{2, 9, 5\}$ and *dom* $\mathcal{B} = \{\mathsf{x}, \mathsf{n}, \mathsf{y}, \mathsf{z}\}$
- $\mathcal{B}(\mathsf{x})$ = cell(3,4)

- $\mathcal{T}(2) = 4$

Sometimes is convenient to use the following abbreviations:

- $[y'/n]$ instead of $[y'/\textsf{the \_ } \#n]$
- $[y'/k]$ instead of $[y'/\textsf{the \_ bound to } k]$

As illustrated in Example 5.8, substitutions are used to keep a record of eliminated transients and bindings.

We formalize binding elimination using inference rules. The formalization introduces two judgements: one corresponding to actions and one to yielders. The judgement

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}' \tag{5.10}$$

states that: in the presence of sort information $\mathcal{K}$, input transient substitution $\mathcal{T}$, input binding substitution $\mathcal{B}$ and input storage allocation context $\mathcal{S}$, action a' is obtained from a by eliminating transients, eliminating bindings, and transforming '**allocate**' actions in a. The output transient substitution $\mathcal{T}'$ and output binding substitution $\mathcal{B}'$ record the transients and bindings eliminated, respectively. The output storage allocation context $\mathcal{S}'$ reflects the next cell available for use in a particular nesting level, and in which context '**allocate**' actions are classified. It is a triple (l, q, e) where l is the current nesting level, q the next available cell, and e a boolean telling if '**allocate**' actions in the current storage allocation context can be transformed (*true*) or not (*false*). The sort information $\mathcal{K}$ can be used to infer the sort of a or any of its subactions.

For yielders we have the following simpler judgement:

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y' \tag{5.11}$$

Here we do not need the output substitutions because yielders do not deliver transients or bindings. Also the current storage allocation context does not change when yielders are transformed. Input substitutions are still needed for yielders. As we shall see shortly, sort information and input storage allocation context are necessary because of abstractions.

In the sequence we describe the main elimination rules.

(PRESERVE)

give all	$[\,]$	$=$	complete
give all	$(\text{uneliminated}/n \cdot \mathcal{T})$	$=$	give all $\mathcal{T}$
give all	$(y/n \cdot \mathcal{T})$	$=$	give y label $\#n$ and give all $\mathcal{T}$, $y \neq$ uneliminated
bind all	$[\,]$	$=$	complete
bind all	$(\text{uneliminated}/k \cdot \mathcal{B})$	$=$	bind all $\mathcal{B}$
bind all	$(y/k \cdot \mathcal{B})$	$=$	bind k to y and bind all $\mathcal{B}$, $y \neq$ uneliminated

Figure 5.4: Preservation actions.

5.8.1 Program Action

We apply the following rule to the program action:

$$\frac{\mathcal{K}, [\,], [\,], (0, 0, \mathit{true}) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K} \vdash a \Rrightarrow a' \text{ and then } (\text{give all } \mathcal{T}' \text{ and bind all } \mathcal{B}')} \tag{5.12}$$

We assume that the initial transient and binding input substitutions are empty, that is, the program action requires no transients and no bindings. If the program action produces bindings (or gives transients) we preserve them in order to maintain the action's (observational) behaviour; this is achieved by appending preservation actions to the target action. The preservation actions' definitions are shown in Figure 5.4. They just "put back" transients or bindings eliminated from an action a from which we could not eliminate them. Notice that we do not preserve transients and bindings which were not eliminated (second line of each definition in Figure 5.4). As we shall see later, there are other cases where the use of preservation actions will be necessary.

5.8.2 Basic

The elimination rules for basic actions are shown in Figure 5.5. As the 'complete' and 'fail' actions do not propagate transients and bindings, the transformation rules specify empty transient and binding output substitutions for them. Compare the rules with the semantic rules for the actions given in Figure 3.2.

Apart from the presence of the sort information and storage allocation context, the rule for 'and then' also looks very similar to its semantic rule in Figure 3.2: we transform a_1; we then transform a_2 using the input substitutions to the compound action and the

(COMPLETE)

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{complete} \Rightarrow \textsf{complete}, [\,], [\,], \mathcal{S} \tag{5.13}$$

(FAIL)

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{fail} \Rightarrow \textsf{fail}, [\,], [\,], \mathcal{S} \tag{5.14}$$

(AND)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a'_1, \mathcal{T}'_1, \mathcal{B}'_1, \mathcal{S}'_1 \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S}'_1 \vdash a_2 \Rightarrow a'_2, \mathcal{T}'_2, \mathcal{B}'_2, \mathcal{S}'_2}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \textsf{ and } a_2 \Rightarrow a'_1 \textsf{ and } a'_2, \mathcal{T}'_1 \oplus \mathcal{T}'_2, \mathcal{B}'_1 \oplus \mathcal{B}'_2, \mathcal{S}'_2} \tag{5.15}$$

(AND THEN)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a'_1, \mathcal{T}'_1, \mathcal{B}'_1, \mathcal{S}'_1 \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S}'_1 \vdash a_2 \Rightarrow a'_2, \mathcal{T}'_2, \mathcal{B}'_2, \mathcal{S}'_2}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \textsf{ and then } a_2 \Rightarrow a'_1 \textsf{ and then } a'_2, \mathcal{T}'_1 \oplus \mathcal{T}'_2, \mathcal{B}'_1 \oplus \mathcal{B}'_2, \mathcal{S}'_2} \tag{5.16}$$

(UNFOLDING)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B} \vdash a \sim \mathcal{T}_u, \mathcal{B}_u \qquad \mathcal{K}, \mathcal{T}_u, \mathcal{B}_u, (l, q, \mathit{false}) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, e) \vdash \textsf{unfolding } a \Rightarrow a'' \textsf{ thence unfolding } a', \mathcal{T}', \mathcal{B}', \mathcal{S}'} \tag{5.17}$$

where $a'' = \textsf{give all } (\mathcal{T} \ominus \mathcal{T}_u) \textsf{ and bind all } (\mathcal{B} \ominus \mathcal{B}_u)$

(UNFOLD)

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{unfold} \Rightarrow \textsf{unfold}, [\,], [\,], \mathcal{S} \tag{5.18}$$

(OR)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, \mathit{false}) \vdash a_1 \Rightarrow a'_1, \mathcal{T}'_1, \mathcal{B}'_1, \mathcal{S}'_1 \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, \mathit{false}) \vdash a_2 \Rightarrow a'_2, \mathcal{T}'_2, \mathcal{B}'_2, \mathcal{S}'_2}{\mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, e) \vdash a_1 \textsf{ or } a_2 \Rightarrow a''_1 \textsf{ or } a''_2, [\,], [\,], \mathit{greater}(\mathcal{S}'_1, \mathcal{S}'_2)} \tag{5.19}$$

where $a''_1 = a'_1 \textsf{ and then (give all } \mathcal{T}'_1 \textsf{ and bind all } \mathcal{B}'_1)$
$a''_2 = a'_2 \textsf{ and then (give all } \mathcal{T}'_2 \textsf{ and bind all } \mathcal{B}'_2)$

Figure 5.5: Basic elimination rules.

output storage allocation context of a_1. The resulting action is the '**and then**' combination of the transformed actions, merged transients and bindings output substitutions, and a_2's output storage allocation context.

We motivate now with an example, the elimination rule for '**unfolding**'. Consider the following source action:

```
| give 4
then
| unfolding
| | | bind "x" to successor (the integer)
| | hence
| | | give sum (the integer bound to "x",1) then unfold
```

A naive transformation would give the following target action (having in mind only transient elimination):

```
| complete
then
| unfolding
| | | bind "x" to successor (4)
| | hence
| | | give sum (the integer bound to "x",1) then unfold
```

which has a different behaviour than the source action. The problem is that at every iteration the unfolded action receives a different transient. Because of this, one can only use as the input transient substitution to the unfolded action those transients from the input substitution to the '**unfolding**' action which reach all '**unfold**' actions. A similar explanation applies to bindings.

Rule 5.17 specifies the elimination rule for '**unfolding**'. The left antecedent is an analysis on the unfolded action which tells what transients in the input transient substitution to the '**unfolding**' action reach all '**unfold**' actions. (In Chapter 6 we will see how this analysis can be performed.) These transients will be used as the input transient substitution to the unfolded action ($\mathcal{T}_u$ in the right antecedent of Rule 5.17). The '**thence**' combinator used in the rule, behaves as '**then**' for transients, and as '**hence**' for bindings.

Using now Rule 5.17, the above source action will be transformed into:

```
| complete
then
| | give 4
| thence
| | unfolding
| | | | bind "x" to successor (the integer)
| | | hence
| | | | give sum (the integer bound to "x",1) then unfold
```

In the rule for 'unfold' (Rule 5.18) we assumed that the unfolded action produces no bindings (empty output substitutions in the rule).

The action 'a_1 or a_2' requires some more thought. If the subactions produce bindings (they should be for the same tokens as required by the sort rules), we cannot eliminate these bindings as only one of the subactions will be performed. Although we cannot eliminate the bindings produced by a_1 and a_2, we can still eliminate bindings produced by subactions of a_1 and a_2. As we did for the program action, the bindings produced by a_1 and by a_2 must be preserved. Again, by using the preservation actions the transients given, and bindings produced by the source action are preserved. Notice that, although we do not classify 'allocate' actions inside the subaction of an 'or' (current storage allocation context *false*), bindings can still be eliminated. This elimination may imply some storage allocation. As the number of allocations for each branch may be different, we take the biggest of the next available cell using the *greater* operation.

Even when the 'or' combinator produces bindings we can identify three cases for which bindings could still be eliminated. The first case is when identical tokens are bound to the same individual d (an integer or truth-value individual). The second case is when identical tokens are bound to a datum of the same sort: we could store the datum in a cell and pass the substitution [cell $(l,q)/k$] on. The third case is when bindings produced by the 'or' action are not used by subsequent actions. We could perfectly eliminate them in this case.

5.8.3 Functional

The rules for functional action notation (Figure 5.6) specify (partial) transient elimination. A transient is only created by a 'give' action. (This 'give' action can be inside an abstraction which when enacted gives the transient.) Although, in general, transients have a very short life, they are sometimes difficult to eliminate. For example, the source action

| give *op* (d_1, d_2)
then
| bind "x" to it and bind "y" to it

could be transformed into

bind "x" to *op* (d_1, d_2) and bind "y" to *op* (d_1, d_2)

But *op* might be a very expensive data operation (d_1 and d_2 are individuals), which makes the target action *less efficient* than the source action. Another alternative target action could be:

| store *op* (d_1, d_2) in cell (0,0)
then
| | bind "x" to the datum stored in cell (0,0)
| and
| | bind "y" to the datum stored in cell (0,0)

which again is less efficient than the source action. Storing transients in memory is not a good policy as this is more expensive than to use machine registers to keep them. Therefore regarding transient elimination we cannot always assert that the final target action will always be more efficient than the original source action. However, when transients are primitive values (like integer or truth-value individuals), we can be sure that a target action will not be less efficient than its source action as there is no cost associated to the evaluation of such individuals.

Rules 5.20 and 5.21 in Figure 5.6 cater for such cases. The first rule is applied whenever the sort of the '**give**' action tells us that it gives an individual datum d labelled n (see the antecedent of the rule). Note the output transient substitution reflecting the elimination of the '**give**' action. The second rule applies whenever the given transient is unknown (proper sort S). In this case we transform the yielder y and keep the '**give**' action. As there is no elimination, the output transient substitution is [**uneliminated**/n]. The output binding substitution is empty, as a '**give**' action does not produce or propagate bindings.

The rule for the '**then**' combinator is straight forward. Note the use of $\mathcal{T}_1'$ as a_2's input transient substitution. This is exactly the behaviour of transients for the combinator.

The rules for the functional yielder '**the** $s\#n$' tells more about the nature of the elimination rules. The first one shows how a reference to an eliminated transient is treated: the occurrence of '**the** $s\#n$' is substituted by the individual associated to (label) n in the

(GIVE)

$$\frac{\mathcal{K} \vdash \mathsf{give}\ y\ \mathsf{label}\ \#n : (\tau, \beta) \hookrightarrow (\{n : d\}, \{\})}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \mathsf{give}\ y\ \mathsf{label}\ \#n \Rightarrow \mathsf{complete}, [d/n], [\,], \mathcal{S}} \tag{5.20}$$

$$\frac{\mathcal{K} \vdash \mathsf{give}\ y\ \mathsf{label}\ \#n : (\tau, \beta) \hookrightarrow (\{n : S\}, \{\}) \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \mathsf{give}\ y\ \mathsf{label}\ \#n \Rightarrow \mathsf{give}\ y'\ \mathsf{label}\ \#n, [\mathsf{uneliminated}/n], [\,], \mathcal{S}} \tag{5.21}$$

(CHECK)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \mathsf{check}\ y \Rightarrow \mathsf{check}\ y', [\,], [\,], \mathcal{S}} \tag{5.22}$$

(THEN)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a_1', \mathcal{T}_1', \mathcal{B}_1', \mathcal{S}_1' \qquad \mathcal{K}, \mathcal{T}_1', \mathcal{B}, \mathcal{S}_1' \vdash a_2 \Rightarrow a_2', \mathcal{T}_2', \mathcal{B}_2', \mathcal{S}_2'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1\ \mathsf{then}\ a_2 \Rightarrow a_1'\ \mathsf{then}\ a_2', \mathcal{T}_2', \mathcal{B}_1' \oplus \mathcal{B}_2', \mathcal{S}_2'} \tag{5.23}$$

(THE)

$$\mathcal{K}, [d/\mathsf{the}\ \_\#n, \ldots], \mathcal{B}, \mathcal{S} \vdash\ \mathsf{the}\ s\#n \Rightarrow d \tag{5.24}$$

$$\mathcal{K}, [\mathsf{uneliminated}/n, \ldots], \mathcal{B}, \mathcal{S} \vdash\ \mathsf{the}\ s\#n \Rightarrow\ \mathsf{the}\ s\#n \tag{5.25}$$

$$\frac{n \notin dom\ \mathcal{T}}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash\ \mathsf{the}\ s\#n \Rightarrow\ \mathsf{the}\ s\#n} \tag{5.26}$$

Figure 5.6: Functional elimination rules.

input transient substitution. Notice that we did not check if d is of sort s because we assume that our actions are well-sorted, so d is guaranteed to be an individual of sort s. (Otherwise we would have to replace '**the** $s\#n$' by 's & d'.) If the '**give**' action for the transient labelled n was not eliminated — n maps to '**uneliminated**' in the input transient substitution — we leave the functional yielder unchanged (Rule 5.25). If n is not in the input transient substitution we leave the functional yielder unchanged (Rule 5.26). As we shall see in Section 5.8.6, this last case may happen because of incorporated actions.

5.8.4 Declarative

Figure 5.7 shows the elimination rules for declarative actions. As expected, they play a central role in the binding elimination process.

Rule 5.27 for the '**bind**' action is analogous to Rule 5.20 of the '**give**' action. The second one (5.28) covers the cases where the datum bound to k is unknown: the transformed yielder (y') is "stored" in a newly allocated cell (**cell** (l,q)), and the output binding substitution is made to reflect the elimination. Notice that the *allocation* of the cell is reflected in the output storage allocation context, which implies that the target action uses one more cell than its source action. Rule 5.29 is used in a context where we cannot classify '**allocate**' actions.

The rules for the declarative combinators are very intuitive. An interesting rule is the one for 'a_1 **before** a_2'. The input binding substitution to a_2 is the input binding substitution to the compound action overlayed by the output binding substitution of a_1. Again, it is worth to mention the similarity of the flow of substitutions and operations on them to the transients and bindings flow and operations in the semantic rule for '**before**' (see Rule 3.31).

Notice that, in the case of the '**furthermore** a', even when total binding elimination is not achieved for a the output binding substitution will be the correct one: information about a non-eliminated '**bind**' action for token k in a will be in the output binding substitution for a as [**uneliminated**/k] which will overlay any binding for k in the input binding substitution for the '**furthermore**' action ('$\mathcal{B}' \oslash \mathcal{B}$' in Rule 5.30). An analogous explanation applies to '**moreover**' and '**before**'.

The elimination rules for the declarative yielder '**the** s **bound to** k' are similar to those for the functional yielder. The difference is that now we can find components like '**the** _

(BIND)

$$\frac{\mathcal{K} \vdash \textsf{bind } k \textsf{ to } y : (\tau, \beta) \hookrightarrow (\{\}, \{k : d\})}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{bind } k \textsf{ to } y \Rightarrow \textsf{complete}, [\,], [d/k], \mathcal{S}} \tag{5.27}$$

$$\frac{\mathcal{K} \vdash \textsf{bind } k \textsf{ to } y : (\tau, \beta) \hookrightarrow (\{\}, \{k : S\}) \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, \mathit{true}) \vdash y \Rightarrow y'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, \mathit{true}) \vdash \textsf{bind } k \textsf{ to } y \Rightarrow a', [\,], [\textsf{the \_ stored in cell } (l, q)/k], (l, q+1, \mathit{true})} \tag{5.28}$$

where $a' = \textsf{store } y' \textsf{ in cell } (l, q)$

$$\frac{\mathcal{K} \vdash \textsf{bind } k \textsf{ to } y : (\tau, \beta) \hookrightarrow (\{\}, \{k : S\}) \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, (l, q, \mathit{false}) \vdash \textsf{bind } k \textsf{ to } y \Rightarrow \textsf{bind } k \textsf{ to } y', [\,], [\textsf{uneliminated}/k], (l, q, \mathit{false})} \tag{5.29}$$

(FURTHERMORE)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{furthermore } a \Rightarrow \textsf{furthermore } a', \mathcal{T}', \mathcal{B}' \oslash \mathcal{B}, \mathcal{S}'} \tag{5.30}$$

(HENCE)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a_1', \mathcal{T}_1', \mathcal{B}_1', \mathcal{S}_1' \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}_1', \mathcal{S}_1' \vdash a_2 \Rightarrow a_2', \mathcal{T}_2', \mathcal{B}_2', \mathcal{S}_2'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \textsf{ hence } a_2 \Rightarrow a_1' \textsf{ hence } a_2', \mathcal{T}_1' \oplus \mathcal{T}_2', \mathcal{B}_2', \mathcal{S}_2'} \tag{5.31}$$

(MOREOVER)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a_1', \mathcal{T}_1', \mathcal{B}_1', \mathcal{S}_1' \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S}_1' \vdash a_2 \Rightarrow a_2', \mathcal{T}_2', \mathcal{B}_2', \mathcal{S}_2'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \textsf{ moreover } a_2 \Rightarrow a_1' \textsf{ moreover } a_2', \mathcal{T}_1' \oplus \mathcal{T}_2', \mathcal{B}_2' \oslash \mathcal{B}_1', \mathcal{S}_2'} \tag{5.32}$$

(BEFORE)

$$\frac{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \Rightarrow a_1', \mathcal{T}_1', \mathcal{B}_1', \mathcal{S}_1' \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}_1' \oslash \mathcal{B}, \mathcal{S}_1' \vdash a_2 \Rightarrow a_2', \mathcal{T}_2', \mathcal{B}_2', \mathcal{S}_2'}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a_1 \textsf{ before } a_2 \Rightarrow a_1' \textsf{ before } a_2', \mathcal{T}_1' \oplus \mathcal{T}_2', \mathcal{B}_2' \oslash \mathcal{B}_1', \mathcal{S}_2'} \tag{5.33}$$

(BOUND)

$$\mathcal{K}, \mathcal{T}, [d/k, \ldots], \mathcal{S} \vdash \textsf{ the } s \textsf{ bound to } k \Rightarrow d \tag{5.34}$$

$$\mathcal{K}, \mathcal{T}, [\textsf{the \_ stored in } c/k, \ldots], \mathcal{S} \vdash \textsf{ the } s \textsf{ bound to } k \Rightarrow \textsf{the } s \textsf{ stored in } c \tag{5.35}$$

$$\mathcal{K}, \mathcal{T}, [\textsf{uneliminated}/k, \ldots], \mathcal{S} \vdash \textsf{ the } s \textsf{ bound to } k \Rightarrow \textsf{ the } s \textsf{ bound to } k \tag{5.36}$$

$$\frac{k \notin \mathit{dom}\ \mathcal{B}}{\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{ the } s \textsf{ bound to } k \Rightarrow \textsf{ the } s \textsf{ bound to } k} \tag{5.37}$$

Figure 5.7: Declarative elimination rules.

stored in c / k' in the input binding substitution. For such cases the yielder is replaced by 'the s stored in c' (Rule 5.35).

If total binding elimination is achieved, the declarative action combinators can be replaced as follows:

- 'furthermore a' can be replaced by 'a'.
- 'a_1 hence a_2' can be replaced by 'a_1 and then a_2'.
- 'a_1 moreover a_2' can be replaced by 'a_1 and a_2'.
- 'a_1 before a_2' can be replaced by 'a_1 and then a_2'.

This can easily be deduced by examination of the semantics of the combinators involved with respect to transient flow (see Figure 3.6).

5.8.5 Imperative

Figure 5.8 shows the elimination rules for imperative action notation. Output substitutions are empty as imperative actions do not give transients and do not produce bindings.

The rules for the 'allocate' action[2] formalises the classification procedure described in Section 5.4. Rule 5.39 applies when the 'allocate' action occurs in a context where it is safe to allocate a known address (l, q) statically. Notice that (l, q) is taken from the input storage allocation context, and that the cell counter in the output storage allocation context is incremented by 1. The second rule (5.40) applies when the 'allocate' action occurs in a context where allocations must remain dynamic.

Notice that if an 'allocate' action was classified then its corresponding 'deallocate' action is transformed to deallocate the particular classified cell. For example,

| allocate a cell
then
| a and then deallocate the cell

is transformed into

| give cell (l,q)
then
| a and then deallocate cell (l,q)

(STORE)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y_1 \Rightarrow y_1' \qquad \mathcal{K},\mathcal{T},\mathcal{B},\mathcal{C} \vdash y_c \Rightarrow y_c'}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash \textsf{store } y_1 \textsf{ in } y_2 \Rightarrow \textsf{store } y_1' \textsf{ in } y_2', [\,], [\,], \mathcal{S}} \tag{5.38}$$

(ALLOCATE)

$$\mathcal{K},\mathcal{T},\mathcal{B},(l,q,true) \vdash \textsf{allocate a cell} \Rightarrow \textsf{complete}, [\textsf{cell } (l,q)/0], [\,], (l, q+1, true) \tag{5.39}$$

$$\mathcal{K},\mathcal{T},\mathcal{B},(l,q,false) \vdash \textsf{allocate a cell} \Rightarrow \textsf{allocate a cell}, [\,], [\,], (l, q, false) \tag{5.40}$$

(DEALLOCATE)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash \textsf{deallocate } y \Rightarrow \textsf{deallocate } y', [\,], [\,], \mathcal{S}} \tag{5.41}$$

(STORED)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash \textsf{the } s \textsf{ stored in } y \Rightarrow \textsf{the } s \textsf{ stored in } y'} \tag{5.42}$$

Figure 5.8: Imperative elimination rules.

For the imperative yielder '**the** s **stored in** y' we just transform y into y' (Rule 5.42).

5.8.6 Reflective

Figure 5.9 shows the reflective elimination rules. If an abstraction gives a transient and/or produces bindings, we do not eliminate them. (In other words, if the enaction of an abstraction produces bindings then the '**bind**' actions which produce them are not eliminated.) In fact they are eliminated from their original positions, but "preserved" for the incorporated action. This is expressed (Rule 5.43) by the empty output substitutions for the '**enact**' action, and the use of the preservation actions for '**abstraction**', '**with**' and '**closure**'.

The important point is to understand how an abstraction is handled by the transformations. We have to consider two cases:

- **Binding flow into the abstraction.** If an abstraction is not closed when it is formed (**abstraction** a), then, in general, it can be closed in many different points with many different bindings. The bindings received by the incorporated action,

[2] Although the 'allocate' action is considered hybrid action notation, its rules are given here for presentation purpose only.

(ENACT)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y_b \Rightarrow y'_b}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash \mathsf{enact}\ y_b \Rightarrow \mathsf{enact}\ y'_b, [\,], [\,], \mathcal{S}} \tag{5.43}$$

(ABSTRACTION)

$$\frac{\mathcal{K}, [\,], [\,], (l+1, 0, true) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash \mathsf{abstraction}\ a \Rightarrow \mathsf{abstraction}\ a''} \tag{5.44}$$

where $a'' = (a'\ \mathsf{and\ then}\ (\mathsf{give\ all}\ \mathcal{T}'\ \mathsf{and\ bind\ all}\ \mathcal{B}'))$

(WITH)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash y \Rightarrow y' \qquad \mathcal{K}, [\,], [\,], (l+1, 0, true) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash \mathsf{abstraction}\ a\ \mathsf{with}\ y \Rightarrow \mathsf{abstraction}\ a''\ \mathsf{with}\ y'} \tag{5.45}$$

where $a'' = (a'\ \mathsf{and\ then}\ (\mathsf{give\ all}\ \mathcal{T}'\ \mathsf{and\ bind\ all}\ \mathcal{B}'))$

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y_b \Rightarrow y'_b \qquad \mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y_b\ \mathsf{with}\ y \Rightarrow y'_b\ \mathsf{with}\ y'} \tag{5.46}$$

(CLOSURE)

$$\frac{\mathcal{K}, [\,], \mathcal{B}, (l+1, 0, true) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash \mathsf{closure}\ (\mathsf{abstraction}\ a) \Rightarrow \mathsf{closure}\ (\mathsf{abstraction}\ a'')} \tag{5.47}$$

where $a'' = (a'\ \mathsf{and\ then}\ (\mathsf{give\ all}\ \mathcal{T}'\ \mathsf{and\ bind\ all}\ \mathcal{B}'))$

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash y \Rightarrow y' \qquad \mathcal{K}, [\,], \mathcal{B}, (l+1, 0, true) \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash \mathsf{closure}\ (\mathsf{abstraction}\ a\ \mathsf{with}\ y) \Rightarrow \mathsf{closure}\ (\mathsf{abstraction}\ a''\ \mathsf{with}\ y')} \tag{5.48}$$

where $a'' = (a'\ \mathsf{and\ then}\ (\mathsf{give\ all}\ \mathcal{T}'\ \mathsf{and\ bind\ all}\ \mathcal{B}'))$

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash y_b \Rightarrow y'_b}{\mathcal{K},\mathcal{T},\mathcal{B},\mathcal{S} \vdash \mathsf{closure}\ y_b \Rightarrow \mathsf{abstraction}\ (\mathsf{bind\ all}\ \mathcal{B})\ \mathsf{hence}\ y'_b} \tag{5.49}$$

Figure 5.9: Reflective elimination rules.

if any, are not known at the formation point. Free occurrences of 'the s bound to k'[3] for the incorporated action of an unclosed abstraction are not simplified, and bindings at every closing point have to be preserved in order to be produced to the abstraction (closure y_b). These two observations are reflected by the empty input binding substitution in the antecedent of Rule 5.44 and the action 'bind all' in the consequent of Rule 5.49, respectively.

If the abstraction is closed at the formation point (closure (abstraction a)), then we can transform a using the input binding substitution current at this point. This is reflected in the antecedent of Rule 5.47. Notice that the input binding substitution is used to simplify all (assuming total binding elimination) the free occurrences of 'the s bound to k' for the incorporated action.

- **Binding flow out of the abstraction.** If an abstraction produces a binding, no matter whether it is a closed or unclosed abstraction, we do not know, in general, where this binding will be used. So each reference to this binding is a reference to an unkown datum. This is a real problem because we can have declarative yielders which cannot be eliminated. The transformations preserve all bindings produced by an incorporated action. This is reflected in rules 5.44, 5.45, 5.47 and 5.48 by the use of the preservation action 'bind all' in the target actions.

A similar explanation applies to transients. Moreover, 'give all' in Rule 5.49 is not necessary because the 'with' operation expects a datum to be passed to the abstraction.

The empty output substitution in the rule for 'enact' is consistent with the preservation policy for incorporated actions.

As a final observation, notice that 'allocate' actions inside incorporated actions (for example, see Rule 5.44) are transformed (if the storage allocation context is *true*).

Example 5.9. The two 'bind' actions in

```
| | give abstraction (bind "x" to 4) or give abstraction (bind "y" to true)
| then
| | enact the abstraction
hence
| give the integer bound to "x" or give the truth-value bound to "y"
```

[3] An occurrence of 'the s bound to k' is free in the incorporated action if there is no 'bind' action which produced a binding for k (see Section 6.3.4).

(ELSE)

$$\frac{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,\mathit{false}) \vdash a_1 \Rightarrow a_1',\mathcal{T}_1',\mathcal{B}_1',\mathcal{S}_1' \qquad \mathcal{K},\mathcal{T},\mathcal{B},(l,q,\mathit{false}) \vdash a_2 \Rightarrow a_2',\mathcal{T}_2',\mathcal{B}_2',\mathcal{S}_2'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash a_1 \text{ else } a_2 \Rightarrow a_1'' \text{ else } a_2'',[\,],[\,],\mathit{greater}(\mathcal{S}_1',\mathcal{S}_2')} \tag{5.50}$$

where $a_1'' = a_1'$ and then (give all $\mathcal{T}_1'$ and bind all $\mathcal{B}_1'$)
$a_2'' = a_2'$ and then (give all $\mathcal{T}_2'$ and bind all $\mathcal{B}_2'$)

(RECURSIVELY BIND)

$$\frac{\mathcal{K},[\,],\mathcal{B}' \oslash \mathcal{B},(l,q,e) \vdash \text{closure (abstraction } a) \Rightarrow y'}{\mathcal{K},\mathcal{T},\mathcal{B},(l,q,e) \vdash \text{recursively bind } k \text{ to closure (abstraction } a) \Rightarrow a',[\,],\mathcal{B}',(l,q+1,e)} \tag{5.51}$$

where $a' =$ store y' in cell (l,q)
$\mathcal{B}' =$ [the _ stored in cell(l,q)/the _ bound to k]

Figure 5.10: Elimination rule for 'else' and 'recursively bind'.

clearly bind tokens to known values (4 and true), but as we do not know which abstraction will be enacted, we cannot eliminate the tokens 'x' and 'y' in the last line of the action. Notice that the above action remains unchanged by the transformations. □

The example above, and some of the above observations, are pathological situations which are unlikely to be found in program actions.

5.8.7 Hybrid

Figure 5.10 shows the elimination rule for 'else' and 'recursively bind'. In Rule 5.51, the important point (see the antecedent) is that we pre-allocate a cell to store the closed abstraction, and transform the incorporated action having this in mind.

5.9 Sort Updating Rules

As mentioned before, action transformations are partly guided by sort information. This sort information is affected by the elimination rules. For example, the action 'bind "x" to the integer', whose sort, before binding elimination, might be:

$$(\{0:4\},\{\}) \hookrightarrow (\{\},\{x:4\})$$

is transformed into the action 'complete', whose sort is:

$$(\{\}, \{\}) \hookrightarrow (\{\}, \{\}).$$

Clearly the transformation affected sort information. We have defined a set of rules which specify the updating of sort information after a transformation is performed. For example, for the 'bind' action we have:

$$\frac{\begin{array}{c}\mathcal{K} \vdash \textsf{bind } k \textsf{ to } y : (\tau, \beta) \hookrightarrow (\{\}, \{k : d\}) \\ \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash \textsf{bind } k \textsf{ to } y \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'\end{array}}{\mathcal{K} \vdash a' : (\tau \ominus \mathit{dom}\ \mathcal{T}, \beta \ominus \mathit{dom}\ \mathcal{B}) \hookrightarrow (\{\}, \{\})} \tag{5.52}$$

which justifies and calculates the new sort (in this case the sort of the '**complete**' action).

The *sort updating rules* for actions have the following form:

$$\frac{\mathcal{K} \vdash a : s \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'}{\mathcal{K} \vdash a' : s'} \tag{5.53}$$

where the new sort s' is calculated in terms of the previous sort s, and the input and output substitutions. For yielders we have:

$$\frac{\mathcal{K} \vdash y : s \qquad \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y'}{\mathcal{K} \vdash y' : s'} \tag{5.54}$$

where s' is now calculated in terms of s and the input substitutions.

An implementation for the rules (in ACTRESS) must not throw away sort information, because the code generator still needs it. Notice that the sort updating rules are simpler than the original sort rules [14] as they do not rely on any sort inference process such as the one built into the sort checker. We start from an already-decorated action tree and just calculate sorts having in mind the sort rules.

5.10 Some Examples

We give here some examples of the application of the transformations defined in this chapter. The first ones are interesting actions we came across during the implementation of the rules, and actions typically found in program actions of languages like PASCAL. The two last examples are program actions for the *loop* program of Figure 4.3 and the *factorial* program of Figure 4.24.

Example 5.10. Using elimination Rule 5.20 and Law 5.3, the action

```
| give 3 label #1 and give sum (5,4) label #2
then
| give the integer#2
```

is transformed into

```
| give sum (5,4) label #2
then
| give the integer#2
```

Notice that, in this particular example, there was no substitution to be made because the transient '3' is not used by the second subaction of 'then'. We have eliminated a dead action! □

Example 5.11. Using elimination rules 5.31, 5.27, 5.30, 5.21, 5.34 and laws 5.7 and 5.5 the action

```
| bind "x" to 1
hence
| | furthermore bind "x" to 3
| hence
| | give successor (the integer bound to "x")
```

is transformed into

```
give successor (3)
```

The 'furthermore' action became a dead action after the elimination of 'bind "x" to 1' and was eliminated (Law 5.5). □

Example 5.12. Using elimination rules 5.31, 5.27, 5.30, 5.21, 5.34, 5.12, and laws 5.6 and 5.5 the action

```
| | bind "x" to 1
| hence
| | furthermore give successor (the integer bound to "x")
hence
| bind "y" to the integer bound to "x"
```

is transformed into

```
| give successor (1)
and then
| bind "y" to 1
```

We have eliminated only the binding for 'x'. The target action still produces a binding for 'y'. We had to keep this binding (see Rule 5.12) because it is part of the observational behaviour of the original source action. Notice again that the '**furthermore**' action was eliminated because there were no more bindings to propagate after the '**bind**' action for 'x' was eliminated. On a whole, although the binding elimination was partial, the obtained action is a lot simpler than the one we started with! □

Example 5.13. The source action

```
| | allocate a cell then give it label #1
| and
| | allocate a cell then give it label #2
then
| | bind "x" to the cell#1
| and
| | bind "y" to the cell#2
| and
| | store the cell#2 in the cell#1
| and
| | store 5 in the cell#2
```

is transformed into

```
| | store cell (0,1) in cell (0,0)
| and
| | store 5 in cell (0,1)
and then
| | bind "x" to cell (0,0)
| and
| | bind "y" to cell (0,1)
```

Again, although we had not eliminate all bindings, the target action is smaller and simpler than the source action. Notice the transformation of the two '**allocate**' actions using Rule 5.39. □

Example 5.14. The source action

```
| | | bind "c" to 100
| | and
| | | allocate a cell then bind "x" to it
| before
| | | furthermore
| | | | allocate a cell then bind "x" to it
| | hence
| | | store the integer bound to "c" in the cell bound to "x"
hence
| store successor (the integer bound to "c") in the cell bound to "x"
```

is transformed into

```
| store 100 in cell (0,1)
hence
| store successor (100) in cell (0,0)
```

Notice here the drastic simplification of the source action and the total elimination of bindings. Also the '**hence**' combinator could well be replaced by the '**and then**' combinator (although in terms of code generation in ACTRESS, no difference exists between '**hence**' and '**and then**' for this case). □

Example 5.15. The source action

```
| | give 4 label #1
| and
| | give abstraction (give successor (the integer bound to "x")) label #2
then
| | bind "x" to 7 and give the integer#1 label #1
| hence
| | enact closure (the abstraction#2)
```

is transformed into

```
| | give abstraction (give successor (the integer bound to "x")) label #2
| then
| | enact (abstraction (bind "x" to 7) hence (the abstraction#2))
and then
| give 4 label #1
```

Notice the preservation of the '**give**' action in line 1 of the source action (last line of the target action). The '**bind**' action in line 5 of the source action was initially eliminated, but it had to be reintroduced because of the '**closure**' operation in line 7 (Rule 5.49). The free occurrence of 'the integer bound to "x"' in the incorporated action (line 1 of the target action) was not simplified (Rule 5.44). □

Example 5.16. The source action

```
| | allocate a cell then bind "x" to it
| before
| | bind "p" to closure abstraction
| | | | furthermore bind "x" to the cell
| | | hence
| | | | give successor (3) then store it in the cell bound to "x"
hence
| | store 5 in the cell bound to "x"
| and then
| | enact (the abstraction bound to "p" with the cell bound to "x")
| and then
| | give the integer stored in the cell bound to "x"
```

is transformed into

```
| store abstraction
| | | store the cell in cell (1,0)
| | hence
| | | | give successor(3)
| | | then
| | | | store the datum in the cell stored in cell (1,0)
| in
| | cell (0,1)
hence
| | | store 5 in cell (0,0)
| | and then
| | | enact (the abstraction stored in cell (0,1) with cell (0,0))
| and then
| | give the integer stored in cell (0,0)
```

The abstraction bound to 'p' in the source action was stored in 'cell (0,1)' in the target action. Notice how the 'bind' action for 'x' in line 1 of the source action was eliminated (rules 5.39 and 5.27), and how all the references to the binding it produces were eliminated from the source action. □

Example 5.17. Figure 5.11 and Figure 5.12 show the transformed program action and transformed object code, respectively, for the *loop* program of Figure 4.3. Both are simpler

```
| store 0 in cell(0,0)
hence
| | store 1000000 in cell(0,0)
| and then
| | unfolding
| | | | | | give the value stored in cell(0,0)
| | | | | then
| | | | | | give the value label #1
| | | | then
| | | | | give isGreaterThan(the value#1,0)
| | | then
| | | | | | | | | give the value stored in cell(0,0)
| | | | | | | | then
| | | | | | | | | give the value label #1
| | | | | | | then
| | | | | | | | give difference(the integer#1,1)
| | | | | | then
| | | | | | | store the value in cell(0,0)
| | | | | and then
| | | | | | unfold
| | | | else
| | | | | complete
```

Figure 5.11: Transformed program action for the *loop* program.

and smaller than the original program action (Figure 4.2) and original object code (Figure 4.27). (As we shall see in Section 7.1 the obtained object code is also much efficient than the original one.) □

Example 5.18. The transformed program action for the *factorial* program of Figure 4.24 is shown in Figure 5.13. The generated object code is shown in Figure 5.14. Notice the elimination of all run-time binding operations in the transformed object code (compare with Figure 5.2). □

5.11 Implementation

The implementation of the transformations introduces a new component in ACTRESS, called the *transformer*, between the sort checker and code generator. Thus, the action notation compiler becomes:

$$anc_t = code_t \circ transform \circ check \circ parse \tag{5.55}$$

```
DATUM _d1, _d2, _d3;

int main()
{
        _d1 = _MAKE_CELL(&static_area[0]);
        static_area[1] = _d1;
        *_d1.datum.cell = _MAKE_INTEGER(0);
        *static_area[1].datum.cell = _MAKE_INTEGER(1000000);

_repeat_1:
        _d1 = *static_area[1].datum.cell;
        _d2 = _IS_GREATER_THAN(_d1, _MAKE_INTEGER(0));
        if (_d2.datum.truth_value) {
                _d1 = *static_area[1].datum.cell;
                _d3 = _DIFFERENCE(_d1, _MAKE_INTEGER(1));
                *static_area[1].datum.cell = _d3;
                goto _repeat_1;
        } else {
                ;               /* complete */
        };
        exit(0);
_failure_0:
        exit(1);
}
```

Figure 5.12: Generated object code for the *loop* program after transformation.

```
| | | give cell(0,0) label #2
| | then
| | | | store the cell#2 in cell(0,1)
| | | and
| | | | store 0 in the cell#2
| before
| | store (abstraction
| | | | | give headOf(the list)
| | | | then
| | | | | | | give sort-check(the integer) label #1
| | | | | | and
| | | | | | | give cell(1,0) label #2
| | | | | then
| | | | | | | store the cell#2 in cell(1,1)
| | | | | | and
| | | | | | | store the integer#1 in the cell#2
| | | hence
| | | | | | | | | | give the integer stored in the cell stored in cell(1,1)
| | | | | | | | | then
| | | | | | | | | | give the integer label #1
| | | | | | | | then
| | | | | | | | | give (the integer#1 is 0)
| | | | | | | then
| | | | | | | | | | store 1 in the cell stored in cell(0,1)
| | | | | | | | | else
| | | | | | | | | | | | | | | | | | give the integer stored in
| | | | | | | | | | | | | | | | | | the cell stored in cell(1,1)
| | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | give the integer label #1
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | give difference(the integer#1,1)
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give listOf(the datum)
| | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | enact (the abstraction stored in cell(0,2) with the list)
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | | give the integer stored in
| | | | | | | | | | | | | | | | | | | | | the cell stored in cell(1,1)
| | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | give the integer label #1
| | | | | | | | | | | | | | | | | | | and
| | | | | | | | | | | | | | | | | | | | | give the integer stored in
| | | | | | | | | | | | | | | | | | | | | the cell stored in cell(0,1)
| | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | give the integer label #2
| | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | give product(the integer#1,the integer#2)
| | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | store the integer in
| | | | | | | | | | | | | | | | | | | the cell stored in cell(0,1)) in cell(0,2)
| | | | | | | | | | | | hence
| | | | | | | | | | | | | | give listOf(10)
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | enact (the abstraction stored in cell(0,2) with the list)
```

Figure 5.13: Transformed program action for the *factorial* program.

```
int _abs0(_din, _bin, _dout, _bout)
        DATUM _din; BINDINGS _bin; DATUM *_dout; BINDINGS *_bout;
{       DATUM _d1, _d2, _d3, _d4, _d5, _d6; BINDINGS _b1;
        /* call sequence omitted */
        _d1 = _din;
        _d2 = _HEAD_OF(_d1);
        _d3 = _SORT_CHECK(_d2, 4L);
        _d4 = _MAKE_CELL(display[1] + 0);
        *(display[1] + 1) = _d4;
        *_d4.datum.cell = _d3;
        _d2 = *(*(display[1] + 1)).datum.cell;
        _d3 = _IS(_d2, _MAKE_INTEGER(0));
        if (_d3.datum.truth_value) {
                *static_area[1].datum.cell = _MAKE_INTEGER(1);
        } else {_d2 = *(*(display[1] + 1)).datum.cell;
                _d4 = _DIFFERENCE(_d2, _MAKE_INTEGER(1));
                _d2 = _LIST(_d4);
                _d4 = _WITH(static_area[2], _d2);
                if ((_d4.datum.abs->codeact)
                        (_d4.datum.abs->datum, _d4.datum.abs->bindings, &_d5, &_b1))
                                goto _failure_0;
                _d2 = *(*(display[1] + 1)).datum.cell;
                _d4 = *static_area[1].datum.cell;
                _d6 = _PRODUCT(_d2, _d4);
                *static_area[1].datum.cell = _d6;
        };
        *_bout = NULL;
        /* return sequence omitted */
        return (0);
_failure_0:
        return (1);
}

DATUM _d1, _d2, _d3; BINDINGS _b1;

int main()
{       _d1 = _MAKE_CELL(&static_area[0]);
        static_area[1] = _d1;
        *_d1.datum.cell = _MAKE_INTEGER(0);
        static_area[2] = _ABSTRACTION(_abs0);
        _d1 = _LIST(_MAKE_INTEGER(10));
        _d2 = _WITH(static_area[2], _d1);
        if ((_d2.datum.abs->codeact)
                (_d2.datum.abs->datum, _d2.datum.abs->bindings, &_d3, &_b1))
                        goto _failure_0;
        exit(0);
_failure_0:
        exit(1);
}
```

Figure 5.14: Object code for the *factorial* program after action transformation.

(PROGRAM ACTION)

```
simp a =
   let
       fun initSortVarCounter () = TransformerBase.lastSortVar := 0
       val _ = initSortVarCounter ()
       val (a',T',B',(l',q',e',nx')) = simpAction a emptysub emptysub (0,0,true,1)
       val a'' = preserve T' B'
       val a'_sd = getDecoration a'
       val a''_sd = getDecoration a''
       val and_then_sd = Action
           {trans_in  = merge (transients_in a'_sd)  (transients_in a''_sd),
            binds_in  = merge (bindings_in a'_sd)    (bindings_in a''_sd),
            trans_out = merge (transients_out a'_sd) (transients_out a''_sd),
            binds_out = merge (bindings_out a'_sd)   (bindings_out a''_sd)}
       val a''' = AND_THEN(a',a'',and_then_sd)
   in
       INFO(a''',l',q',nx')
   end
```

Figure 5.15: Implementation of the elimination rule for the program action.

where the transformer is denoted by the function *transform*.

The introduction of the transformer also requires some modifications to the code generator and run-time system. In Equation 5.55, $code_t$ is an extension to *code* in Equation 4.1 which includes these modifications.

5.11.1 Action Notation Transformer

The transformer is divided into two modules: one implements the elimination rules together with the sort updating rules, and the other implements the action notation laws. The transformer is a composition of two functions:

$$transform = \ law \ \circ \ simp \tag{5.56}$$

with the following types:

$$\begin{array}{lcl} transform & : & sort\ AST \texttt{ -> } sort\ AST \\ law & : & sort\ AST \texttt{ -> } sort\ AST \\ simp & : & sort\ AST \texttt{ -> } sort\ AST \end{array}$$

The simplification function *simp* implements the program action elimination rule, and it is defined as shown in Figure 5.15. The transformed program action a''' is enclosed by an information node, *INFO*(*a*,0,*q*,*nx*), which contains the number of cells q statically allocated at level 0. nx is the maximum nesting level of abstractions in a which will be needed by the extended code generator (see Section 5.11.2)[4]. It defines how many display registers will be needed at run-time. Every '`abstraction`' operation in a will also be enclosed by an information node. The information node is used by the code generator to implement the cell reservations needed as explained in Section 5.4. The '`preserve`' function in line 6 is used to build the preservation actions. '`a'_sd`' in line 7 is the sort decoration for action `a'`.

The judgements

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}' \tag{5.57}$$

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash y \Rightarrow y' \tag{5.58}$$

are implemented, respectively, by the functions *simpAction* and *simpYielder*, whose types are:

sort AST `->` *transient substitution* `->` *binding substitution* `->` *int* $*$ *int* $*$ *bool* $*$ *int*
`->`
(*sort AST* $*$ *transient substitution* $*$ *binding substitution* $*$ (*int* $*$ *int* $*$ *bool* $*$ *int*))

and

sort AST `->` *transient substitution* `->` *binding substitution* `->` *int* $*$ *int* $*$ *bool* $*$ *int*
`->`
sort AST

respectively.

The elimination pass is performed in a single traversal of the decorated action tree. Figure 5.16 shows a fragment of the function *simpAction* correspondent to the implementation of rules 5.13, 5.15, and 5.31. Figure 5.17 shows the implementation of rules 5.24, 5.25, 5.34, 5.35, 5.36, 5.42 and 5.44. Notice in both cases how the form of the implementation is close to the rules' definition.

Figure 5.18 shows a fragment of the implemention of the *law* function which implements rules 5.3 and 5.9.

[4]We have omitted nx from the elimination rules as it does not add anything to the them.

(COMPLETE)

```
simpAction (COMPLETE sd) T B C = (COMPLETE complete_sort,emptysub,emptysub,C)
```

(AND)

```
simpAction (AND(a1,a2,sd)) T B (C as (l,q,e,nx)) =
    let
        val (a1',T1',B1',C1' as (l1',q1',e1',nx1')) = simpAction a1 T B C
        val (a2',T2',B2',C2') = simpAction a2 T B (l,q1',e,nx1')
        val T' = merge_sub T1' T2'
        val B' = merge_sub B1' B2'
        val sd' = Action {trans_in  = deldom (labelsOfSub T)  (transients_in sd),
                          binds_in  = deldom (tokensOfSub B)  (bindings_in sd),
                          trans_out = deldom (labelsOfSub T') (transients_out sd),
                          binds_out = deldom (tokensOfSub B') (bindings_out sd)}
    in
        (AND(a1',a2',sd'),T',B',C2')
    end
```

(HENCE)

```
simpAction (HENCE(a1,a2,sd)) T B (C as (l,q,e,nx)) =
    let
        val (a1',T1',B1',C1' as (l1',q1',e1',nx1')) = simpAction a1 T B C
        val (a2',T2',B2',C2') = simpAction a2 T B1' (l,q1',e,nx1')
        val T' = merge_sub T1' T2'
        val B' = B2'
        val sd' = Action {trans_in  = deldom (labelsOfSub T)  (transients_in sd),
                          binds_in  = deldom (tokensOfSub B)  (bindings_in sd),
                          trans_out = deldom (labelsOfSub T') (transients_out sd),
                          binds_out = deldom (tokensOfSub B') (bindings_out sd)}
    in
        (HENCE(a1',a2',sd'),T',B',C2')
    end
```

Figure 5.16: Implementation of elimination rules (actions).

(THE)

```
simpYielder (THE(s,n,sd)) T B (l,q,e,nx) =
   ((case (apl_sub (THEVAR n) T) of VOID => (THE(s,n,sd),nx)
                                  | d    => (d,nx))
    handle Apl => (THE(s,n,sd),nx))
```

(BOUND)

```
simpYielder (BOUND_TO(s,INDIVIDUAL(TOKEN k,indsd),sd)) T B (l,q,e,nx) =
   ((case (apl_sub (BOUNDVAR k) B) of VOID => (BOUND_TO
                                                 (s,INDIVIDUAL(TOKEN k,indsd),sd),nx)
                                    | d    => (d,nx))
    handle Apl => (BOUND_TO(s,INDIVIDUAL(TOKEN k,indsd),sd),nx))
```

(STORED)

```
simpYielder (STORED_IN(s,y,sd)) T B C =
   let
      val (y',nx') = simpYielder y T B C
      val sd' = Dependent {trans_in  = deldom (labelsOfSub T) (transients_in sd),
                           binds_in  = deldom (tokensOfSub B) (bindings_in sd),
                           sort_out = sort_out sd}
   in
      (STORED_IN(s,y',sd'),nx')
   end
```

(ABSTRACTION)

```
simpYielder (ABSTRACTION(a,sd)) T B (l,q,e,nx) =
   let
      val (a',T',B',(l',q',e',nx')) =
          simpAction a emptysub emptysub (l+1,0,e,max(nx,l+1))
      val sd' = (lastSortVar := !lastSortVar + 1;
                 SortVar (~(!lastSortVar),
                 ref (Abstraction
                        {trans_in  = deldom (labelsOfSub T) (transients_in sd),
                         binds_in  = deldom (tokensOfSub B) (bindings_in sd),
                         trans_out = transients_out sd,
                         binds_out = bindings_out sd})))
   in
      (INFO(ABSTRACTION(AND_THEN(a',preserve T' B',Nothing),sd'),l+1,q',nx - nx'),nx')
   end
```

Figure 5.17: Implementation of elimination rules (yielders).

(AND)

```
law (AND(a1,a2,sd)) =
let
   val a1' = law a1
   val a2' = law a2
in
   case (a1',a2') of (COMPLETE _,a) => law a
                   | (a,COMPLETE _) => law a
                   | (FAIL sd,a)    => FAIL sd
                   | (x,y)          => AND(x,y,sd)
end
```

(CLOSURE)

```
law (CLOSURE(y,sd)) = let
                         val beta = bindings_in sd
                      in
                         if beta = emptymap
                            then law y
                         else CLOSURE(law y,sd)
                      end
```

Figure 5.18: Implementation of laws.

We do not have any problem with application order of the rules, as most of them are mutually exclusive. The implementation (by pattern matching in ML) is deterministic, the rules being ordered in such a way as to match the larger patterns first. The cases where a transformation originates possibilities of further transformations are handled satisfactorily by the composition of Equation 5.56.

5.11.2 Changes in the Code Generator

The code generator presented in Chapter 4 was extended to use the information node and to recognize and translate adequately the term cell (l,q). The changes are as follows:

- All the statically allocated cells for the program action are located in a datum array (*static_area*). If the information node of the program action is $INFO(a,0,q,nx)$ the static array will be of size q.

- For every abstraction in the program action we designed a *entry sequence* and a *return sequence*. To generate code for an abstraction's entry sequence, the code generator makes use of the abstraction's information node.

- Access to local and non-local cells are made using the abstraction nesting level l, the cell q and a display register that is kept at run-time. The frames for each abstraction are still heap allocated though (see Section 5.13). The display is set up as a global array, and it is updated by the entry and return sequences.
- Modifications in the run-time environment. These include definition of the static area array, handle of entry and return sequences, etc.

Notice that, if total binding elimination is guaranteed, there is no need for translation of the '**closure**' operation as the program action will not have one (eliminated by Law 5.9). Also, in this case, there would be no need for the binding field in the run-time representation for abstractions (see Section 4.3.2), and the translation for '**enact**' would be simpler. Finally, all run-time binding operations could be removed from the run-time environment.

5.12 Exploring Relationships

The exploration of relationships between dynamic semantics, static semantics and transformation semantics, could reveal important properties. We would like to relate semantic rules, sort rules, and the various kinds of transformation rules (laws, elimination rules and sort updating rules).

The sort of an action can be related to its input and output substitutions. The following proposition establishes this relationship.

Proposition 5.1 (Transformation invariant) *Let a_s be a subaction of an arbitrary action a. If*

$$\mathcal{K} \vdash a_s : (\tau_s, \beta_s) \hookrightarrow (\tau'_s, \beta'_s)$$

and

$$\mathcal{K}, \mathcal{T}_s, \mathcal{B}_s, \mathcal{S}_s \vdash a_s \Rightarrow a'_s, \mathcal{T}'_s, \mathcal{B}'_s, \mathcal{S}'_s$$

holds, then

$$dom\ \mathcal{T}'_s \subseteq dom\ \tau'_s \wedge dom\ \mathcal{B}'_s \subseteq dom\ \beta'_s$$

holds.

In general we can say nothing about the domain of β and $\mathcal{B}$. However, for some special cases, such as the program action and incorporated actions, we could state some relation. For the program action:

$$dom\ \mathcal{B} = dom\ \beta$$

because we assume that program actions are closed. For an incorporated action a_i of an unclosed abstraction we have:

$$dom\ \mathcal{B}_i = \{\}$$

so

$$dom\ \mathcal{B}_i \subseteq dom\ \beta_i$$

But if the incorporated action expects no bindings, we have

$$dom\ \mathcal{B}_i = dom\ \beta_i = \{\}$$

On the other hand, an action can produce bindings which are not used by subsequent actions. In this case, and if we do not consider incorporated actions and unfolded actions, we could write:

$$dom\ \mathcal{T} \supseteq dom\ \tau \wedge dom\ \mathcal{B} \supseteq dom\ \beta$$

The following defines *bind-free action* in terms of binding substitutions.

Definition 5.3 (Bind-free action) *Let a_s be a subaction of an action a. If*

$$\mathcal{K} \vdash a : (\tau, \{\}) \hookrightarrow (\tau', \{\})$$

$$\mathcal{K} \vdash a_s : (\tau_s, \beta_s) \hookrightarrow (\tau'_s, \beta'_s)$$

and for every transformational step

$$\mathcal{K}, \mathcal{T}_s, \mathcal{B}_s, \mathcal{S}_s \vdash a_s \Rightarrow a'_s, \mathcal{T}'_s, \mathcal{B}'_s, \mathcal{S}'_s$$

the relations

$$dom\ \beta_s \subseteq dom\ \mathcal{B}_s \wedge dom\ \beta'_s \subseteq dom\ \mathcal{B}'_s$$

and

$$\mathsf{uneliminated} \notin range\ \mathcal{B}_s \wedge \mathsf{uneliminated} \notin range\ \mathcal{B}'_s$$

holds, then a is a bind-free action.

Consider the income bindings to an action and its transformational step. If for the transformational step all "tokens" required by the action (domain of β) are in the input binding substitution ($\mathcal{B}$), and none of the tokens received by the action is bound to '$\mathsf{uneliminated}$' (this would indicate that there is at least 'bind' action which was not eliminated), then all the free occurrences of '$\mathsf{the}\ s\ \mathsf{bound\ to}\ k$' are eliminated in this step. Consider now the outcome bindings from the action. If for the transformational step all "tokens" produced by the action (domain of β') are in the output binding substitution ($\mathcal{B}$), and none of the "tokens" produced by the action is bound to '**uneliminated**' (this would indicate that there is at least a '**bind**' action which was not eliminated in the action, or outside it, if the action propagate bindings), then all the '**bind**' actions of '**the** s **bound to** k' are eliminated in this step.

Proposition 5.2 *If a' is a bind-free action which satisfies*

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'$$

then all subactions a_s of a' have sorts of the form

$$\mathcal{K} \vdash a_s : (\tau_s, \{\}) \hookrightarrow (\tau_s', \{\})$$

for some τ_s and τ_s'.

Action transformations do not introduce ill-sorted actions. In practice we do not need to sort-check a target action; it can be delivered straight to the code generator.

Proposition 5.3 (Sort preservation) *If a is a well-sorted action and*

$$\mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{S} \vdash a \Rightarrow a', \mathcal{T}', \mathcal{B}', \mathcal{S}'$$

then a' is a well-sorted action.

Action transformations, in some sense, preserve semantics as defined by the semantic rules of Chapter 3. However, in order to make this statement precise, we need a non-trivial definition of action equivalence.

Definition 5.4 (Functional action equivalence) *We say that action a_1 is functionally equivalent to action a_2, $a_1 \cong a_2$, if and only if for all transients t, bindings b and storage s, if*

$$t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1$$

and

$$t, b, s \vdash a_2 \rhd o_2, t_2, b_2, s_2$$

then

$$o_1 = o_2 \wedge t_1 = t_2 \wedge b_1 = b_2$$

holds.

Although we have used the term "functional" equivalence, as one can see from the definition it means equivalence for the functional and declarative facets of actions[5].

Proposition 5.4 (Semantic faithfulness) *If a is a well-sorted program action and*

$$\mathcal{K} \vdash a \Rrightarrow a'$$

then $a \cong a'$.

We leave the proofs of the above propositions for future work.

5.13 Improvements

We identify below some points which deserve more study in order to improve binding elimination and storage allocation.

- **Classification of 'allocate' actions.** If a source program contains variable declarations within a block within a conditional or iterative command, for example, it is well known that static allocation is possible. The corresponding allocations in the program action will occur inside an 'or' or 'unfolding' action, however, and our existing transformer will not recognise that static allocation is safely possible. Further work is needed to detect this important special case.

- **Storage allocation for blocks.** As we do not identify source language's blocks in the program action, the use of storage is not the best one. For example, we do not reuse the deallocated cells on exit of a block. This would require a way to identify blocks in action notation and to modify the elimination rule for 'deallocate'. The use of 'deallocate' actions explicitly in action semantic descriptions is desirable, as it allows a better use of storage. Also some analysis is necessary so a memory block deallocation could be used instead of isolated deallocations.

- **Stack allocation.** Statically allocated cells for abstractions are allocated in the heap. They are not deallocated at abstraction exit (although they can be deallocated by explicit deallocate actions). We need to keep these cells because references to them could outlive the incorporated action performance. For example, a local statically

[5] The functional equivalence just defined is not a congruence; we recognize that we ought to (somehow) consider the imperative facet.

allocated cell can be assigned to a global variable. Also, if cells are ordinary values, they could be passed as transients, passed in bindings produced by the abstraction, or stored in global cells. The impact — of the non-deallocation of statically allocated cells — in the object code is that it allocates more storage than object code for hand-written compilers. Detection whether a stack storage organization could be used is desirable. Maybe life-time analysis can be a source of inspiration to improve this situation.

- **Bindings produced by abstractions.** The transformations do not eliminate bindings produced by abstractions. In general, this is an impossible task. But probably, for some special cases, these bindings could be eliminated. To consider this we would need a way to identify abstractions, to record the binding substitution for them and to use that substitution whenever the abstraction is enacted.

- **Eager strategy.** We have no evidence that the *eager strategy* adopted in the transformations — '**bind**' actions are eliminated and later, if we find out that they are not eliminable, are preserved — is the better one. Notice that, when a '**bind**' action has to be preserved the strategy introduces no overheads. In fact, in this case we replaced the '**bind**' action by a '**store**' action which is more efficient, and replaces all '**the** s **bound to** k' by store lookups, which again are more efficient (a binding lookup represents a search over a binding environment at run-time). We think these efficient lookups compensate the preservation of the non-eliminable '**bind**' action.

Finally, it would be desirable to have a condition on the programming language being defined, that is, a condition on the action semantic description, which guarantees total binding elimination. For example, if the source language is statically scoped according to some definition, then total binding elimination is guaranteed for every source program. In the next chapter, we will address the problem of establish a statically scoped condition that is language-based.

Chapter 6

Binding Analysis in Action Semantics

This chapter studies bindings in action semantics. We begin by reviewing the binding concepts in programming languages. In the sequel we show that action notation has similar concepts. We continue by giving a precise definition of what we mean by a statically scoped action, and set sufficient conditions which identifies a statically scoped subset of action notation. The final part of the chapter extends this condition to action semantic descriptions and gives a sufficient condition for a statically scoped programming language.

6.1 Initial Observations

We want to state a condition which allows the identification of statically scoped languages based on their action semantic descriptions. That is, by analysing an action semantic description, and checking that it satisfies the condition, we will be able to say that the language it defines is statically scoped. The main result is a sufficient statically scoped condition. Although the condition is not a definitive result, we believe that it covers most of conventional languages and can be of some practical use.

Total binding elimination can only be achieved for statically scoped languages. For dynamically scoped languages it is possible to eliminate some bindings, but not all of them. In the Actress context, a generated compiler could incorporate different versions of the transformer, code generator and run-time system, depending on the binding discipline of the source language. Which version to use could be decided at compiler generation time, using the statically scoped condition: if it tells that the language is statically scoped then the specialized versions for statically scoped languages are incorporated, otherwise the default versions are used. We expect that this parameterization will improve the compilation time of generated compilers.

A simple initial and known condition is that the 'closure' operation is applied to every abstraction as soon as it is formed — 'closure (abstraction a)'. This guarantees that all the bindings to be used by the incorporated action a will be frozen at abstraction formation time. Unfortunately, we will see that one can have a language which satisfies this initial condition but is not statically scoped.

6.2 Bindings in Programming Languages

The notion of binding in programming languages is derived from a mathematical concept [102]. For example, consider the following equations:

$$x = 4$$

$$y = x^2 + 3$$

One can see the first equation as establishing a *binding* between the variable x and the number 4. Similarly the second equation establishes a binding between variable y and the right-hand side expression. To evaluate this expression we need to know what is the value of x or, put in another way, what is *bound* to x. Clearly x is *free* in the right-hand side expression.

These terms (binding, bound, free) are used with the same meaning for programming languages. As pointed out in [102] it seems that programmers feel more ease with assignment-related concepts (updating, locations, stores, commands, lifetime) than with these binding-related concepts. In (denotational) semantics of programming languages and functional programming however, binding-related concepts are widely used and understood. These concepts will be examined in detail in the following subsections.

6.2.1 What is a Binding?

The first concept is that of *binding* itself. A binding is an association of an identifier to the value (entity) it denotes. In general it is produced by a declaration of the identifier. For example, the SPECIMEN declaration

$$\textbf{var}\ x\ :\ int\ \texttt{:=}\ \texttt{4}$$

establishes a binding of the identifier x to a storage cell. (It also stores the integer 4 in the cell.) One can only say this based on the semantics of SPECIMEN (a simple inspection of the declaration does not tell this). Notice also that the actual binding will be made at run-time, that is, it is a *late binding*. In principle, we do not know what particular cell will be bound to x. For efficiency reasons it is important to detect whether a late binding can be turned into an *early binding*, that is, a binding made at compile time. In this particular example, the actual binding can be made earlier to a (relative) *known* storage location. The objective is to detect as many early bindings as possible. But we can not treat all bindings as early bindings.

6.2.2 Binding and Applied Occurrences

The second concept is that of *binding occurrence* and *applied occurrence* of identifiers. A binding occurrence is one that creates a binding for the identifier; an applied occurrence is one where the identifier is used. In the SPECIMEN program of Figure 6.1 the occurrences of y in line 7 and *fact* in line 9 are binding occurrences whereas the one in lines 11 and 14, and 13 and 20 are applied occurrences.

Usually the meaning of an applied occurrence of an identifier is established at a *unique* and *explicit* binding occurrence of the same identifier; the applied occurrence is in the *scope* of a binding occurrence for the identifier. (As we shall see, there are cases where this uniqueness is not satisfied.) For example, the applied occurrence of y in line 11 of Figure 6.1 is said to be in the scope of the binding occurrence of y in line 7. Alternatively, one could say that a binding occurrence establishes a region on the program text where all occurrences of the bound identifier have the meaning given by the binding occurrence, that is, the identifier is *visible*, with its declared meaning, in this region. For example, the scope of the declaration of n in line 9 goes from line 10 to line 15. Which binding

```
program loopfact is
    const
        n : int = 1000000;
    var
        x : int := 0;
    var
        y : int := 0;
    proc
        fact (n : int) =
            if (n = 0)
                then y := 1
            else
                call fact (n - 1);
                y := n * y
            end
in
    x := n;
    while (x > 0)
        do
            call fact (10);
            x := x - 1
        end
end
```

Figure 6.1: The SPECIMEN *loopfact* program.

occurrence of an identifier applies to an applied occurrence is determined by the language's *scope rules*.

6.2.3 Environment

An *environment* is a collection of bindings. In denotational semantics, for example, it is modelled as a function from identifiers to their semantic values. In general, expressions are evaluated relative to an environment, for example, the evaluation of the expression

$$x^2 + 3y + 20$$

in the environment[1] $[x \mapsto 4, y \mapsto 0, z \mapsto -1]$ yields 36.

6.2.4 Free Identifiers

Finally we have the important notion of *free occurrence* of an identifier in a phrase. An identifier is said to be free if it has an applied occurrence in a phrase which is not in the

[1] For convenience we have presented the environment as a mapping.

scope of any binding occurrence local to that phrase. All occurences of y inside procedure *fact* in Figure 6.1 are free occurrences.

6.2.5 Static and Dynamic Bindings

Programming languages are usually equipped with some form of abstraction. PASCAL, for example, has two forms of abstractions: procedures and functions. In functional languages, the abstraction mechanism is provided by means of functions. In general, an abstraction can be seen as the abstraction name, its input parameters, its output parameters, and its body which is a phrase such as an expression or statement.

In [102], Tennent begins the explanation of the notion of free identifiers of procedural abstracts with the following:

> In PASCAL, free identifier occurrences are bound in the environment of the *abstract*. This is known as *static* binding, because the binding occurrence is determined “statically”, that is to say, without executing the program; furthermore, the binding occurrence does not change during program execution.

Let us examine in more detail the above quotation.

Firstly, Tennent mentions only *free* variables. That is because binding occurrences for the non-free (locally defined) identifiers are in the abstraction's body and they are the same no matter from where the abstraction is called.

Secondly, he refers only to abstractions (“abstracts”). We can justify this because only abstractions can be called from different places; blocks (such as expressions) cannot be “called” from anywhere else, so their free identifiers' binding occurrences are always established statically. For example, the (block) expression in lines 10–14 of the program shown Figure 6.2 has a free identifier, x, whose binding occurrence is at line 3; this binding occurrence does not vary when we run the program. The value assigned to y in line 13 will always be 20 plus the value most recently assigned to x.

Finally, Tennent refers to the environment where the abstraction was defined. So, regarding abstractions, there are the following important points in the program text: the point where an abstraction is defined, and the points where the abstraction is invoked. This is exactly where the difference between static and dynamic bindings resides. In languages with a *dynamic* binding discipline, free identifiers in the abstraction are bound

```
program locale is
    const c : int = 3;
    var x : int := 4                                    (3)
in
    local
        var x : int := 5
    in
        x := 10 + c
    end;
    local                                               (10)
        var y : int := 6
    in
        y := 20 + x                                     (13)
    end;                                                (14)
    x := x + c
end
```

Figure 6.2: The SPECIMEN *locale* program.

in the invocation environment. Binding occurrences of free identifiers can therefore change during program execution. In imperative languages with dynamic bindings, abstractions are functions not only of the store but also of the environment of their invocations.

Unfortunately, the absence of abstraction mechanisms does not guarantee a static binding discipline of a language, as the following example illustrates.

Example 6.1. Consider the following SPECIMEN (artificial) expression:

```
let                                                     (1)
  const x = if b                                        (2)
               then let const v = 2 in 4 + v end        (3)
            else                                        (4)
               let const v = 8 in 5 end                 (5)
            end                                         (6)
in                                                      (7)
  x + v                                                 (8)
end                                                     (9)
```

According to the semantics given in Appendix B, the above expression evaluates to $6 + v$ if b evaluates to *true*, or to $5 + v$ if b evaluates to *false*. v (in line 8) and b are free identifiers defined somewhere else outside the expression.

We can change the semantics of the 'let' expression, so that bindings defined in it escape. This is achieved by simply using the '**before**' combinator instead of '**hence**' in the semantic equation for 'let':

```
evaluate ⟦ "let" D:Declaration "in" E:Expression "end" ⟧ =
    | furthermore elaborate D
    before
    | evaluate E .
```

Now the evaluation of the expression changes: if b evaluates to *true* then the expression evaluates to 8, otherwise it evaluates to 13. Furthermore, one cannot determine statically what is the binding occurrence for the applied occurrence of v at line 8. With the change above we turned SPECIMEN into a dynamically scoped language. □

Another example could be a language with conditional declarations.

6.3 Bindings in Action Notation

As a formal language for describing programming languages, action notation is general enough to describe both statically scoped and dynamically scoped languages. So, in general, action notation is dynamically scoped. However one can define properties which identify statically scoped actions. This is one of the objectives of this chapter. Before we state the statically scoped condition, we examine the main binding concepts in the case of action notation, and give some definitions.

6.3.1 Binding

A binding in action notation is created by the 'bind' action. Syntactically, there are precise points in an action where bindings are created. For example, in the action

```
| | bind "c" to 1000
| before
| | | | give 0 label #1
| | | and
| | | | | allocate a cell
| | | | then
| | | | | give the cell label #2
| | then
| | | | bind "x" to the cell#2
| | | and
| | | | store the value#1 in the cell#2
hence
| | give the value bound to "c"
| then
| | store the value in the cell bound to "x"
```

the 'bind' actions in line 1 and 9 create the bindings $\{c \mapsto 1000\}$ and $\{x \mapsto c\}$ respectively (assuming that 'allocate a cell' at line 5 allocates cell c).

But when we consider abstractions, although the points where bindings can be produced are quite clear, the points where they are used are not. This is because the enaction of an abstraction may produce bindings, and the points where these bindings will take effect are not always statically determined. Consider for example the action

```
| ...
and
| give closure abstraction
| | | bind "p" to the abstraction
| | hence
| | | | enact the abstraction bound to "p"
| | | hence
| | | | give successor (the integer bound to "x")
and
| ...
```

The yielder 'the abstraction' (line 4) evaluates to an unknown abstraction. Because of this, we do not know which abstraction will be enacted at line 6, so we do not know which 'bind' action produced the binding expected by the 'give' action in line 8.

A binding $\{k \mapsto d\}$ does not change during the performance of an action. It can only be hidden or overriden by a new 'bind' action for k. For example, in the action

```
| bind "x" to 4
hence
| | a1
| hence
| | bind "x" to true hence a2
```

and assuming that there are no 'bind' actions for 'x' in a_1 and a_2, the bindings $\{x \mapsto 4\}$ and $\{x \mapsto \text{true}\}$ will remain constant during the performance of a_1 and a_2 respectively.

Notice that a binding for the same token k can be created many times. A token k can also be bound to a different value each time a binding for k is created. Consider, for example, the action

```
unfolding
| | bind "x" to the integer
| hence
| | give successor (the integer bound to "x") then unfold
```

and assume it is given an integer d. For each iteration of the unfolded action, a new binding for 'x' is created. In the first iteration, the binding $\{\mathsf{x} \mapsto d\}$ is created. In subsequent iterations, the bindings $\{\mathsf{x} \mapsto succ^n\ d\}$, are successively created ($n = 1, 2, \ldots$). The binding for 'x' is not changing; a new one is being created at each iteration.

6.3.2 Binding and Applied Occurrences

In action notation, a binding occurrence corresponds to the production of a binding. As we saw before, this is achieved by the 'bind' action. That is, an occurrence of the action 'bind k to y' defines a binding occurrence for token k in its scope.

An applied occurrence is an occurrence of the declarative yielder 'the s bound to k'. It defines points where a binding for k is used, or, more precisely, the datum bound to k is used.

For example, in the action of Figure 6.3, we identify a binding occurrence for token 'x' in line 5. By following the flow of bindings through the action, applied occurrences for 'x' are identified in lines 23, 25, 34 and 43. The binding occurrence for all these applied occurrences is the one in line 5. To find out the binding occurrence for each applied occurrence of 'x' we have used implicitly our knowledge of the scope rules for bindings in action notation.

6.3.3 Binding Environment

The outcome of an action performance depends on the *binding environment* for the action. For example, the action

give successor (the integer bound to "x")

has outcome (*completed*, $\{\mathsf{0} \mapsto \mathsf{5}\}, \{\}, \{\}$) when performed in the binding environment $\{\mathsf{x} \mapsto \mathsf{4}\}$, and outcome (*completed*, $\{\mathsf{0} \mapsto \mathsf{10}\}, \{\}, \{\}$) when performed in the binding environment $\{\mathsf{x} \mapsto \mathsf{9}\}$.

The binding environment, at a particular point in the action, is determined by the flow of bindings during action performance. The action combinators define precisely this binding flow. The semantic rules of Chapter 3 formalize this behaviour. However, due to the presence of abstractions (as discussed in Section 6.3.1), the flow of bindings is not always explicit.

```
| | | | give 4 then give the value label #1
| | | and
| | | | allocate a cell then give the cell label #2                    (3)
| | then
| | | | bind "x" to the cell#2                                          (5)
| | | and
| | | | store the value#1 in the cell#2
| before
| | recursively bind "add" to closure
| | | abstraction
| | | | | furthermore
| | | | | | | give headOf (the fun-argument-list)
| | | | | | then
| | | | | | | bind "n" to the value
| | | | hence
| | | | | | | | | | give the value bound to "n"
| | | | | | | | | or
| | | | | | | | | | give the primitive-value stored
| | | | | | | | | | in the cell bound to "n"
| | | | | | | | then
| | | | | | | | | give the value label #1
| | | | | | | and
| | | | | | | | | | give the value bound to "x"                        (23)
| | | | | | | | | or
| | | | | | | | | | give the primitive-value stored
| | | | | | | | | | in the cell bound to "x"                           (26)
| | | | | | | | then
| | | | | | | | | give the value label #2
| | | | | | then
| | | | | | | give sum (the integer#1,the integer#2)
| | | | | then
| | | | | | give the fun-result
hence
| | give 8 then store the value in the cell bound to "x"                (34)
| and then
| | | | give 3 then give list (the datum)
| | | then
| | | | enact
| | | | | | the function bound to "add"
| | | | | with
| | | | | | the fun-argument-list
| | then
| | | store the value in the cell bound to "x"                          (43)
```

Figure 6.3: Binding and applied occurrences for an action.

The action in lines 34-43 in Figure 6.3 is performed in the following binding environment:

$$\{\mathsf{x} \mapsto c, \mathsf{add} \mapsto \mathit{abstraction}(...)\}$$

if we assume that '**allocate a cell**' in line 3 allocates cell c when performed.

6.3.4 Free Tokens

Action notation has a notion of *free tokens* which corresponds to that of *free variables*. In the '**give**' action of the previous section, because there is no binding occurrence for the token 'x', we say that *the token 'x' is free in the action.*

Important occurrences of free tokens are the ones inside incorporated actions. As these actions are frozen and can be performed in different places, the question we immediately ask is from what binding environment the incorporated action inherits its free tokens. For an abstraction whose incorporated action expects no bindings, that is, one that has no free tokens, one can determine the binding occurrence for any applied occurrence of a token k, for any k in the incorporated action top level.

We can be more precise about free tokens in actions using sort information:

Definition 6.1 (Action free tokens) *If action a has sort*

$$(\tau, \{\}) \rightsquigarrow (\tau', \beta')$$

then it has no free tokens.

Notice that all closed abstractions have no free tokens, as their sorts are $(\tau_i, \{\}) \rightsquigarrow (\tau_i', \beta_i')$.

We know exactly where in an (program) action one can have free tokens:

- In the top level. But we do not allow this. Program actions are closed.
- In incorporated actions. The abstraction needs to be closed somewhere before enaction to "eliminate" its free tokens.
- After the enaction of an unknown abstraction which produces bindings. For example, in

```
| enact the abstraction bound to "f"
hence
| give successor (the integer bound to "x")
```

the applied occurrence of token 'x' in line 3 is free.

6.3.5 Static and Dynamic Bindings

Finally the concept of static scopedness and dynamic scopedness applies naturally to action notation. The action in Figure 6.3 is statically scoped because one can determine the binding occurrence for every applied occurrence of a token in the action.

The action below is dynamically scoped:

```
| bind "x" to 4 or bind "x" to 7
hence
| give the integer bound to "x"
```

The binding environment for the 'give' action is either $\{x \mapsto 4\}$ or $\{x \mapsto 7\}$. The actual binding environment will only be known at performance time.

The following is another example of a dynamically scoped action:

```
| give abstraction (give successor (the integer bound to "x"))
then
| | | bind "x" to 4
| | hence
| | | enact closure (the abstraction) then give it label #1
| and
| | | bind "x" to 6
| | hence
| | | enact closure (the abstraction) then give it label #2
```

In this example there is not a unique binding occurrence for the applied occurrence of token 'x' at line 1.

6.4 Statically Scoped Actions

From the analogy made with programming languages, we have learned about bindings in action notation and have developed our intuition. Now we can define (informally) and state the statically scoped condition.

Definition 6.2 (Statically scoped action) *An action a is statically scoped if for every applied occurrence* 'the s bound to k' *there exists a unique* 'bind' *action which produces the binding for token k.*

Definition 6.3 (Dynamically scoped action) *An action a which is not statically scoped is said to be dynamically scoped.*

The above definition is very general. A test to verify if an action is statically scoped could be a decision procedure which accepts an action as its input, and tells if the action is definitely statically scoped or dynamically scoped. If an action is dynamically scoped, this procedure could go one step further and say which bindings in the action are static and which ones are dynamic. This kind of annotation could be useful as a basis for action transformations.

If the decision procedure does not give a definite result, then we could have a statically scoped action which would not be classified as so.

6.4.1 Statically Scoped Condition

In establishing a statically scoped condition, the first point we have to consider is concerned with unclosed abstractions. We will have to check whether abstractions are closed immediately after they are formed. All occurrences of the 'abstraction' operation must be immediately enclosed by the 'closure' operation. With this restriction, any free tokens in the incorporated action become non-free, and the enaction of the abstraction becomes independent of the binding environment. If the incorporated action has no free tokens then it does not have to be closed.

If conditional actions ('or' and 'else') produce bindings, and these bindings are used, then we will certainly have dynamic bindings. So we have to check whether conditional actions produce bindings and whether these bindings get used. For example, the action

```
| bind "x" to 4 or bind "x" to 30
hence
| give the integer bound to "x"
```

is dynamically scoped, whereas

```
| bind "x" to 4 or bind "x" to 30
hence
| give successor (1)
```

is statically scoped.

For 'unfolding' we have to consider two points. The first point is that, if an unfolded action produces a binding for a token, the binding must be produced by the same 'bind' action no matter how many times the unfolded action is performed. For example, the action

```
unfolding
| | unfold hence unfold
| or
| | bind ”x” to 3
```

is statically scoped, whereas

```
unfolding
| | unfold hence bind ”x” to true
| or
| | bind ”x” to 3
```

is dynamically scoped (notice that both actions diverge). In the first action, no matter the number of iterations for the unfolded action, the binding for ‘x’ produced will be always the one created by the ‘bind’ action in line 4. In the case of the second action, the binding for ‘x’ produced by the unfolded action can be created by the ‘bind’ action in line 2 or the ‘bind’ action in line 4.

The second point is better introduced with an example. Consider the following action:

```
| bind “x” to 0
hence
| unfolding | | bind “x” to successor (the integer bound to “x”)
|           | hence
|           | | | check (the integer bound to “x” is 10) and then complete
|           | | or
|           | | | check not (the integer bound to “x” is 10) and then unfold
```

Clearly the ‘unfolding’ action does not produce bindings, but clearly there is not a unique ‘bind’ action for ‘the integer bound to “x”’ in line 3. Therefore the action is dynamically scoped. This is however a feature hard to find in real programming languages. In contrast, the following action is statically scoped:

```
| bind “x” to 0
hence
| unfolding | | furthermore (give successor (the integer bound to “x”))
|           | hence
|           | | | check (the integer bound to “x” is 10) and then complete
|           | | or
|           | | | check not (the integer bound to “x” is 10) and then unfold
```

We noticed that when the unfolded action has free tokens, and when the bindings received by the free '**unfold**' actions do not remain static during the performance of '**unfolding**', then there is no unique binding occurrence for those free tokens. That is, for all iterations, the bindings which reach all '**unfold**' actions and contain free tokens of the unfolded action, must have been produced by the same '**bind**' action as the ones that reaches the unfolded action.

Finally, we observed that incorporated actions that produce bindings are a source of dynamic bindings. This is true because we cannot always determine the identity of the abstraction being enacted, so we do not know, in general, where the bindings produced by an '**enact**' action came from. For example, the action

```
| give closure abstraction (bind "x" to 1) or give closure abstraction (bind "x" to 2)
then
| | enact the abstraction
| hence
| | ... the s bound to "x" ...
```

is dynamically scoped (we do not know which is the binding occurrence for token 'x' in the last line).

An alternative would be to prohibit conditional actions from giving abstractions, *and* to prohibit abstractions from being storable. If, for an action a, conditional actions do not give abtractions, then each abstraction can be uniquely identified by following the flow of data through the action. (The possibility of abstractions bound to tokens in conditional actions is caught by the restriction on the bindings produced by these actions.) An action which stores abstractions has the potential to originate dynamic bindings. A stored abstraction can be fetched from store and enacted. In general we do not know the content of the cell, so we do not know which particular abstraction will be enacted.

For an example illustrating how storable abstractions can introduce dynamic bindings, consider the action

```
| | allocate a cell
| then
| | | bind “x” to the cell
| | and
| | | store closure abstraction (bind “y” to 12) in the cell
| | and
| | | store abstraction (bind “y” to 30) in the cell
hence
| | enact the abstraction stored in the cell bound to “x”
| hence
| | give successor (the integer bound to “y”)
```

The ‘store’ action in line 7 stores a new abstraction in the cell allocated in line 1, which is the abstraction enacted in line 9. The ‘give’ action in line 11 “sees” the binding $\{\mathsf{y} \mapsto 30\}$ instead of $\{\mathsf{y} \mapsto 12\}$.

We state now the statically scoped condition for actions:

Condition 6.1 (Statically scoped action) *If in an action a:*

1 *every occurrence of the* ‘abstraction *a*’*, where a contains free tokens, is enclosed by the* ‘closure’ *operation,*

2 *no conditional action produces bindings which are used,*

3 *no* ‘unfolding’ *action produces bindings which were not produced by a unique* ‘bind’ *action,*

4 *every* ‘unfold’ *action receives the same bindings as the enclosing unfolded action, and*

5 *no incorporated action produces bindings,*

then a is statically scoped.

In the next section we will be more precise about the observations made in this section.

6.4.2 Formalisation

Before we formalise the definition of statically scoped action we introduce some machinery. We will formalise the knowledge of bindings and the uniqueness of binding occurrences using the following judgement:

$$\mathcal{K}, \mathcal{B}, n \vdash a \downarrow \mathcal{B}', n'$$

where

- $\mathcal{K}$ is sort information concerned only with bindings.

- $\mathcal{B}$ and $\mathcal{B}'$ are mappings from tokens to the powerset of natural numbers. For an element $\{k \mapsto \{n_1, n_2, \ldots, n_m\}\}$ of $\mathcal{B}$, we can assert that there are m '**bind**' actions (or m binding occurrences) for the free applied occurrences '**the** s **bound to** k' of a. Moreover, these '**bind**' actions are uniquely tagged $n_1, n_2, \ldots, n_m$. A similar explanation applies to $\mathcal{B}'$ (but regarding the binding occurrences produced by a).

- n and n' are natural numbers.

- a is a well-sorted action.

The interpretation of the judgement is that $\mathcal{B}$ is the set of binding occurrences received by a and $\mathcal{B}'$ the binding occurrences produced by a.

For a well-sorted yielder we have

$$\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n'$$

One can think of $\mathcal{B}$ and $\mathcal{B}'$ as *binding occurrence annotations* on actions, and $\mathcal{B}$ and $[\,]$ as binding occurrence annotations on yielders (the evaluation of a yielder produces no binding occurrences).

The following observations can be easily inferred from the judgements above:

- If $\mathcal{K} \vdash a : \beta \hookrightarrow \beta'$ then $dom\ \beta \subseteq dom\ \mathcal{B}$ and $dom\ \beta' = dom\ \mathcal{B}'$.

- $\mid \mathcal{B}(k) \mid \ = 1$ means that there is only a unique '**bind**' action (binding occurrence) for token k, and this '**bind**' action is known[2].

- $\mid \mathcal{B}(k) \mid \ > 1$ means that there is potentially more than one '**bind**' action (binding occurrence) for token k.

- If a is well-sorted then $\beta(k)$ is always defined for all occurrences of k in a. This implies that $\mathcal{B}(k)$ is also always defined.

Binding annotation for Actress action notation is shown in Figures 6.5–6.10. Most of the rules are straight forward and can be obtained directly from the semantic rules of Chapter 3 considering only binding information. In the following we will discuss the interesting cases.

[2] $|s|$ means the cardinality of set s.

The 'combine' operation ($\diamond$) used in the rules is defined as:

$$\{\} \diamond \mathcal{B}_2 = \mathcal{B}_2 \tag{6.1}$$

$$\mathcal{B}_1 \diamond \{\} = \mathcal{B}_1 \tag{6.2}$$

$$(k \mapsto w_1 \cdot \mathcal{B}_1) \diamond (k \mapsto w_2 \cdot \mathcal{B}_2) = \{k \mapsto (w_1 \cup w_2)\} \oslash (\mathcal{B}_1 \diamond \mathcal{B}_2) \tag{6.3}$$

$$(k \mapsto w_1 \cdot \mathcal{B}_1) \diamond \mathcal{B}_2 = \{k \mapsto w_1\} \oslash (\mathcal{B}_1 \diamond \mathcal{B}_2), \text{ if } k \notin dom\ \mathcal{B}_2 \tag{6.4}$$

Figure 6.5 contains the annotation rules for basic action notation. In the rule for 'unfolding' we use the "unfolded" judgement which rules are in Figure 6.4. The place holders in the *unfolded annotation rules* are for $\mathcal{K}$, $\mathcal{B}$ and n, and are to be interpreted as saying that these variables have the same behaviour as that specified by the annotation rules. In the rule for 'unfold', the variables $\mathcal{A}$ and $\mathcal{U}$ are auxiliary binding annotations. The former is used to *record* the output binding annotation for the unfolded action. The latter is used to record the *combined* output binding annotations for all free 'unfold' actions in the unfolded action.

Each antecedent in the annotation rule for 'unfolding' (Rule 6.13) represents an analysis iteration over the unfolded action. In the first iteration (first antecedent), as one does not know what is the output binding annotation for the unfolded action, nor the combined output binding annotation for the 'unfold' actions, it has empty annotations for these variables. From the second iteration on (see the second antecedent) the input binding annotation for the unfolded action is the combination of the previous one ($\mathcal{B}$) and the combined output binding annotation for all 'unfold' actions ($\mathcal{U}_1$). Finally, the output binding annotation for the 'unfolding' action combines the output obtained at each iteration.

Rule 6.14 in Figure 6.5 specifies the binding annotation for the 'or' combinator. When a conditional action produces a binding for k, we do not know which 'bind' action produced this binding, so we associate k to the union of the sets associated to k by a_1 and a_2. For example, for the action

bind "x" to 1 or bind "x" to true or bind "x" to list (3)

we would have $\{\mathsf{x} \mapsto \{1, 2, 3\}\}$ as its output binding occurrence annotation.

(UNFOLDED)

$$\mathcal{K}, \mathcal{B}, \mathcal{A}, \mathcal{U}, n \vdash \mathsf{unfold} \looparrowright \mathcal{A}, \mathcal{A}, \mathcal{B} \diamond \mathcal{U}, n \tag{6.5}$$

$$\_, \_, \mathcal{A}, \mathcal{U}, \_ \vdash \textit{primitive-action} \looparrowright \_, \mathcal{A}, \mathcal{U}, \_ \tag{6.6}$$

$$\frac{\_, \_, \mathcal{A}, \mathcal{U}, \_ \vdash a \looparrowright \_, \mathcal{A}', \mathcal{U}', \_}{\_, \_, \mathcal{A}, \mathcal{U}, \_ \vdash \textit{prefix-combinator}\ a \looparrowright \_, \mathcal{A}', \mathcal{U}', \_} \tag{6.7}$$

$$\frac{\_, \_, \mathcal{A}, \mathcal{U}, \_ \vdash a_1 \looparrowright \_, \mathcal{A}_1', \mathcal{U}_1', \_ \qquad \_, \_, \mathcal{A}_1', \mathcal{U}_1', \_ \vdash a_2 \looparrowright \_, \mathcal{A}_2', \mathcal{U}_2', \_}{\_, \_, \mathcal{A}, \mathcal{U}, \_ \vdash a_1\ \textit{infix-combinator}\ a_2 \looparrowright \_, \mathcal{A}_2', \mathcal{U}_2', \_} \tag{6.8}$$

Figure 6.4: Unfolded annotation rules.

(COMPLETE)

$$\mathcal{K}, \mathcal{B}, n \vdash \mathsf{complete} \downarrow \{\}, n \tag{6.9}$$

(FAIL)

$$\mathcal{K}, \mathcal{B}, n \vdash \mathsf{fail} \downarrow \{\}, n \tag{6.10}$$

(AND)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K}, \mathcal{B}, n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K}, \mathcal{B}, n \vdash a_1\ \mathsf{and}\ a_2 \downarrow \mathcal{B}_1' \oplus \mathcal{B}_2', n_2'} \tag{6.11}$$

(AND THEN)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K}, \mathcal{B}, n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K}, \mathcal{B}, n \vdash a_1\ \mathsf{and\ then}\ a_2 \downarrow \mathcal{B}_1' \oplus \mathcal{B}_2', n_2'} \tag{6.12}$$

(UNFOLDING)

$$\frac{\begin{array}{c}\mathcal{K}, \mathcal{B}, \{\}, \{\}, n \vdash a \looparrowright \mathcal{B}_1, \mathcal{A}_1, \mathcal{U}_1, n_1 \\ \mathcal{K}, \mathcal{B} \diamond \mathcal{U}_1, \{\} \diamond \mathcal{B}_1, \{\} \diamond \mathcal{U}_1, n \vdash a \looparrowright \mathcal{B}_2, \mathcal{A}_2, \mathcal{U}_2, n_2 \\ \cdots \\ \mathcal{K}, \mathcal{B} \diamond \mathcal{U}_1 \diamond \ldots \diamond \mathcal{U}_{m-1}, \{\} \diamond \mathcal{B}_1 \diamond \ldots \diamond \mathcal{B}_{m-1}, \{\} \diamond \mathcal{U}_1 \diamond \ldots \diamond \mathcal{U}_{m-1}, n \vdash a \looparrowright \mathcal{B}_m, \mathcal{A}_m, \mathcal{U}_m, n_m\end{array}}{\mathcal{K}, \mathcal{B}, n \vdash \mathsf{unfolding}\ a \downarrow \mathcal{B}_1 \diamond \mathcal{B}_2 \diamond \ldots \diamond \mathcal{B}_m, n_m} \tag{6.13}$$

(OR)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K}, \mathcal{B}, n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K}, \mathcal{B}, n \vdash a_1\ \mathsf{or}\ a_2 \downarrow \mathcal{B}_1' \diamond \mathcal{B}_2', n_2'} \tag{6.14}$$

Figure 6.5: Annotation rules for basic action notation.

(GIVE)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{give } y \textsf{ label } \#n \downarrow \{\}, n'} \tag{6.15}$$

(CHECK)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{check } y \downarrow \{\}, n'} \tag{6.16}$$

(THEN)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a_1 \downarrow \mathcal{B}'_1, n'_1 \qquad \mathcal{K}, \mathcal{B}, n'_1 \vdash a_2 \downarrow \mathcal{B}'_2, n'_2}{\mathcal{K}, \mathcal{B}, n \vdash a_1 \textsf{ then } a_2 \downarrow \mathcal{B}'_1 \oplus \mathcal{B}'_2, n'_2} \tag{6.17}$$

(THE)

$$\mathcal{K}, \mathcal{B}, n' \vdash \textsf{ the } s\#n \downarrow n' \tag{6.18}$$

Figure 6.6: Annotation rules for functional action notation.

Figure 6.7 shows the annotation rules for declarative action notation. Notice in the rule for the '**bind**' action (Rule 6.19), that n is incremented to reflect the tagging of the '**bind**' action.

Figure 6.9 shows the binding occurrence annotations for reflective action notation. The output binding occurrence annotation for the '**enact**' action (Rule 6.28) is based on sort information for the action. The rule specifies that the output binding occurrence annotation is formed by associating each token produced by the action with the set of all natural numbers ($\mathbb{N}$). The interpretation is that as we do not know which '**bind**' actions produced the binding for these tokens, we say that every '**bind**' action potentially produces bindings for those tokens. Notice that, if in '**enact** y', y forms an abstraction, we could be more precise about these '**bind**' actions.

We distinguish two cases for the '**abstraction**' operation. One is when it occurs inside a '**closure**' operation (Rule 6.31), and the other is when it occurs in isolation (Rule 6.29). In the first case the input annotation for the incorporated action is the input annotation for the '**closure**' operation. In the second case, the incorporated action input annotation is unknown and we proceed as in the rule for '**enact**', except that here we are interested in the input bindings to the incorporated action.

We are ready now to formalize Definition 6.2 (statically scoped action):

(BIND)

$$\frac{\mathcal{K},\mathcal{B},n \vdash y \downarrow n'}{\mathcal{K},\mathcal{B},n \vdash \textsf{bind}\ k\ \textsf{to}\ y \downarrow \{k \mapsto \{n'\}\}, n'+1} \tag{6.19}$$

(FURTHERMORE)

$$\frac{\mathcal{K},\mathcal{B},n \vdash a \downarrow \mathcal{B}', n'}{\mathcal{K},\mathcal{B},n \vdash \textsf{furthermore}\ a \downarrow \mathcal{B}' \oslash \mathcal{B}, n'} \tag{6.20}$$

(HENCE)

$$\frac{\mathcal{K},\mathcal{B},n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K},\mathcal{B}_1',n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K},\mathcal{B},n \vdash a_1\ \textsf{hence}\ a_2 \downarrow \mathcal{B}_2', n_2'} \tag{6.21}$$

(MOREOVER)

$$\frac{\mathcal{K},\mathcal{B},n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K},\mathcal{B},n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K},\mathcal{B},n \vdash a_1\ \textsf{moreover}\ a_2 \downarrow \mathcal{B}_2' \oslash \mathcal{B}_1', n_2'} \tag{6.22}$$

(BEFORE)

$$\frac{\mathcal{K},\mathcal{B},n \vdash a_1 \downarrow \mathcal{B}_1', n_1' \qquad \mathcal{K},\mathcal{B}_1' \oslash \mathcal{B},n_1' \vdash a_2 \downarrow \mathcal{B}_2', n_2'}{\mathcal{K},\mathcal{B},n \vdash a_1\ \textsf{before}\ a_2 \downarrow \mathcal{B}_2' \oslash \mathcal{B}_1', n_2'} \tag{6.23}$$

(BOUND)

$$\mathcal{K},\mathcal{B},n \vdash\ \textsf{the}\ s\ \textsf{bound to}\ k \downarrow n \tag{6.24}$$

Figure 6.7: Annotation rules for declarative action notation.

(STORE)

$$\frac{\mathcal{K},\mathcal{B},n \vdash y_1 \downarrow n_1' \qquad \mathcal{K},\mathcal{B},n_1' \vdash y_2 \downarrow n_2'}{\mathcal{K},\mathcal{B},n \vdash \textsf{store}\ y_1\ \textsf{in}\ y_2 \downarrow \{\}, n_2'} \tag{6.25}$$

(DEALLOCATE)

$$\frac{\mathcal{K},\mathcal{B},n \vdash y \downarrow n'}{\mathcal{K},\mathcal{B},n \vdash \textsf{deallocate}\ y \downarrow \{\}, n'} \tag{6.26}$$

(STORED)

$$\frac{\mathcal{K},\mathcal{B},n \vdash y \downarrow n'}{\mathcal{K},\mathcal{B},n \vdash \textsf{the}\ s\ \textsf{stored in}\ y \downarrow n'} \tag{6.27}$$

Figure 6.8: Annotation rules for imperative action notation.

(ENACT)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n' \qquad \mathcal{K} \vdash \textsf{enact}\ y : \beta \hookrightarrow \beta' \qquad \mathcal{B}' = \{k \mapsto \mathbb{N} \mid k \leftarrow dom\ \beta'\}}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{enact}\ y \downarrow \mathcal{B}', n'} \tag{6.28}$$

(ABSTRACTION)

$$\frac{\mathcal{K} \vdash a : \beta \hookrightarrow \beta' \qquad \mathcal{B}' = \{k \mapsto \mathbb{N} \mid k \leftarrow dom\ \beta\} \qquad \mathcal{K}, \mathcal{B}', n \vdash a \downarrow \mathcal{B}'', n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{abstraction}\ a \downarrow n'} \tag{6.29}$$

(WITH)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y_1 \downarrow n'_1 \qquad \mathcal{K}, \mathcal{B}, n'_1 \vdash y_2 \downarrow n'_2}{\mathcal{K}, \mathcal{B}, n \vdash y_1\ \textsf{with}\ y_2 \downarrow n'_2} \tag{6.30}$$

(CLOSURE)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a \downarrow \mathcal{B}', n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{closure}\ (\textsf{abstraction}\ a) \downarrow n'} \tag{6.31}$$

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{closure}\ y \downarrow n'} \tag{6.32}$$

Figure 6.9: Annotation rules for reflective action notation.

(ELSE)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash a_1 \downarrow \mathcal{B}'_1, n'_1 \qquad \mathcal{K}, \mathcal{B}, n'_1 \vdash a_2 \downarrow \mathcal{B}'_2, n'_2}{\mathcal{K}, \mathcal{B}, n \vdash a_1\ \textsf{else}\ a_2 \downarrow \mathcal{B}'_1 \diamond \mathcal{B}'_2, n'_2} \tag{6.33}$$

(RECURSIVELY BIND)

$$\frac{\mathcal{K}, \{k \mapsto \{n\}\} \oslash \mathcal{B}, n+1 \vdash y \downarrow n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{recursively bind}\ k\ \textsf{to}\ y \downarrow \{k \mapsto \{n\}\}, n'} \tag{6.34}$$

(ALLOCATE)

$$\frac{\mathcal{K}, \mathcal{B}, n \vdash y \downarrow n'}{\mathcal{K}, \mathcal{B}, n \vdash \textsf{allocate}\ y \downarrow \{\}, n'} \tag{6.35}$$

Figure 6.10: Annotation rules for hybrid actions.

Definition 6.4 (Statically scoped action) *Let a be a well-sorted (program) action. If*

$$\mathcal{K}, \{\}, n \vdash a \downarrow \mathcal{B}', n'$$

and for every occurrence of '$\textsf{the}\ s\ \textsf{bound to}\ k$*' in a we have*

$$\mathcal{K}, \mathcal{B}, n \vdash \textsf{the}\ s\ \textsf{bound to}\ k \downarrow n'$$

and $\mid \mathcal{B}(k) \mid\ = 1$*, then a is statically scoped.*

If an action satisfies Condition 6.1 (statically scoped action), then the action is statically scoped according to Definition 6.4.

We can easily identify the cases which break Definition 6.4 by inspection of the annotation rules:

- There is a possibility of introduction of multiple binding occurrences for free tokens of incorporated actions of unclosed abtractions (see Rule 6.29 in Figure 6.9, all tokens k in $\mathcal{B}'$ satisfy $\mid \mathcal{B}(k) \mid\ > 1$). This can be avoided by insisting that all abstractions are closed immediatelly after they are formed (see Rule 6.31 in Figure 6.9). This agrees with Condition 6.1.1.

- By inspection of rules 6.14 and 6.33 one can see, by the presence of the combine operation, that conditional actions can introduce the possibility of $\mid \mathcal{B}(k) \mid\ > 1$. This only happens if the subactions produce bindings and these bindings are used. So, if conditional actions do not produce bindings no possibility of multiple binding occurrences arises from them. Condition 6.1.2 states exactly this.

- An '**unfolding**' action is another point where $\mid \mathcal{B}(k) \mid\ > 1$ can be introduced. By looking at the annotation rule for '**unfolding**' (Rule 6.13), we can eliminate this possibility by restricting the unfolded action from producing any binding. This is exactly what Condition 6.1.3 says.

- If Condition 6.1.4 is satisfied then the input binding occurrence annotation to the unfolded action is the same for all iterations (rules 6.13 and 6.5).

- Finally, in the rule for '**enact**' we can see that if $\beta' \neq \{\}$ then there is the possibility of introducing multiple binding occurrences for the free tokens in the scope of the '**enact**' action. If we restrict $\mathcal{B}'$ to be empty in Rule 6.28, that is, no abstraction produces bindings, then we eliminate this possibility. This is exactly what is stated by Condition 6.1.5.

6.4.3 Implementation

A prototype implementation of the static scopedness condition was done in two steps. The first step annotates the action according to the annotation rules. The second step just inspects the annotations for the applied occurrences (yielders of the form 'the s bound to k'). If, for all such applied occurrences, $|\mathcal{B}(k)| = 1$ holds, then the action is statically scoped, otherwise we do not know. Figure 6.11 shows an extract of the ML program which implements the binding occurrences annotation. Figure 6.12 shows an extract of the implementation of the statically scoped condition.

6.5 Statically Scoped Languages

We extend here the statically scoped condition from individual actions to action semantic descriptions, and define what a statically scoped language is, based on the language's action semantic description.

6.5.1 Statically Scoped Condition

Figure 6.13 shows the abstract syntax for an action semantic description. Each semantic function composes part of the program action which denotes the source program. By making sure that, given an action semantic description of a source language, only statically scoped actions are composed, we can say that the language is statically scoped. If in the right hand side of one or more semantic equation, there exists a possibility that a dynamically scoped action can be composed, then the action semantic description has the possibility to be of a dynamically scoped language.

Condition 6.2 (Statically scoped action semantic description) *If in an action semantic description $\mathcal{D}$ for a programming language $\mathcal{L}$, the right hand side of every semantic equation is such that*

1 *every occurrence of the* 'abstraction' *operation is enclosed by the* 'closure' *operation,*

2 *no conditional action produces bindings,*

3 *no* 'unfolding' *action produces bindings which were not produced by a unique* 'bind' *action,*

(TYPES)

```
annotateAction : (string,int set) map -> (string,int set) map ->
   (string,int set) map -> int -> ((string,'a) map * (string,'b) map) AST ->
   ((string,int set) map * (string,int set) map) AST * (string,int set) map *
   (string,int set) map * (string,int set) map * (string,int set) map * int

annotateYielder : (string,int set) map -> int ->
   ((string,'a) map * (string,'b) map) AST ->
   ((string,int set) map * (string,int set) map) AST * int
```

(OR)

```
annotateAction B Ua Ui n (OR(a1,a2,_)) =
   let
      val (a1',Bi1',Bo1',Ua1',Ui1',n1') = annotateAction B Ua Ui n a1
      val (a2',Bi2',Bo2',Ua2',Ui2',n2') = annotateAction B Ua1' Ui1' n1' a2
   in
      (OR(a1',a2',(B,combine Bo1' Bo2')),B,combine Bo1' Bo2',Ua2',Ui2',n2')
   end
```

(BIND)

```
annotateAction B Ua Ui n (BIND_TO(INDIVIDUAL(TOKEN k,_),y,_)) =
   let
      val (y',n') = annotateYielder B n y
      val anot = (B,singlemap(k,singleton n'))
      val anotk = (B,emptymap)
   in
      (BIND_TO(INDIVIDUAL(TOKEN k,anotk),y',anot),
       B,singlemap(k,singleton n'),Ua,Ui,n'+1)
   end
```

(ENACT)

```
annotateAction B Ua Ui n (ENACT(y,(beta,beta'))) =
   let
      val (y',n') = annotateYielder B n y
      val B' = makeUnknownBindings beta'
   in
      (ENACT(y',(B,B')),B,B',Ua,Ui,n')
   end
```

(BOUND)

```
annotateYielder B n (BOUND_TO(s,INDIVIDUAL(TOKEN k,_),(_,_))) =
   let
      val (s',n') = annotateYielder B n s
      val annotation = (B,emptymap)
   in
      (BOUND_TO(s',INDIVIDUAL(TOKEN k,annotation),annotation),n)
   end
```

Figure 6.11: The implementation of binding occurrences annotation (extract).

(TYPES)

```
static : ((string,'a set) map * 'b) AST -> bool
```

(STATICALLY SCOPED CONDITION)

```
static (ABSTRACTION(a,d))           = static a
static (AND(a1,a2,d))               = (static a1) andalso (static a2)
static (AND_THEN(a1,a2,d))          = (static a1) andalso (static a2)
static (AND_THEN_MOREOVER(a1,a2,d)) = (static a1) andalso (static a2)
static (BIND_TO(k,y,d))             = static y
static (BOUND_TO(r,INDIVIDUAL(TOKEN k,_),(B,B'))) =
    if (card (apl k B)) = 1 then true else false
```

Figure 6.12: The implementation of the statically scoped condition (extract).

asd	::=	$sf_1 \ldots sf_n$	(semantic description)
sf	::=	$fy\ se_1 \ldots se_n$	(semantic function definition)
fy	::=	$\mathcal{F} :: binfo$	(functionality)
se	::=	$\mathcal{F}\ [\![\ op\ v_1 : S_1 \ldots v_n : S_n\ ]\!] = a$	(semantic equation)
a	::=	complete	(action)
	|	fail	
	|	...	
	|	$\mathcal{F}\ v$	(semantic function application)
$binfo$	::=	*bindings*	
	|	*no-bindings*	

Figure 6.13: Abstract syntax of action semantic descriptions.

4 *every '*unfold*' action receives the same bindings as its unfolded action, and*

5 *no incorporated action produces bindings,*

then $\mathcal{D}$ is statically scoped.

Definition 6.5 (Statically scoped language) *Let $\mathcal{D}$ be an action semantic description for a language $\mathcal{L}$. If $\mathcal{D}$ is statically scoped then we say that $\mathcal{L}$ is a statically scoped language.*

Proposition 6.1 *If a language $\mathcal{L}$ is statically scoped then all $\mathcal{L}$ program actions are statically scoped.*

Proposition 6.2 *If for a language $\mathcal{L}$ there exists a program action that is dynamically scoped then $\mathcal{L}$ is dynamically scoped.*

The proofs of the above propositions are left for future work.

6.5.2 Formalisation and Implementation

We defer for future work the formalisation and implementation of an analysis which determines whether a language is statically scoped.

One basic difference from the analysis employed for actions is that we do not have precise binding information in an action semantic description. At most we know whether an action produces or does not produce bindings. Tokens are not present in a semantic description (only token variables). Another difference is the presence of semantic function applications on the right hand sides of semantic equations.

We have done some exploration work towards the formalisation of the statically scoped condition on action semantic descriptions. Below some points which might be of some help in tackling the problem.

An analysis could use binding information, provided by the specifier in the form of the functionality for semantic functions, to analyse occurrences of semantic function applications in the right hand sides of semantic equations. Restrictions could be imposed (enforced) based on Condition 6.2. For example, for the 'or' action we could have:

$$\frac{\mathcal{E} \vdash a_1 \Downarrow \mathit{bindings}, h_1 \qquad \mathcal{E} \vdash a_2 \Downarrow \mathit{bindings}, h_2}{\mathcal{E} \vdash a_1 \ \mathsf{or}\ a_2 \Downarrow \mathit{bindings}, \mathit{dynamic}} \tag{6.36}$$

where $\mathcal{E}$ is a semantic function environment used to keep information about semantic functions. The constant *bindings* indicates that the action produces bindings, and *dynamic* that the action is dynamically scoped. The rule says that, if for the action 'a_1 **or** a_2', a_1 and a_2 may produce bindings, then, no matter if the subactions are statically scoped or dynamically scoped (h_1 and h_2), the '**or**' is not considered to be a statically scoped action.

For semantic function application we could have:

$$\mathcal{F} : (b, h) \cdot \mathcal{E} \vdash \mathcal{F}\ v \Downarrow (b, h) \tag{6.37}$$

That is, for a semantic function application in the right hand side of a semantic equation, we look up in the semantic function environment the "type" for $\mathcal{F}$. After the analysis of each right hand side of a semantic equation, its type is updated.

6.6 Discussion and Applicability

By stating the statically scoped condition, we have defined a subset of ACTRESS action notation which exhibits the statically scoped property.

The study of bindings in action semantics made in this chapter, might be useful when incorporated in an *action semantic description analyser* (see Chapter 7), which could inform the language designer about the binding discipline of the source language.

The annotation rules might be also used to check if the bindings which reach every '**unfold**' are the same which reach the unfolded action (for all iterations), which is what is required by the elimination rule for '**unfolding**' (Rule 5.17).

Notice that our condition for statically scoped actions can be used to tell if a program $\mathcal{P}$ (program action) of a dynamically scoped language is statically scoped.

It would be nice if one could safely identify cases of dynamically scoped source languages. Also, it remains to be seen if a definitive result can be achieved.

In general the binding discipline for a programming language is specified informally and using syntactic examples. Our analysis is solely based on the formal semantic description, and identifies, formally, a statically scoped language. We recognize however that much work is required to the formalisation and implementation of the statically scoped condition.

Chapter 7

Conclusions and Future Work

If it can't be expressed in figures, it is not Science; it is opinion.
The Notebooks of Lazarus Long.

This chapter presents the results we achieved and possibilities for future work. We start by showing figures for some benchmark programs used to compare an ACTRESS-generated compiler with a hand-written one, and to assess the effectiveness of action transformations. We continue by discussing what we think was achieved and how the work compares with others. Topics for further improvements and some open questions conclude the thesis.

7.1 Assessment

We have assessed the effectiveness of action transformations and the binding elimination technique using the programs *loop*, *loopfact*, *bindings* and *block* (see figures 4.3, 6.1, A.1 and A.2 respectively). The figures obtained are summarized in tables 7.1, 7.2 and 7.3. We have compared them against the figures obtained for equivalent programs written in PASCAL and compiled using the Sun PASCAL compiler. All programs were run under SunOS 4.1.3 on a Sun SPARCstation ELC. The measurements were obtained using the UNIX `tcsh` built-in `time` command. The figures shown are the arithmetic means of the CPU user times for five consecutive runs. The compilation time figures for SPECIMEN are shown in three columns, which correspond to compilation of the source program to C, compilation of the C object code by the GNU C compiler, and the sum of these two figures.

Program	Compilation time					Running time		
	PASCAL	SPECIMEN			Penalty	PASCAL	SPECIMEN	Penalty
loop	1.0	2.1	2.1	4.2	4.2	0.6	23.4	39.0
loopfact	1.0	3.1	2.6	5.7	5.7	0.8	11.6	14.5
bindings	1.1	10.8	3.8	14.6	13.3	0.4	44.3	110.7
block	1.0	2.6	2.4	5.0	5.0	0.7	43.6	62.3

Table 7.1: Compilation time and run time figures (in seconds).

The penalty column is the time obtained for SPECIMEN divided by the correponding one for PASCAL. It represents the factor by which compilation time and running time for an ACTRESS-generated compiler are worse when comparing with a hand-written compiler.

Table 7.1 shows the figures for a compiler generated by the preliminary version of ACTRESS. We could say that compilation time is about 4–15 times larger than the compilation time for hand-written compilers. Running time of object code is slower by factor of about 15–110. Although high, this penalty compares favorably with the compiler generators based on denotational semantics (classical systems) discussed in Chapter 2. The compilation time penalty for the *bindings* program is very anomalous when comparing with similar figures for the other programs. We suspect that this might be caused by the manipulation of some inefficient data structure, used by the generated compiler, at compilation time. The high running time penalty for the *bindings* program can be explained in terms of the high numbers of binding lookups (calls to the run-time function *_BOUND*) during the performance of the program.

Table 7.2 shows the result when we use a version of ACTRESS which incorporates the transformer into the generated compiler. The *loopfact* object code now runs faster by a factor of 2. The *bindings* program runs faster by a factor of 10! We expect that the gain will be even better for larger programs with a lot of bindings. At least for the benchmark programs, the transformer introduced no significant time overhead in the generated compiler. It seems that its time overhead is compensated by the smaller program action which is input to the code generator and the smaller C object program which is input to the C compiler. However we would need a larger sample of benchmark programs to be sure about this statement.

Finally, Table 7.3 shows the figures obtained using an optimized C compiler. These figures assume that there are no sort checks at run time (tag-free run time environment).

Based on Table 7.3 we can say that ACTRESS-generated compilers have a compilation penalty about 5–10 and a running time penalty about 5–30.

7.2 What was Achieved?

We consider the discovery and use of action transformations as the principal result of this thesis. In particular, by using them, we solved the problem of binding elimination, transient elimination and storage allocation in ACTRESS. Action transformations can also be seen in a wider context of *static performance of actions*. Although actions are dynamic entities, and action semantics does not distinguish what is static from what is dynamic in an action, we have made such distinction in a formal and systematic way, and have explored it in order to obtain actions which perform better because part of their performance was carried out statically. The figures presented in Section 7.1 demonstrated the effectiveness of action transformations.

This thesis can also be seen as a study of bindings in action notation and action semantic descriptions. The conditions stated in Chapter 6, as far as we know, were never stated for other semantic formalisms. We have identified if a language has a static binding discipline from its action semantic description. Again, we see this result as part of a wider spectrum, that of useful analyses which are built into a processor for language descriptions. As the static analysis of a program by a compiler can detect many inconsistencies without the need to run the program, analyses built into a semantics-based compiler generator should assist the language designer (and implementor) not only to guarantee the consistency of the semantic description but also to give information about the binding discipline, type discipline and other important properties of a language. All this before

Program	Compilation time					Running time		
	PASCAL	SPECIMEN			Penalty	PASCAL	SPECIMEN	Penalty
loop	1.1	2.3	2.0	4.3	3.9	0.6	7.6	12.7
loopfact	1.0	3.0	2.3	5.3	5.3	0.8	6.1	7.6
bindings	1.1	11.1	3.3	14.4	13.1	0.4	4.6	11.5
block	1.0	2.4	2.4	4.8	4.8	0.7	21.8	31.1

• SPECIMEN compiler including transformations.

Table 7.2: Compilation time and run time with action transformations (in seconds).

Program	Compilation time					Running time		
	PASCAL	SPECIMEN			Penalty	PASCAL	SPECIMEN	Penalty
loop	1.0	2.3	2.3	4.6	4.6	0.4	2.8	7.0
loopfact	1.1	3.0	3.0	6.0	5.5	0.7	3.7	5.3
bindings	1.6	11.1	5.1	16.2	10.1	0.3	1.5	5.0
block	1.1	2.4	2.8	5.2	4.7	0.4	11.4	28.5

- Optimized PASCAL compiler.
- SPECIMEN compiler including transformations, optimized C compiler, and no run-time sort checkings.

Table 7.3: Final figures (in seconds).

an implementation is ready! By exploring these properties, more efficient compilers can be generated. The binding analysis of Chapter 6 is a start point in the direction of building useful language analysis into a semantics-based compiler generator. These *semantic description analysers* would be similar in spirit to the analyses found in programming language compilers. This is better expressed by the following relation:

$$\frac{\textit{semantic description analysers}}{\textit{languages}} = \frac{\textit{compiler static analysers}}{\textit{programs}}$$

The use of inference rules to specify the action notation code generator was rewarding. Besides its importance as a precise presentation of the code generator, it revealed some bugs of the implementation[1].

Our contribution to the implementation of ACTRESS can be described concretely by the number of lines of code written. For ANC, including 800 lines of C code of the run time system, but excluding the parser and sort checker, approximately 5,000 lines of ML code were written. ANI has 3,500 lines and the actioneer generator 1,400 lines. The prototype implementation for the binding analysis presented in Chapter 6 has about 500 lines.

7.3 Comparison with other Systems

ACTRESS (with action transformations), to the best of our knowledge, compares favorably with any other semantics-directed compiler generator. There is no doubt that its object code runs better than the code produced by the classical systems. Below we consider some

[1] That was because the specification presented in Chapter 4 was only done after the implementation of the code generator.

other systems.

Palsberg's compiler generator CANTOR is broadly similar to ACTRESS, but it accepts a different subset of action notation. It restricts itself to statically-scoped source languages, however. The current version does not actually eliminate bindings. ACTRESS-generated compilers compare favorably to CANTOR's ones. Object code produced by CANTOR-generated compilers has a running time penalty about 148–369 and a compilation time penalty about 136–542 [89]. Although ACTRESS does not treat commitments and escape actions, we believe that their inclusion would not cause much lost in efficiency. Furthermore, we think that some improvement could be expected if we generated machine code instead of C. A nice fact about CANTOR is that its correctness proof is given [90]. At present there is no correctness proof for ACTRESS.

At the core of both ACTRESS and CANTOR is a compiler for actions, hand-written in both cases. More recently Palsberg and Bondorf have applied partial evaluation to obtain an action compiler [12]. The compiler is obtained by partial evaluation of an action interpreter. The interpreter is written in SCHEME, and the compiler is obtained by applying the SIMILIX [11] partial evaluator to it. The generated compiler works by specializing an action interpreter with respect to the input action. It is reported that the produced SCHEME code runs as fast as that produced by the previous action compilers.

Doh's prototyping system [27], based on a category-sorted algebraic model for action semantics [28], extracts a binding-time semantics from an action-semantic description. It generates a syntax-directed translator that translates the source program to a program action annotated with binding-time information. This annotation will assist a static evaluator to identify which parts of the program action can be statically performed. Doh's method is more strongly influenced by partial evaluation than ours: in the source action of Example 5.16, he would unfold the abstraction rather than leave it to be enacted. As compared with Doh's method, it seems that our method can eliminate more bindings, and is applicable to a larger subset of action notation. Doh reported, for a particular program action, an improvement in efficiency by a factor of 2 using his approach. However, much assessment would be required to see how effective his approach is. As he suggested in [27], a good exercise would be to incorporate his system in ACTRESS and assess the result.

Using MESS, Lee and Pleban have constructed compilers whose object code is excellent. As reported in [65], they compare well with code generated by hand-written compilers. Unfortunately, the language specifier has to design a new semantic algebra for each source language, as high-level semantics has no standard notation as in action semantics, and must manually implement the translation from this semantic algebra to the target machine code. The good performance of generated compilers comes from a distinct separation between compile-time and run-time objects (macrosemantics and microsemantics, respectively) [65].

7.4 Improving ACTRESS Further

There is some room for improvements and extensions to the current work. We identify below some of the points which deserves more study.

- **Transient elimination.** We have paid slight attention to transient elimination. As stated in Chapter 5 transformations involving transients do not always lead to gain in efficiency. But certainly, transient elimination could be explored in more detail.

- **Compiler transformations.** There is a possibility to explore standard compiler transformations — such as common subexpression elimination and code motion — in terms of action notation. This could have the advantage of bringing these transformations to a formal basis, on which eventually their correctness could be proved.

- **Code generator.** Better C object code could be generated. It seems that our choice to translate yielders to C expressions does not give good flexibility in the use of C as a target language. A more assembly-like approach would be to translate a yielder to C statements. Another possibility is to retarget the code generator to some machine independent format such as ANDF [103].

- **Commitments and escape actions.** The retargeting of the action notation code generator could be accompanied by an extension of ACTRESS action notation to include commitments and escape actions. (These are the most important action notation concepts — apart from the communicative facet — excluded from the ACTRESS subset.)

- **Translation correctness.** The specification of the action notation code generator has helped our understanding of the topic. However one can notice a semantic gap between a program action and its corresponding C object code. It is desirable to prove the correctness of the translation process. We could start by given a semantics to the C subset used. This work could contribute to a proof of ACTRESS' correctness.

- **Better actioneer generator.** The version of the actioneer generator described in this thesis is very simple. In a future version a deeper analysis of the semantic description could be implemented. At present, some consistency checks are deferred to be performed when the generated actioneer is itself compiled (by the ML compiler). Some future enhancements could include: automatic extraction of a ML datatype from the abstract syntax definition; consistency checks on the semantic equations against the abstract syntax definition; sort checking; support for specifier-defined sorts; and an automatically-generated TEX output (similar to the one in Appendix B). Some of these improvements have already been studied by Brown, and are described in [14].

- **Type analysis of the source language.** For a particular language it can happen that we do not need any run-time sort checks. So the code generator should be parameterized regarding this. If we do not need such checks, the generated code can be optimized, e.g., the data do not need tags at run time. A simple approach to this problem is to examine the decorated action tree, for a particular program action, looking for *SORT_CHECK* nodes. If there are no such nodes then the run-time environment for that program may be tag-free. This is a per-program solution and not so elegant. A better approach is to define a type analysis on the source language. If this analysis concludes, for example, that the source language is statically typed, ACTRESS could generate, based on this result, a better compiler which would generate better object code for all source programs in the language.

- **Better storage allocation.** Storage for classified '**allocate**' actions of incorporated actions is allocated in the heap. For a language where abstractions are first-class values, frames must be allocated in the heap. However, for a language where a stack based storage organization can be used, '**allocate**' actions of incorporated actions could be allocated in the stack. It seems that we need a condition on the language description to test if its abstractions are first-class values, so we could be sure when a stack based or heap based storage organization can be used. A condition for stack allocation must also consider the possibility of cells as values in the source language. Much study is required here too.

- **Correctness of the transformations.** We would like to show that a target action is *equivalent* to its source action. This would require a deeper study on the theme of action equivalence. The transformations preserve (for program actions) what we have called functional equivalence of actions. We need however a notion of equivalence which includes the imperative facet. Using this notion together with some convenient semantics for action notation, one could in principle prove that action transformations preserve action equivalence.

- **Semantic rules.** Although possible, we did not give a semantics for ACTRESS action notation which takes into account the introduction of statically allocated cells (cell (l,n)). This could be an interesting work to be carried out in future.

7.5 Open Questions

Although we did the best effort to test our ideas using practical examples, and a non-trivial programming language, it remains to be seen how the system would work for a real language. A good exercise would be to use the PASCAL action semantics [83] as an input to ACTRESS, and test the effectiveness of the transformations for real and large programs. As demonstrated by the figures for the *bindings* program (Table 7.2), we expect that binding elimination will be responsible for a big improvement in the running time of those programs.

Some initial experiments with a functional subset of STANDARD ML (MICROML), showed that binding elimination can be applied successfully to functional languages [8, 53]. Notice that, although it is expected that an action semantic description for a functional language has no imperative actions, the binding elimination technique does introduce imperative actions. We believe this does not break the functional behaviour of the original semantics. Conventional implementations of functional languages also use the store! Again, although initial figures for the object code generated for MICROML show a performance comparable to one generated by a hand-written compiler, it remains to be seen how generated compilers for full functional languages behave.

It would be nice to have a formal notion of *action efficiency*. Given an action a, we would like to answer if an action a' obtained from a by some action transformations is more efficient than a. This notion could help in transient elimination where it is not always clear if the target action is more efficient than the source action.

After the work on the code generator, we thought about the possibility of definition of an *abstract machine for actions*. This could help to obtain a more independent back end for the action notation compiler, as well as in a possible correctness proof for Actress. Also it could reveal desirable properties a real machine for action performance could have.

7.6 Final Words

The Actress system was the first compiler generator built using action semantics [16, 15]. Although a relatively new formalism, action semantics has also been used to build other systems [90, 27], which demonstrates its potential in the area of semantics-directed compiler generation. The action primitives and combinators correspond quite closely to the operational concepts in terms of which languages are implemented. The store is by definition single-threaded, and bindings are by definition scoped. Action notation has more structure than λ-notation, which gives a better handle on the problem in an action semantics directed compiler generator than in a denotational-semantics based system. Finally, action transformations (including algebraic properties of action notation) provide a rigorous foundation for code-improving transformations.

We believe that the present work on action transformations represents a modest step towards the development of a high-quality semantics-directed compiler generator based on action semantics.

Bibliography

[1] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1. Ellis Horwood Limited, 1987.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming.* Springer-Verlag, August 1991.

[4] H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1985.

[5] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in cooperation with Addison-Wesley, 1989.

[6] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, June 1991.

[7] P. Bird. An implementation of a code generator specification language for table driven code generators. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 44–55, June 1982. SIGPLAN Notices, Volume 17, Number 6.

[8] R. A. Bird and P. L. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[9] J. M. Bodwin, L. Bradley, K. Kanda, D. Litle, and U. F. Pleban. Experience with an experimental compiler generator based on denotational semantics. *SIGPLAN Notices (SIGPLAN '82 Symp. On Compiler Construction)*, 17(6), June 1982.

[10] S. Bondesen and S. Laursen. An action semantics for Joyce. Internal Report DAIMI IR-72, Computer Science Department, Aarhus University, 1987. Out of print.

[11] A. Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, April 1993. Included in Similix 5.0 distribution.

[12] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 308–317, June 1993.

[13] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *SIGPLAN Notices*, 24(2):14–24, February 1989. Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

[14] D. F. Brown. *Sort Inference in Action Semantic Specifications.* PhD thesis, University of Glasgow, 1993. In preparation.

[15] D. F. Brown, H. Moura, and D. A. Watt. Towards a realistic semantics-directed compiler generator. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Functional Programming, Glasgow 1991*, pages 51–55. Springer-Verlag, August 1991. Workshops in Computing Series.

[16] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, pages 95–109. Springer-Verlag, October 1992. Lecture Notes in Computer Science, volume 641.

[17] BSI. Specification for computer programming language Pascal, 1982. BS 6192, British Standards Institution, Milton Keynes, England.

[18] S. Christensen and M. H. Olsen. Action semantics of “CCS” and “CSP”. Internal Report DAIMI IR-82, Computer Science Department, Aarhus University, 1988.

[19] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. Rapports de Recherche 529, INRIA Sophia Antipolis, May 1986.

[20] F. Q. B. da Silva. Correctness proofs of compilers and debuggers: an overview of an approach based on structural operational semantics. LFCS Report Series ECS-LFCS-92-233, University of Edinburgh, September 1992.

[21] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars (Definitions, Systems and Bibliography)*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[22] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier, 1990. Volume B of Handbook of Theoretical Computer Science.

[23] P. Deschamp. PERLUETTE: a compiler production system using abstract data types. In *International Symposium on Programming*, Turin, April 1982.

[24] J. Despeyroux. Proof of translation in natural semantics. In *LICS '86, First Symposium on Logic in Computer Science*, June 1986.

[25] T. Despeyroux. Executable specification of static semantics. Rapports de Recherche 295, INRIA Sophia Antipolis, May 1984.

[26] T. Despeyroux. Typol: a formalism to implement natural semantics. Rapports Techniques 94, INRIA Sophia Antipolis, March 1988.

[27] Kyung-Goo Doh. *Action Semantics-Directed Prototyping.* PhD thesis, Kansas State University, 1992.

[28] S. Even and D. A. Schmidt. Category sorted algebra-based action semantics. *Theoretical Computer Science*, (77):73–96, 1990.

[29] R. Farrow. LINGUIST-86: Yet Another Translator Writing System Based on Attribute Grammars. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 160–171, 1982. SIGPLAN Notices, Volume 17, Number 6.

[30] R. Farrow. Generating a production compiler from an attribute grammar. *IEEE Software*, pages 77–93, 1984.

[31] C. N. Fisher and R. J. LeBlanc, Jr. *Crafting a Compiler.* The Benjamin/Cummings Publishing Company, Inc., 1988.

[32] R. W. Floyd. Assigning meanings to programs. In T. Schwartz, editor, *Proceedings of the Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[33] D. P. Friedman, C. T. Haynes, E. Kohlbecker, and M. Wand. The Scheme 84 reference manual. Technical report, Indiana University, Computer Science Department, March 1984.

[34] M. Ganapathi. *Retargetable Code Generation and Optimization Using Attribute Grammars.* PhD thesis, University of Wisconsin, Madison, Wisconsin, 1980.

[35] H. Ganzinger, R. Giegerich, V. Möncke, and R. Wilhelm. A Truly Generative Semantics-Directed Compiler Generator. *SIGPLAN Notices (SIGPLAN '82 Symp. On Compiler Construction)*, 17(6), June 1982.

[36] M. C. Gaudel. Specification of compilers as abstract data types representations. In N. D. Jones, editor, *Semantics Directed Compiler Generation.* Springer-Verlag, 1980. Lecture Notes in Computer Science, volume 94.

[37] M. C. Gaudel. Compiler generation from formal definition of programming languages: a survey. In J. Diaz and I. Ramos, editors, *Formalization of Programming Concepts*, pages 96–114. Springer-Verlag, 1981. Lecture Notes in Computer Science, volume 107.

[38] R. S. Glanville and S. Graham. A New Method for Compiler Code Generation. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 231–239. ACM, 1978.

[39] C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programmming*, 1(1):21–69, January 1991.

[40] C. A. Gunter. *Semantics of Programming Languages.* Foundations of Computing. The MIT Press, 1992.

[41] J. Hannan. Making abstract machines less abstract. In *Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 618–635, 1991. Lecture Notes in Computer Science, volume 523.

[42] J. Hannan. Staging transformations for abstract machines. In *ACM SIGPLAN Symposium of Partial Evaluation and Semantics Based Program Manipulation*, pages 130–141, 1991.

[43] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, December 1992.

[44] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[46] J. W. Thatcher J. A. Goguen and E. G. Wagner. An Initial Algebra Approach to the Specification Correctness, and Implementation of Abstract Data Types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV*. Prentice-Hall, 1978.

[47] I. Jacobs and L. Rideau-Gallot. A CENTAUR tutorial. Rapports Techniques 140, INRIA Sophia Antipolis, July 1992.

[48] K. Jensen and N. Wirth. *Pascal – User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*. Springer-Verlag, second edition, 1975.

[49] S. C. Johnson. Yacc - Yet Another Compiler Compiler. Technical report, Bell Laboratories, Murray Hill, 1979. UNIX manual.

[50] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[51] N. D. Jones and S. S. Muchnick. *TEMPO: A unified treatment of binding time and paramenter passing concepts in programming languages*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[52] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation (revised version). Research Report 87/8, DIKU, University of Copenhagen, 1987.

[53] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1987.

[54] G. Kahn. Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, February 1987.

[55] U. Kastens. The GAG-System – a tool for compiler construction. In B. Lorho, editor, *Methods and tools for compiler construction*. Cambridge University Press, 1984.

[56] U. Kastens, B. Hutt, and E. Zimmerman. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[57] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989. Association for Computing Machinery.

[58] P. Klint. A meta-environment for generating programming environments. *ACM Transcations on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[59] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–152, 1968.

[60] K. Koskimies. A specification language for one-pass semantic analysis. In *Proceedings of the SIGPLAN '84 Symposium On Compiler Construction*, pages 179–189, June 1984. SIGPLAN Notices, volume 19, number 6.

[61] K. Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The design of the language processor generator HLP84. Technical Report A-1986-4, Department of Computer Science, University of Helsinki, November 1986.

[62] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, July 1986. SIGPLAN Notices, Volume 21, Number 7.

[63] P. J. Landim. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[64] J. Launchbury. Notes on partial evaluation, June 1991. PEPM'91 tutorial notes.

[65] P. Lee. *Realistic Compiler Generation*. MIT Press, Cambridge, Massachusetts, 1989.

[66] P. Lee and U. F. Pleban. On the Use of LISP in Implementing Denotational Semantics. In *Proceedings of 1986 ACM Conference on LISP and Functional Programming*, pages 233–248, Cambridge, Mass., 1986. ACM.

[67] M. Lesk and E. Schmidt. Lex – A Lexical Analyzer Generator. Technical report, Bell Laboratories, Murray Hill, 1979. UNIX manual.

[68] B. Lohro. Semantics attributes processing in the system DELTA. In A. Ershov and C. H. A. Koster, editors, *Methods of Algorithmic Language Implementation*, pages 21–40. Springer-Verlag, 1977.

[69] S. McKeever. A framework for generating compilers from natural semantics specifications. Technical Report 10, Programming Research Group, Oxford University, 1993.

[70] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, England, 1976.

[71] P. D. Mosses. *Mathematical Semantics and Compiler Generation*. PhD thesis, Oxford University, 1975.

[72] P. D. Mosses. Making denotational semantics less concrete. In *Proc. Int. Workshop on Semantics of Programming Languages*, 1977.

[73] P. D. Mosses. SIS – Semantics Implementation System, Reference Manual and User Guide. Technical Report DAIMI MD-30, Aarhus University, 1979.

[74] P. D. Mosses. A semantic algebra for binding constructs. In *Proc. Int. Coll. on Formalization of Programming Concepts*. Springer-Verlag, 1981. Lecture Notes in Computer Science **107**.

[75] P. D. Mosses. Abstract semantic algebras! In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 45–72, Amsterdan, 1983. North-Holland.

[76] P. D. Mosses. A basic abstract semantic algebra. In *Semantics of Data Types*. Springer-Verlag, 1984. Lecture Notes in Computer Science, volume 173.

[77] P. D. Mosses. Unified algebras and action semantics. In *Proceedings of STACS'89*. Springer-Verlag, 1989. Lecture Notes in Computer Science, volume 349.

[78] P. D. Mosses. Unified algebras and modules. In *Proceedings of the 16th Annual ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.

[79] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Formal Models and Semantics*, chapter 11, pages 575–631. Elsevier, 1990. Volume B of Handbook of Theoretical Computer Science.

[80] P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[81] P. D. Mosses and D. A. Watt. The potential use of action semantics in standards. Departamental Research Report CSC/86/R1, Glasgow University, Computing Science Department, 1986.

[82] P. D. Mosses and D. A. Watt. The use of action semantics. In M. Wirsing, editor, *Formal Description of Programming Concepts*, Amsterdam, 1987. North Holland.

[83] P. D. Mosses and D. A. Watt. Pascal action semantics, 1993. In preparation.

[84] H. Moura. An implementation of action semantics (summary). In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 477–478. Springer-Verlag, August 1992. Lecture Notes in Computer Science 631.

[85] M. A. Musicante. Another action semantics of SML, 1993. Draft.

[86] L. Naish. The MU-Prolog 3.2 reference manual. Technical Report 85/11, Department of Computer Science, University of Melbourne, October 1985.

[87] Deutsche Industrie Norma. Programmiersprache PEARL, 1980. Normentwurf DIN 66253, Teil 2, Reuth-Verlag.

[88] J. Palsberg. An action semantics for inheritance. Master's thesis, Aarhus University, 1988.

[89] J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proceedings of ICCL '92, Fourth IEEE International Conference on Computer Languages*, San Francisco, California, April 1992.

[90] J. Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Aarhus University, October 1992.

[91] J. Palsberg. A provably correct compiler generator. In *In Proceedings of ESOP '92, European Symposium on Programming*, pages 418–434, Rennes, France, February 1992. Lecture Notes in Computer Science.

[92] L. Paulson. *A Compiler Generator for Semantic Grammars.* PhD thesis, Stanford University, 1981.

[93] L. Paulson. A Semantics-Directed Compiler Generator. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 224–239, Albuquerque, NM, January 1982.

[94] M. Pettersson and P. Fritzson. DML – a meta-language and system for the generation of practical and efficient compilers from denotational specifications. In *International Conference on Computer Languages*, Oakland, California, April 1992.

[95] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, September 1981.

[96] K. Räihä. Experiences with the compiler writing system HLP. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 350–362. Springer-Verlag, 1980.

[97] D. A. Schmidt. *Denotational Semantics.* Allyn & Bacon, 1986.

[98] D. A. Schmidt and S. Even. Type Inference for Action Semantics. In N. Jones, editor, *ESOP '90, 3rd European Symposium on Programming*, pages 118–133. Springer-Verlag, May 1990. Lecture Notes in Computer Science, Volume 432.

[99] R. M. Stallman. *Using and porting GNU CC*, May 1992. Manual for version 2.2.

[100] C. Strachey. The varieties of programming language. Technical Monograph PRG-10, Oxford University Computing Laboratory, March 1973.

[101] D. R. Tarditi and A. W. Appel. *ML-Yacc, version 2.0*, 1990. Documentation for Release Version.

[102] R. D. Tennent. *Principles of Programming Languages.* International Series in Computer Science. Prentice Hall, 1981.

[103] J. U. Toft. Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report TR2.1.2-02, DDC International A/S, 1993.

[104] M. Tofte. *Compiler Generators.* Springer-Verlag, 1990. EATCS Monographs on Theoretical Computer Science, volume 19.

[105] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., 1991.

[106] M. Wand. A semantic prototyping system. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 213–221, June 1984.

[107] M. Wand. Type inference for record concatenation and simple objects. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, 1989.

[108] D. A. Watt. Executable semantic descriptions. *Software – Practice and Experience*, 16(1):13–43, 1986.

[109] D. A. Watt. An action semantics of Standard ML. In *Mathematical Foundations of Programming Language Semantics LNCS 298*. Springer-Verlag, 1988.

[110] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[111] D. A. Watt. Modular description of programming languages. *The Computer Journal*, 35, 1992.

[112] D. A. Watt. *Programming Language Processors.* Prentice Hall International Series in Computer Science. Prentice Hall, 1993.

[113] D. A. Watt and P. D. Mosses. Action semantics in action. Unpublished, 1987.

[114] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1992.

[115] P. Weis. *Le Système SAM: Mètacompilation très Efficace à l'aide d'Opérateurs Sèmantiques.* PhD thesis, INRIA, L'Université Paris VII, 1987. In French.

[116] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. Volume B of Handbook of Theoretical Computer Science.

Appendix A

Informal Description of SPECIMEN

This appendix and the next one define SPECIMEN. We present here an informal description of SPECIMEN; the formal definition is given in the next appendix. This organization was inspired by the one found in [110] and it enphasizes the complementary aspect of informal and formal descriptions.

A.1 Informal Description

SPECIMEN is a simple imperative programming language. It has two types of abstractions: procedures and functions, which can be recursive. Functions are higher-order and free of side effects, although they can access non-local variables. Integers, booleans, and arrays are the only data types in the language.

A.1.1 Programs

Syntax

$$\mathsf{Program} = [\![\ \text{“}\texttt{program}\text{”}\ \mathsf{Identifier}\ \text{“}\texttt{is}\text{”}\ \mathsf{Declaration}\ \text{“}\texttt{in}\text{”}\ \mathsf{Command}\ \text{“}\texttt{end}\text{”}\ ]\!]\ .$$

Semantics

- A SPECIMEN program is a declaration followed by a command. The execution of a program consists of the elaboration of the declaration, followed by the execution of the command using the bindings produced by the declaration. These bindings are visible throughout the command (global bindings); holes in the scope of global bindings can occur by local declarations of the same global identifier.

Examples

Figure A.1 and A.2 show two examples of SPECIMEN programs.

```
program bindings is
    const ca : bool = true;
    const cb : int = 346;
    const cc : int = 3;
    var ba : bool := true;
    var bb : bool := true;
    var bc : bool := true;
    var bd : bool := true;
    var be : bool := true;
    var bf : bool := true;
    var bg : bool := true;
    var bh : bool := true;
    var bi : bool := true;
    var bj : bool := true;
    var bk : bool := true;
    var bl : bool := true;
    var bm : bool := true;
    var bn : bool := true;
    var ia : int := 56;
    var ib : int := 0;
    var ic : int := 3;
    var id : int := 20;
    var ie : int := 406;
    var iF : int := 78;
    var ix : int := 4;
    var iy : int := 45;
    var iz : int := 4;
    var counter : int := 1000
in
    while (counter > 0) do
        ix := ix + cc;
        iz := ia + ib + ic + id + ie;
        bb := bc and bf and bh and bn;
        iy := ie + (ia + ib + ic + iF);
        counter := counter - 1
    end
end
```

Figure A.1: The *bindings* program.

```
program block is
    const c : int = 3;
    var x : int := 4;
    var counter : int := 0;
    proc block () =
            local
                var y : int := 6
            in
                y := 20 + x
            end
in
    while (counter < 500000)
        do
            call block ();
            counter := counter + 1
        end
end
```

Figure A.2: The *block* program.

A.1.2 Type Denoters

Syntax

(1) ValueType = PrimitiveType | FunctionType .

(2) PrimitiveType = "bool" | "int" .

(3) FunctionType = ⟦ ValueType "->" ValueType ⟧ |
⟦ "(" TupleType ")" "->" ValueType ⟧ .

(4) TupleType = ⟦ ⟧ | ⟦ ValueType ⟧ | ⟦ ValueType "," TupleType ⟧ .

(5) ArrayType = ⟦ "array" "[" Numeral "]" "of" PrimitiveType ⟧ .

Semantics

Types of constants, variables, procedures, functions and arrays must be explicitly declared. A SPECIMEN expression always evaluate to a value of type ValueType. Booleans and integers are primitive values; they have type PrimitiveValue. Array components are primitive values.

Examples

- *int* is the type of integer values.
- **array** [20] **of** *bool* is the type of an array of 20 components of type *bool*.
- (*int*, *int*) -> *bool* is the type of a function which expects two arguments of type *int* and returns a result of type *bool*.
- *bool* -> (*int* -> *int*) is the type of a function which expects one argument of type *bool* and returns a function of type *int* -> *int* as its result.

A.1.3 Declarations

Syntax

(1) Declaration = ⟦ "const" Identifier ":" PrimitiveType "=" Expression ⟧ **|**
⟦ "var" Identifier ":" PrimitiveType ":=" Expression ⟧ **|**
⟦ "varray" Identifier ":" ArrayType ":=" "[" ArrayComps "]" ⟧ **|**
⟦ "proc" Identifier "(" ProcFormals ")" "=" Command ⟧ **|**
⟦ "fun" Identifier "(" FunFormals ")" ":" ValueType "=" Expression ⟧ **|**
⟨ Declaration ⟨ ";" Declaration ⟩* ⟩ .

(2) ArrayComps = Expression **|** ⟦ Expression "," ArrayComps ⟧ .

(3) ProcFormals = ⟦ ⟧ **|** ⟦ ProcFormal ⟧ **|** ⟦ ProcFormal "," ProcFormals ⟧ .

(4) ProcFormal = ⟦ Identifier ":" ValueType ⟧ **|** ⟦ "var" Identifier ":" PrimitiveType ⟧ .

(5) FunFormals = ⟦ ⟧ **|** ⟦ FunFormal ⟧ **|** ⟦ FunFormal "," FunFormals ⟧ .

(6) FunFormal = ⟦ Identifier ":" ValueType ⟧ .

Semantics

Declarations allow the programmer to make bindings. SPECIMEN has four types of declarations: constant, variable, procedure and function declarations. Procedure and function declarations can be recursive. A SPECIMEN program must have at least one declaration.

- A constant declaration "**const** I:T = E" binds the constant identifier I to the primitive value yielded by the evaluation of expression E. I and E have primitive type T.

- A variable declaration "**var** I:T := E" binds the variable identifier I to a newly allocated cell. The result of the evaluation of the expression E is stored in this cell. I and E have primitive type T.

- An array declaration "**varray** I:**array** [N] **of** T := E" binds the array variable identifier I to a list of N newly allocated cell. The expression E, an array aggregate with N components, is evaluated and every resultant component is stored in the correspondent cell. I and E have array type "**array** [N] **of** T", where T is a primitive type.

- A procedure declaration "proc I (FS) = C" binds the procedure identifier I to a procedure. A procedure is a *command abstraction*. Zero or more formal parameters can be defined in the procedure declaration. Parameters can be passed by value or by reference (var parameters). When passed by value, the argument, a value, is bound to the correspondent formal parameter; when passed by reference the argument, a cell, is bound to the correspondent formal variable. The body of a procedure is just a command. The command is executed in an the same environment of the procedure declaration overlaid by the bindings produced by the elaboration of the procedure and its formal parameters. No result is returned, a procedure works by its side effects only. Procedure can be recursive. Procedures cannot be passed as parameters (of procedures or functions) or return as result of functions.

- A function declaration "`fun` I `(`FS`):`T `=` E" binds the function identifier I to a function. A function is an *expression abstraction*. SPECIMEN functions can have zero or more parameters. Parameters are passed by value only. The body of a function is just an expression which is evaluated in the same environment of the function declaration overlaid by the bindings produced by the elaboration of the function and its formal parameters. An application of function I always returns a value of value type T. Functions can be recursive.

- The sequential declaration "D_1 ; D_2" is elaborated by elaborating D_1 and then elaborating D_2. D_2 is elaborated in the environment of the sequential declaration overlaid by the bindings produced by D_1. The resultant bindings are the ones produced by D_1 overlaid by those produced by D_2.

Examples

- **const** *year* : *int* `=` `1991`
- **const** *yearPlusOne* : *int* `=` *year* `+ 1`
- **const** *factOfFour* : *int* = *fact* `(4)`
- **var** *x* : *bool* `:=` *false*
- **var** *y* : *bool* `:=` *not* `(`x`)`
- **varray** *vec* : **array** [4] **of** *int* `:=` `[3,5,9,0]`
- **proc** *inc* (**var** x : *int*) `=` x`:=` x `+ 1`
- **fun** *isGreaterThan* (a : *int*, b : *int*) : bool **=** **if** a **>** b **then** *true* **else** *false* **end**

A.1.4 Commands

A command is used to update variables. SPECIMEN has six types of commands: assignment, conditional, while, procedure call, block and sequential.

Syntax

(1) Command = ⟦ Identifier ":=" Expression ⟧ |
⟦ Identifier "[" Expression "]" ":=" Expression ⟧ |
⟦ "if" Expression "then" Command ⟨ "else" Command ⟩? "end" ⟧ |
⟦ "while" Expression "do" Command "end" ⟧ |
⟦ "call" Identifier "(" ProcActuals ")" ⟧ |
⟦ "local" Declaration "in" Command "end" ⟧ |
⟨ Command ⟨ ";" Command ⟩* ⟩ .

(2) ProcActuals = ⟦ ⟧ | ⟦ ProcActual ⟧ | ⟦ ProcActual "," ProcActuals ⟧ .

(3) ProcActual = ⟦ Expression ⟧ | ⟦ "var" Identifier ⟧ .

Semantics

- The assignment "I `:=` E is the simplest form of command. It assigns the value of expression E to variable I. The expression is evaluated and the resultant value is stored in the cell bound to the variable identifier.
- The array assignment "I`[`E_1`]` `:=` E_2" updates a component of array I. The component is the one indexed by the result of evalution of E_1 which must be a positive integer. The primitive value resultant from the evaluation of E_2 will be the new component.
- The conditional command "**if** E **then** C_1 **else** C_2 **end**" is executed as follows: the expression E is evaluated; if the evaluation result is true then command C_1 is executed; otherwise command C_2 is executed.
- The while command "**while** E **do** C **end**" is executed as follows: (a) the expression E is evaluated; (b) if its value is true then the subcommand C is executed and then we start from step (a) again; otherwise, the while-command is terminated.
- A procedure call, "**call** I `(`A`)`", causes the execution of the procedure bound to identifier I. A parameter can be passed by value or by reference (`var` formal). By value, "I `(`E`)`", the actual parameter expression E is evaluated and its value is bound to the formal paramenter; by reference, "I_1 `(var` I_2`)`", the cell bound to variable I_2 is bound to the formal parameter. In the latter case the actual parameter must be a variable so the value which is bound to the formal paramenter is a cell. Functions can be actual parameters of procedures.
- The block command "**local** D **in** C **end**" is executed as follows. The declaration D is elaborated. The command C is then executed in the environment of the block command overlaid by the bindings produced by D.
- The sequential command "C_1 `;` C_2" is executed by first executing C_1 and then executing C_2.

Examples

- i `:= 314`
- x `:=` x `+ 1`
- **if** $x = 0$ **then** x `:=` x `+ 1` **else** x `:=` x `- 1` **end**
- **while** x `=` *true* **do** x `:=` *true and true* **end**
- *vec*`[`*t*`] := 345 ;` **call** p `(3 + 5)`
- **local var** x `:` *int* `:= 2` **in** x `:=` x `+` y **end**

A.1.5 Expressions

The evaluation of an expression yields a value (an integer, a boolean, or a function value). Expressions are completely free of side effects. Functions are first class: they can be passed as arguments to procedures and functions and returned as a result of a function application.

Syntax

```
(1) Expression = "true" |
                 "false" |
                 Numeral |
                 Identifier |
                 ⟦ Identifier "[" Expression "]" ⟧ |
                 ⟦ "if" Expression "then" Expression ⟨ "else" Expression ⟩? "end" ⟧ |
                 ⟦ Identifier "(" FunActuals ")" ⟧ |
                 ⟦ Expression Operator Expression ⟧ |
                 ⟦ Operator Expression ⟧
                 ⟦ "let" Declaration "in" Expression "end" ⟧ .

(2) FunActuals = ⟦ ⟧ | ⟦ FunActual ⟧ | ⟦ FunActual "," FunActuals ⟧ .

(3) FunActual = ⟦ Expression ⟧ .

(4) Operator = "not" | "and" | "or" |
               "~" | "+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" |
               "=" | "<>" .
```

Semantics

- A literal expression is one that does not need additional evaluation (it is already in a canonical form). SPECIMEN has two types of literal expressions: boolean and integer.

- An identifier is an expression. Its evaluation gives the primitive value bound to it, in the case of a constant; or the function bound to it, in the case of a function identifier, respectively; or the content of the cell bound to it, in the case of a variable.

- The expression "I[E]" is evaluated as follows: expression E is evaluated and its result, a positive integer, say n, is used to access the nth component of array I which is the result of the whole expression.

- The conditional expression "**if** E_1 **then** E_2 **else** E_3 **end**" is evaluates as follows: the expression E_1 is evaluated; if the evaluation result is true then E_2 is evaluate and the result value is the result of the conditional expression; otherwise E_3 is evaluate and the result value is the result of the conditional expression.

- The function application, I ($E_1, \ldots, E_n$), evaluates as follows: each parameter E_i is evaluated and a list of arguments is formed with the resultant values. The function bound to I is then applied to the list of arguments. Each occurrence of the formal parameter inside the function body is replaced by the correspondent resultant value; the function body is evaluated and its result is returned as the result of the function application.

- Constants, variables and functions can be defined locally using a let expression. (Procedures can also be defined locally but they cannot be used, so in practice there is no sense in defining them.)

- Conventional operations on booleans and integers are provided. Equality and inequality between primitive values are present.

Examples

- `true`
- `4563738`
- x
- **if** *tag* **then** *process* (*tag*) **else** *unprocess* (*tag*) **end**
- g (x + 3) - f (x - 3)
- `~3`
- `3 + 4 - 4 * 3 - 9`
- `(3 + 4) - (4 * 3 - 9)`

Appendix B

The Action Semantic Description of SPECIMEN

This appendix presents the complete action semantics of SPECIMEN. The description is organized in four modules (sections): abstract syntax, semantic entities, semantic functions and lexical syntax.

B.1 Abstract Syntax

needs: **Lexical Syntax** .

closed

grammar:

B.1.1 Programs

(1) Program = ⟦ "program" Identifier "is" Declaration "in" Command "end" ⟧ .

B.1.2 Declarations

(1) Declaration = ⟦ "const" Identifier ":" PrimitiveType "=" Expression ⟧ |
⟦ "var" Identifier ":" PrimitiveType ":=" Expression ⟧ |
⟦ "varray" Identifier ":" ArrayType ":=" "[" ArrayComps "]" ⟧ |
⟦ "proc" Identifier "(" ProcFormals ")" "=" Command ⟧ |
⟦ "fun" Identifier "(" FunFormals ")" ":" ValueType "="
Expression ⟧ |
⟨ Declaration ⟨ ";" Declaration ⟩* ⟩ .

(2) ArrayComps = Expression | ⟦ Expression "," ArrayComps ⟧ .

(3) ProcFormals = ⟦ ⟧ | ⟦ ProcFormal ⟧ | ⟦ ProcFormal "," ProcFormals ⟧ .

(4) ProcFormal = ⟦ Identifier ":" ValueType ⟧ | ⟦ "var" Identifier ":" PrimitiveType ⟧ .

(5) FunFormals = ⟦ ⟧ | ⟦ FunFormal ⟧ | ⟦ FunFormal "," FunFormals ⟧ .

(6) FunFormal = ⟦ Identifier ":" ValueType ⟧ .

B.1.3 Commands

(1) Command = ⟦ Identifier ":=" Expression ⟧ |
⟦ Identifier "[" Expression "]" ":=" Expression ⟧ |
⟦ "if" Expression "then" Command ⟨ "else" Command ⟩$^?$ "end" ⟧ |
⟦ "while" Expression "do" Command "end" ⟧ |
⟦ "call" Identifier "(" ProcActuals ")" ⟧ |
⟦ "local" Declaration "in" Command "end" ⟧ |
⟨ Command ⟨ ";" Command ⟩* ⟩ .

(2) ProcActuals = ⟦ ⟧ | ⟦ ProcActual ⟧ | ⟦ ProcActual "," ProcActuals ⟧ .

(3) ProcActual = ⟦ Expression ⟧ | ⟦ "var" Identifier ⟧ .

B.1.4 Expressions

(1) Expression = "true" |
"false" |
Numeral |
Identifier |
⟦ Identifier "[" Expression "]" ⟧ |
⟦ "if" Expression "then" Expression ⟨ "else" Expression ⟩$^?$ "end" ⟧ |
⟦ Identifier "(" FunActuals ")" ⟧ |
⟦ Expression Operator Expression ⟧ |
⟦ Operator Expression ⟧
⟦ "let" Declaration "in" Expression "end" ⟧ .

(2) FunActuals = ⟦ ⟧ | ⟦ FunActual ⟧ | ⟦ FunActual "," FunActuals ⟧ .

(3) FunActual = ⟦ Expression ⟧ .

(4) Operator = "not" | "and" | "or" |
"~" | "+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" |
"=" | "<>" .

B.1.5 Type Denoters

(1) ValueType = PrimitiveType | FunctionType .

(2) PrimitiveType = "bool" | "int" .

(3) FunctionType = ⟦ ValueType "->" ValueType ⟧ |
⟦ "(" TupleType ")" "->" ValueType ⟧ .

(4) TupleType = ⟦ ⟧ | ⟦ ValueType ⟧ | ⟦ ValueType "," TupleType ⟧ .

(5) ArrayType = ⟦ "array" "[" Numeral "]" "of" PrimitiveType ⟧ .

B.2 Semantic Entities

includes: **Action Notation** .

B.2.1 Values

introduces: primitive-value , value .

(1) primitive-value = truth-value | integer .

(2) value = primitive-value | function .

B.2.2 Bindings

(1) alpha = lowercase-letter | uppercase-letter .

(2) token = string-of (alpha,(alpha | digit)*) .

(3) bindable = value | procedure | cell | array .

B.2.3 Storage

(1) cell = cell [truth-value] | cell [integer] .

(2) storable = primitive-value .

B.2.4 Procedures and Functions

introduces: procedure , proc-argument , proc-argument-list ,
function , fun-argument , fun-argument-list , fun-result .

(1) procedure = abstraction .

(2) proc-argument = value | cell .

(3) proc-argument-list = list [proc-argument] .

(4) function = abstraction .

(5) fun-argument = value .

(6) fun-argument-list = list [fun-argument] .

(7) fun-result = value .

B.2.5 Arrays

introduces: array .

(1) array = list [cell] .

B.3 Semantic Functions

needs: **Abstract Syntax** , **Semantic Entities** .

B.3.1 Programs

introduces: run _ .

- run _ :: Program → action .

(1) run ⟦ "`program`" I:Identifier "`is`" D:Declaration "`in`" C:Command "`end`" ⟧ =
    | elaborate D
    hence
    | execute C .

B.3.2 Declarations

introduces: elaborate _ , elaborate-formals _ , elaborate-formal _ ,
    evaluate-array-components _ .

B.3.2.1 Elaborating Declarations

- elaborate _ :: Declaration → action .

(1) elaborate ⟦ "`const`" I:Identifier "`:`" T:PrimitiveType "`=`" E:Expression ⟧ =
    evaluate E then bind token-of I to the primitive-value .

(2) elaborate ⟦ "`var`" I:Identifier "`:`" T:PrimitiveType "`:=`" E:Expression ⟧ =
    | | evaluate E then give the primitive-value label #1
    | and
    | | allocate-for-primitive-value T then give the cell label #2
    then
    | | bind token-of I to the cell#2
    | and
    | | store the primitive-value#1 in the cell#2 .

```
(3) elaborate [[ "varray" I:Identifier ":" T:ArrayType ":=" A:ArrayComps ]] =
        evaluate-array-components A then give the list label #1
      and
        allocate-for-array T then give the list label #2
    then
        unfolding
              check (the list#1 is empty-list)
            and then
              complete
          or
              check not (the list#1 is empty-list)
            and then
                store head-of (the list#1) in head-of (the list#2)
              and then
                    give tail-of (the list#1) label #1
                  and
                    give tail-of (the list#2) label #2
                then
                  unfold
      and
        bind token-of I to the list#2 .

(4) elaborate [[ "proc" I:Identifier "(" FS:ProcFormals ")" "=" C:Command ]] =
    recursively bind token-of I to closure abstraction
          furthermore elaborate-proc-formals FS
        hence
          execute C .

(5) elaborate [[ "fun" I:Identifier "(" FS:FunFormals ")" ":" T:ValueType "="
          E:Expression ]] =
    recursively bind token-of I to closure abstraction
          furthermore elaborate-fun–formals FS
        hence
          evaluate E then give the fun-result .

(6) elaborate [[ D1:Declaration ";" D2:Declaration ]] =
      elaborate D1
    before
      elaborate D2 .
```

B.3.2.2 Evaluating Array Components

- evaluate-array-components _ :: ArrayComps → action .

```
(1) evaluate-array-components [[ E:Expression ]] =
    evaluate E then give list (primitive-value) .
```

(2) evaluate-array-components ⟦ E:Expression , A:ArrayComps ⟧ =
| | evaluate E then give list (primitive-value) label #1
| and
| | evaluate-array-components A then give the list label #2
then
| give concatenation (the list#1,the list#2) .

B.3.2.3 Elaborating Procedure Formal Parameters

- elaborate-proc-formals _ :: ProcFormals → action .

(1) elaborate-proc-formals ⟦ ⟧ = complete .

(2) elaborate-proc-formals ⟦ F:ProcFormal ⟧ =
give head-of (the proc-argument-list) then elaborate-proc-formal F .

(3) elaborate-proc-formals ⟦ F:ProcFormal "," FS:ProcFormals ⟧ =
| give head-of (the proc-argument-list) then elaborate-proc-formal F
and then
| give tail-of (the proc-argument-list) then elaborate-proc-formals FS .

B.3.2.4 Elaborating Procedure Formal Parameter

- elaborate-proc-formal _ :: ProcFormal → action .

(1) elaborate-proc-formal ⟦ I:Identifier ":" T:PrimitiveType ⟧ =
| | give the primitive-value label #1
| and
| | allocate-for-primitive-value T then give the cell label #2
then
| bind token-of I to the cell#2 and store the primitive-value#1 in the cell#2 .

(2) elaborate-proc-formal ⟦ I:Identifier ":" T:FunctionType ⟧ =
bind token-of I to the function .

(3) elaborate-proc-formal ⟦ "`var`" I:Identifier ":" T:Type ⟧ = bind token-of I to the cell .

B.3.2.5 Elaborating Function Formal Parameters

- elaborate-fun-formals _ :: FunFormals → action .

(1) elaborate-fun-formals ⟦ ⟧ = complete .

(2) elaborate-fun-formals ⟦ F:FunFormal ⟧ =
give head-of (the fun-argument-list) then elaborate-fun-formal F .

(3) elaborate-fun-formals ⟦ F:FunFormal "," FS:FunFormals ⟧ =
| give head-of (the fun-argument-list) then elaborate-fun-formal F
and then
| give tail-of (the fun-argument-list) then elaborate-fun-formals FS .

B.3.2.6 Elaborating Function Formal Parameter

- elaborate-fun-formal _ :: FunFormal → action .

(1) elaborate-fun-formal ⟦ I:Identifier ":" T:ValueType ⟧ =
    bind token-of I to the fun-argument .

B.3.2.7 Allocating Storage

introduces: allocate-for-primitive-value _ , allocate-for-array _ .

- allocate-for-primitive-value _ :: PrimitiveType → action .

(1) allocate-for-primitive-value ⟦ "`bool`" ⟧ = allocate a truth-value-cell .

(2) allocate-for-primitive-value ⟦ "`int`" ⟧ = allocate an integer-cell .

- allocate-for-array _ :: ArrayType → action .

(3) allocate-for-array ⟦ "`array`" "`[`" N:Numeral "`]`" "`of`" T:PrimitiveType ⟧ =

```
    | | give valuation-of N label #1
    | and
    | | give 1 label #2
    | and
    | | give empty-list label #3
    then
    | unfolding
    | | | | | check not (the integer#1 is the integer#2)
    | | | | and then
    | | | | | | give the integer#1 label #1
    | | | | | and
    | | | | | | give successor (the integer#2) label #2
    | | | | | and
    | | | | | | | allocate-for-primitive-value T and give the list#3 label #1
    | | | | | | then
    | | | | | | | give concatenation (the list#1,list (the cell)) label #3
    | | | then
    | | | | unfold
    | | or
    | | | | | check (the integer#1 is the integer#2)
    | | | | and then
    | | | | | allocate-for-primitive-value T and give the list#3 label #1
    | | | then
    | | | | give concatenation(the list#1,list (the cell)) .
```

B.3.3 Commands

introduces: execute _ , evaluate-actuals _ , evaluate-actual _.

B.3.3.1 Executing Commands

- execute _ :: Command → action .

(1) execute ⟦ I:Identifier ":=" E:Expression ⟧ =
  | evaluate E
  then
  | store the primitive-value in the cell bound to token-of I .

(2) execute ⟦ I:Identifier "[" E_1:Expression "]" ":=" E_2:Expression ⟧ =
  | | evaluate E_1 then give the integer label #1
  | and
  | | evaluate E_2 then give the primitive-value label #2
  | and
  | | give the list bound to token-of I label #3
  | and
  | | give 1 label #4
  then
  | unfolding
  | | | | check (the integer#1 is the integer#2)
  | | | and then
  | | | | store the primitive-value#2 in head-of (the list#3)
  | | or
  | | | | check not (the integer#1 is the integer#2)
  | | | and then
  | | | | | | give the integer#1 label #1
  | | | | | and
  | | | | | | give the primitive-value#2 label #2
  | | | | | and
  | | | | | | give tail-of (the list#3) label #3
  | | | | | and
  | | | | | | give successor (the integer#4) label #4
  | | | | then
  | | | | | unfold .

(3) execute ⟦ "if" E:Expression "then" C:Command "end" ⟧ =
  | evaluate E
  then
  | | execute C
  | else
  | | complete .

(4) execute ⟦ "if" E:Expression "then" C_1:Command "else" C_2:Command "end" ⟧ =
  | evaluate E
  then
  | | execute C_1
  | else
  | | execute C_2 .

(5) execute ⟦ "`while`" E:Expression "`do`" C:Command "`end`" ⟧ =
  unfolding
  | | evaluate E
  | then
  | | | execute C and then unfold
  | | else
  | | | complete .

(6) execute ⟦ "`call`" I:Identifier "`(`" AS:ProcActuals "`)`" ⟧ =
  | evaluate-proc-actuals AS
  then
  | enact (the procedure bound to token-of I with the proc-argument-list) .

(7) execute ⟦ "`local`" D:Declaration "`in`" C:Command "`end`" ⟧ =
  | furthermore elaborate D
  hence
  | execute C .

(8) execute ⟦ C_1:Command "`;`" C_2:Command ⟧ = execute C_1 and then execute C_2 .

B.3.3.2 Evaluating Procedure Actual Parameters

- evaluate-proc-actuals _ :: ProcActuals → action .

(1) evaluate-proc-actuals ⟦ ⟧ = give empty-list .

(2) evaluate-proc-actuals ⟦ A:ProcActual ⟧ = evaluate-proc-actual A then give list (it) .

(3) evaluate-proc-actuals ⟦ A:ProcActual "`,`" AS:ProcActuals ⟧ =
  | | evaluate-proc-actual A then give list (it) label #1
  | and
  | | evaluate-proc-actuals AS then give it label #2
  then
  | | give concatenation (the list#1,the list#2) .

B.3.3.3 Evaluating Procedure Actual Parameter

- evaluate-proc-actual _ :: ProcActual → action .

(1) evaluate-proc-actual ⟦ E:Expression ⟧ = evaluate E .

(2) evaluate-proc-actual ⟦ "`var`" I:Identifier ⟧ = give the cell bound to token-of I .

B.3.4 Expressions

introduces: evaluate _ , apply-operator _ .

B.3.4.1 Evaluating Expressions

- evaluate _ :: Expression → action .

(1) evaluate ⟦ "`true`" ⟧ = give true .

(2) evaluate ⟦ "`false`" ⟧ = give false .

(3) evaluate ⟦ N:Numeral ⟧ = give valuation-of N .

(4) evaluate ⟦ I:Identifier ⟧ =
    | give the value bound to token-of I
    or
    | give the primitive-value stored in the cell bound to token-of I .

(5) evaluate ⟦ I:Identifier "`[`" E:Expression "`]`" ⟧ =
    | | evaluate E then give the integer label #1
    | and
    | | give the list bound to token-of I label #2
    | and
    | | give 1 label #3
    then
    | unfolding
    | | | | check (the integer#1 is the integer#3)
    | | | and then
    | | | | give the primitive-value stored in head-of (the list#2)
    | | or
    | | | | check not (the integer#1 is the integer#3)
    | | | and then
    | | | | | | give the integer#1 label #1
    | | | | | and
    | | | | | | give tail-of (the list#2) label #2
    | | | | | and
    | | | | | | give successor (the integer#3) label #3
    | | | | then
    | | | | | unfold .

(6) evaluate ⟦ "`if`" E_1:Expression "`then`" E_2:Expression "`end`" ⟧ =
    | evaluate E_1
    then
    | | evaluate E_2
    | else
    | | complete .

(7) evaluate ⟦ "`if`" E_1:Expression "`then`" E_2:Expression "`else`" E_3:Expression "`end`" ⟧ =
    | evaluate E_1
    then
    | | evaluate E_2
    | else
    | | evaluate E_3 .

(8) evaluate ⟦ I:Identifier "`(`" AS:FunActuals "`)`" ⟧ =
    | evaluate-fun-actuals AS
    then
    | enact (the function bound to token-of I with the fun-argument-list) .

(9) evaluate ⟦ E_1:Expression O:Operator E_2:Expression ⟧ =
| | evaluate E_1 then give the value label #1
| and
| | evaluate E_2 then give the value label #2
then
| apply-operator O .

(10) evaluate ⟦ O:Operator E:Expression ⟧ =
| evaluate E then give the value
then
| apply-operator O .

(11) execute ⟦ "`let`" D:Declaration "`in`" E:Expression "`end`" ⟧ =
| furthermore elaborate D
hence
| evaluate E .

B.3.4.2 Evaluating Function Actual Parameters

- evaluate-fun-actuals _ :: FunActuals → action .

(1) evaluate-fun-actuals ⟦ ⟧ = give empty-list .

(2) evaluate-fun-actuals ⟦ A:Actual ⟧ = evaluate-fun-actual A then give list (it) .

(3) evaluate-fun-actuals ⟦ A:Actual "," AS:Actuals ⟧ =
| | evaluate-fun-actual A then give list (it) label #1
| and
| | evaluate-fun-actuals AS then give it label #2
then
| | give concatenation (the list#1,the list#2) .

B.3.4.3 Evaluating Function Actual Parameter

- evaluate-fun-actual _ :: FunActual → action .

(1) evaluate-fun-actual ⟦ E:Expression ⟧ = evaluate E .

B.3.4.4 Applying Operators

- apply-operator _ :: Operator → action .

(1) apply-operator ⟦ "`not`" ⟧ = give not (the truth-value) .

(2) apply-operator ⟦ "`and`" ⟧ = give both (the truth-value#1, the truth-value#2) .

(3) apply-operator ⟦ "`or`" ⟧ = give either (the truth-value#1, the truth-value#2) .

(4) apply-operator ⟦ "~" ⟧ = give negation (the integer) .

(5) apply-operator ⟦ "+" ⟧ = give sum (the integer#1, the integer#2) .

(6) apply-operator ⟦ "-" ⟧ = give difference (the integer#1, the integer#2) .

(7) apply-operator ⟦ “*” ⟧ = give product (the integer#1, the integer#2) .

(8) apply-operator ⟦ “/” ⟧ = give integer-quotient (the integer#1, the integer#2) .

(9) apply-operator ⟦ “<” ⟧ = give is-less-than (the integer#1, the integer#2) .

(10) apply-operator ⟦ “>” ⟧ = give is-greater-than (the integer#1, the integer#2) .

(11) apply-operator ⟦ “<=” ⟧ = give either (is-less-than (the integer#1, the integer#2),
the integer#1 is the integer#2) .

(12) apply-operator ⟦ “>=” ⟧ = give either (is-greater-than (the integer#1, the integer#2),
the integer#1 is the integer#2) .

(13) apply-operator ⟦ “=” ⟧ = give (the primitive-value#1 is the primitive-value#2) .

(14) apply-operator ⟦ “<>” ⟧ = give not (the primitive-value#1 is the primitive-value#2) .

B.4 Lexical Syntax

introduces: token-of _ , valuation-of _ .

closed

grammar:

(1) Identifier = letter | Identifier letter | Identifier digit .

(2) Numeral = digit | Numeral digit .

(3) digit = 0 | 1 | . . . | 9 .

(4) letter = `A` | `B` | . . . | `Z` | `a` | . . . | `z` .

B.4.1 Identifiers

- token-of _ :: Identifier → token .

(1) token-of ⟦ I ⟧ = I .

B.4.2 Numerals

- valuation-of _ :: Numeral → integer .

(1) valuation-of ⟦ 0 ⟧ = 0 .

(2) valuation-of ⟦ 1 ⟧ = 1 .

(3) valuation-of ⟦ 2 ⟧ = 2 .

(4) valuation-of ⟦ 3 ⟧ = 3 .

(5) valuation-of ⟦ 4 ⟧ = 4 .

(6) valuation-of ⟦ 5 ⟧ = 5 .

(7) valuation-of $[\![$ 6 $]\!]$ = 6 .

(8) valuation-of $[\![$ 7 $]\!]$ = 7 .

(9) valuation-of $[\![$ 8 $]\!]$ = 8 .

(10) valuation-of $[\![$ 9 $]\!]$ = 9 .

(11) valuation-of $[\![$ N D $]\!]$ = sum (product (valuation-of N, 10), valuation-of D) .

Index